QEMU Documentation

Release 9.0.50

The QEMU Project Developers

May 18, 2024

CONTENTS:

CHAPTER

ONE

ABOUT QEMU

QEMU is a generic and open source machine emulator and virtualizer.

QEMU can be used in several different ways. The most common is for *System Emulation*, where it provides a virtual model of an entire machine (CPU, memory and emulated devices) to run a guest OS. In this mode the CPU may be fully emulated, or it may work with a hypervisor such as KVM, Xen or Hypervisor.Framework to allow the guest to run directly on the host CPU.

The second supported way to use QEMU is *User Mode Emulation*, where QEMU can launch processes compiled for one CPU on another CPU. In this mode the CPU is always emulated.

QEMU also provides a number of standalone *command line utilities*, such as the `qemu-img` disk image utility that allows you to create, convert and modify disk images.

1.1 Supported build platforms

QEMU aims to support building and executing on multiple host OS platforms. This appendix outlines which platforms are the major build targets. These platforms are used as the basis for deciding upon the minimum required versions of 3rd party software QEMU depends on. The supported platforms are the targets for automated testing performed by the project when patches are submitted for review, and tested before and after merge.

If a platform is not listed here, it does not imply that QEMU won't work. If an unlisted platform has comparable software versions to a listed platform, there is every expectation that it will work. Bug reports are welcome for problems encountered on unlisted platforms unless they are clearly older vintage than what is described here.

Note that when considering software versions shipped in distros as support targets, QEMU considers only the version number, and assumes the features in that distro match the upstream release with the same version. In other words, if a distro backports extra features to the software in their distro, QEMU upstream code will not add explicit support for those backports, unless the feature is auto-detectable in a manner that works for the upstream releases too.

The Repology site is a useful resource to identify currently shipped versions of software in various operating systems, though it does not cover all distros listed below.

1.1.1 Supported host architectures

Those hosts are officially supported, with various accelerators:

CPU Architecture	Accelerators
Arm	kvm (64 bit only), tcg, xen
MIPS (little endian only)	kvm, tcg
PPC	kvm, tcg
RISC-V	kvm, tcg
s390x	kvm, tcg
SPARC	tcg
x86	hvf (64 bit only), kvm, nvmm, tcg, whpx (64 bit only), xen

Other host architectures are not supported. It is possible to build QEMU system emulation on an unsupported host architecture using the configure `--enable-tcg-interpreter` option to enable the TCI support, but note that this is very slow and is not recommended for normal use. QEMU user emulation requires host-specific support for signal handling, therefore TCI won't help on unsupported host architectures.

Non-supported architectures may be removed in the future following the *deprecation process*.

1.1.2 Linux OS, macOS, FreeBSD, NetBSD, OpenBSD

The project aims to support the most recent major version at all times for up to five years after its initial release. Support for the previous major version will be dropped 2 years after the new major version is released or when the vendor itself drops support, whichever comes first. In this context, third-party efforts to extend the lifetime of a distro are not considered, even when they are endorsed by the vendor (eg. Debian LTS); the same is true of repositories that contain packages backported from later releases (e.g. Debian backports). Within each major release, only the most recent minor release is considered.

For the purposes of identifying supported software versions available on Linux, the project will look at CentOS, Debian, Fedora, openSUSE, RHEL, SLES and Ubuntu LTS. Other distros will be assumed to ship similar software versions.

For FreeBSD and OpenBSD, decisions will be made based on the contents of the respective ports repository, while NetBSD will use the pkgsrc repository.

For macOS, Homebrew will be used, although MacPorts is expected to carry similar versions.

Some build dependencies may follow less conservative rules:

Python runtime
: Distributions with long-term support often provide multiple versions of the Python runtime. While QEMU will initially aim to support the distribution's default runtime, it may later increase its minimum version to any newer python that is available as an option from the vendor. In this case, it will be necessary to use the `--python` command line option of the `configure` script to point QEMU to a supported version of the Python runtime.

  As of QEMU 9.0.50, the minimum supported version of Python is 3.7.

Python build dependencies
: Some of QEMU's build dependencies are written in Python. Usually these are only packaged by distributions for the default Python runtime. If QEMU bumps its minimum Python version and a non-default runtime is required, it may be necessary to fetch python modules from the Python Package Index (PyPI) via `pip`, in order to build QEMU.

Optional build dependencies
: Build components whose absence does not affect the ability to build QEMU may not be available in distros, or may be too old for QEMU's requirements. Many of these, such as the Avocado testing framework or various linters, are written in Python and therefore can also be installed using `pip`. Cross compilers are another example

of optional build-time dependency; in this case it is possible to download them from repositories such as EPEL, to use container-based cross compilation using `docker` or `podman`, or to use pre-built binaries distributed with QEMU.

1.1.3 Windows

The project aims to support the two most recent versions of Windows that are still supported by the vendor. The minimum Windows API that is currently targeted is "Windows 8", so theoretically the QEMU binaries can still be run on older versions of Windows, too. However, such old versions of Windows are not tested anymore, so it is recommended to use one of the latest versions of Windows instead.

The project supports building QEMU with current versions of the MinGW toolchain, either hosted on Linux (Debian/Fedora) or via MSYS2 on Windows. A more recent Windows version is always preferred as it is less likely to have problems with building via MSYS2. The building process of QEMU involves some Python scripts that call os.symlink() which needs special attention for the build process to successfully complete. On newer versions of Windows 10, unprivileged accounts can create symlinks if Developer Mode is enabled. When Developer Mode is not available/enabled, the SeCreateSymbolicLinkPrivilege privilege is required, or the process must be run as an administrator.

Only 64-bit Windows is supported.

1.2 Emulation

QEMU's Tiny Code Generator (TCG) provides the ability to emulate a number of CPU architectures on any supported host platform. Both *System Emulation* and *User Mode Emulation* are supported depending on the guest architecture.

Table 1: Supported Guest Architectures for Emulation

Architecture (qemu name)	System	User	Notes
Alpha	Yes	Yes	Legacy 64 bit RISC ISA developed by DEC
Arm (arm, aarch64)	*Yes*	Yes	Wide range of features, see *A-profile CPU architecture support* for details
AVR	*Yes*	No	8 bit micro controller, often used in maker projects
Cris	Yes	Yes	Embedded RISC chip developed by AXIS
Hexagon	No	Yes	Family of DSPs by Qualcomm
PA-RISC (hppa)	Yes	Yes	A legacy RISC system used in HP's old minicomputers
x86 (i386, x86_64)	*Yes*	Yes	The ubiquitous desktop PC CPU architecture, 32 and 64 bit.
Loongarch	Yes	Yes	A MIPS-like 64bit RISC architecture developed in China
m68k	*Yes*	Yes	Motorola 68000 variants and ColdFire
Microblaze	Yes	Yes	RISC based soft-core by Xilinx
MIPS (mips*)	*Yes*	Yes	Venerable RISC architecture originally out of Stanford University
OpenRISC	*Yes*	Yes	Open source RISC architecture developed by the OpenRISC community
Power (ppc, ppc64)	*Yes*	Yes	A general purpose RISC architecture now managed by IBM
RISC-V	*Yes*	Yes	An open standard RISC ISA maintained by RISC-V International
RX	*Yes*	No	A 32 bit micro controller developed by Renesas
s390x	*Yes*	Yes	A 64 bit CPU found in IBM's System Z mainframes
sh4	Yes	Yes	A 32 bit RISC embedded CPU developed by Hitachi
SPARC (sparc, sparc64)	*Yes*	Yes	A RISC ISA originally developed by Sun Microsystems
Tricore	Yes	No	A 32 bit RISC/uController/DSP developed by Infineon
Xtensa	*Yes*	Yes	A configurable 32 bit soft core now owned by Cadence

A number of features are only available when running under emulation including *Record/Replay* and *QEMU TCG Plugins*.

1.2.1 Semihosting

Semihosting is a feature defined by the owner of the architecture to allow programs to interact with a debugging host system. On real hardware this is usually provided by an In-circuit emulator (ICE) hooked directly to the board. QEMU's implementation allows for semihosting calls to be passed to the host system or via the `gdbstub`.

Generally semihosting makes it easier to bring up low level code before a more fully functional operating system has been enabled. On QEMU it also allows for embedded micro-controller code which typically doesn't have a full libc to be run as "bare-metal" code under QEMU's user-mode emulation. It is also useful for writing test cases and indeed a number of compiler suites as well as QEMU itself use semihosting calls to exit test code while reporting the success state.

Semihosting is only available using TCG emulation. This is because the instructions to trigger a semihosting call are typically reserved causing most hypervisors to trap and fault on them.

Warning: Semihosting inherently bypasses any isolation there may be between the guest and the host. As a result a program using semihosting can happily trash your host system. Some semihosting calls (e.g. `SYS_READC`) can block execution indefinitely. You should only ever run trusted code with semihosting enabled.

Redirection

Semihosting calls can be re-directed to a (potentially remote) gdb during debugging via the *gdbstub*. Output to the semihosting console is configured as a `chardev` so can be redirected to a file, pipe or socket like any other `chardev` device.

Supported Targets

Most targets offer similar semihosting implementations with some minor changes to define the appropriate instruction to encode the semihosting call and which registers hold the parameters. They tend to presents a simple POSIX-like API which allows your program to read and write files, access the console and some other basic interactions.

For full details of the ABI for a particular target, and the set of calls it provides, you should consult the semihosting specification for that architecture.

Note: QEMU makes an implementation decision to implement all file access in `O_BINARY` mode. The user-visible effect of this is regardless of the text/binary mode the program sets QEMU will always select a binary mode ensuring no line-terminator conversion is performed on input or output. This is because gdb semihosting support doesn't make the distinction between the modes and magically processing line endings can be confusing.

Table 2: Guest Architectures supporting Semihosting

Architecture	Modes	Specification
Arm	System and User-mode	https://github.com/ARM-software/abi-aa/blob/main/semihosting/semihosting.rst
m68k	System	https://sourceware.org/git/?p=newlib-cygwin.git;a=blob;f=libgloss/m68k/m68k-semi.txt;hb=HEAD
MIPS	System	Unified Hosting Interface (MD01069)
RISC-V	System and User-mode	https://github.com/riscv/riscv-semihosting-spec/blob/main/riscv-semihosting-spec.adoc
Xtensa	System	Tensilica ISS SIMCALL

1.3 Deprecated features

In general features are intended to be supported indefinitely once introduced into QEMU. In the event that a feature needs to be removed, it will be listed in this section. The feature will remain functional for the release in which it was deprecated and one further release. After these two releases, the feature is liable to be removed. Deprecated features may also generate warnings on the console when QEMU starts up, or if activated via a monitor command, however, this is not a mandatory requirement.

Prior to the 2.10.0 release there was no official policy on how long features would be deprecated prior to their removal, nor any documented list of which features were deprecated. Thus any features deprecated prior to 2.10.0 will be treated as if they were first deprecated in the 2.10.0 release.

What follows is a list of all features currently marked as deprecated.

1.3.1 System emulator command line arguments

Short-form boolean options (since 6.0)

Boolean options such as `share=on`/`share=off` could be written in short form as `share` and `noshare`. This is now deprecated and will cause a warning.

`delay` option for socket character devices (since 6.0)

The replacement for the `nodelay` short-form boolean option is `nodelay=on` rather than `delay=off`.

Plugin argument passing through `arg=<string>` (since 6.1)

Passing TCG plugins arguments through `arg=` is redundant is makes the command-line less readable, especially when the argument itself consist of a name and a value, e.g. `-plugin plugin_name,arg="arg_name=arg_value"`. Therefore, the usage of `arg` is redundant. Single-word arguments are treated as short-form boolean values, and passed to plugins as `arg_name=on`. However, short-form booleans are deprecated and full explicit `arg_name=on` form is preferred.

`-smp` (Unsupported "parameter=1" SMP configurations) (since 9.0)

Specified CPU topology parameters must be supported by the machine.

In the SMP configuration, users should provide the CPU topology parameters that are supported by the target machine.

However, historically it was allowed for users to specify the unsupported topology parameter as "1", which is meaningless. So support for this kind of configurations (e.g. -smp drawers=1,books=1,clusters=1 for x86 PC machine) is marked deprecated since 9.0, users have to ensure that all the topology members described with -smp are supported by the target machine.

1.3.2 `-runas` (since 9.1)

Use `-run-with user=..` instead.

1.3.3 User-mode emulator command line arguments

`-p` (since 9.0)

The `-p` option pretends to control the host page size. However, it is not possible to change the host page size, and using the option only causes failures.

1.3.4 QEMU Machine Protocol (QMP) commands

`blockdev-open-tray`, `blockdev-close-tray` argument `device` (since 2.8)

Use argument `id` instead.

`eject` argument `device` (since 2.8)

Use argument `id` instead.

`blockdev-change-medium` argument `device` (since 2.8)

Use argument `id` instead.

`block_set_io_throttle` argument `device` (since 2.8)

Use argument `id` instead.

`blockdev-add` empty string argument `backing` (since 2.10)

Use argument value `null` instead.

`block-commit` arguments `base` and `top` (since 3.1)

Use arguments `base-node` and `top-node` instead.

`nbd-server-add` and `nbd-server-remove` (since 5.2)

Use the more generic commands `block-export-add` and `block-export-del` instead. As part of this deprecation, where `nbd-server-add` used a single `bitmap`, the new `block-export-add` uses a list of `bitmaps`.

`query-qmp-schema` return value member `values` (since 6.2)

Member `values` in return value elements with meta-type `enum` is deprecated. Use `members` instead.

`drive-backup` (since 6.2)

Use `blockdev-backup` in combination with `blockdev-add` instead. This change primarily separates the creation/opening process of the backup target with explicit, separate steps. `blockdev-backup` uses mostly the same arguments as `drive-backup`, except the `format` and `mode` options are removed in favor of using explicit `blockdev-create` and `blockdev-add` calls. See *Live Block Device Operations* for details.

Incorrectly typed `device_add` arguments (since 6.2)

Due to shortcomings in the internal implementation of `device_add`, QEMU incorrectly accepts certain invalid arguments: Any object or list arguments are silently ignored. Other argument types are not checked, but an implicit conversion happens, so that e.g. string values can be assigned to integer device properties or vice versa.

This is a bug in QEMU that will be fixed in the future so that previously accepted incorrect commands will return an error. Users should make sure that all arguments passed to `device_add` are consistent with the documented property types.

1.3.5 QEMU Machine Protocol (QMP) events

`MEM_UNPLUG_ERROR` (since 6.2)

Use the more generic event `DEVICE_UNPLUG_GUEST_ERROR` instead.

`vcpu` trace events (since 8.1)

The ability to instrument QEMU helper functions with vCPU-aware trace points was removed in 7.0. However QMP still exposed the vcpu parameter. This argument has now been deprecated and the remaining remaining trace points that used it are selected just by name.

1.3.6 Host Architectures

BE MIPS (since 7.2)

As Debian 10 ("Buster") moved into LTS the big endian 32 bit version of MIPS moved out of support making it hard to maintain our cross-compilation CI tests of the architecture. As we no longer have CI coverage support may bitrot away before the deprecation process completes. The little endian variants of MIPS (both 32 and 64 bit) are still a supported host architecture.

System emulation on 32-bit x86 hosts (since 8.0)

Support for 32-bit x86 host deployments is increasingly uncommon in mainstream OS distributions given the widespread availability of 64-bit x86 hardware. The QEMU project no longer considers 32-bit x86 support for system emulation to be an effective use of its limited resources, and thus intends to discontinue it. Since all recent x86 hardware from the past >10 years is capable of the 64-bit x86 extensions, a corresponding 64-bit OS should be used instead.

1.3.7 System emulator CPUs

`power5+` and `power7+` CPU names (since 9.0)

The character "+" in device (and thus also CPU) names is not allowed in the QEMU object model anymore. `power5+`, `power5+_v2.1`, `power7+` and `power7+_v2.1` are currently still supported via an alias, but for consistency these will get removed in a future release, too. Use `power5p_v2.1` and `power7p_v2.1` instead.

`Sun-UltraSparc-IIIi+` and `Sun-UltraSparc-IV+` CPU names (since 9.1)

The character "+" in device (and thus also CPU) names is not allowed in the QEMU object model anymore. `Sun-UltraSparc-IIIi+` and `Sun-UltraSparc-IV+` are currently still supported via a workaround, but for consistency these will get removed in a future release, too. Use `Sun-UltraSparc-IIIi-plus` and `Sun-UltraSparc-IV-plus` instead.

CRIS CPU architecture (since 9.0)

The CRIS architecture was pulled from Linux in 4.17 and the compiler is no longer packaged in any distro making it harder to run the `check-tcg` tests. Unless we can improve the testing situation there is a chance the code will bitrot without anyone noticing.

1.3.8 System emulator machines

Arm `virt` machine `dtb-kaslr-seed` property (since 7.1)

The `dtb-kaslr-seed` property on the `virt` board has been deprecated; use the new name `dtb-randomness` instead. The new name better reflects the way this property affects all random data within the device tree blob, not just the `kaslr-seed` node.

`pc-i440fx-2.0` up to `pc-i440fx-2.3` (since 8.2)

These old machine types are quite neglected nowadays and thus might have various pitfalls with regards to live migration. Use a newer machine type instead.

`shix` (since 9.0)

The machine is no longer in existence and has been long unmaintained in QEMU. This also holds for the TC51828 16MiB flash that it uses.

`pseries-2.1` up to `pseries-2.12` (since 9.0)

Older pseries machines before version 3.0 have undergone many changes to correct issues, mostly regarding migration compatibility. These are no longer maintained and removing them will make the code easier to read and maintain. Use versions 3.0 and above as a replacement.

Arm machines `akita`, `borzoi`, `cheetah`, `connex`, `mainstone`, `n800`, `n810`, `spitz`, `terrier`, `tosa`, `verdex`, `z2` (since 9.0)

QEMU includes models of some machine types where the QEMU code that emulates their SoCs is very old and unmaintained. This code is now blocking our ability to move forward with various changes across the codebase, and over many years nobody has been interested in trying to modernise it. We don't expect any of these machines to have a large number of users, because they're all modelling hardware that has now passed away into history. We are therefore dropping support for all machine types using the PXA2xx and OMAP2 SoCs. We are also dropping the `cheetah` OMAP1 board, because we don't have any test images for it and don't know of anybody who does; the `sx1` and `sx1-v1` OMAP1 machines remain supported for now.

PPC 405 `ref405ep` machine (since 9.1)

The `ref405ep` machine and PPC 405 CPU have no known users, firmware images are not available, OpenWRT dropped support in 2019, U-Boot in 2017, Linux also is dropping support in 2024. It is time to let go of this ancient hardware and focus on newer CPUs and platforms.

1.3.9 Backend options

Using non-persistent backing file with pmem=on (since 6.1)

This option is used when `memory-backend-file` is consumed by emulated NVDIMM device. However enabling `memory-backend-file.pmem` option, when backing file is (a) not DAX capable or (b) not on a filesystem that support direct mapping of persistent memory, is not safe and may lead to data loss or corruption in case of host crash. Options are:

- modify VM configuration to set `pmem=off` to continue using fake NVDIMM (without persistence guaranties) with backing file on non DAX storage
- move backing file to NVDIMM storage and keep `pmem=on` (to have NVDIMM with persistence guaranties).

1.3.10 Device options

Emulated device options

`-device virtio-blk,scsi=on|off` (since 5.0)

The virtio-blk SCSI passthrough feature is a legacy VIRTIO feature. VIRTIO 1.0 and later do not support it because the virtio-scsi device was introduced for full SCSI support. Use virtio-scsi instead when SCSI passthrough is required.

Note this also applies to `-device virtio-blk-pci,scsi=on|off`, which is an alias.

`-device nvme-ns,eui64-default=on|off` (since 7.1)

In QEMU versions 6.1, 6.2 and 7.0, the `nvme-ns` generates an EUI-64 identifier that is not globally unique. If an EUI-64 identifier is required, the user must set it explicitly using the `nvme-ns` device parameter `eui64`.

`-device nvme,use-intel-id=on|off` (since 7.1)

The `nvme` device originally used a PCI Vendor/Device Identifier combination from Intel that was not properly allocated. Since version 5.2, the controller has used a properly allocated identifier. Deprecate the `use-intel-id` machine compatibility parameter.

`-device cxl-type3,memdev=xxxx` (since 8.0)

The `cxl-type3` device initially only used a single memory backend. With the addition of volatile memory support, it is now necessary to distinguish between persistent and volatile memory backends. As such, memdev is deprecated in favor of persistent-memdev.

`-fsdev proxy` and `-virtfs proxy` (since 8.1)

The 9p `proxy` filesystem backend driver has been deprecated and will be removed (along with its proxy helper daemon) in a future version of QEMU. Please use `-fsdev local` or `-virtfs local` for using the 9p `local` filesystem backend, or alternatively consider deploying virtiofsd instead.

The 9p `proxy` backend was originally developed as an alternative to the 9p `local` backend. The idea was to enhance security by dispatching actual low level filesystem operations from 9p server (QEMU process) over to a separate process (the virtfs-proxy-helper binary). However this alternative never gained momentum. The proxy backend is much slower than the local backend, hasn't seen any development in years, and showed to be less secure, especially due to the fact that its helper daemon must be run as root, whereas with the local backend QEMU is typically run as unprivileged user and allows to tighten behaviour by mapping permissions et al by using its 'mapped' security model option.

Nowadays it would make sense to reimplement the `proxy` backend by using QEMU's `vhost` feature, which would eliminate the high latency costs under which the 9p `proxy` backend currently suffers. However as of to date nobody has indicated plans for such kind of reimplementation unfortunately.

RISC-V 'any' CPU type `-cpu any` (since 8.2)

The 'any' CPU type was introduced back in 2018 and has been around since the initial RISC-V QEMU port. Its usage has always been unclear: users don't know what to expect from a CPU called 'any', and in fact the CPU does not do anything special that isn't already done by the default CPUs rv32/rv64.

After the introduction of the 'max' CPU type, RISC-V now has a good coverage of generic CPUs: rv32 and rv64 as default CPUs and 'max' as a feature complete CPU for both 32 and 64 bit builds. Users are then discouraged to use the 'any' CPU type starting in 8.2.

RISC-V CPU properties which start with capital 'Z' (since 8.2)

All RISC-V CPU properties which start with capital 'Z' are being deprecated starting in 8.2. The reason is that they were wrongly added with capital 'Z' in the past. CPU properties were later added with lower-case names, which is the format we want to use from now on.

Users which try to use these deprecated properties will receive a warning recommending to switch to their stable counterparts:

- "Zifencei" should be replaced with "zifencei"
- "Zicsr" should be replaced with "zicsr"
- "Zihintntl" should be replaced with "zihintntl"
- "Zihintpause" should be replaced with "zihintpause"
- "Zawrs" should be replaced with "zawrs"
- "Zfa" should be replaced with "zfa"
- "Zfh" should be replaced with "zfh"
- "Zfhmin" should be replaced with "zfhmin"

- "Zve32f" should be replaced with "zve32f"
- "Zve64f" should be replaced with "zve64f"
- "Zve64d" should be replaced with "zve64d"

Block device options

`"backing": ""` (since 2.12)

In order to prevent QEMU from automatically opening an image's backing chain, use `"backing": null` instead.

`rbd` keyvalue pair encoded filenames: `""` (since 3.1)

Options for `rbd` should be specified according to its runtime options, like other block drivers. Legacy parsing of keyvalue pair encoded filenames is useful to open images with the old format for backing files; These image files should be updated to use the current format.

Example of legacy encoding:

```
json:{"file.driver":"rbd", "file.filename":"rbd:rbd/name"}
```

The above, converted to the current supported format:

```
json:{"file.driver":"rbd", "file.pool":"rbd", "file.image":"name"}
```

`iscsi,password=xxx` (since 8.0)

Specifying the iSCSI password in plain text on the command line using the `password` option is insecure. The `password-secret` option should be used instead, to refer to a `--object secret...` instance that provides a password via a file, or encrypted.

Character device options

Backend `memory` (since 9.0)

`memory` is a deprecated synonym for `ringbuf`.

CPU device properties

`pcommit` on x86 (since 9.1)

The PCOMMIT instruction was never included in any physical processor. It was implemented as a no-op instruction in TCG up to QEMU 9.0, but only with `-cpu max` (which does not guarantee migration compatibility across versions).

`pmu-num=n` on RISC-V CPUs (since 8.2)

In order to support more flexible counter configurations this has been replaced by a `pmu-mask` property. If set of counters is continuous then the mask can be calculated with `((2 ^ n) - 1) << 3`. The least significant three bits must be left clear.

1.3.11 Backwards compatibility

Runnability guarantee of CPU models (since 4.1)

Previous versions of QEMU never changed existing CPU models in ways that introduced additional host software or hardware requirements to the VM. This allowed management software to safely change the machine type of an existing VM without introducing new requirements ("runnability guarantee"). This prevented CPU models from being updated to include CPU vulnerability mitigations, leaving guests vulnerable in the default configuration.

The CPU model runnability guarantee won't apply anymore to existing CPU models. Management software that needs runnability guarantees must resolve the CPU model aliases using the `alias-of` field returned by the `query-cpu-definitions` QMP command.

While those guarantees are kept, the return value of `query-cpu-definitions` will have existing CPU model aliases point to a version that doesn't break runnability guarantees (specifically, version 1 of those CPU models). In future QEMU versions, aliases will point to newer CPU model versions depending on the machine type, so management software must resolve CPU model aliases before starting a virtual machine.

1.3.12 QEMU guest agent

`--blacklist` command line option (since 7.2)

`--blacklist` has been replaced by `--block-rpcs` (which is a better wording for what this option does). The short form `-b` still stays the same and thus is the preferred way for scripts that should run with both, older and future versions of QEMU.

`blacklist` config file option (since 7.2)

The `blacklist` config file option has been renamed to `block-rpcs` (to be in sync with the renaming of the corresponding command line option).

1.3.13 Migration

`fd:` URI when used for file migration (since 9.1)

The `fd:` URI can currently provide a file descriptor that references either a socket or a plain file. These are two different types of migration. In order to reduce ambiguity, the `fd:` URI usage of providing a file descriptor to a plain file has been deprecated in favor of explicitly using the `file:` URI with the file descriptor being passed as an `fdset`. Refer to the `add-fd` command documentation for details on the `fdset` usage.

1.4 Removed features

What follows is a record of recently removed, formerly deprecated features that serves as a record for users who have encountered trouble after a recent upgrade.

1.4.1 System emulator command line arguments

`-hdachs` (removed in 2.12)

The geometry defined by `-hdachs c,h,s,t` should now be specified via `-device ide-hd,drive=dr,cyls=c,heads=h,secs=s,bios-chs-trans=t` (together with `-drive if=none,id=dr,...`).

`-net channel` (removed in 2.12)

This option has been replaced by `-net user,guestfwd=...`.

`-net dump` (removed in 2.12)

`-net dump[,vlan=n][,file=filename][,len=maxlen]` has been replaced by `-object filter-dump,id=id,netdev=dev[,file=filename][,maxlen=maxlen]`. Note that the new syntax works with netdev IDs instead of the old "vlan" hubs.

`-no-kvm-pit` (removed in 2.12)

This was just a dummy option that has been ignored, since the in-kernel PIT cannot be disabled separately from the irqchip anymore. A similar effect (which also disables the KVM IOAPIC) can be obtained with `-M kernel_irqchip=split`.

`-tdf` (removed in 2.12)

There is no replacement, the `-tdf` option has just been ignored since the behaviour that could be changed by this option in qemu-kvm is now the default when using the KVM PIT. It still can be requested explicitly using `-global kvm-pit.lost_tick_policy=delay`.

`-drive secs=s`, `-drive heads=h` & `-drive cyls=c` (removed in 3.0)

The drive geometry should now be specified via `-device ...,drive=dr,cyls=c,heads=h,secs=s` (together with `-drive if=none,id=dr,...`).

-drive serial=, -drive trans= & -drive addr= (removed in 3.0)

Use -device ...,drive=dr,serial=r,bios-chs-trans=t,addr=a instead (together with -drive if=none, id=dr,...).

-net ...,vlan=x (removed in 3.0)

The term "vlan" was very confusing for most users in this context (it's about specifying a hub ID, not about IEEE 802.1Q or something similar), so this has been removed. To connect one NIC frontend with a network backend, either use -nic ... (e.g. for on-board NICs) or use -netdev ...,id=n together with -device ...,netdev=n (for full control over pluggable NICs). To connect multiple NICs or network backends via a hub device (which is what vlan did), use -nic hubport,hubid=x,... or -netdev hubport,id=n,hubid=x,... (with -device ...,netdev=n) instead.

-no-kvm-irqchip (removed in 3.0)

Use -machine kernel_irqchip=off instead.

-no-kvm-pit-reinjection (removed in 3.0)

Use -global kvm-pit.lost_tick_policy=discard instead.

-balloon (removed in 3.1)

The -balloon virtio option has been replaced by -device virtio-balloon. The -balloon none option was a no-op and has no replacement.

-bootp (removed in 3.1)

The -bootp /some/file argument is replaced by either -netdev user,id=x,bootp=/some/file (for pluggable NICs, accompanied with -device ...,netdev=x), or -nic user,bootp=/some/file (for on-board NICs). The new syntax allows different settings to be provided per NIC.

-redir (removed in 3.1)

The -redir [tcp|udp]:hostport:[guestaddr]:guestport option is replaced by either -netdev user,id=x,hostfwd=[tcp|udp]:[hostaddr]:hostport-[guestaddr]:guestport (for pluggable NICs, accompanied with -device ...,netdev=x) or by the option -nic user, hostfwd=[tcp|udp]:[hostaddr]:hostport-[guestaddr]:guestport (for on-board NICs). The new syntax allows different settings to be provided per NIC.

-smb (removed in 3.1)

The -smb /some/dir argument is replaced by either -netdev user,id=x,smb=/some/dir (for pluggable NICs, accompanied with -device ...,netdev=x), or -nic user,smb=/some/dir (for on-board NICs). The new syntax allows different settings to be provided per NIC.

-tftp (removed in 3.1)

The -tftp /some/dir argument is replaced by either -netdev user,id=x,tftp=/some/dir (for pluggable NICs, accompanied with -device ...,netdev=x), or -nic user,tftp=/some/dir (for embedded NICs). The new syntax allows different settings to be provided per NIC.

-localtime (removed in 3.1)

Replaced by -rtc base=localtime.

-nodefconfig (removed in 3.1)

Use -no-user-config instead.

-rtc-td-hack (removed in 3.1)

Use -rtc driftfix=slew instead.

-startdate (removed in 3.1)

Replaced by -rtc base=date.

-vnc ...,tls=..., -vnc ...,x509=... & -vnc ...,x509verify=... (removed in 3.1)

The "tls-creds" option should be used instead to point to a "tls-creds-x509" object created using "-object".

-mem-path fallback to RAM (removed in 5.0)

If guest RAM allocation from file pointed by mem-path failed, QEMU was falling back to allocating from RAM, which might have resulted in unpredictable behavior since the backing file specified by the user as ignored. Currently, users are responsible for making sure the backing storage specified with -mem-path can actually provide the guest RAM configured with -m and QEMU fails to start up if RAM allocation is unsuccessful.

-net ...,name=... (removed in 5.1)

The name parameter of the -net option was a synonym for the id parameter, which should now be used instead.

-numa node,mem=... (removed in 5.1)

The parameter mem of -numa node was used to assign a part of guest RAM to a NUMA node. But when using it, it's impossible to manage a specified RAM chunk on the host side (like bind it to a host node, setting bind policy, ...), so the guest ends up with the fake NUMA configuration with suboptiomal performance. However since 2014 there is an alternative way to assign RAM to a NUMA node using parameter memdev, which does the same as mem and adds means to actually manage node RAM on the host side. Use parameter memdev with *memory-backend-ram* backend as replacement for parameter mem to achieve the same fake NUMA effect or a properly configured *memory-backend-file* backend to actually benefit from NUMA configuration. New machine versions (since 5.1) will not accept the option but it will still work with old machine types. User can check the QAPI schema to see if the legacy option is supported by looking at MachineInfo::numa-mem-supported property.

-numa node (without memory specified) (removed in 5.2)

Splitting RAM by default between NUMA nodes had the same issues as mem parameter with the difference that the role of the user plays QEMU using implicit generic or board specific splitting rule. Use memdev with *memory-backend-ram* backend or mem (if it's supported by used machine type) to define mapping explicitly instead. Users of existing VMs, wishing to preserve the same RAM distribution, should configure it explicitly using -numa node,memdev options. Current RAM distribution can be retrieved using HMP command info numa and if separate memory devices (pc|nv-dimm) are present use info memory-device and subtract device memory from output of info numa.

-smp (invalid topologies) (removed in 5.2)

CPU topology properties should describe whole machine topology including possible CPUs.

However, historically it was possible to start QEMU with an incorrect topology where *n* <= *sockets* * *cores* * *threads* < *maxcpus*, which could lead to an incorrect topology enumeration by the guest. Support for invalid topologies is removed, the user must ensure topologies described with -smp include all possible cpus, i.e. *sockets* * *cores* * *threads* = *maxcpus*.

-machine enforce-config-section=on|off (removed in 5.2)

The enforce-config-section property was replaced by the -global migration.send-configuration={on|off} option.

-no-kvm (removed in 5.2)

The -no-kvm argument was a synonym for setting -machine accel=tcg.

`-realtime` (removed in 6.0)

The `-realtime mlock=on|off` argument has been replaced by the `-overcommit mem-lock=on|off` argument.

`-show-cursor` option (removed in 6.0)

Use `-display sdl,show-cursor=on`, `-display gtk,show-cursor=on` or `-display default, show-cursor=on` instead.

`-tb-size` option (removed in 6.0)

QEMU 5.0 introduced an alternative syntax to specify the size of the translation block cache, `-accel tcg,tb-size=`.

`-usbdevice audio` (removed in 6.0)

This option lacked the possibility to specify an audio backend device. Use `-device usb-audio` now instead (and specify a corresponding USB host controller or `-usb` if necessary).

`-vnc acl` (removed in 6.0)

The `acl` option to the `-vnc` argument has been replaced by the `tls-authz` and `sasl-authz` options.

`-mon ...,control=readline,pretty=on|off` (removed in 6.0)

The `pretty=on|off` switch has no effect for HMP monitors and its use is rejected.

`-drive file=json:{...{'driver':'file'}}` (removed in 6.0)

The 'file' driver for drives is no longer appropriate for character or host devices and will only accept regular files (S_IFREG). The correct driver for these file types is 'host_cdrom' or 'host_device' as appropriate.

Floppy controllers' drive properties (removed in 6.0)

Use `-device floppy,...` instead. When configuring onboard floppy controllers

```
-global isa-fdc.driveA=...
-global sysbus-fdc.driveA=...
-global SUNW,fdtwo.drive=...
```

become

```
-device floppy,unit=0,drive=...
```

and

```
-global isa-fdc.driveB=...
-global sysbus-fdc.driveB=...
```

become

```
-device floppy,unit=1,drive=...
```

When plugging in a floppy controller

```
-device isa-fdc,...,driveA=...
```

becomes

```
-device isa-fdc,...
-device floppy,unit=0,drive=...
```

and

```
-device isa-fdc,...,driveB=...
```

becomes

```
-device isa-fdc,...
-device floppy,unit=1,drive=...
```

`-drive` with bogus interface type (removed in 6.0)

Drives with interface types other than `if=none` are for onboard devices. Drives the board doesn't pick up can no longer be used with -device. Use `if=none` instead.

`-usbdevice ccid` (removed in 6.0)

This option was undocumented and not used in the field. Use `-device usb-ccid` instead.

RISC-V firmware not booted by default (removed in 5.1)

QEMU 5.1 changes the default behaviour from `-bios none` to `-bios default` for the RISC-V `virt` machine and `sifive_u` machine.

`-no-quit` (removed in 7.0)

The `-no-quit` was a synonym for `-display ...,window-close=off` which should be used instead.

`--enable-fips` (removed in 7.1)

This option restricted usage of certain cryptographic algorithms when the host is operating in FIPS mode.

If FIPS compliance is required, QEMU should be built with the `libgcrypt` or `gnutls` library enabled as a cryptography provider.

Neither the `nettle` library, or the built-in cryptography provider are supported on FIPS enabled hosts.

`-writeconfig` (removed in 7.1)

The `-writeconfig` option was not able to serialize the entire contents of the QEMU command line. It is thus considered a failed experiment and removed without a replacement.

`loaded` property of `secret` and `secret_keyring` objects (removed in 7.1)

The `loaded=on` option in the command line or QMP `object-add` either had no effect (if `loaded` was the last option) or caused options to be effectively ignored as if they were not given. The property is therefore useless and should simply be removed.

`opened` property of `rng-*` objects (removed in 7.1)

The `opened=on` option in the command line or QMP `object-add` either had no effect (if `opened` was the last option) or caused errors. The property is therefore useless and should simply be removed.

`-display sdl,window_close=...` (removed in 7.1)

Use `-display sdl,window-close=...` instead (i.e. with a minus instead of an underscore between "window" and "close").

`-alt-grab` and `-display sdl,alt_grab=on` (removed in 7.1)

Use `-display sdl,grab-mod=lshift-lctrl-lalt` instead.

`-ctrl-grab` and `-display sdl,ctrl_grab=on` (removed in 7.1)

Use `-display sdl,grab-mod=rctrl` instead.

`-sdl` (removed in 7.1)

Use `-display sdl` instead.

`-curses` (removed in 7.1)

Use `-display curses` instead.

Creating sound card devices using `-soundhw` (removed in 7.1)

Sound card devices should be created using `-device` or `-audio`. The exception is `pcspk` which can be activated using `-machine pcspk-audiodev=<name>`.

`-watchdog` (since 7.2)

Use `-device` instead.

Hexadecimal sizes with scaling multipliers (since 8.0)

Input parameters that take a size value should only use a size suffix (such as 'k' or 'M') when the base is written in decimal, and not when the value is hexadecimal. That is, '0x20M' should be written either as '32M' or as '0x2000000'.

`-chardev` backend aliases `tty` and `parport` (removed in 8.0)

`tty` and `parport` used to be aliases for `serial` and `parallel` respectively. The actual backend names should be used instead.

`-drive if=none` for the sifive_u OTP device (removed in 8.0)

Use `-drive if=pflash` to configure the OTP device of the sifive_u RISC-V machine instead.

`-spice password=string` (removed in 8.0)

This option was insecure because the SPICE password remained visible in the process listing. This was replaced by the new `password-secret` option which lets the password be securely provided on the command line using a `secret` object instance.

`QEMU_AUDIO_` environment variables and `-audio-help` (removed in 8.2)

The `-audiodev` and `-audio` command line options are now the only way to specify audio backend settings.

Using `-audiodev` to define the default audio backend (removed in 8.2)

If no audiodev property is specified, previous versions would use the first `-audiodev` command line option as a fallback. Starting with version 8.2, audio backends created with `-audiodev` will only be used by clients (sound cards, machines with embedded sound hardware, VNC) that refer to it in an `audiodev=` property.

In order to configure a default audio backend, use the `-audio` command line option without specifying a `model`; while previous versions of QEMU required a model, starting with version 8.2 QEMU does not require a model and will not create any sound card in this case.

Note that the default audio backend must be configured on the command line if the `-nodefaults` options is used.

`-no-hpet` (removed in 9.0)

The HPET setting has been turned into a machine property. Use `-machine hpet=off` instead.

-no-acpi (removed in 9.0)

The -no-acpi setting has been turned into a machine property. Use -machine acpi=off instead.

-async-teardown (removed in 9.0)

Use -run-with async-teardown=on instead.

-chroot (removed in 9.0)

Use -run-with chroot=dir instead.

-singlestep (removed in 9.0)

The -singlestep option has been turned into an accelerator property, and given a name that better reflects what it actually does. Use -accel tcg,one-insn-per-tb=on instead.

-smp ("parameter=0" SMP configurations) (removed in 9.0)

Specified CPU topology parameters must be greater than zero.

In the SMP configuration, users should either provide a CPU topology parameter with a reasonable value (greater than zero) or just omit it and QEMU will compute the missing value.

However, historically it was implicitly allowed for users to provide a parameter with zero value, which is meaningless and could also possibly cause unexpected results in the -smp parsing. So support for this kind of configurations (e.g. -smp 8,sockets=0) is removed since 9.0, users have to ensure that all the topology members described with -smp are greater than zero.

-global migration.decompress-error-check (removed in 9.1)

Removed along with the compression migration capability.

1.4.2 User-mode emulator command line arguments

-singlestep (removed in 9.0)

The -singlestep option has been given a name that better reflects what it actually does. For both linux-user and bsd-user, use the -one-insn-per-tb option instead.

1.4.3 QEMU Machine Protocol (QMP) commands

block-dirty-bitmap-add "autoload" parameter (removed in 4.2)

The "autoload" parameter has been ignored since 2.12.0. All bitmaps are automatically loaded from qcow2 images.

`cpu-add` (removed in 5.2)

Use `device_add` for hotplugging vCPUs instead of `cpu-add`. See documentation of `query-hotpluggable-cpus` for additional details.

`change` (removed in 6.0)

Use `blockdev-change-medium` or `change-vnc-password` or `display-update` instead.

`query-events` (removed in 6.0)

The `query-events` command has been superseded by the more powerful and accurate `query-qmp-schema` command.

`migrate_set_cache_size` and `query-migrate-cache-size` (removed in 6.0)

Use `migrate_set_parameter` and `info migrate_parameters` instead.

`migrate_set_downtime` and `migrate_set_speed` (removed in 6.0)

Use `migrate_set_parameter` instead.

`query-cpus` (removed in 6.0)

The `query-cpus` command is replaced by the `query-cpus-fast` command.

`query-cpus-fast` `arch` output member (removed in 6.0)

The `arch` output member of the `query-cpus-fast` command is replaced by the `target` output member.

chardev client socket with `wait` option (removed in 6.0)

Character devices creating sockets in client mode should not specify the 'wait' field, which is only applicable to sockets in server mode

`query-named-block-nodes` result `encryption_key_missing` (removed in 6.0)

Removed with no replacement.

`query-block` result `inserted.encryption_key_missing` (removed in 6.0)

Removed with no replacement.

`query-named-block-nodes` and `query-block` result dirty-bitmaps[i].status (removed in 6.0)

The `status` field of the `BlockDirtyInfo` structure, returned by these commands is removed. Two new boolean fields, `recording` and `busy` effectively replace it.

`query-block` result field `dirty-bitmaps` (removed in 6.0)

The `dirty-bitmaps` field of the `BlockInfo` structure, returned by the query-block command is itself now removed. The `dirty-bitmaps` field of the `BlockDeviceInfo` struct should be used instead, which is the type of the `inserted` field in query-block replies, as well as the type of array items in query-named-block-nodes.

`object-add` option `props` (removed in 6.0)

Specify the properties for the object as top-level arguments instead.

`query-sgx` return value member `section-size` (removed in 8.0)

Member `section-size` in the return value of `query-sgx` was superseded by `sections`.

`query-sgx-capabilities` return value member `section-size` (removed in 8.0)

Member `section-size` in the return value of `query-sgx-capabilities` was superseded by `sections`.

`query-migrate` return value member `skipped` (removed in 9.1)

Member `skipped` of the `MigrationStats` struct hasn't been used for more than 10 years. Removed with no replacement.

`migrate` command option `inc` (removed in 9.1)

Use blockdev-mirror with NBD instead. See "QMP invocation for live storage migration with `blockdev-mirror` + NBD" in docs/interop/live-block-operations.rst for a detailed explanation.

`migrate` command option `blk` (removed in 9.1)

Use blockdev-mirror with NBD instead. See "QMP invocation for live storage migration with `blockdev-mirror` + NBD" in docs/interop/live-block-operations.rst for a detailed explanation.

`migrate-set-capabilities` `block` option (removed in 9.1)

Block migration has been removed. For a replacement, see "QMP invocation for live storage migration with `blockdev-mirror` + NBD" in docs/interop/live-block-operations.rst.

`migrate-set-parameter compress-level` option (removed in 9.1)

Use `multifd-zlib-level` or `multifd-zstd-level` instead.

`migrate-set-parameter compress-threads` option (removed in 9.1)

Use `multifd-channels` instead.

`migrate-set-parameter compress-wait-thread` option (removed in 9.1)

Removed with no replacement.

`migrate-set-parameter decompress-threads` option (removed in 9.1)

Use `multifd-channels` instead.

`migrate-set-capability compress` option (removed in 9.1)

Use `multifd-compression` instead.

1.4.4 Human Monitor Protocol (HMP) commands

`usb_add` and `usb_remove` (removed in 2.12)

Replaced by `device_add` and `device_del` (use `device_add help` for a list of available devices).

`host_net_add` and `host_net_remove` (removed in 2.12)

Replaced by `netdev_add` and `netdev_del`.

The `hub_id` parameter of `hostfwd_add` / `hostfwd_remove` (removed in 5.0)

The `[hub_id name]` parameter tuple of the 'hostfwd_add' and 'hostfwd_remove' HMP commands has been replaced by `netdev_id`.

`cpu-add` (removed in 5.2)

Use `device_add` for hotplugging vCPUs instead of `cpu-add`. See documentation of `query-hotpluggable-cpus` for additional details.

change vnc TARGET (removed in 6.0)

No replacement. The `change vnc password` and `change DEVICE MEDIUM` commands are not affected.

acl_show, acl_reset, acl_policy, acl_add, acl_remove (removed in 6.0)

The `acl_show`, `acl_reset`, `acl_policy`, `acl_add`, and `acl_remove` commands were removed with no replacement. Authorization for VNC should be performed using the pluggable QAuthZ objects.

migrate-set-cache-size and info migrate-cache-size (removed in 6.0)

Use `migrate-set-parameters` and `info migrate-parameters` instead.

migrate_set_downtime and migrate_set_speed (removed in 6.0)

Use `migrate-set-parameters` instead.

info cpustats (removed in 6.1)

This command didn't produce any output already. Removed with no replacement.

singlestep (removed in 9.0)

The `singlestep` command has been replaced by the `one-insn-per-tb` command, which has the same behaviour but a less misleading name.

migrate command -i option (removed in 9.1)

Use blockdev-mirror with NBD instead. See "QMP invocation for live storage migration with `blockdev-mirror` + NBD" in docs/interop/live-block-operations.rst for a detailed explanation.

migrate command -b option (removed in 9.1)

Use blockdev-mirror with NBD instead. See "QMP invocation for live storage migration with `blockdev-mirror` + NBD" in docs/interop/live-block-operations.rst for a detailed explanation.

migrate_set_capability block option (removed in 9.1)

Block migration has been removed. For a replacement, see "QMP invocation for live storage migration with `blockdev-mirror` + NBD" in docs/interop/live-block-operations.rst.

`migrate_set_parameter compress-level` option (removed in 9.1)

Use `multifd-zlib-level` or `multifd-zstd-level` instead.

`migrate_set_parameter compress-threads` option (removed in 9.1)

Use `multifd-channels` instead.

`migrate_set_parameter compress-wait-thread` option (removed in 9.1)

Removed with no replacement.

`migrate_set_parameter decompress-threads` option (removed in 9.1)

Use `multifd-channels` instead.

`migrate_set_capability compress` option (removed in 9.1)

Use `multifd-compression` instead.

1.4.5 Host Architectures

System emulation on 32-bit Windows hosts (removed in 9.0)

Windows 11 has no support for 32-bit host installs, and Windows 10 did not support new 32-bit installs, only upgrades. 32-bit Windows support has now been dropped by the MSYS2 project. QEMU also is deprecating and dropping support for 32-bit x86 host deployments in general. 32-bit Windows is therefore no longer a supported host for QEMU. Since all recent x86 hardware from the past >10 years is capable of the 64-bit x86 extensions, a corresponding 64-bit OS should be used instead.

1.4.6 Guest Emulator ISAs

RISC-V ISA privilege specification version 1.09.1 (removed in 5.1)

The RISC-V ISA privilege specification version 1.09.1 has been removed. QEMU supports both the newer version 1.10.0 and the ratified version 1.11.0, these should be used instead of the 1.09.1 version.

1.4.7 System emulator CPUS

KVM guest support on 32-bit Arm hosts (removed in 5.2)

The Linux kernel has dropped support for allowing 32-bit Arm systems to host KVM guests as of the 5.7 kernel, and was thus removed from QEMU as well. Running 32-bit guests on a 64-bit Arm host remains supported.

RISC-V ISA Specific CPUs (removed in 5.1)

The RISC-V cpus with the ISA version in the CPU name have been removed. The four CPUs are: `rv32gcsu-v1.9.1`, `rv32gcsu-v1.10.0`, `rv64gcsu-v1.9.1` and `rv64gcsu-v1.10.0`. Instead the version can be specified via the CPU `priv_spec` option when using the `rv32` or `rv64` CPUs.

RISC-V no MMU CPUs (removed in 5.1)

The RISC-V no MMU cpus have been removed. The two CPUs: `rv32imacu-nommu` and `rv64imacu-nommu` can no longer be used. Instead the MMU status can be specified via the CPU `mmu` option when using the `rv32` or `rv64` CPUs.

`compat` property of server class POWER CPUs (removed in 6.0)

The `max-cpu-compat` property of the `pseries` machine type should be used instead.

`moxie` CPU (removed in 6.1)

Nobody was using this CPU emulation in QEMU, and there were no test images available to make sure that the code is still working, so it has been removed without replacement.

`lm32` CPUs (removed in 6.1)

The only public user of this architecture was the milkymist project, which has been dead for years; there was never an upstream Linux port. Removed without replacement.

`unicore32` CPUs (removed in 6.1)

Support for this CPU was removed from the upstream Linux kernel, and there is no available upstream toolchain to build binaries for it. Removed without replacement.

x86 `Icelake-Client` CPU (removed in 7.1)

There isn't ever Icelake Client CPU, it is some wrong and imaginary one. Use `Icelake-Server` instead.

Nios II CPU (removed in 9.1)

QEMU Nios II architecture was orphan; Intel has EOL'ed the Nios II processor IP (see Intel discontinuance notification).

1.4.8 System accelerators

Userspace local APIC with KVM (x86, removed in 8.0)

`-M kernel-irqchip=off` cannot be used on KVM if the CPU model includes a local APIC. The `split` setting is supported, as is using `-M kernel-irqchip=off` when the CPU does not have a local APIC.

HAXM (`-accel hax`) (removed in 8.2)

The HAXM project has been retired (see https://github.com/intel/haxm#status). Use "whpx" (on Windows) or "hvf" (on macOS) instead.

MIPS "Trap-and-Emulate" KVM support (removed in 8.0)

The MIPS "Trap-and-Emulate" KVM host and guest support was removed from Linux in 2021, and is not supported anymore by QEMU either.

1.4.9 System emulator machines

`s390-virtio` (removed in 2.6)

Use the `s390-ccw-virtio` machine instead.

The m68k `dummy` machine (removed in 2.9)

Use the `none` machine with the `loader` device instead.

`xlnx-ep108` (removed in 3.0)

The EP108 was an early access development board that is no longer used. Use the `xlnx-zcu102` machine instead.

`spike_v1.9.1` and `spike_v1.10` (removed in 5.1)

The version specific Spike machines have been removed in favour of the generic `spike` machine. If you need to specify an older version of the RISC-V spec you can use the `-cpu rv64gcsu,priv_spec=v1.10.0` command line argument.

mips `r4k` platform (removed in 5.2)

This machine type was very old and unmaintained. Users should use the `malta` machine type instead.

mips `fulong2e` machine alias (removed in 6.0)

This machine has been renamed `fuloong2e`.

`pc-0.10` up to `pc-i440fx-1.7` (removed in 4.0 up to 8.2)

These machine types were very old and likely could not be used for live migration from old QEMU versions anymore. Use a newer machine type instead.

Raspberry Pi `raspi2` and `raspi3` machines (removed in 6.2)

The Raspberry Pi machines come in various models (A, A+, B, B+). To be able to distinguish which model QEMU is implementing, the `raspi2` and `raspi3` machines have been renamed `raspi2b` and `raspi3b`.

Aspeed `swift-bmc` machine (removed in 7.0)

This machine was removed because it was unused. Alternative AST2500 based OpenPOWER machines are `witherspoon-bmc` and `romulus-bmc`.

ppc `taihu` machine (removed in 7.2)

This machine was removed because it was partially emulated and 405 machines are very similar. Use the `ref405ep` machine instead.

Nios II `10m50-ghrd` and `nios2-generic-nommu` machines (removed in 9.1)

The Nios II architecture was orphan.

1.4.10 linux-user mode CPUs

`tilegx` CPUs (removed in 6.0)

The `tilegx` guest CPU support has been removed without replacement. It was only implemented in linux-user mode, but support for this CPU was removed from the upstream Linux kernel in 2018, and it has also been dropped from glibc, so there is no new Linux development taking place with this architecture. For running the old binaries, you can use older versions of QEMU.

`ppc64abi32` CPUs (removed in 7.0)

The `ppc64abi32` architecture has a number of issues which regularly tripped up the CI testing and was suspected to be quite broken. For that reason the maintainers strongly suspected no one actually used it.

`nios2` CPU (removed in 9.1)

QEMU Nios II architecture was orphan; Intel has EOL'ed the Nios II processor IP (see Intel discontinuance notification).

1.4.11 TCG introspection features

TCG trace-events (since 6.2)

The ability to add new TCG trace points had bit rotted and as the feature can be replicated with TCG plugins it was removed. If any user is currently using this feature and needs help with converting to using TCG plugins they should contact the qemu-devel mailing list.

1.4.12 System emulator devices

`spapr-pci-vfio-host-bridge` (removed in 2.12)

The `spapr-pci-vfio-host-bridge` device type has been replaced by the `spapr-pci-host-bridge` device type.

`ivshmem` (removed in 4.0)

Replaced by either the `ivshmem-plain` or `ivshmem-doorbell`.

`ide-drive` (removed in 6.0)

The 'ide-drive' device has been removed. Users should use 'ide-hd' or 'ide-cd' as appropriate to get an IDE hard disk or CD-ROM as needed.

`scsi-disk` (removed in 6.0)

The 'scsi-disk' device has been removed. Users should use 'scsi-hd' or 'scsi-cd' as appropriate to get a SCSI hard disk or CD-ROM as needed.

`sga` (removed in 8.0)

The `sga` device loaded an option ROM for x86 targets which enabled SeaBIOS to send messages to the serial console. SeaBIOS 1.11.0 onwards contains native support for this feature and thus use of the option ROM approach was obsolete. The native SeaBIOS support can be activated by using `-machine graphics=off`.

pvrdma and the RDMA subsystem (removed in 9.1)

The 'pvrdma' device and the whole RDMA subsystem have been removed.

1.4.13 Related binaries

qemu-nbd --partition (removed in 5.0)

The `qemu-nbd --partition $digit` code (also spelled `-P`) could only handle MBR partitions, and never correctly handled logical partitions beyond partition 5. Exporting a partition can still be done by utilizing the `--image-opts` option with a raw blockdev using the `offset` and `size` parameters layered on top of any other existing blockdev. For example, if partition 1 is 100MiB long starting at 1MiB, the old command:

```
qemu-nbd -t -P 1 -f qcow2 file.qcow2
```

can be rewritten as:

```
qemu-nbd -t --image-opts driver=raw,offset=1M,size=100M,file.driver=qcow2,file.file.
↪driver=file,file.file.filename=file.qcow2
```

qemu-img convert -n -o (removed in 5.1)

All options specified in `-o` are image creation options, so they are now rejected when used with `-n` to skip image creation.

qemu-img create -b bad file $size (removed in 5.1)

When creating an image with a backing file that could not be opened, `qemu-img create` used to issue a warning about the failure but proceed with the image creation if an explicit size was provided. However, as the `-u` option exists for this purpose, it is safer to enforce that any failure to open the backing image (including if the backing file is missing or an incorrect format was specified) is an error when `-u` is not used.

qemu-img amend to adjust backing file (removed in 6.1)

The use of `qemu-img amend` to modify the name or format of a qcow2 backing image was never fully documented or tested, and interferes with other amend operations that need access to the original backing image (such as deciding whether a v3 zero cluster may be left unallocated when converting to a v2 image). Any changes to the backing chain should be performed with `qemu-img rebase -u` either before or after the remaining changes being performed by amend, as appropriate.

qemu-img backing file without format (removed in 6.1)

The use of `qemu-img create`, `qemu-img rebase`, or `qemu-img convert` to create or modify an image that depends on a backing file now requires that an explicit backing format be provided. This is for safety: if QEMU probes a different format than what you thought, the data presented to the guest will be corrupt; similarly, presenting a raw image to a guest allows a potential security exploit if a future probe sees a non-raw image based on guest writes.

To avoid creating unsafe backing chains, you must pass `-o backing_fmt=` (or the shorthand `-F` during create) to specify the intended backing format. You may use `qemu-img rebase -u` to retroactively add a backing format to an existing image. However, be aware that there are already potential security risks to blindly using `qemu-img info` to probe the format of an untrusted backing image, when deciding what format to add into an existing image.

1.4.14 Block devices

VXHS backend (removed in 5.1)

The VXHS code did not compile since v2.12.0. It was removed in 5.1.

sheepdog driver (removed in 6.0)

The corresponding upstream server project is no longer maintained. Users are recommended to switch to an alternative distributed block device driver such as RBD.

1.4.15 Tools

virtiofsd (removed in 8.0)

There is a newer Rust implementation of `virtiofsd` at `https://gitlab.com/virtio-fs/virtiofsd`; this has been stable for some time and is now widely used. The command line and feature set is very close to the removed C implementation.

1.5 License

QEMU is a trademark of Fabrice Bellard.

QEMU is released under the GNU General Public License, version 2. Parts of QEMU have specific licenses, see file LICENSE.

CHAPTER

TWO

SYSTEM EMULATION

This section of the manual is the overall guide for users using QEMU for full system emulation (as opposed to user-mode emulation). This includes working with hypervisors such as KVM, Xen or Hypervisor.Framework.

2.1 Introduction

2.1.1 Virtualisation Accelerators

QEMU's system emulation provides a virtual model of a machine (CPU, memory and emulated devices) to run a guest OS. It supports a number of hypervisors (known as accelerators) as well as a JIT known as the Tiny Code Generator (TCG) capable of emulating many CPUs.

Table 1: Supported Accelerators

Accelerator	Host OS	Host Architectures
KVM	Linux	Arm (64 bit only), MIPS, PPC, RISC-V, s390x, x86
Xen	Linux (as dom0)	Arm, x86
Hypervisor Framework (hvf)	MacOS	x86 (64 bit only), Arm (64 bit only)
Windows Hypervisor Platform (whpx)	Windows	x86
NetBSD Virtual Machine Monitor (nvmm)	NetBSD	x86
Tiny Code Generator (tcg)	Linux, other POSIX, Windows, MacOS	Arm, x86, Loongarch64, MIPS, PPC, s390x, Sparc64

2.1.2 Feature Overview

System emulation provides a wide range of device models to emulate various hardware components you may want to add to your machine. This includes a wide number of VirtIO devices which are specifically tuned for efficient operation under virtualisation. Some of the device emulation can be offloaded from the main QEMU process using either vhost-user (for VirtIO) or *Multi-process QEMU*. If the platform supports it QEMU also supports directly passing devices through to guest VMs to eliminate the device emulation overhead. See *Device Emulation* for more details.

There is a full *featured block layer* which allows for construction of complex storage topology which can be stacked across multiple layers supporting redirection, networking, snapshots and migration support.

The flexible `chardev` system allows for handling IO from character like devices using stdio, files, unix sockets and TCP networking.

QEMU provides a number of management interfaces including a line based *Human Monitor Protocol (HMP)* that allows you to dynamically add and remove devices as well as introspect the system state. The *QEMU Monitor Protocol* (QMP) is a well defined, versioned, machine usable API that presents a rich interface to other tools to create, control and manage Virtual Machines. This is the interface used by higher level tools interfaces such as Virt Manager using the libvirt framework.

For the common accelerators QEMU, supported debugging with its *gdbstub* which allows users to connect GDB and debug system software images.

2.1.3 Running

QEMU provides a rich and complex API which can be overwhelming to understand. While some architectures can boot something with just a disk image, those examples elide a lot of details with defaults that may not be optimal for modern systems.

For a non-x86 system where we emulate a broad range of machine types, the command lines are generally more explicit in defining the machine and boot behaviour. You will find often find example command lines in the *QEMU System Emulator Targets* section of the manual.

While the project doesn't want to discourage users from using the command line to launch VMs, we do want to highlight that there are a number of projects dedicated to providing a more user friendly experience. Those built around the `libvirt` framework can make use of feature probing to build modern VM images tailored to run on the hardware you have.

That said, the general form of a QEMU command line can be expressed as:

```
$ qemu-system-x86_64 [machine opts] \
                [cpu opts] \
                [accelerator opts] \
                [device opts] \
                [backend opts] \
                [interface opts] \
                [boot opts]
```

Most options will generate some help information. So for example:

```
$ qemu-system-x86_64 -M help
```

will list the machine types supported by that QEMU binary. `help` can also be passed as an argument to another option. For example:

```
$ qemu-system-x86_64 -device scsi-hd,help
```

will list the arguments and their default values of additional options that can control the behaviour of the `scsi-hd` device.

Table 2: Options Overview

Options	
Machine	Define the machine type, amount of memory etc
CPU	Type and number/topology of vCPUs. Most accelerators offer a `host` cpu option which simply passes through your host CPU configuration without filtering out any features.
Accelerator	This will depend on the hypervisor you run. Note that the default is TCG, which is purely emulated, so you must specify an accelerator type to take advantage of hardware virtualization.
Devices	Additional devices that are not defined by default with the machine type.
Backends	Backends are how QEMU deals with the guest's data, for example how a block device is stored, how network devices see the network or how a serial device is directed to the outside world.
Interfaces	How the system is displayed, how it is managed and controlled or debugged.
Boot	How the system boots, via firmware or direct kernel boot.

In the following example we first define a `virt` machine which is a general purpose platform for running Aarch64 guests. We enable virtualisation so we can use KVM inside the emulated guest. As the `virt` machine comes with some built in pflash devices we give them names so we can override the defaults later.

```
$ qemu-system-aarch64 \
   -machine type=virt,virtualization=on,pflash0=rom,pflash1=efivars \
   -m 4096 \
```

We then define the 4 vCPUs using the `max` option which gives us all the Arm features QEMU is capable of emulating. We enable a more emulation friendly implementation of Arm's pointer authentication algorithm. We explicitly specify TCG acceleration even though QEMU would default to it anyway.

```
-cpu max,pauth-impdef=on \
-smp 4 \
-accel tcg \
```

As the `virt` platform doesn't have any default network or storage devices we need to define them. We give them ids so we can link them with the backend later on.

```
-device virtio-net-pci,netdev=unet \
-device virtio-scsi-pci \
-device scsi-hd,drive=hd \
```

We connect the user-mode networking to our network device. As user-mode networking isn't directly accessible from the outside world we forward localhost port 2222 to the ssh port on the guest.

```
-netdev user,id=unet,hostfwd=tcp::2222-:22 \
```

We connect the guest visible block device to an LVM partition we have set aside for our guest.

```
-blockdev driver=raw,node-name=hd,file.driver=host_device,file.filename=/dev/lvm-disk/
↪debian-bullseye-arm64 \
```

We then tell QEMU to multiplex the *QEMU Monitor* with the serial port output (we can switch between the two using *Keys in the character backend multiplexer*). As there is no default graphical device we disable the display as we can work entirely in the terminal.

```
-serial mon:stdio \
-display none \
```

Finally we override the default firmware to ensure we have some storage for EFI to persist its configuration. That firmware is responsible for finding the disk, booting grub and eventually running our system.

```
-blockdev node-name=rom,driver=file,filename=(pwd)/pc-bios/edk2-aarch64-code.fd,read-
↪only=true \
-blockdev node-name=efivars,driver=file,filename=$HOME/images/qemu-arm64-efivars
```

2.2 Invocation

```
qemu-system-x86_64 [options] [disk_image]
```

disk_image is a raw hard disk image for IDE hard disk 0. Some targets do not need a disk image.

When dealing with options parameters as arbitrary strings containing commas, such as in "file=my,file" and "string=a,b", it's necessary to double the commas. For instance,"-fw_cfg name=z,string=a,,b" will be parsed as "-fw_cfg name=z,string=a,b".

2.2.1 Standard options

-h
Display help and exit

-version
Display version information and exit

-machine [type=]name[,prop=value[,...]]
Select the emulated machine by name. Use `-machine help` to list available machines.

For architectures which aim to support live migration compatibility across releases, each release will introduce a new versioned machine type. For example, the 2.8.0 release introduced machine types "pc-i440fx-2.8" and "pc-q35-2.8" for the x86_64/i686 architectures.

To allow live migration of guests from QEMU version 2.8.0, to QEMU version 2.9.0, the 2.9.0 version must support the "pc-i440fx-2.8" and "pc-q35-2.8" machines too. To allow users live migrating VMs to skip multiple intermediate releases when upgrading, new releases of QEMU will support machine types from many previous versions.

Supported machine properties are:

accel=accels1[:accels2[:...]]
This is used to enable an accelerator. Depending on the target architecture, kvm, xen, hvf, nvmm, whpx or tcg can be available. By default, tcg is used. If there is more than one accelerator specified, the next one is used if the previous one fails to initialize.

vmport=on|off|auto
Enables emulation of VMWare IO port, for vmmouse etc. auto says to select the value based on accel. For accel=xen the default is off otherwise the default is on.

dump-guest-core=on|off
Include guest memory in a core dump. The default is on.

mem-merge=on|off
Enables or disables memory merge support. This feature, when supported by the host, de-duplicates identical memory pages among VMs instances (enabled by default).

aes-key-wrap=on|off
Enables or disables AES key wrapping support on s390-ccw hosts. This feature controls whether AES wrapping keys will be created to allow execution of AES cryptographic functions. The default is on.

dea-key-wrap=on|off
Enables or disables DEA key wrapping support on s390-ccw hosts. This feature controls whether DEA wrapping keys will be created to allow execution of DEA cryptographic functions. The default is on.

nvdimm=on|off
Enables or disables NVDIMM support. The default is off.

memory-encryption=
Memory encryption object to use. The default is none.

hmat=on|off
Enables or disables ACPI Heterogeneous Memory Attribute Table (HMAT) support. The default is off.

memory-backend='id'
An alternative to legacy `-mem-path` and `mem-prealloc` options. Allows to use a memory backend as main RAM.

For example:

```
-object memory-backend-file,id=pc.ram,size=512M,mem-path=/hugetlbfs,prealloc=on,
↪share=on
-machine memory-backend=pc.ram
-m 512M
```

Migration compatibility note:

- as backend id one shall use value of 'default-ram-id', advertised by machine type (available via `query-machines` QMP command), if migration to/from old QEMU (<5.0) is expected.
- for machine types 4.0 and older, user shall use `x-use-canonical-path-for-ramblock-id=off` backend option if migration to/from old QEMU (<5.0) is expected.

For example:

```
-object memory-backend-ram,id=pc.ram,size=512M,x-use-canonical-path-for-
↪ramblock-id=off
-machine memory-backend=pc.ram
-m 512M
```

cxl-fmw.0.targets.0=firsttarget,cxl-fmw.0.targets.1=secondtarget,cxl-fmw.0.size=size[,cxl-fmw.0.interleave-granularity=granularity]
Define a CXL Fixed Memory Window (CFMW).

Described in the CXL 2.0 ECN: CEDT CFMWS & QTG _DSM.

They are regions of Host Physical Addresses (HPA) on a system which may be interleaved across one or more CXL host bridges. The system software will assign particular devices into these windows and configure the downstream Host-managed Device Memory (HDM) decoders in root ports, switch ports and devices appropriately to meet the interleave requirements before enabling the memory devices.

`targets.X=target` provides the mapping to CXL host bridges which may be identified by the id provided in the -device entry. Multiple entries are needed to specify all the targets when the fixed memory window represents interleaved memory. X is the target index from 0.

`size=size` sets the size of the CFMW. This must be a multiple of 256MiB. The region will be aligned to 256MiB but the location is platform and configuration dependent.

interleave-granularity=granularity sets the granularity of interleave. Default 256 (bytes). Only 256, 512, 1k, 2k, 4k, 8k and 16k granularities supported.

Example:

```
-machine cxl-fmw.0.targets.0=cxl.0,cxl-fmw.0.targets.1=cxl.1,cxl-fmw.0.
↪size=128G,cxl-fmw.0.interleave-granularity=512
```

sgx-epc.0.memdev=@var{memid},sgx-epc.0.node=@var{numaid}
Define an SGX EPC section.

-cpu model
Select CPU model (-cpu help for list and additional feature selection)

-accel name[,prop=value[,...]]
This is used to enable an accelerator. Depending on the target architecture, kvm, xen, hvf, nvmm, whpx or tcg can be available. By default, tcg is used. If there is more than one accelerator specified, the next one is used if the previous one fails to initialize.

igd-passthru=on|off
When Xen is in use, this option controls whether Intel integrated graphics devices can be passed through to the guest (default=off)

kernel-irqchip=on|off|split
Controls KVM in-kernel irqchip support. The default is full acceleration of the interrupt controllers. On x86, split irqchip reduces the kernel attack surface, at a performance cost for non-MSI interrupts. Disabling the in-kernel irqchip completely is not recommended except for debugging purposes.

kvm-shadow-mem=size
Defines the size of the KVM shadow MMU.

one-insn-per-tb=on|off
Makes the TCG accelerator put only one guest instruction into each translation block. This slows down emulation a lot, but can be useful in some situations, such as when trying to analyse the logs produced by the -d option.

split-wx=on|off
Controls the use of split w^x mapping for the TCG code generation buffer. Some operating systems require this to be enabled, and in such a case this will default on. On other operating systems, this will default off, but one may enable this for testing or debugging.

tb-size=n
Controls the size (in MiB) of the TCG translation block cache.

thread=single|multi
Controls number of TCG threads. When the TCG is multi-threaded there will be one thread per vCPU therefore taking advantage of additional host cores. The default is to enable multi-threading where both the back-end and front-ends support it and no incompatible TCG features have been enabled (e.g. icount/replay).

dirty-ring-size=n
When the KVM accelerator is used, it controls the size of the per-vCPU dirty page ring buffer (number of entries for each vCPU). It should be a value that is power of two, and it should be 1024 or bigger (but still less than the maximum value that the kernel supports). 4096 could be a good initial value if you have no idea which is the best. Set this value to 0 to disable the feature. By default, this feature is disabled (dirty-ring-size=0). When enabled, KVM will instead record dirty pages in a bitmap.

eager-split-size=n
KVM implements dirty page logging at the PAGE_SIZE granularity and enabling dirty-logging on a huge-page requires breaking it into PAGE_SIZE pages in the first place. KVM on ARM does this splitting lazily by default. There are performance benefits in doing huge-page split eagerly, especially in situations where

TLBI costs associated with break-before-make sequences are considerable and also if guest workloads are read intensive. The size here specifies how many pages to break at a time and needs to be a valid block size which is 1GB/2MB/4KB, 32MB/16KB and 512MB/64KB for 4KB/16KB/64KB PAGE_SIZE respectively. Be wary of specifying a higher size as it will have an impact on the memory. By default, this feature is disabled (eager-split-size=0).

notify-vmexit=run|internal-error|disable,notify-window=n
Enables or disables notify VM exit support on x86 host and specify the corresponding notify window to trigger the VM exit if enabled. `run` option enables the feature. It does nothing and continue if the exit happens. `internal-error` option enables the feature. It raises a internal error. `disable` option doesn't enable the feature. This feature can mitigate the CPU stuck issue due to event windows don't open up for a specified of time (i.e. notify-window). Default: notify-vmexit=run,notify-window=0.

device=path
Sets the path to the KVM device node. Defaults to `/dev/kvm`. This option can be used to pass the KVM device to use via a file descriptor by setting the value to `/dev/fdset/NN`.

-smp [[cpus=]n][,maxcpus=maxcpus][,sockets=sockets][,dies=dies][,clusters=clusters][,cores=cores][,threads=threads]
Simulate a SMP system with '`n`' CPUs initially present on the machine type board. On boards supporting CPU hotplug, the optional '`maxcpus`' parameter can be set to enable further CPUs to be added at runtime. When both parameters are omitted, the maximum number of CPUs will be calculated from the provided topology members and the initial CPU count will match the maximum number. When only one of them is given then the omitted one will be set to its counterpart's value. Both parameters may be specified, but the maximum number of CPUs must be equal to or greater than the initial CPU count. Product of the CPU topology hierarchy must be equal to the maximum number of CPUs. Both parameters are subject to an upper limit that is determined by the specific machine type chosen.

To control reporting of CPU topology information, values of the topology parameters can be specified. Machines may only support a subset of the parameters and different machines may have different subsets supported which vary depending on capacity of the corresponding CPU targets. So for a particular machine type board, an expected topology hierarchy can be defined through the supported sub-option. Unsupported parameters can also be provided in addition to the sub-option, but their values must be set as 1 in the purpose of correct parsing.

Either the initial CPU count, or at least one of the topology parameters must be specified. The specified parameters must be greater than zero, explicit configuration like "cpus=0" is not allowed. Values for any omitted parameters will be computed from those which are given.

For example, the following sub-option defines a CPU topology hierarchy (2 sockets totally on the machine, 2 cores per socket, 2 threads per core) for a machine that only supports sockets/cores/threads. Some members of the option can be omitted but their values will be automatically computed:

```
-smp 8,sockets=2,cores=2,threads=2,maxcpus=8
```

The following sub-option defines a CPU topology hierarchy (2 sockets totally on the machine, 2 dies per socket, 2 cores per die, 2 threads per core) for PC machines which support sockets/dies/cores/threads. Some members of the option can be omitted but their values will be automatically computed:

```
-smp 16,sockets=2,dies=2,cores=2,threads=2,maxcpus=16
```

The following sub-option defines a CPU topology hierarchy (2 sockets totally on the machine, 2 clusters per socket, 2 cores per cluster, 2 threads per core) for ARM virt machines which support sockets/clusters /cores/threads. Some members of the option can be omitted but their values will be automatically computed:

```
-smp 16,sockets=2,clusters=2,cores=2,threads=2,maxcpus=16
```

Historically preference was given to the coarsest topology parameters when computing missing values (ie sockets preferred over cores, which were preferred over threads), however, this behaviour is considered liable to change.

Prior to 6.2 the preference was sockets over cores over threads. Since 6.2 the preference is cores over sockets over threads.

For example, the following option defines a machine board with 2 sockets of 1 core before 6.2 and 1 socket of 2 cores after 6.2:

```
-smp 2
```

Note: The cluster topology will only be generated in ACPI and exposed to guest if it's explicitly specified in -smp.

-numa node[,mem=size][,cpus=firstcpu[-lastcpu]][,nodeid=node][,initiator=initiator]

-numa node[,memdev=id][,cpus=firstcpu[-lastcpu]][,nodeid=node][,initiator=initiator]

-numa dist,src=source,dst=destination,val=distance

-numa cpu,node-id=node[,socket-id=x][,core-id=y][,thread-id=z]

-numa hmat-lb,initiator=node,target=node,hierarchy=hierarchy,data-type=type[,latency=lat][,bandwidth=bw]

-numa hmat-cache,node-id=node,size=size,level=level[,associativity=str][,policy=str][,line=size]

Define a NUMA node and assign RAM and VCPUs to it. Set the NUMA distance from a source node to a destination node. Set the ACPI Heterogeneous Memory Attributes for the given nodes.

Legacy VCPU assignment uses '`cpus`' option where firstcpu and lastcpu are CPU indexes. Each '`cpus`' option represent a contiguous range of CPU indexes (or a single VCPU if lastcpu is omitted). A non-contiguous set of VCPUs can be represented by providing multiple '`cpus`' options. If '`cpus`' is omitted on all nodes, VCPUs are automatically split between them.

For example, the following option assigns VCPUs 0, 1, 2 and 5 to a NUMA node:

```
-numa node,cpus=0-2,cpus=5
```

'`cpu`' option is a new alternative to '`cpus`' option which uses '`socket-id|core-id|thread-id`' properties to assign CPU objects to a node using topology layout properties of CPU. The set of properties is machine specific, and depends on used machine type/'`smp`' options. It could be queried with '`hotpluggable-cpus`' monitor command. '`node-id`' property specifies node to which CPU object will be assigned, it's required for node to be declared with '`node`' option before it's used with '`cpu`' option.

For example:

```
-M pc \
-smp 1,sockets=2,maxcpus=2 \
-numa node,nodeid=0 -numa node,nodeid=1 \
-numa cpu,node-id=0,socket-id=0 -numa cpu,node-id=1,socket-id=1
```

'`memdev`' option assigns RAM from a given memory backend device to a node. It is recommended to use '`memdev`' option over legacy '`mem`' option. This is because '`memdev`' option provides better performance and more control over the backend's RAM (e.g. '`prealloc`' parameter of '`-memory-backend-ram`' allows memory preallocation).

For compatibility reasons, legacy '`mem`' option is supported in 5.0 and older machine types. Note that '`mem`' and '`memdev`' are mutually exclusive. If one node uses '`memdev`', the rest nodes have to use '`memdev`' option, and vice versa.

Users must specify memory for all NUMA nodes by '`memdev`' (or legacy '`mem`' if available). In QEMU 5.2, the support for '`-numa node`' without memory specified was removed.

'initiator' is an additional option that points to an initiator NUMA node that has best performance (the lowest latency or largest bandwidth) to this NUMA node. Note that this option can be set only when the machine property 'hmat' is set to 'on'.

Following example creates a machine with 2 NUMA nodes, node 0 has CPU. node 1 has only memory, and its initiator is node 0. Note that because node 0 has CPU, by default the initiator of node 0 is itself and must be itself.

```
-machine hmat=on \
-m 2G,slots=2,maxmem=4G \
-object memory-backend-ram,size=1G,id=m0 \
-object memory-backend-ram,size=1G,id=m1 \
-numa node,nodeid=0,memdev=m0 \
-numa node,nodeid=1,memdev=m1,initiator=0 \
-smp 2,sockets=2,maxcpus=2  \
-numa cpu,node-id=0,socket-id=0 \
-numa cpu,node-id=0,socket-id=1
```

source and destination are NUMA node IDs. distance is the NUMA distance from source to destination. The distance from a node to itself is always 10. If any pair of nodes is given a distance, then all pairs must be given distances. Although, when distances are only given in one direction for each pair of nodes, then the distances in the opposite directions are assumed to be the same. If, however, an asymmetrical pair of distances is given for even one node pair, then all node pairs must be provided distance values for both directions, even when they are symmetrical. When a node is unreachable from another node, set the pair's distance to 255.

Note that the -numa option doesn't allocate any of the specified resources, it just assigns existing resources to NUMA nodes. This means that one still has to use the -m, -smp options to allocate RAM and VCPUs respectively.

Use 'hmat-lb' to set System Locality Latency and Bandwidth Information between initiator and target NUMA nodes in ACPI Heterogeneous Attribute Memory Table (HMAT). Initiator NUMA node can create memory requests, usually it has one or more processors. Target NUMA node contains addressable memory.

In 'hmat-lb' option, node are NUMA node IDs. hierarchy is the memory hierarchy of the target NUMA node: if hierarchy is 'memory', the structure represents the memory performance; if hierarchy is 'first-level|second-level|third-level', this structure represents aggregated performance of memory side caches for each domain. type of 'data-type' is type of data represented by this structure instance: if 'hierarchy' is 'memory', 'data-type' is 'access|read|write' latency or 'access|read|write' bandwidth of the target memory; if 'hierarchy' is 'first-level|second-level|third-level', 'data-type' is 'access|read|write' hit latency or 'access|read|write' hit bandwidth of the target memory side cache.

lat is latency value in nanoseconds. bw is bandwidth value, the possible value and units are NUM[M|G|T], mean that the bandwidth value are NUM byte per second (or MB/s, GB/s or TB/s depending on used suffix). Note that if latency or bandwidth value is 0, means the corresponding latency or bandwidth information is not provided.

In 'hmat-cache' option, node-id is the NUMA-id of the memory belongs. size is the size of memory side cache in bytes. level is the cache level described in this structure, note that the cache level 0 should not be used with 'hmat-cache' option. associativity is the cache associativity, the possible value is 'none/direct(direct-mapped)/complex(complex cache indexing)'. policy is the write policy. line is the cache Line size in bytes.

For example, the following options describe 2 NUMA nodes. Node 0 has 2 cpus and a ram, node 1 has only a ram. The processors in node 0 access memory in node 0 with access-latency 5 nanoseconds, access-bandwidth is 200 MB/s; The processors in NUMA node 0 access memory in NUMA node 1 with access-latency 10 nanoseconds, access-bandwidth is 100 MB/s. And for memory side cache information, NUMA node 0 and 1 both have 1 level memory cache, size is 10KB, policy is write-back, the cache Line size is 8 bytes:

```
-machine hmat=on \
-m 2G \
```

(continues on next page)

(continued from previous page)

```
-object memory-backend-ram,size=1G,id=m0 \
-object memory-backend-ram,size=1G,id=m1 \
-smp 2,sockets=2,maxcpus=2 \
-numa node,nodeid=0,memdev=m0 \
-numa node,nodeid=1,memdev=m1,initiator=0 \
-numa cpu,node-id=0,socket-id=0 \
-numa cpu,node-id=0,socket-id=1 \
-numa hmat-lb,initiator=0,target=0,hierarchy=memory,data-type=access-latency,
↪latency=5 \
-numa hmat-lb,initiator=0,target=0,hierarchy=memory,data-type=access-bandwidth,
↪bandwidth=200M \
-numa hmat-lb,initiator=0,target=1,hierarchy=memory,data-type=access-latency,
↪latency=10 \
-numa hmat-lb,initiator=0,target=1,hierarchy=memory,data-type=access-bandwidth,
↪bandwidth=100M \
-numa hmat-cache,node-id=0,size=10K,level=1,associativity=direct,policy=write-back,
↪line=8 \
-numa hmat-cache,node-id=1,size=10K,level=1,associativity=direct,policy=write-back,
↪line=8
```

-add-fd fd=fd,set=set[,opaque=opaque]

Add a file descriptor to an fd set. Valid options are:

fd=fd

This option defines the file descriptor of which a duplicate is added to fd set. The file descriptor cannot be stdin, stdout, or stderr.

set=set

This option defines the ID of the fd set to add the file descriptor to.

opaque=opaque

This option defines a free-form string that can be used to describe fd.

You can open an image using pre-opened file descriptors from an fd set:

```
qemu-system-x86_64 \
 -add-fd fd=3,set=2,opaque="rdwr:/path/to/file" \
 -add-fd fd=4,set=2,opaque="rdonly:/path/to/file" \
 -drive file=/dev/fdset/2,index=0,media=disk
```

-set group.id.arg=value

Set parameter arg for item id of type group

-global driver.prop=value

-global driver=driver,property=property,value=value

Set default value of driver's property prop to value, e.g.:

```
qemu-system-x86_64 -global ide-hd.physical_block_size=4096 disk-image.img
```

In particular, you can use this to set driver properties for devices which are created automatically by the machine model. To create a device which is not created automatically and set properties on it, use -device.

-global driver.prop=value is shorthand for -global driver=driver,property=prop,value=value. The longhand syntax works even when driver contains a dot.

-boot [order=drives][,once=drives][,menu=on|off][,splash=sp_name][,splash-time=sp_time][,reboot-timeout=rb_timeout][,strict=on|off]

Specify boot order drives as a string of drive letters. Valid drive letters depend on the target architecture. The

x86 PC uses: a, b (floppy 1 and 2), c (first hard disk), d (first CD-ROM), n-p (Etherboot from network adapter 1-4), hard disk boot is the default. To apply a particular boot order only on the first startup, specify it via `once`. Note that the `order` or `once` parameter should not be used together with the `bootindex` property of devices, since the firmware implementations normally do not support both at the same time.

Interactive boot menus/prompts can be enabled via `menu=on` as far as firmware/BIOS supports them. The default is non-interactive boot.

A splash picture could be passed to bios, enabling user to show it as logo, when option splash=sp_name is given and menu=on, If firmware/BIOS supports them. Currently Seabios for X86 system support it. limitation: The splash file could be a jpeg file or a BMP file in 24 BPP format(true color). The resolution should be supported by the SVGA mode, so the recommended is 320x240, 640x480, 800x640.

A timeout could be passed to bios, guest will pause for rb_timeout ms when boot failed, then reboot. If rb_timeout is '-1', guest will not reboot, qemu passes '-1' to bios by default. Currently Seabios for X86 system support it.

Do strict boot via `strict=on` as far as firmware/BIOS supports it. This only effects when boot priority is changed by bootindex options. The default is non-strict boot.

```
# try to boot from network first, then from hard disk
qemu-system-x86_64 -boot order=nc
# boot from CD-ROM first, switch back to default order after reboot
qemu-system-x86_64 -boot once=d
# boot with a splash picture for 5 seconds.
qemu-system-x86_64 -boot menu=on,splash=/root/boot.bmp,splash-time=5000
```

Note: The legacy format '-boot drives' is still supported but its use is discouraged as it may be removed from future versions.

-m [size=]megs[,slots=n,maxmem=size]
Sets guest startup RAM size to megs megabytes. Default is 128 MiB. Optionally, a suffix of "M" or "G" can be used to signify a value in megabytes or gigabytes respectively. Optional pair slots, maxmem could be used to set amount of hotpluggable memory slots and maximum amount of memory. Note that maxmem must be aligned to the page size.

For example, the following command-line sets the guest startup RAM size to 1GB, creates 3 slots to hotplug additional memory and sets the maximum memory the guest can reach to 4GB:

```
qemu-system-x86_64 -m 1G,slots=3,maxmem=4G
```

If slots and maxmem are not specified, memory hotplug won't be enabled and the guest startup RAM will never increase.

-mem-path path
Allocate guest RAM from a temporarily created file in path.

-mem-prealloc
Preallocate memory when using -mem-path.

-k language
Use keyboard layout language (for example `fr` for French). This option is only needed where it is not easy to get raw PC keycodes (e.g. on Macs, with some X11 servers or with a VNC or curses display). You don't normally need to use it on PC/Linux or PC/Windows hosts.

The available layouts are:

```
ar  de-ch  es  fo     fr-ca  hu  ja  mk     no  pt-br  sv
da  en-gb  et  fr     fr-ch  is  lt  nl     pl  ru     th
de  en-us  fi  fr-be  hr     it  lv  nl-be  pt  sl     tr
```

The default is `en-us`.

-audio [driver=]driver[,model=value][,prop[=value][,...]]

If the `model` option is specified, `-audio` is a shortcut for configuring both the guest audio hardware and the host audio backend in one go. The guest hardware model can be set with `model=modelname`. Use `model=help` to list the available device types.

The following two example do exactly the same, to show how `-audio` can be used to shorten the command line length:

```
qemu-system-x86_64 -audiodev pa,id=pa -device sb16,audiodev=pa
qemu-system-x86_64 -audio pa,model=sb16
```

If the `model` option is not specified, `-audio` is used to configure a default audio backend that will be used whenever the `audiodev` property is not set on a device or machine. In particular, `-audio none` ensures that no audio is produced even for machines that have embedded sound hardware.

In both cases, the driver option is the same as with the corresponding `-audiodev` option below. Use `driver=help` to list the available drivers.

-audiodev [driver=]driver,id=id[,prop[=value][,...]]

Adds a new audio backend driver identified by id. There are global and driver specific properties. Some values can be set differently for input and output, they're marked with `in|out.`. You can set the input's property with `in.prop` and the output's property with `out.prop`. For example:

```
-audiodev alsa,id=example,in.frequency=44110,out.frequency=8000
-audiodev alsa,id=example,out.channels=1 # leaves in.channels unspecified
```

NOTE: parameter validation is known to be incomplete, in many cases specifying an invalid option causes QEMU to print an error message and continue emulation without sound.

Valid global options are:

id=identifier

Identifies the audio backend.

timer-period=period

Sets the timer period used by the audio subsystem in microseconds. Default is 10000 (10 ms).

in|out.mixing-engine=on|off

Use QEMU's mixing engine to mix all streams inside QEMU and convert audio formats when not supported by the backend. When off, fixed-settings must be off too. Note that disabling this option means that the selected backend must support multiple streams and the audio formats used by the virtual cards, otherwise you'll get no sound. It's not recommended to disable this option unless you want to use 5.1 or 7.1 audio, as mixing engine only supports mono and stereo audio. Default is on.

in|out.fixed-settings=on|off

Use fixed settings for host audio. When off, it will change based on how the guest opens the sound card. In this case you must not specify frequency, channels or format. Default is on.

in|out.frequency=frequency

Specify the frequency to use when using fixed-settings. Default is 44100Hz.

in|out.channels=channels

Specify the number of channels to use when using fixed-settings. Default is 2 (stereo).

in|out.format=format

Specify the sample format to use when using fixed-settings. Valid values are: `s8`, `s16`, `s32`, `u8`, `u16`, `u32`, `f32`. Default is `s16`.

in|out.voices=voices

Specify the number of voices to use. Default is 1.

in|out.buffer-length=usecs
Sets the size of the buffer in microseconds.

-audiodev none,id=id[,prop[=value][,...]]
Creates a dummy backend that discards all outputs. This backend has no backend specific properties.

-audiodev alsa,id=id[,prop[=value][,...]]
Creates backend using the ALSA. This backend is only available on Linux.

ALSA specific options are:

in|out.dev=device
Specify the ALSA device to use for input and/or output. Default is `default`.

in|out.period-length=usecs
Sets the period length in microseconds.

in|out.try-poll=on|off
Attempt to use poll mode with the device. Default is on.

threshold=threshold
Threshold (in microseconds) when playback starts. Default is 0.

-audiodev coreaudio,id=id[,prop[=value][,...]]
Creates a backend using Apple's Core Audio. This backend is only available on Mac OS and only supports playback.

Core Audio specific options are:

in|out.buffer-count=count
Sets the count of the buffers.

-audiodev dsound,id=id[,prop[=value][,...]]
Creates a backend using Microsoft's DirectSound. This backend is only available on Windows and only supports playback.

DirectSound specific options are:

latency=usecs
Add extra usecs microseconds latency to playback. Default is 10000 (10 ms).

-audiodev oss,id=id[,prop[=value][,...]]
Creates a backend using OSS. This backend is available on most Unix-like systems.

OSS specific options are:

in|out.dev=device
Specify the file name of the OSS device to use. Default is `/dev/dsp`.

in|out.buffer-count=count
Sets the count of the buffers.

in|out.try-poll=on|of
Attempt to use poll mode with the device. Default is on.

try-mmap=on|off
Try using memory mapped device access. Default is off.

exclusive=on|off
Open the device in exclusive mode (vmix won't work in this case). Default is off.

dsp-policy=policy
Sets the timing policy (between 0 and 10, where smaller number means smaller latency but higher CPU

usage). Use -1 to use buffer sizes specified by `buffer` and `buffer-count`. This option is ignored if you do not have OSS 4. Default is 5.

`-audiodev pa,id=id[,prop[=value][,...]]`
Creates a backend using PulseAudio. This backend is available on most systems.

PulseAudio specific options are:

`server=server`
Sets the PulseAudio server to connect to.

`in|out.name=sink`
Use the specified source/sink for recording/playback.

`in|out.latency=usecs`
Desired latency in microseconds. The PulseAudio server will try to honor this value but actual latencies may be lower or higher.

`-audiodev pipewire,id=id[,prop[=value][,...]]`
Creates a backend using PipeWire. This backend is available on most systems.

PipeWire specific options are:

`in|out.latency=usecs`
Desired latency in microseconds.

`in|out.name=sink`
Use the specified source/sink for recording/playback.

`in|out.stream-name`
Specify the name of pipewire stream.

`-audiodev sdl,id=id[,prop[=value][,...]]`
Creates a backend using SDL. This backend is available on most systems, but you should use your platform's native backend if possible.

SDL specific options are:

`in|out.buffer-count=count`
Sets the count of the buffers.

`-audiodev sndio,id=id[,prop[=value][,...]]`
Creates a backend using SNDIO. This backend is available on OpenBSD and most other Unix-like systems.

Sndio specific options are:

`in|out.dev=device`
Specify the sndio device to use for input and/or output. Default is `default`.

`in|out.latency=usecs`
Sets the desired period length in microseconds.

`-audiodev spice,id=id[,prop[=value][,...]]`
Creates a backend that sends audio through SPICE. This backend requires `-spice` and automatically selected in that case, so usually you can ignore this option. This backend has no backend specific properties.

`-audiodev wav,id=id[,prop[=value][,...]]`
Creates a backend that writes audio to a WAV file.

Backend specific options are:

`path=path`
Write recorded audio into the specified file. Default is `qemu.wav`.

-device driver[,prop[=value][,...]]
Add device driver. prop=value sets driver properties. Valid properties depend on the driver. To get help on possible drivers and properties, use `-device help` and `-device driver,help`.

Some drivers are:

-device ipmi-bmc-sim,id=id[,prop[=value][,...]]
Add an IPMI BMC. This is a simulation of a hardware management interface processor that normally sits on a system. It provides a watchdog and the ability to reset and power control the system. You need to connect this to an IPMI interface to make it useful

The IPMI slave address to use for the BMC. The default is 0x20. This address is the BMC's address on the I2C network of management controllers. If you don't know what this means, it is safe to ignore it.

id=id
The BMC id for interfaces to use this device.

slave_addr=val
Define slave address to use for the BMC. The default is 0x20.

sdrfile=file
file containing raw Sensor Data Records (SDR) data. The default is none.

fruareasize=val
size of a Field Replaceable Unit (FRU) area. The default is 1024.

frudatafile=file
file containing raw Field Replaceable Unit (FRU) inventory data. The default is none.

guid=uuid
value for the GUID for the BMC, in standard UUID format. If this is set, get "Get GUID" command to the BMC will return it. Otherwise "Get GUID" will return an error.

-device ipmi-bmc-extern,id=id,chardev=id[,slave_addr=val]
Add a connection to an external IPMI BMC simulator. Instead of locally emulating the BMC like the above item, instead connect to an external entity that provides the IPMI services.

A connection is made to an external BMC simulator. If you do this, it is strongly recommended that you use the "reconnect=" chardev option to reconnect to the simulator if the connection is lost. Note that if this is not used carefully, it can be a security issue, as the interface has the ability to send resets, NMIs, and power off the VM. It's best if QEMU makes a connection to an external simulator running on a secure port on localhost, so neither the simulator nor QEMU is exposed to any outside network.

See the "lanserv/README.vm" file in the OpenIPMI library for more details on the external interface.

-device isa-ipmi-kcs,bmc=id[,ioport=val][,irq=val]
Add a KCS IPMI interface on the ISA bus. This also adds a corresponding ACPI and SMBIOS entries, if appropriate.

bmc=id
The BMC to connect to, one of ipmi-bmc-sim or ipmi-bmc-extern above.

ioport=val
Define the I/O address of the interface. The default is 0xca0 for KCS.

irq=val
Define the interrupt to use. The default is 5. To disable interrupts, set this to 0.

-device isa-ipmi-bt,bmc=id[,ioport=val][,irq=val]
Like the KCS interface, but defines a BT interface. The default port is 0xe4 and the default interrupt is 5.

-device pci-ipmi-kcs,bmc=id
Add a KCS IPMI interface on the PCI bus.

bmc=id
The BMC to connect to, one of ipmi-bmc-sim or ipmi-bmc-extern above.

-device pci-ipmi-bt,bmc=id
Like the KCS interface, but defines a BT interface on the PCI bus.

-device intel-iommu[,option=...]
This is only supported by `-machine q35`, which will enable Intel VT-d emulation within the guest. It supports below options:

intremap=on|off (default: auto)
This enables interrupt remapping feature. It's required to enable complete x2apic. Currently it only supports kvm kernel-irqchip modes `off` or `split`, while full kernel-irqchip is not yet supported. The default value is "auto", which will be decided by the mode of kernel-irqchip.

caching-mode=on|off (default: off)
This enables caching mode for the VT-d emulated device. When caching-mode is enabled, each guest DMA buffer mapping will generate an IOTLB invalidation from the guest IOMMU driver to the vIOMMU device in a synchronous way. It is required for `-device vfio-pci` to work with the VT-d device, because host assigned devices requires to setup the DMA mapping on the host before guest DMA starts.

device-iotlb=on|off (default: off)
This enables device-iotlb capability for the emulated VT-d device. So far virtio/vhost should be the only real user for this parameter, paired with ats=on configured for the device.

aw-bits=39|48 (default: 39)
This decides the address width of IOVA address space. The address space has 39 bits width for 3-level IOMMU page tables, and 48 bits for 4-level IOMMU page tables.

Please also refer to the wiki page for general scenarios of VT-d emulation in QEMU: https://wiki.qemu.org/Features/VT-d.

-device virtio-iommu-pci[,option=...]
This is only supported by `-machine q35` (x86_64) and `-machine virt` (ARM). It supports below options:

granule=val (possible values are 4k, 8k, 16k, 64k and host; default: host)
This decides the default granule to be be exposed by the virtio-iommu. If host, the granule matches the host page size.

aw-bits=val (val between 32 and 64, default depends on machine)
This decides the address width of the IOVA address space.

-name name
Sets the name of the guest. This name will be displayed in the SDL window caption. The name will also be used for the VNC server. Also optionally set the top visible process name in Linux. Naming of individual threads can also be enabled on Linux to aid debugging.

-uuid uuid
Set system UUID.

2.2.2 Block device options

The QEMU block device handling options have a long history and have gone through several iterations as the feature set and complexity of the block layer have grown. Many online guides to QEMU often reference older and deprecated options, which can lead to confusion.

The most explicit way to describe disks is to use a combination of `-device` to specify the hardware device and `-blockdev` to describe the backend. The device defines what the guest sees and the backend describes how QEMU handles the data. It is the only guaranteed stable interface for describing block devices and as such is recommended for management tools and scripting.

The `-drive` option combines the device and backend into a single command line option which is a more human friendly. There is however no interface stability guarantee although some older board models still need updating to work with the modern blockdev forms.

Older options like `-hda` are essentially macros which expand into `-drive` options for various drive interfaces. The original forms bake in a lot of assumptions from the days when QEMU was emulating a legacy PC, they are not recommended for modern configurations.

-fda file

-fdb file
Use file as floppy disk 0/1 image (see the *Disk Images* chapter in the System Emulation Users Guide).

-hda file

-hdb file

-hdc file

-hdd file
Use file as hard disk 0, 1, 2 or 3 image on the default bus of the emulated machine (this is for example the IDE bus on most x86 machines, but it can also be SCSI, virtio or something else on other target architectures). See also the *Disk Images* chapter in the System Emulation Users Guide.

-cdrom file
Use file as CD-ROM image on the default bus of the emulated machine (which is IDE1 master on x86, so you cannot use `-hdc` and `-cdrom` at the same time there). On systems that support it, you can use the host CD-ROM by using `/dev/cdrom` as filename.

-blockdev option[,option[,option[,...]]]
Define a new block driver node. Some of the options apply to all block drivers, other options are only accepted for a specific block driver. See below for a list of generic options and options for the most common block drivers.

Options that expect a reference to another node (e.g. `file`) can be given in two ways. Either you specify the node name of an already existing node (file=node-name), or you define a new node inline, adding options for the referenced node after a dot (file.filename=path,file.aio=native).

A block driver node created with `-blockdev` can be used for a guest device by specifying its node name for the `drive` property in a `-device` argument that defines a block device.

Valid options for any block driver node:

driver
Specifies the block driver to use for the given node.

node-name
This defines the name of the block driver node by which it will be referenced later. The name must be unique, i.e. it must not match the name of a different block driver node, or (if you use `-drive` as well) the ID of a drive.

If no node name is specified, it is automatically generated. The generated node name is not intended to be predictable and changes between QEMU invocations. For the top level, an explicit node name must be specified.

read-only
Open the node read-only. Guest write attempts will fail.

Note that some block drivers support only read-only access, either generally or in certain configurations. In this case, the default value `read-only=off` does not work and the option must be specified explicitly.

auto-read-only
If `auto-read-only=on` is set, QEMU may fall back to read-only usage even when `read-only=off` is requested, or even switch between modes as needed, e.g. depending on whether the image file is writable or whether a writing user is attached to the node.

force-share
Override the image locking system of QEMU by forcing the node to utilize weaker shared access for permissions where it would normally request exclusive access. When there is the potential for multiple instances to have the same file open (whether this invocation of QEMU is the first or the second instance), both instances must permit shared access for the second instance to succeed at opening the file.

Enabling `force-share=on` requires `read-only=on`.

cache.direct
The host page cache can be avoided with `cache.direct=on`. This will attempt to do disk IO directly to the guest's memory. QEMU may still perform an internal copy of the data.

cache.no-flush
In case you don't care about data integrity over host failures, you can use `cache.no-flush=on`. This option tells QEMU that it never needs to write any data to the disk but can instead keep things in cache. If anything goes wrong, like your host losing power, the disk storage getting disconnected accidentally, etc. your image will most probably be rendered unusable.

discard=discard
discard is one of "ignore" (or "off") or "unmap" (or "on") and controls whether `discard` (also known as `trim` or `unmap`) requests are ignored or passed to the filesystem. Some machine types may not support discard requests.

detect-zeroes=detect-zeroes
detect-zeroes is "off", "on" or "unmap" and enables the automatic conversion of plain zero writes by the OS to driver specific optimized zero write commands. You may even choose "unmap" if discard is set to "unmap" to allow a zero write to be converted to an `unmap` operation.

Driver-specific options for file
This is the protocol-level block driver for accessing regular files.

filename
The path to the image file in the local filesystem

aio
Specifies the AIO backend (threads/native/io_uring, default: threads)

locking
Specifies whether the image file is protected with Linux OFD / POSIX locks. The default is to use the Linux Open File Descriptor API if available, otherwise no lock is applied. (auto/on/off, default: auto)

Example:

```
-blockdev driver=file,node-name=disk,filename=disk.img
```

Driver-specific options for raw

This is the image format block driver for raw images. It is usually stacked on top of a protocol level block driver such as `file`.

file

Reference to or definition of the data source block driver node (e.g. a `file` driver node)

Example 1:

```
-blockdev driver=file,node-name=disk_file,filename=disk.img
-blockdev driver=raw,node-name=disk,file=disk_file
```

Example 2:

```
-blockdev driver=raw,node-name=disk,file.driver=file,file.filename=disk.img
```

Driver-specific options for qcow2

This is the image format block driver for qcow2 images. It is usually stacked on top of a protocol level block driver such as `file`.

file

Reference to or definition of the data source block driver node (e.g. a `file` driver node)

backing

Reference to or definition of the backing file block device (default is taken from the image file). It is allowed to pass `null` here in order to disable the default backing file.

lazy-refcounts

Whether to enable the lazy refcounts feature (on/off; default is taken from the image file)

cache-size

The maximum total size of the L2 table and refcount block caches in bytes (default: the sum of l2-cache-size and refcount-cache-size)

l2-cache-size

The maximum size of the L2 table cache in bytes (default: if cache-size is not specified - 32M on Linux platforms, and 8M on non-Linux platforms; otherwise, as large as possible within the cache-size, while permitting the requested or the minimal refcount cache size)

refcount-cache-size

The maximum size of the refcount block cache in bytes (default: 4 times the cluster size; or if cache-size is specified, the part of it which is not used for the L2 cache)

cache-clean-interval

Clean unused entries in the L2 and refcount caches. The interval is in seconds. The default value is 600 on supporting platforms, and 0 on other platforms. Setting it to 0 disables this feature.

pass-discard-request

Whether discard requests to the qcow2 device should be forwarded to the data source (on/off; default: on if discard=unmap is specified, off otherwise)

pass-discard-snapshot

Whether discard requests for the data source should be issued when a snapshot operation (e.g. deleting a snapshot) frees clusters in the qcow2 file (on/off; default: on)

pass-discard-other

Whether discard requests for the data source should be issued on other occasions where a cluster gets freed (on/off; default: off)

discard-no-unref
When enabled, data clusters will remain preallocated when they are no longer used, e.g. because they are discarded or converted to zero clusters. As usual, whether the old data is discarded or kept on the protocol level (i.e. in the image file) depends on the setting of the pass-discard-request option. Keeping the clusters preallocated prevents qcow2 fragmentation that would otherwise be caused by freeing and re-allocating them later. Besides potential performance degradation, such fragmentation can lead to increased allocation of clusters past the end of the image file, resulting in image files whose file length can grow much larger than their guest disk size would suggest. If image file length is of concern (e.g. when storing qcow2 images directly on block devices), you should consider enabling this option.

overlap-check
Which overlap checks to perform for writes to the image (none/constant/cached/all; default: cached). For details or finer granularity control refer to the QAPI documentation of `blockdev-add`.

Example 1:

```
-blockdev driver=file,node-name=my_file,filename=/tmp/disk.qcow2
-blockdev driver=qcow2,node-name=hda,file=my_file,overlap-check=none,cache-
↪size=16777216
```

Example 2:

```
-blockdev driver=qcow2,node-name=disk,file.driver=http,file.filename=http://
↪example.com/image.qcow2
```

Driver-specific options for other drivers
Please refer to the QAPI documentation of the `blockdev-add` QMP command.

-drive option[,option[,option[,...]]]
Define a new drive. This includes creating a block driver node (the backend) as well as a guest device, and is mostly a shortcut for defining the corresponding `-blockdev` and `-device` options.

`-drive` accepts all options that are accepted by `-blockdev`. In addition, it knows the following options:

file=file
This option defines which disk image (see the *Disk Images* chapter in the System Emulation Users Guide) to use with this drive. If the filename contains comma, you must double it (for instance, "file=my,,file" to use file "my,file").

Special files such as iSCSI devices can be specified using protocol specific URLs. See the section for "Device URL Syntax" for more information.

if=interface
This option defines on which type on interface the drive is connected. Available types are: ide, scsi, sd, mtd, floppy, pflash, virtio, none.

bus=bus,unit=unit
These options define where is connected the drive by defining the bus number and the unit id.

index=index
This option defines where the drive is connected by using an index in the list of available connectors of a given interface type.

media=media
This option defines the type of the media: disk or cdrom.

snapshot=snapshot
snapshot is "on" or "off" and controls snapshot mode for the given drive (see `-snapshot`).

cache=cache
cache is "none", "writeback", "unsafe", "directsync" or "writethrough" and controls how the host cache is used to access block data. This is a shortcut that sets the `cache.direct` and `cache.no-flush` options (as in `-blockdev`), and additionally `cache.writeback`, which provides a default for the `write-cache` option of block guest devices (as in `-device`). The modes correspond to the following settings:

	cache.writeback	cache.direct	cache.no-flush
writeback	on	off	off
none	on	on	off
writethrough	off	off	off
directsync	off	on	off
unsafe	on	off	on

The default mode is `cache=writeback`.

aio=aio
aio is "threads", "native", or "io_uring" and selects between pthread based disk I/O, native Linux AIO, or Linux io_uring API.

format=format
Specify which disk format will be used rather than detecting the format. Can be used to specify format=raw to avoid interpreting an untrusted format header.

werror=action,rerror=action
Specify which action to take on write and read errors. Valid actions are: "ignore" (ignore the error and try to continue), "stop" (pause QEMU), "report" (report the error to the guest), "enospc" (pause QEMU only if the host disk is full; report the error to the guest otherwise). The default setting is `werror=enospc` and `rerror=report`.

copy-on-read=copy-on-read
copy-on-read is "on" or "off" and enables whether to copy read backing file sectors into the image file.

bps=b,bps_rd=r,bps_wr=w
Specify bandwidth throttling limits in bytes per second, either for all request types or for reads or writes only. Small values can lead to timeouts or hangs inside the guest. A safe minimum for disks is 2 MB/s.

bps_max=bm,bps_rd_max=rm,bps_wr_max=wm
Specify bursts in bytes per second, either for all request types or for reads or writes only. Bursts allow the guest I/O to spike above the limit temporarily.

iops=i,iops_rd=r,iops_wr=w
Specify request rate limits in requests per second, either for all request types or for reads or writes only.

iops_max=bm,iops_rd_max=rm,iops_wr_max=wm
Specify bursts in requests per second, either for all request types or for reads or writes only. Bursts allow the guest I/O to spike above the limit temporarily.

iops_size=is
Let every is bytes of a request count as a new request for iops throttling purposes. Use this option to prevent guests from circumventing iops limits by sending fewer but larger requests.

group=g
Join a throttling quota group with given name g. All drives that are members of the same group are accounted for together. Use this option to prevent guests from circumventing throttling limits by using many small disks instead of a single larger disk.

By default, the `cache.writeback=on` mode is used. It will report data writes as completed as soon as the data is present in the host page cache. This is safe as long as your guest OS makes sure to correctly flush disk caches

where needed. If your guest OS does not handle volatile disk write caches correctly and your host crashes or loses power, then the guest may experience data corruption.

For such guests, you should consider using `cache.writeback=off`. This means that the host page cache will be used to read and write data, but write notification will be sent to the guest only after QEMU has made sure to flush each write to the disk. Be aware that this has a major impact on performance.

When using the `-snapshot` option, unsafe caching is always used.

Copy-on-read avoids accessing the same backing file sectors repeatedly and is useful when the backing file is over a slow network. By default copy-on-read is off.

Instead of `-cdrom` you can use:

```
qemu-system-x86_64 -drive file=file,index=2,media=cdrom
```

Instead of `-hda`, `-hdb`, `-hdc`, `-hdd`, you can use:

```
qemu-system-x86_64 -drive file=file,index=0,media=disk
qemu-system-x86_64 -drive file=file,index=1,media=disk
qemu-system-x86_64 -drive file=file,index=2,media=disk
qemu-system-x86_64 -drive file=file,index=3,media=disk
```

You can open an image using pre-opened file descriptors from an fd set:

```
qemu-system-x86_64 \
 -add-fd fd=3,set=2,opaque="rdwr:/path/to/file" \
 -add-fd fd=4,set=2,opaque="rdonly:/path/to/file" \
 -drive file=/dev/fdset/2,index=0,media=disk
```

You can connect a CDROM to the slave of ide0:

```
qemu-system-x86_64 -drive file=file,if=ide,index=1,media=cdrom
```

If you don't specify the "file=" argument, you define an empty drive:

```
qemu-system-x86_64 -drive if=ide,index=1,media=cdrom
```

Instead of `-fda`, `-fdb`, you can use:

```
qemu-system-x86_64 -drive file=file,index=0,if=floppy
qemu-system-x86_64 -drive file=file,index=1,if=floppy
```

By default, interface is "ide" and index is automatically incremented:

```
qemu-system-x86_64 -drive file=a -drive file=b
```

is interpreted like:

```
qemu-system-x86_64 -hda a -hdb b
```

-mtdblock file
Use file as on-board Flash memory image.

-sd file
Use file as SecureDigital card image.

-snapshot
Write to temporary files instead of disk image files. In this case, the raw disk image you use is not written back. You can however force the write back by pressing C-a s (see the *Disk Images* chapter in the System Emulation Users Guide).

Warning: snapshot is incompatible with `-blockdev` (instead use qemu-img to manually create snapshot images to attach to your blockdev). If you have mixed `-blockdev` and `-drive` declarations you can use the 'snapshot' property on your drive declarations instead of this global option.

-fsdev local,id=id,path=path,security_model=security_model
[,writeout=writeout][,readonly=on][,fmode=fmode][,dmode=dmode]
[,throttling.option=value[,throttling.option=value[,...]]]

-fsdev proxy,id=id,socket=socket[,writeout=writeout][,readonly=on]

-fsdev proxy,id=id,sock_fd=sock_fd[,writeout=writeout][,readonly=on]

-fsdev synth,id=id[,readonly=on]

Define a new file system device. Valid options are:

local
Accesses to the filesystem are done by QEMU.

proxy
Accesses to the filesystem are done by virtfs-proxy-helper(1). This option is deprecated (since QEMU 8.1) and will be removed in a future version of QEMU. Use `local` instead.

synth
Synthetic filesystem, only used by QTests.

id=id
Specifies identifier for this device.

path=path
Specifies the export path for the file system device. Files under this path will be available to the 9p client on the guest.

security_model=security_model
Specifies the security model to be used for this export path. Supported security models are "passthrough", "mapped-xattr", "mapped-file" and "none". In "passthrough" security model, files are stored using the same credentials as they are created on the guest. This requires QEMU to run as root. In "mapped-xattr" security model, some of the file attributes like uid, gid, mode bits and link target are stored as file attributes. For "mapped-file" these attributes are stored in the hidden .virtfs_metadata directory. Directories exported by this security model cannot interact with other unix tools. "none" security model is same as passthrough except the sever won't report failures if it fails to set file attributes like ownership. Security model is mandatory only for local fsdriver. Other fsdrivers (like proxy) don't take security model as a parameter.

writeout=writeout
This is an optional argument. The only supported value is "immediate". This means that host page cache will be used to read and write data but write notification will be sent to the guest only when the data has been reported as written by the storage subsystem.

readonly=on
Enables exporting 9p share as a readonly mount for guests. By default read-write access is given.

socket=socket
Enables proxy filesystem driver to use passed socket file for communicating with virtfs-proxy-helper(1).

sock_fd=sock_fd
Enables proxy filesystem driver to use passed socket descriptor for communicating with virtfs-proxy-helper(1). Usually a helper like libvirt will create socketpair and pass one of the fds as sock_fd.

fmode=fmode
Specifies the default mode for newly created files on the host. Works only with security models "mapped-xattr" and "mapped-file".

dmode=dmode
Specifies the default mode for newly created directories on the host. Works only with security models "mapped-xattr" and "mapped-file".

throttling.bps-total=b,throttling.bps-read=r,throttling.bps-write=w
Specify bandwidth throttling limits in bytes per second, either for all request types or for reads or writes only.

throttling.bps-total-max=bm,bps-read-max=rm,bps-write-max=wm
Specify bursts in bytes per second, either for all request types or for reads or writes only. Bursts allow the guest I/O to spike above the limit temporarily.

throttling.iops-total=i,throttling.iops-read=r, throttling.iops-write=w
Specify request rate limits in requests per second, either for all request types or for reads or writes only.

throttling.iops-total-max=im,throttling.iops-read-max=irm, throttling.iops-write-max=iwm
Specify bursts in requests per second, either for all request types or for reads or writes only. Bursts allow the guest I/O to spike above the limit temporarily.

throttling.iops-size=is
Let every is bytes of a request count as a new request for iops throttling purposes.

-fsdev option is used along with -device driver "virtio-9p-…".

-device virtio-9p-type,fsdev=id,mount_tag=mount_tag
Options for virtio-9p-… driver are:

type
Specifies the variant to be used. Supported values are "pci", "ccw" or "device", depending on the machine type.

fsdev=id
Specifies the id value specified along with -fsdev option.

mount_tag=mount_tag
Specifies the tag name to be used by the guest to mount this export point.

-virtfs local,path=path,mount_tag=mount_tag ,security_model=security_model[,writeout=writeout][,readonly=on] [,fmode=fmode][,dmode=dmode][,multidevs=multidevs]

-virtfs proxy,socket=socket,mount_tag=mount_tag [,writeout=writeout][,readonly=on]

-virtfs proxy,sock_fd=sock_fd,mount_tag=mount_tag [,writeout=writeout][,readonly=on]

-virtfs synth,mount_tag=mount_tag
Define a new virtual filesystem device and expose it to the guest using a virtio-9p-device (a.k.a. 9pfs), which essentially means that a certain directory on host is made directly accessible by guest as a pass-through file system by using the 9P network protocol for communication between host and guests, if desired even accessible, shared by several guests simultaneously.

Note that `-virtfs` is actually just a convenience shortcut for its generalized form `-fsdev -device virtio-9p-pci`.

The general form of pass-through file system options are:

local
Accesses to the filesystem are done by QEMU.

proxy
Accesses to the filesystem are done by virtfs-proxy-helper(1). This option is deprecated (since QEMU 8.1) and will be removed in a future version of QEMU. Use `local` instead.

synth
Synthetic filesystem, only used by QTests.

id=id
Specifies identifier for the filesystem device

path=path
Specifies the export path for the file system device. Files under this path will be available to the 9p client on the guest.

security_model=security_model
Specifies the security model to be used for this export path. Supported security models are "passthrough", "mapped-xattr", "mapped-file" and "none". In "passthrough" security model, files are stored using the same credentials as they are created on the guest. This requires QEMU to run as root. In "mapped-xattr" security model, some of the file attributes like uid, gid, mode bits and link target are stored as file attributes. For "mapped-file" these attributes are stored in the hidden .virtfs_metadata directory. Directories exported by this security model cannot interact with other unix tools. "none" security model is same as passthrough except the sever won't report failures if it fails to set file attributes like ownership. Security model is mandatory only for local fsdriver. Other fsdrivers (like proxy) don't take security model as a parameter.

writeout=writeout
This is an optional argument. The only supported value is "immediate". This means that host page cache will be used to read and write data but write notification will be sent to the guest only when the data has been reported as written by the storage subsystem.

readonly=on
Enables exporting 9p share as a readonly mount for guests. By default read-write access is given.

socket=socket
Enables proxy filesystem driver to use passed socket file for communicating with virtfs-proxy-helper(1). Usually a helper like libvirt will create socketpair and pass one of the fds as sock_fd.

sock_fd
Enables proxy filesystem driver to use passed 'sock_fd' as the socket descriptor for interfacing with virtfs-proxy-helper(1).

fmode=fmode
Specifies the default mode for newly created files on the host. Works only with security models "mapped-xattr" and "mapped-file".

dmode=dmode
Specifies the default mode for newly created directories on the host. Works only with security models "mapped-xattr" and "mapped-file".

mount_tag=mount_tag
Specifies the tag name to be used by the guest to mount this export point.

multidevs=multidevs
Specifies how to deal with multiple devices being shared with a 9p export. Supported behaviours are either "remap", "forbid" or "warn". The latter is the default behaviour on which virtfs 9p expects only one device to be shared with the same export, and if more than one device is shared and accessed via the same 9p export then only a warning message is logged (once) by qemu on host side. In order to avoid file ID collisions on guest you should either create a separate virtfs export for each device to be shared with guests (recommended way) or you might use "remap" instead which allows you to share multiple devices with only one export instead, which is achieved by remapping the original inode numbers from host to guest in a way that would prevent such collisions. Remapping inodes in such use cases is required because the original device IDs from host are never passed and exposed on guest. Instead all files of an export shared with virtfs always share the same device id on guest. So two files with identical inode numbers but from actually different devices on host would otherwise cause a file ID collision and hence potential misbehaviours on

guest. “forbid” on the other hand assumes like “warn” that only one device is shared by the same export, however it will not only log a warning message but also deny access to additional devices on guest. Note though that “forbid” does currently not block all possible file access operations (e.g. readdir() would still return entries from other devices).

-iscsi
Configure iSCSI session parameters.

2.2.3 USB convenience options

-usb
Enable USB emulation on machine types with an on-board USB host controller (if not enabled by default). Note that on-board USB host controllers may not support USB 3.0. In this case `-device qemu-xhci` can be used instead on machines with PCI.

-usbdevice devname
Add the USB device devname, and enable an on-board USB controller if possible and necessary (just like it can be done via `-machine usb=on`). Note that this option is mainly intended for the user’s convenience only. More fine-grained control can be achieved by selecting a USB host controller (if necessary) and the desired USB device via the `-device` option instead. For example, instead of using `-usbdevice mouse` it is possible to use `-device qemu-xhci -device usb-mouse` to connect the USB mouse to a USB 3.0 controller instead (at least on machines that support PCI and do not have an USB controller enabled by default yet). For more details, see the chapter about *Connecting USB devices* in the System Emulation Users Guide. Possible devices for devname are:

braille
Braille device. This will use BrlAPI to display the braille output on a real or fake device (i.e. it also creates a corresponding `braille` chardev automatically beside the `usb-braille` USB device).

keyboard
Standard USB keyboard. Will override the PS/2 keyboard (if present).

mouse
Virtual Mouse. This will override the PS/2 mouse emulation when activated.

tablet
Pointer device that uses absolute coordinates (like a touchscreen). This means QEMU is able to report the mouse position without having to grab the mouse. Also overrides the PS/2 mouse emulation when activated.

wacom-tablet
Wacom PenPartner USB tablet.

2.2.4 Display options

-display type
Select type of display to use. Use `-display help` to list the available display types. Valid values for type are

spice-app[,gl=on|off]
Start QEMU as a Spice server and launch the default Spice client application. The Spice server will redirect the serial consoles and QEMU monitors. (Since 4.0)

dbus
Export the display over D-Bus interfaces. (Since 7.0)

The connection is registered with the “org.qemu” name (and queued when already owned).

addr=<dbusaddr> : D-Bus bus address to connect to.

p2p=yes|no : Use peer-to-peer connection, accepted via QMP add_client.

gl=on|off|core|es : Use OpenGL for rendering (the D-Bus interface will share framebuffers with DMABUF file descriptors).

sdl

Display video output via SDL (usually in a separate graphics window; see the SDL documentation for other possibilities). Valid parameters are:

grab-mod=<mods> : Used to select the modifier keys for toggling the mouse grabbing in conjunction with the "g" key. <mods> can be either lshift-lctrl-lalt or rctrl.

gl=on|off|core|es : Use OpenGL for displaying

show-cursor=on|off : Force showing the mouse cursor

window-close=on|off : Allow to quit qemu with window close button

gtk

Display video output in a GTK window. This interface provides drop-down menus and other UI elements to configure and control the VM during runtime. Valid parameters are:

full-screen=on|off : Start in fullscreen mode

gl=on|off : Use OpenGL for displaying

grab-on-hover=on|off : Grab keyboard input on mouse hover

show-tabs=on|off

[Display the tab bar for switching between the] various graphical interfaces (e.g. VGA and virtual console character devices) by default.

show-cursor=on|off : Force showing the mouse cursor

window-close=on|off : Allow to quit qemu with window close button

show-menubar=on|off : Display the main window menubar, defaults to "on"

zoom-to-fit=on|off

[Expand video output to the window size,] defaults to "off"

curses[,charset=<encoding>]

Display video output via curses. For graphics device models which support a text mode, QEMU can display this output using a curses/ncurses interface. Nothing is displayed when the graphics device is in graphical mode or if the graphics device does not support a text mode. Generally only the VGA device models support text mode. The font charset used by the guest can be specified with the charset option, for example charset=CP850 for IBM CP850 encoding. The default is CP437.

cocoa

Display video output in a Cocoa window. Mac only. This interface provides drop-down menus and other UI elements to configure and control the VM during runtime. Valid parameters are:

full-grab=on|off

[Capture all key presses, including system combos.] This requires accessibility permissions, since it performs a global grab on key events. (default: off) See https://support.apple.com/en-in/guide/mac-help/mh32356/mac

swap-opt-cmd=on|off

[Swap the Option and Command keys so that their] key codes match their position on non-Mac keyboards and you can use Meta/Super and Alt where you expect them. (default: off)

`show-cursor=on|off` : Force showing the mouse cursor

`left-command-key=on|off` : Disable forwarding left command key to host

`full-screen=on|off` : Start in fullscreen mode

zoom-to-fit=on|off
[Expand video output to the window size,] defaults to "off"

egl-headless[,rendernode=<file>]
Offload all OpenGL operations to a local DRI device. For any graphical display, this display needs to be paired with either VNC or SPICE displays.

vnc=<display>
Start a VNC server on display <display>

none
Do not display video output. The guest will still see an emulated graphics card, but its output will not be displayed to the QEMU user. This option differs from the -nographic option in that it only affects what is done with video output; -nographic also changes the destination of the serial and parallel port data.

-nographic
Normally, if QEMU is compiled with graphical window support, it displays output such as guest graphics, guest console, and the QEMU monitor in a window. With this option, you can totally disable graphical output so that QEMU is a simple command line application. The emulated serial port is redirected on the console and muxed with the monitor (unless redirected elsewhere explicitly). Therefore, you can still use QEMU to debug a Linux kernel with a serial console. Use C-a h for help on switching between the console and monitor.

-spice option[,option[,...]]
Enable the spice remote desktop protocol. Valid options are

port=<nr>
Set the TCP port spice is listening on for plaintext channels.

addr=<addr>
Set the IP address spice is listening on. Default is any address.

ipv4=on|off; ipv6=on|off; unix=on|off
Force using the specified IP version.

password-secret=<secret-id>
Set the ID of the `secret` object containing the password you need to authenticate.

sasl=on|off
Require that the client use SASL to authenticate with the spice. The exact choice of authentication method used is controlled from the system / user's SASL configuration file for the 'qemu' service. This is typically found in /etc/sasl2/qemu.conf. If running QEMU as an unprivileged user, an environment variable SASL_CONF_PATH can be used to make it search alternate locations for the service config. While some SASL auth methods can also provide data encryption (eg GSSAPI), it is recommended that SASL always be combined with the 'tls' and 'x509' settings to enable use of SSL and server certificates. This ensures a data encryption preventing compromise of authentication credentials.

disable-ticketing=on|off
Allow client connects without authentication.

disable-copy-paste=on|off
Disable copy paste between the client and the guest.

disable-agent-file-xfer=on|off
Disable spice-vdagent based file-xfer between the client and the guest.

tls-port=<nr>
Set the TCP port spice is listening on for encrypted channels.

x509-dir=<dir>
Set the x509 file directory. Expects same filenames as -vnc $display,x509=$dir

x509-key-file=<file>; x509-key-password=<file>; x509-cert-file=<file>; x509-cacert-file=<file>; x509-dh-key-file=<file>
The x509 file names can also be configured individually.

tls-ciphers=<list>
Specify which ciphers to use.

tls-channel=[main|display|cursor|inputs|record|playback]; plaintext-channel=[main|display|cursor|inputs|record|playback]
Force specific channel to be used with or without TLS encryption. The options can be specified multiple times to configure multiple channels. The special name "default" can be used to set the default mode. For channels which are not explicitly forced into one mode the spice client is allowed to pick tls/plaintext as he pleases.

image-compression=[auto_glz|auto_lz|quic|glz|lz|off]
Configure image compression (lossless). Default is auto_glz.

jpeg-wan-compression=[auto|never|always]; zlib-glz-wan-compression=[auto|never|always]
Configure wan image compression (lossy for slow links). Default is auto.

streaming-video=[off|all|filter]
Configure video stream detection. Default is off.

agent-mouse=[on|off]
Enable/disable passing mouse events via vdagent. Default is on.

playback-compression=[on|off]
Enable/disable audio stream compression (using celt 0.5.1). Default is on.

seamless-migration=[on|off]
Enable/disable spice seamless migration. Default is off.

gl=[on|off]
Enable/disable OpenGL context. Default is off.

rendernode=<file>
DRM render node for OpenGL rendering. If not specified, it will pick the first available. (Since 2.9)

-portrait
Rotate graphical output 90 deg left (only PXA LCD).

-rotate deg
Rotate graphical output some deg left (only PXA LCD).

-vga type
Select type of VGA card to emulate. Valid values for type are

cirrus
Cirrus Logic GD5446 Video card. All Windows versions starting from Windows 95 should recognize and use this graphic card. For optimal performances, use 16 bit color depth in the guest and the host OS. (This card was the default before QEMU 2.2)

std
Standard VGA card with Bochs VBE extensions. If your guest OS supports the VESA 2.0 VBE extensions (e.g. Windows XP) and if you want to use high resolution modes (>= 1280x1024x16) then you should use this option. (This card is the default since QEMU 2.2)

vmware
VMWare SVGA-II compatible adapter. Use it if you have sufficiently recent XFree86/XOrg server or Windows guest with a driver for this card.

qxl
QXL paravirtual graphic card. It is VGA compatible (including VESA 2.0 VBE support). Works best with qxl guest drivers installed though. Recommended choice when using the spice protocol.

tcx
(sun4m only) Sun TCX framebuffer. This is the default framebuffer for sun4m machines and offers both 8-bit and 24-bit colour depths at a fixed resolution of 1024x768.

cg3
(sun4m only) Sun cgthree framebuffer. This is a simple 8-bit framebuffer for sun4m machines available in both 1024x768 (OpenBIOS) and 1152x900 (OBP) resolutions aimed at people wishing to run older Solaris versions.

virtio
Virtio VGA card.

none
Disable VGA card.

-full-screen
Start in full screen.

-g *widthxheight*[x*depth*]
Set the initial graphical resolution and depth (PPC, SPARC only).

For PPC the default is 800x600x32.

For SPARC with the TCX graphics device, the default is 1024x768x8 with the option of 1024x768x24. For cgthree, the default is 1024x768x8 with the option of 1152x900x8 for people who wish to use OBP.

-vnc display[,option[,option[,...]]]
Normally, if QEMU is compiled with graphical window support, it displays output such as guest graphics, guest console, and the QEMU monitor in a window. With this option, you can have QEMU listen on VNC display display and redirect the VGA display over the VNC session. It is very useful to enable the usb tablet device when using this option (option `-device usb-tablet`). When using the VNC display, you must use the `-k` parameter to set the keyboard layout if you are not using en-us. Valid syntax for the display is

to=L
With this option, QEMU will try next available VNC displays, until the number L, if the originally defined "-vnc display" is not available, e.g. port 5900+display is already used by another application. By default, to=0.

host:d
TCP connections will only be allowed from host on display d. By convention the TCP port is 5900+d. Optionally, host can be omitted in which case the server will accept connections from any host.

unix:path
Connections will be allowed over UNIX domain sockets where path is the location of a unix socket to listen for connections on.

none
VNC is initialized but not started. The monitor `change` command can be used to later start the VNC server.

Following the display value there may be one or more option flags separated by commas. Valid options are

reverse=on|off
Connect to a listening VNC client via a "reverse" connection. The client is specified by the display. For

reverse network connections (host:d,``reverse``), the d argument is a TCP port number, not a display number.

websocket=on|off
Opens an additional TCP listening port dedicated to VNC Websocket connections. If a bare websocket option is given, the Websocket port is 5700+display. An alternative port can be specified with the syntax `websocket`=port.

If host is specified connections will only be allowed from this host. It is possible to control the websocket listen address independently, using the syntax `websocket`=host:port.

Websocket could be allowed over UNIX domain socket, using the syntax `websocket`=unix:path, where path is the location of a unix socket to listen for connections on.

If no TLS credentials are provided, the websocket connection runs in unencrypted mode. If TLS credentials are provided, the websocket connection requires encrypted client connections.

password=on|off
Require that password based authentication is used for client connections.

The password must be set separately using the `set_password` command in the *QEMU Monitor*. The syntax to change your password is: `set_password` <`protocol`> <`password`> where <protocol> could be either "vnc" or "spice".

If you would like to change <protocol> password expiration, you should use `expire_password` <`protocol`> <`expiration-time`> where expiration time could be one of the following options: now, never, +seconds or UNIX time of expiration, e.g. +60 to make password expire in 60 seconds, or 1335196800 to make password expire on "Mon Apr 23 12:00:00 EDT 2012" (UNIX time for this date and time).

You can also use keywords "now" or "never" for the expiration time to allow <protocol> password to expire immediately or never expire.

password-secret=<secret-id>
Require that password based authentication is used for client connections, using the password provided by the `secret` object identified by `secret-id`.

tls-creds=ID
Provides the ID of a set of TLS credentials to use to secure the VNC server. They will apply to both the normal VNC server socket and the websocket socket (if enabled). Setting TLS credentials will cause the VNC server socket to enable the VeNCrypt auth mechanism. The credentials should have been previously created using the `-object tls-creds` argument.

tls-authz=ID
Provides the ID of the QAuthZ authorization object against which the client's x509 distinguished name will validated. This object is only resolved at time of use, so can be deleted and recreated on the fly while the VNC server is active. If missing, it will default to denying access.

sasl=on|off
Require that the client use SASL to authenticate with the VNC server. The exact choice of authentication method used is controlled from the system / user's SASL configuration file for the 'qemu' service. This is typically found in /etc/sasl2/qemu.conf. If running QEMU as an unprivileged user, an environment variable SASL_CONF_PATH can be used to make it search alternate locations for the service config. While some SASL auth methods can also provide data encryption (eg GSSAPI), it is recommended that SASL always be combined with the 'tls' and 'x509' settings to enable use of SSL and server certificates. This ensures a data encryption preventing compromise of authentication credentials. See the *VNC security* section in the System Emulation Users Guide for details on using SASL authentication.

sasl-authz=ID
Provides the ID of the QAuthZ authorization object against which the client's SASL username will vali-

dated. This object is only resolved at time of use, so can be deleted and recreated on the fly while the VNC server is active. If missing, it will default to denying access.

acl=on|off
Legacy method for enabling authorization of clients against the x509 distinguished name and SASL username. It results in the creation of two `authz-list` objects with IDs of `vnc.username` and `vnc.x509dname`. The rules for these objects must be configured with the HMP ACL commands.

This option is deprecated and should no longer be used. The new `sasl-authz` and `tls-authz` options are a replacement.

lossy=on|off
Enable lossy compression methods (gradient, JPEG, ...). If this option is set, VNC client may receive lossy framebuffer updates depending on its encoding settings. Enabling this option can save a lot of bandwidth at the expense of quality.

non-adaptive=on|off
Disable adaptive encodings. Adaptive encodings are enabled by default. An adaptive encoding will try to detect frequently updated screen regions, and send updates in these regions using a lossy encoding (like JPEG). This can be really helpful to save bandwidth when playing videos. Disabling adaptive encodings restores the original static behavior of encodings like Tight.

share=[allow-exclusive|force-shared|ignore]
Set display sharing policy. 'allow-exclusive' allows clients to ask for exclusive access. As suggested by the rfb spec this is implemented by dropping other connections. Connecting multiple clients in parallel requires all clients asking for a shared session (vncviewer: -shared switch). This is the default. 'force-shared' disables exclusive client access. Useful for shared desktop sessions, where you don't want someone forgetting specify -shared disconnect everybody else. 'ignore' completely ignores the shared flag and allows everybody connect unconditionally. Doesn't conform to the rfb spec but is traditional QEMU behavior.

key-delay-ms
Set keyboard delay, for key down and key up events, in milliseconds. Default is 10. Keyboards are low-bandwidth devices, so this slowdown can help the device and guest to keep up and not lose events in case events are arriving in bulk. Possible causes for the latter are flaky network connections, or scripts for automated testing.

audiodev=audiodev
Use the specified audiodev when the VNC client requests audio transmission. When not using an -audiodev argument, this option must be omitted, otherwise is must be present and specify a valid audiodev.

power-control=on|off
Permit the remote client to issue shutdown, reboot or reset power control requests.

2.2.5 i386 target only

-win2k-hack
Use it when installing Windows 2000 to avoid a disk full bug. After Windows 2000 is installed, you no longer need this option (this option slows down the IDE transfers). Synonym of `-global ide-device.win2k-install-hack=on`.

-no-fd-bootchk
Disable boot signature checking for floppy disks in BIOS. May be needed to boot from old floppy disks. Synonym of `-m fd-bootchk=off`.

-acpitable [sig=str][,rev=n][,oem_id=str][,oem_table_id=str][,oem_rev=n]
[,asl_compiler_id=str][,asl_compiler_rev=n][,data=file1[:file2]...]
Add ACPI table with specified header fields and context from specified files. For file=, take whole ACPI table from the specified files, including all ACPI headers (possible overridden by other options). For data=, only data

portion of the table is used, all header information is specified in the command line. If a SLIC table is supplied to QEMU, then the SLIC's oem_id and oem_table_id fields will override the same in the RSDT and the FADT (a.k.a. FACP), in order to ensure the field matches required by the Microsoft SLIC spec and the ACPI spec.

-smbios file=binary
Load SMBIOS entry from binary file.

-smbios type=0[,vendor=str][,version=str][,date=str][,release=%d.%d][,uefi=on|off]
Specify SMBIOS type 0 fields

-smbios type=1[,manufacturer=str][,product=str][,version=str][,serial=str][,uuid=uuid][,sku=str][,family=str]
Specify SMBIOS type 1 fields

-smbios type=2[,manufacturer=str][,product=str][,version=str][,serial=str][,asset=str][,location=str]
Specify SMBIOS type 2 fields

-smbios type=3[,manufacturer=str][,version=str][,serial=str][,asset=str][,sku=str]
Specify SMBIOS type 3 fields

-smbios type=4[,sock_pfx=str][,manufacturer=str][,version=str][,serial=str][,asset=str][,part=str][,processor-family=%d][,processor-id=%d]
Specify SMBIOS type 4 fields

-smbios type=9[,slot_designation=str][,slot_type=%d][,slot_data_bus_width=%d][,current_usage=%d][,slot_length=%d][,slot_id=%d][,slot_characteristics1=%d][,slot_characteristics12=%d][,pci_device=str]
Specify SMBIOS type 9 fields

-smbios type=11[,value=str][,path=filename]
Specify SMBIOS type 11 fields

This argument can be repeated multiple times, and values are added in the order they are parsed. Applications intending to use OEM strings data are encouraged to use their application name as a prefix for the value string. This facilitates passing information for multiple applications concurrently.

The `value=str` syntax provides the string data inline, while the `path=filename` syntax loads data from a file on disk. Note that the file is not permitted to contain any NUL bytes.

Both the `value` and `path` options can be repeated multiple times and will be added to the SMBIOS table in the order in which they appear.

Note that on the x86 architecture, the total size of all SMBIOS tables is limited to 65535 bytes. Thus the OEM strings data is not suitable for passing large amounts of data into the guest. Instead it should be used as a indicator to inform the guest where to locate the real data set, for example, by specifying the serial ID of a block device.

An example passing three strings is

```
-smbios type=11,value=cloud-init:ds=nocloud-net;s=http://10.10.0.1:8000/,\
                value=anaconda:method=http://dl.fedoraproject.org/pub/fedora/linux/
↪releases/25/x86_64/os,\
                path=/some/file/with/oemstringsdata.txt
```

In the guest OS this is visible with the `dmidecode` command

```
$ dmidecode -t 11
Handle 0x0E00, DMI type 11, 5 bytes
OEM Strings
     String 1: cloud-init:ds=nocloud-net;s=http://10.10.0.1:8000/
     String 2: anaconda:method=http://dl.fedoraproject.org/pub/fedora/linux/
```

(continues on next page)

(continued from previous page)

```
releases/25/x86_64/os
    String 3: myapp:some extra data
```

-smbios type=17[,loc_pfx=str][,bank=str][,manufacturer=str][,serial=str][,asset=str][,part=str][,speed=%d]
Specify SMBIOS type 17 fields

-smbios type=41[,designation=str][,kind=str][,instance=%d][,pcidev=str]
Specify SMBIOS type 41 fields

This argument can be repeated multiple times. Its main use is to allow network interfaces be created as `enoX` on Linux, with X being the instance number, instead of the name depending on the interface position on the PCI bus.

Here is an example of use:

```
-netdev user,id=internet \
-device virtio-net-pci,mac=50:54:00:00:00:42,netdev=internet,id=internet-dev \
-smbios type=41,designation='Onboard LAN',instance=1,kind=ethernet,
pcidev=internet-dev
```

In the guest OS, the device should then appear as `eno1`:

..parsed-literal:

```
$ ip -brief l
lo               UNKNOWN        00:00:00:00:00:00 <LOOPBACK,UP,LOWER_UP>
eno1             UP             50:54:00:00:00:42 <BROADCAST,MULTICAST,UP,LOWER_UP>
```

Currently, the PCI device has to be attached to the root bus.

2.2.6 Network options

-nic [tap|bridge|user|l2tpv3|vde|netmap|af-xdp|vhost-user|socket][,...][,mac=macaddr][,model=mn]
This option is a shortcut for configuring both the on-board (default) guest NIC hardware and the host network backend in one go. The host backend options are the same as with the corresponding `-netdev` options below. The guest NIC model can be set with `model=modelname`. Use `model=help` to list the available device types. The hardware MAC address can be set with `mac=macaddr`.

The following two example do exactly the same, to show how `-nic` can be used to shorten the command line length:

```
qemu-system-x86_64 -netdev user,id=n1,ipv6=off -device e1000,netdev=n1,
mac=52:54:98:76:54:32
qemu-system-x86_64 -nic user,ipv6=off,model=e1000,mac=52:54:98:76:54:32
```

-nic none
Indicate that no network devices should be configured. It is used to override the default configuration (default NIC with "user" host network backend) which is activated if no other networking options are provided.

-netdev user,id=id[,option][,option][,...]
Configure user mode host network backend which requires no administrator privilege to run. Valid options are:

id=id
Assign symbolic name for use in monitor commands.

ipv4=on|off and ipv6=on|off
Specify that either IPv4 or IPv6 must be enabled. If neither is specified both protocols are enabled.

net=addr[/mask]
Set IP network address the guest will see. Optionally specify the netmask, either in the form a.b.c.d or as number of valid top-most bits. Default is 10.0.2.0/24.

host=addr
Specify the guest-visible address of the host. Default is the 2nd IP in the guest network, i.e. x.x.x.2.

ipv6-net=addr[/int]
Set IPv6 network address the guest will see (default is fec0::/64). The network prefix is given in the usual hexadecimal IPv6 address notation. The prefix size is optional, and is given as the number of valid top-most bits (default is 64).

ipv6-host=addr
Specify the guest-visible IPv6 address of the host. Default is the 2nd IPv6 in the guest network, i.e. xxxx::2.

restrict=on|off
If this option is enabled, the guest will be isolated, i.e. it will not be able to contact the host and no guest IP packets will be routed over the host to the outside. This option does not affect any explicitly set forwarding rules.

hostname=name
Specifies the client hostname reported by the built-in DHCP server.

dhcpstart=addr
Specify the first of the 16 IPs the built-in DHCP server can assign. Default is the 15th to 31st IP in the guest network, i.e. x.x.x.15 to x.x.x.31.

dns=addr
Specify the guest-visible address of the virtual nameserver. The address must be different from the host address. Default is the 3rd IP in the guest network, i.e. x.x.x.3.

ipv6-dns=addr
Specify the guest-visible address of the IPv6 virtual nameserver. The address must be different from the host address. Default is the 3rd IP in the guest network, i.e. xxxx::3.

dnssearch=domain
Provides an entry for the domain-search list sent by the built-in DHCP server. More than one domain suffix can be transmitted by specifying this option multiple times. If supported, this will cause the guest to automatically try to append the given domain suffix(es) in case a domain name can not be resolved.

Example:

```
qemu-system-x86_64 -nic user,dnssearch=mgmt.example.org,dnssearch=example.org
```

domainname=domain
Specifies the client domain name reported by the built-in DHCP server.

tftp=dir
When using the user mode network stack, activate a built-in TFTP server. The files in dir will be exposed as the root of a TFTP server. The TFTP client on the guest must be configured in binary mode (use the command `bin` of the Unix TFTP client). The built-in TFTP server is read-only; it does not implement any command for writing files. QEMU will not write to this directory.

tftp-server-name=name
In BOOTP reply, broadcast name as the "TFTP server name" (RFC2132 option 66). This can be used to advise the guest to load boot files or configurations from a different server than the host address.

bootfile=file

When using the user mode network stack, broadcast file as the BOOTP filename. In conjunction with `tftp`, this can be used to network boot a guest from a local directory.

Example (using pxelinux):

```
qemu-system-x86_64 -hda linux.img -boot n -device e1000,netdev=n1 \
    -netdev user,id=n1,tftp=/path/to/tftp/files,bootfile=/pxelinux.0
```

smb=dir[,smbserver=addr]

When using the user mode network stack, activate a built-in SMB server so that Windows OSes can access to the host files in `dir` transparently. The IP address of the SMB server can be set to addr. By default the 4th IP in the guest network is used, i.e. x.x.x.4.

In the guest Windows OS, the line:

```
10.0.2.4 smbserver
```

must be added in the file `C:\WINDOWS\LMHOSTS` (for windows 9x/Me) or `C:\WINNT\SYSTEM32\DRIVERS\ETC\LMHOSTS` (Windows NT/2000).

Then `dir` can be accessed in `\\smbserver\qemu`.

Note that a SAMBA server must be installed on the host OS.

hostfwd=[tcp|udp]:[hostaddr]:hostport-[guestaddr]:guestport

Redirect incoming TCP or UDP connections to the host port hostport to the guest IP address guestaddr on guest port guestport. If guestaddr is not specified, its value is x.x.x.15 (default first address given by the built-in DHCP server). By specifying hostaddr, the rule can be bound to a specific host interface. If no connection type is set, TCP is used. This option can be given multiple times.

For example, to redirect host X11 connection from screen 1 to guest screen 0, use the following:

```
# on the host
qemu-system-x86_64 -nic user,hostfwd=tcp:127.0.0.1:6001-:6000
# this host xterm should open in the guest X11 server
xterm -display :1
```

To redirect telnet connections from host port 5555 to telnet port on the guest, use the following:

```
# on the host
qemu-system-x86_64 -nic user,hostfwd=tcp::5555-:23
telnet localhost 5555
```

Then when you use on the host `telnet localhost 5555`, you connect to the guest telnet server.

guestfwd=[tcp]:server:port-dev; guestfwd=[tcp]:server:port-cmd:command

Forward guest TCP connections to the IP address server on port port to the character device dev or to a program executed by cmd:command which gets spawned for each connection. This option can be given multiple times.

You can either use a chardev directly and have that one used throughout QEMU's lifetime, like in the following example:

```
# open 10.10.1.1:4321 on bootup, connect 10.0.2.100:1234 to it whenever
# the guest accesses it
qemu-system-x86_64 -nic user,guestfwd=tcp:10.0.2.100:1234-tcp:10.10.1.1:4321
```

Or you can execute a command on every TCP connection established by the guest, so that QEMU behaves similar to an inetd process for that virtual server:

```
# call "netcat 10.10.1.1 4321" on every TCP connection to 10.0.2.100:1234
# and connect the TCP stream to its stdin/stdout
```

```
qemu-system-x86_64 -nic  'user,id=n1,guestfwd=tcp:10.0.2.100:1234-cmd:netcat 10.
↪10.1.1 4321'
```

-netdev tap,id=id[,fd=h][,ifname=name][,script=file][,downscript=dfile][,br=bridge][,helper=helper]

Configure a host TAP network backend with ID id.

Use the network script file to configure it and the network script dfile to deconfigure it. If name is not provided, the OS automatically provides one. The default network configure script is `/etc/qemu-ifup` and the default network deconfigure script is `/etc/qemu-ifdown`. Use `script=no` or `downscript=no` to disable script execution.

If running QEMU as an unprivileged user, use the network helper to configure the TAP interface and attach it to the bridge. The default network helper executable is `/path/to/qemu-bridge-helper` and the default bridge device is `br0`.

`fd=h` can be used to specify the handle of an already opened host TAP interface.

Examples:

```
#launch a QEMU instance with the default network script
qemu-system-x86_64 linux.img -nic tap
```

```
#launch a QEMU instance with two NICs, each one connected
#to a TAP device
qemu-system-x86_64 linux.img \
        -netdev tap,id=nd0,ifname=tap0 -device e1000,netdev=nd0 \
        -netdev tap,id=nd1,ifname=tap1 -device rtl8139,netdev=nd1
```

```
#launch a QEMU instance with the default network helper to
#connect a TAP device to bridge br0
qemu-system-x86_64 linux.img -device virtio-net-pci,netdev=n1 \
        -netdev tap,id=n1,"helper=/path/to/qemu-bridge-helper"
```

-netdev bridge,id=id[,br=bridge][,helper=helper]

Connect a host TAP network interface to a host bridge device.

Use the network helper helper to configure the TAP interface and attach it to the bridge. The default network helper executable is `/path/to/qemu-bridge-helper` and the default bridge device is `br0`.

Examples:

```
#launch a QEMU instance with the default network helper to
#connect a TAP device to bridge br0
qemu-system-x86_64 linux.img -netdev bridge,id=n1 -device virtio-net,netdev=n1
```

```
#launch a QEMU instance with the default network helper to
#connect a TAP device to bridge qemubr0
qemu-system-x86_64 linux.img -netdev bridge,br=qemubr0,id=n1 -device virtio-net,
↪netdev=n1
```

-netdev socket,id=id[,fd=h][,listen=[host]:port][,connect=host:port]

This host network backend can be used to connect the guest's network to another QEMU virtual machine using a TCP socket connection. If `listen` is specified, QEMU waits for incoming connections on port (host is optional). `connect` is used to connect to another QEMU instance using the `listen` option. `fd=h` specifies an already opened TCP socket.

Example:

```
# launch a first QEMU instance
qemu-system-x86_64 linux.img \
                 -device e1000,netdev=n1,mac=52:54:00:12:34:56 \
```

```
                -netdev socket,id=n1,listen=:1234
    # connect the network of this instance to the network of the first instance
    qemu-system-x86_64 linux.img \
                -device e1000,netdev=n2,mac=52:54:00:12:34:57 \
                -netdev socket,id=n2,connect=127.0.0.1:1234
```

-netdev socket,id=id[,fd=h][,mcast=maddr:port[,localaddr=addr]]

Configure a socket host network backend to share the guest's network traffic with another QEMU virtual machines using a UDP multicast socket, effectively making a bus for every QEMU with same multicast address maddr and port. NOTES:

1. Several QEMU can be running on different hosts and share same bus (assuming correct multicast setup for these hosts).
2. mcast support is compatible with User Mode Linux (argument `ethN=mcast`), see http://user-mode-linux.sf.net.
3. Use `fd=h` to specify an already opened UDP multicast socket.

Example:

```
# launch one QEMU instance
qemu-system-x86_64 linux.img \
                -device e1000,netdev=n1,mac=52:54:00:12:34:56 \
                -netdev socket,id=n1,mcast=230.0.0.1:1234
# launch another QEMU instance on same "bus"
qemu-system-x86_64 linux.img \
                -device e1000,netdev=n2,mac=52:54:00:12:34:57 \
                -netdev socket,id=n2,mcast=230.0.0.1:1234
# launch yet another QEMU instance on same "bus"
qemu-system-x86_64 linux.img \
                -device e1000,netdev=n3,mac=52:54:00:12:34:58 \
                -netdev socket,id=n3,mcast=230.0.0.1:1234
```

Example (User Mode Linux compat.):

```
# launch QEMU instance (note mcast address selected is UML's default)
qemu-system-x86_64 linux.img \
                -device e1000,netdev=n1,mac=52:54:00:12:34:56 \
                -netdev socket,id=n1,mcast=239.192.168.1:1102
# launch UML
/path/to/linux ubd0=/path/to/root_fs eth0=mcast
```

Example (send packets from host's 1.2.3.4):

```
qemu-system-x86_64 linux.img \
                -device e1000,netdev=n1,mac=52:54:00:12:34:56 \
                -netdev socket,id=n1,mcast=239.192.168.1:1102,localaddr=1.2.3.4
```

**-netdev l2tpv3,id=id,src=srcaddr,dst=dstaddr[,srcport=srcport][,dstport=dstport],
txsession=txsession[,rxsession=rxsession][,ipv6=on|off][,udp=on|off][,cookie64][,
counter][,pincounter][,txcookie=txcookie][,rxcookie=rxcookie][,offset=offset]**

Configure a L2TPv3 pseudowire host network backend. L2TPv3 (RFC3931) is a popular protocol to transport Ethernet (and other Layer 2) data frames between two systems. It is present in routers, firewalls and the Linux kernel (from version 3.3 onwards).

This transport allows a VM to communicate to another VM, router or firewall directly.

src=srcaddr
source address (mandatory)

dst=dstaddr
destination address (mandatory)

udp
select udp encapsulation (default is ip).

srcport=srcport
source udp port.

dstport=dstport
destination udp port.

ipv6
force v6, otherwise defaults to v4.

rxcookie=rxcookie; txcookie=txcookie
Cookies are a weak form of security in the l2tpv3 specification. Their function is mostly to prevent misconfiguration. By default they are 32 bit.

cookie64
Set cookie size to 64 bit instead of the default 32

counter=off
Force a 'cut-down' L2TPv3 with no counter as in draft-mkonstan-l2tpext-keyed-ipv6-tunnel-00

pincounter=on
Work around broken counter handling in peer. This may also help on networks which have packet reorder.

offset=offset
Add an extra offset between header and data

For example, to attach a VM running on host 4.3.2.1 via L2TPv3 to the bridge br-lan on the remote Linux host 1.2.3.4:

```
# Setup tunnel on linux host using raw ip as encapsulation
# on 1.2.3.4
ip l2tp add tunnel remote 4.3.2.1 local 1.2.3.4 tunnel_id 1 peer_tunnel_id 1 \
    encap udp udp_sport 16384 udp_dport 16384
ip l2tp add session tunnel_id 1 name vmtunnel0 session_id \
    0xFFFFFFFF peer_session_id 0xFFFFFFFF
ifconfig vmtunnel0 mtu 1500
ifconfig vmtunnel0 up
brctl addif br-lan vmtunnel0


# on 4.3.2.1
# launch QEMU instance - if your network has reorder or is very lossy add ,
↪pincounter

qemu-system-x86_64 linux.img -device e1000,netdev=n1 \
    -netdev l2tpv3,id=n1,src=4.2.3.1,dst=1.2.3.4,udp,srcport=16384,dstport=16384,
↪rxsession=0xffffffff,txsession=0xffffffff,counter
```

-netdev vde,id=id[,sock=socketpath][,port=n][,group=groupname][,mode=octalmode]
Configure VDE backend to connect to PORT n of a vde switch running on host and listening for incoming connections on socketpath. Use GROUP groupname and MODE octalmode to change default ownership and permissions for communication port. This option is only available if QEMU has been compiled with vde support enabled.

Example:

```
# launch vde switch
vde_switch -F -sock /tmp/myswitch
# launch QEMU instance
qemu-system-x86_64 linux.img -nic vde,sock=/tmp/myswitch
```

-netdev af-xdp,id=str,ifname=name[,mode=native|skb][,force-copy=on|off][,queues=n][,start-queue=m][,inhibit=on|off][,sock-fds=x:y:...:z]

Configure AF_XDP backend to connect to a network interface 'name' using AF_XDP socket. A specific program attach mode for a default XDP program can be forced with 'mode', defaults to best-effort, where the likely most performant mode will be in use. Number of queues 'n' should generally match the number or queues in the interface, defaults to 1. Traffic arriving on non-configured device queues will not be delivered to the network backend.

```
# set number of queues to 4
ethtool -L eth0 combined 4
# launch QEMU instance
qemu-system-x86_64 linux.img -device virtio-net-pci,netdev=n1 \
    -netdev af-xdp,id=n1,ifname=eth0,queues=4
```

'start-queue' option can be specified if a particular range of queues [m, m + n] should be in use. For example, this is may be necessary in order to use certain NICs in native mode. Kernel allows the driver to create a separate set of XDP queues on top of regular ones, and only these queues can be used for AF_XDP sockets. NICs that work this way may also require an additional traffic redirection with ethtool to these special queues.

```
# set number of queues to 1
ethtool -L eth0 combined 1
# redirect all the traffic to the second queue (id: 1)
# note: drivers may require non-empty key/mask pair.
ethtool -N eth0 flow-type ether \
    dst 00:00:00:00:00:00 m FF:FF:FF:FF:FF:FE action 1
ethtool -N eth0 flow-type ether \
    dst 00:00:00:00:00:01 m FF:FF:FF:FF:FF:FE action 1
# launch QEMU instance
qemu-system-x86_64 linux.img -device virtio-net-pci,netdev=n1 \
    -netdev af-xdp,id=n1,ifname=eth0,queues=1,start-queue=1
```

XDP program can also be loaded externally. In this case 'inhibit' option should be set to 'on' and 'sock-fds' provided with file descriptors for already open but not bound XDP sockets already added to a socket map for corresponding queues. One socket per queue.

```
qemu-system-x86_64 linux.img -device virtio-net-pci,netdev=n1 \
    -netdev af-xdp,id=n1,ifname=eth0,queues=3,inhibit=on,sock-fds=15:16:17
```

-netdev vhost-user,chardev=id[,vhostforce=on|off][,queues=n]

Establish a vhost-user netdev, backed by a chardev id. The chardev should be a unix domain socket backed one. The vhost-user uses a specifically defined protocol to pass vhost ioctl replacement messages to an application on the other end of the socket. On non-MSIX guests, the feature can be forced with vhostforce. Use 'queues=n' to specify the number of queues to be created for multiqueue vhost-user.

Example:

```
qemu -m 512 -object memory-backend-file,id=mem,size=512M,mem-path=/hugetlbfs,
↪share=on \
     -numa node,memdev=mem \
     -chardev socket,id=chr0,path=/path/to/socket \
     -netdev type=vhost-user,id=net0,chardev=chr0 \
     -device virtio-net-pci,netdev=net0
```

-netdev vhost-vdpa[,vhostdev=/path/to/dev][,vhostfd=h]
Establish a vhost-vdpa netdev.

vDPA device is a device that uses a datapath which complies with the virtio specifications with a vendor specific control path. vDPA devices can be both physically located on the hardware or emulated by software.

-netdev hubport,id=id,hubid=hubid[,netdev=nd]
Create a hub port on the emulated hub with ID hubid.

The hubport netdev lets you connect a NIC to a QEMU emulated hub instead of a single netdev. Alternatively, you can also connect the hubport to another netdev with ID nd by using the `netdev=nd` option.

-net nic[,netdev=nd][,macaddr=mac][,model=type] [,name=name][,addr=addr][,vectors=v]
Legacy option to configure or create an on-board (or machine default) Network Interface Card(NIC) and connect it either to the emulated hub with ID 0 (i.e. the default hub), or to the netdev nd. If model is omitted, then the default NIC model associated with the machine type is used. Note that the default NIC model may change in future QEMU releases, so it is highly recommended to always specify a model. Optionally, the MAC address can be changed to mac, the device address set to addr (PCI cards only), and a name can be assigned for use in monitor commands. Optionally, for PCI cards, you can specify the number v of MSI-X vectors that the card should have; this option currently only affects virtio cards; set v = 0 to disable MSI-X. If no `-net` option is specified, a single NIC is created. QEMU can emulate several different models of network card. Use `-net nic,model=help` for a list of available devices for your target.

-net user|tap|bridge|socket|l2tpv3|vde[,...][,name=name]
Configure a host network backend (with the options corresponding to the same `-netdev` option) and connect it to the emulated hub 0 (the default hub). Use name to specify the name of the hub port.

2.2.7 Character device options

The general form of a character device option is:

-chardev backend,id=id[,mux=on|off][,options]
Backend is one of: `null`, `socket`, `udp`, `msmouse`, `vc`, `ringbuf`, `file`, `pipe`, `console`, `serial`, `pty`, `stdio`, `braille`, `parallel`, `spicevmc`, `spiceport`. The specific backend will determine the applicable options.

Use `-chardev help` to print all available chardev backend types.

All devices must have an id, which can be any string up to 127 characters long. It is used to uniquely identify this device in other command line directives.

A character device may be used in multiplexing mode by multiple front-ends. Specify `mux=on` to enable this mode. A multiplexer is a "1:N" device, and here the "1" end is your specified chardev backend, and the "N" end is the various parts of QEMU that can talk to a chardev. If you create a chardev with `id=myid` and `mux=on`, QEMU will create a multiplexer with your specified ID, and you can then configure multiple front ends to use that chardev ID for their input/output. Up to four different front ends can be connected to a single multiplexed chardev. (Without multiplexing enabled, a chardev can only be used by a single front end.) For instance you could use this to allow a single stdio chardev to be used by two serial ports and the QEMU monitor:

```
-chardev stdio,mux=on,id=char0 \
-mon chardev=char0,mode=readline \
-serial chardev:char0 \
-serial chardev:char0
```

You can have more than one multiplexer in a system configuration; for instance you could have a TCP port multiplexed between UART 0 and UART 1, and stdio multiplexed between the QEMU monitor and a parallel port:

```
-chardev stdio,mux=on,id=char0 \
-mon chardev=char0,mode=readline \
-parallel chardev:char0 \
-chardev tcp,...,mux=on,id=char1 \
-serial chardev:char1 \
-serial chardev:char1
```

When you're using a multiplexed character device, some escape sequences are interpreted in the input. See the chapter about *Keys in the character backend multiplexer* in the System Emulation Users Guide for more details.

Note that some other command line options may implicitly create multiplexed character backends; for instance `-serial mon:stdio` creates a multiplexed stdio backend connected to the serial port and the QEMU monitor, and `-nographic` also multiplexes the console and the monitor to stdio.

There is currently no support for multiplexing in the other direction (where a single QEMU front end takes input and output from multiple chardevs).

Every backend supports the `logfile` option, which supplies the path to a file to record all data transmitted via the backend. The `logappend` option controls whether the log file will be truncated or appended to when opened.

The available backends are:

-chardev null,id=id
A void device. This device will not emit any data, and will drop any data it receives. The null backend does not take any options.

-chardev socket,id=id[,TCP options or unix options][,server=on|off][,wait=on|off][,telnet=on|off][,websocket=on|off][,reconnect=seconds][,tls-creds=id][,tls-authz=id]
Create a two-way stream socket, which can be either a TCP or a unix socket. A unix socket will be created if `path` is specified. Behaviour is undefined if TCP options are specified for a unix socket.

`server=on|off` specifies that the socket shall be a listening socket.

`wait=on|off` specifies that QEMU should not block waiting for a client to connect to a listening socket.

`telnet=on|off` specifies that traffic on the socket should interpret telnet escape sequences.

`websocket=on|off` specifies that the socket uses WebSocket protocol for communication.

`reconnect` sets the timeout for reconnecting on non-server sockets when the remote end goes away. qemu will delay this many seconds and then attempt to reconnect. Zero disables reconnecting, and is the default.

`tls-creds` requests enablement of the TLS protocol for encryption, and specifies the id of the TLS credentials to use for the handshake. The credentials must be previously created with the `-object tls-creds` argument.

`tls-auth` provides the ID of the QAuthZ authorization object against which the client's x509 distinguished name will be validated. This object is only resolved at time of use, so can be deleted and recreated on the fly while the chardev server is active. If missing, it will default to denying access.

TCP and unix socket options are given below:

TCP options:
port=port[,host=host][,to=to][,ipv4=on|off][,ipv6=on|off][,nodelay=on|off]
`host` for a listening socket specifies the local address to be bound. For a connecting socket species the remote host to connect to. `host` is optional for listening sockets. If not specified it defaults to `0.0.0.0`.

`port` for a listening socket specifies the local port to be bound. For a connecting socket specifies the port on the remote host to connect to. `port` can be given as either a port number or a service name. `port` is required.

`to` is only relevant to listening sockets. If it is specified, and `port` cannot be bound, QEMU will attempt to bind to subsequent ports up to and including `to` until it succeeds. `to` must be specified as a port number.

ipv4=on|off and ipv6=on|off specify that either IPv4 or IPv6 must be used. If neither is specified the socket may use either protocol.

nodelay=on|off disables the Nagle algorithm.

unix options: path=path[,abstract=on|off][,tight=on|off]
path specifies the local path of the unix socket. path is required. abstract=on|off specifies the use of the abstract socket namespace, rather than the filesystem. Optional, defaults to false. tight=on|off sets the socket length of abstract sockets to their minimum, rather than the full sun_path length. Optional, defaults to true.

-chardev udp,id=id[,host=host],port=port[,localaddr=localaddr][,localport=localport][,ipv4=on|off][,ipv6=on|off]
Sends all traffic from the guest to a remote host over UDP.

host specifies the remote host to connect to. If not specified it defaults to localhost.

port specifies the port on the remote host to connect to. port is required.

localaddr specifies the local address to bind to. If not specified it defaults to 0.0.0.0.

localport specifies the local port to bind to. If not specified any available local port will be used.

ipv4=on|off and ipv6=on|off specify that either IPv4 or IPv6 must be used. If neither is specified the device may use either protocol.

-chardev msmouse,id=id
Forward QEMU's emulated msmouse events to the guest. msmouse does not take any options.

-chardev vc,id=id[[,width=width][,height=height]][[,cols=cols][,rows=rows]]
Connect to a QEMU text console. vc may optionally be given a specific size.

width and height specify the width and height respectively of the console, in pixels.

cols and rows specify that the console be sized to fit a text console with the given dimensions.

-chardev ringbuf,id=id[,size=size]
Create a ring buffer with fixed size size. size must be a power of two and defaults to 64K.

-chardev file,id=id,path=path[,input-path=input-path]
Log all traffic received from the guest to a file.

path specifies the path of the file to be opened. This file will be created if it does not already exist, and overwritten if it does. path is required.

If input-path is specified, this is the path of a second file which will be used for input. If input-path is not specified, no input will be available from the chardev.

Note that input-path is not supported on Windows hosts.

-chardev pipe,id=id,path=path
Create a two-way connection to the guest. The behaviour differs slightly between Windows hosts and other hosts:

On Windows, a single duplex pipe will be created at \\.pipe\path.

On other hosts, 2 pipes will be created called path.in and path.out. Data written to path.in will be received by the guest. Data written by the guest can be read from path.out. QEMU will not create these fifos, and requires them to be present.

path forms part of the pipe path as described above. path is required.

-chardev console,id=id
Send traffic from the guest to QEMU's standard output. console does not take any options.

console is only available on Windows hosts.

-chardev serial,id=id,path=path
Send traffic from the guest to a serial device on the host.

On Unix hosts serial will actually accept any tty device, not only serial lines.

`path` specifies the name of the serial device to open.

-chardev pty,id=id
Create a new pseudo-terminal on the host and connect to it. `pty` does not take any options.

`pty` is not available on Windows hosts.

-chardev stdio,id=id[,signal=on|off]
Connect to standard input and standard output of the QEMU process.

`signal` controls if signals are enabled on the terminal, that includes exiting QEMU with the key sequence Control-c. This option is enabled by default, use `signal=off` to disable it.

-chardev braille,id=id
Connect to a local BrlAPI server. `braille` does not take any options.

-chardev parallel,id=id,path=path

`parallel` is only available on Linux, FreeBSD and DragonFlyBSD hosts.

Connect to a local parallel port.

`path` specifies the path to the parallel port device. `path` is required.

-chardev spicevmc,id=id,debug=debug,name=name
`spicevmc` is only available when spice support is built in.

`debug` debug level for spicevmc

`name` name of spice channel to connect to

Connect to a spice virtual machine channel, such as vdiport.

-chardev spiceport,id=id,debug=debug,name=name
`spiceport` is only available when spice support is built in.

`debug` debug level for spicevmc

`name` name of spice port to connect to

Connect to a spice port, allowing a Spice client to handle the traffic identified by a name (preferably a fqdn).

2.2.8 TPM device options

The general form of a TPM device option is:

-tpmdev backend,id=id[,options]
The specific backend type will determine the applicable options. The `-tpmdev` option creates the TPM backend and requires a `-device` option that specifies the TPM frontend interface model.

Use `-tpmdev help` to print all available TPM backend types.

The available backends are:

-tpmdev passthrough,id=id,path=path,cancel-path=cancel-path
(Linux-host only) Enable access to the host's TPM using the passthrough driver.

`path` specifies the path to the host's TPM device, i.e., on a Linux host this would be `/dev/tpm0`. `path` is optional and by default `/dev/tpm0` is used.

`cancel-path` specifies the path to the host TPM device's sysfs entry allowing for cancellation of an ongoing TPM command. `cancel-path` is optional and by default QEMU will search for the sysfs entry to use.

Some notes about using the host's TPM with the passthrough driver:

The TPM device accessed by the passthrough driver must not be used by any other application on the host.

Since the host's firmware (BIOS/UEFI) has already initialized the TPM, the VM's firmware (BIOS/UEFI) will not be able to initialize the TPM again and may therefore not show a TPM-specific menu that would otherwise allow the user to configure the TPM, e.g., allow the user to enable/disable or activate/deactivate the TPM. Further, if TPM ownership is released from within a VM then the host's TPM will get disabled and deactivated. To enable and activate the TPM again afterwards, the host has to be rebooted and the user is required to enter the firmware's menu to enable and activate the TPM. If the TPM is left disabled and/or deactivated most TPM commands will fail.

To create a passthrough TPM use the following two options:

```
-tpmdev passthrough,id=tpm0 -device tpm-tis,tpmdev=tpm0
```

Note that the `-tpmdev` id is `tpm0` and is referenced by `tpmdev=tpm0` in the device option.

-tpmdev emulator,id=id,chardev=dev
(Linux-host only) Enable access to a TPM emulator using Unix domain socket based chardev backend.

`chardev` specifies the unique ID of a character device backend that provides connection to the software TPM server.

To create a TPM emulator backend device with chardev socket backend:

```
-chardev socket,id=chrtpm,path=/tmp/swtpm-sock -tpmdev emulator,id=tpm0,
↪chardev=chrtpm -device tpm-tis,tpmdev=tpm0
```

2.2.9 Boot Image or Kernel specific

There are broadly 4 ways you can boot a system with QEMU.

- specify a firmware and let it control finding a kernel
- specify a firmware and pass a hint to the kernel to boot
- direct kernel image boot
- manually load files into the guest's address space

The third method is useful for quickly testing kernels but as there is no firmware to pass configuration information to the kernel the hardware must either be probeable, the kernel built for the exact configuration or passed some configuration data (e.g. a DTB blob) which tells the kernel what drivers it needs. This exact details are often hardware specific.

The final method is the most generic way of loading images into the guest address space and used mostly for `bare metal` type development where the reset vectors of the processor are taken into account.

For x86 machines and some other architectures `-bios` will generally do the right thing with whatever it is given. For other machines the more strict `-pflash` option needs an image that is sized for the flash device for the given machine type.

Please see the *QEMU System Emulator Targets* section of the manual for more detailed documentation.

-bios file
Set the filename for the BIOS.

-pflash file
Use file as a parallel flash image.

The kernel options were designed to work with Linux kernels although other things (like hypervisors) can be packaged up as a kernel executable image. The exact format of a executable image is usually architecture specific.

The way in which the kernel is started (what address it is loaded at, what if any information is passed to it via CPU registers, the state of the hardware when it is started, and so on) is also architecture specific. Typically it follows the specification laid down by the Linux kernel for how kernels for that architecture must be started.

-kernel bzImage
Use bzImage as kernel image. The kernel can be either a Linux kernel or in multiboot format.

-append cmdline
Use cmdline as kernel command line

-initrd file
Use file as initial ram disk.

-initrd "file1 arg=foo,file2"
This syntax is only available with multiboot.

Use file1 and file2 as modules and pass `arg=foo` as parameter to the first module. Commas can be provided in module parameters by doubling them on the command line to escape them:

-initrd "bzImage earlyprintk=xen,,keep root=/dev/xvda1,initrd.img"
Multiboot only. Use bzImage as the first module with "`earlyprintk=xen,keep root=/dev/xvda1`" as its command line, and initrd.img as the second module.

-dtb file
Use file as a device tree binary (dtb) image and pass it to the kernel on boot.

Finally you can also manually load images directly into the address space of the guest. This is most useful for developers who already know the layout of their guest and take care to ensure something sane will happen when the reset vector executes.

The generic loader can be invoked by using the loader device:

`-device loader,addr=<addr>,data=<data>,data-len=<data-len>[,data-be=<data-be>][,cpu-num=<cpu-num>]`

there is also the guest loader which operates in a similar way but tweaks the DTB so a hypervisor loaded via `-kernel` can find where the guest image is:

`-device guest-loader,addr=<addr>[,kernel=<path>,[bootargs=<arguments>]][,initrd=<path>]`

2.2.10 Debug/Expert options

-compat [deprecated-input=@var{input-policy}][,deprecated-output=@var{output-policy}]
Set policy for handling deprecated management interfaces (experimental):

deprecated-input=accept (default)
Accept deprecated commands and arguments

deprecated-input=reject
Reject deprecated commands and arguments

deprecated-input=crash
Crash on deprecated commands and arguments

deprecated-output=accept (default)
Emit deprecated command results and events

deprecated-output=hide
Suppress deprecated command results and events

Limitation: covers only syntactic aspects of QMP.

-compat [unstable-input=@var{input-policy}][,unstable-output=@var{output-policy}]
Set policy for handling unstable management interfaces (experimental):

unstable-input=accept (default)
Accept unstable commands and arguments

unstable-input=reject
Reject unstable commands and arguments

unstable-input=crash
Crash on unstable commands and arguments

unstable-output=accept (default)
Emit unstable command results and events

unstable-output=hide
Suppress unstable command results and events

Limitation: covers only syntactic aspects of QMP.

-fw_cfg [name=]name,file=file
Add named fw_cfg entry with contents from file file. If the filename contains comma, you must double it (for instance, "file=my,,file" to use file "my,file").

-fw_cfg [name=]name,string=str
Add named fw_cfg entry with contents from string str. If the string contains comma, you must double it (for instance, "string=my,,string" to use file "my,string").

The terminating NUL character of the contents of str will not be included as part of the fw_cfg item data. To insert contents with embedded NUL characters, you have to use the file parameter.

The fw_cfg entries are passed by QEMU through to the guest.

Example:

```
-fw_cfg name=opt/com.mycompany/blob,file=./my_blob.bin
```

creates an fw_cfg entry named opt/com.mycompany/blob with contents from ./my_blob.bin.

-serial dev
Redirect the virtual serial port to host character device dev. The default device is `vc` in graphical mode and `stdio` in non graphical mode.

This option can be used several times to simulate multiple serial ports.

You can use `-serial none` to suppress the creation of default serial devices.

Available character devices are:

vc[:WxH]
Virtual console. Optionally, a width and height can be given in pixel with

```
vc:800x600
```

It is also possible to specify width or height in characters:

```
vc:80Cx24C
```

pty
[Linux only] Pseudo TTY (a new PTY is automatically allocated)

none
No device is allocated. Note that for machine types which emulate systems where a serial device is always present in real hardware, this may be equivalent to the `null` option, in that the serial device is still present but all output is discarded. For boards where the number of serial ports is truly variable, this suppresses the creation of the device.

null
A guest will see the UART or serial device as present in the machine, but all output is discarded, and there is no input. Conceptually equivalent to redirecting the output to `/dev/null`.

chardev:id
Use a named character device defined with the `-chardev` option.

/dev/XXX
[Linux only] Use host tty, e.g. `/dev/ttyS0`. The host serial port parameters are set according to the emulated ones.

/dev/parportN
[Linux only, parallel port only] Use host parallel port N. Currently SPP and EPP parallel port features can be used.

file:filename
Write output to filename. No character can be read.

stdio
[Unix only] standard input/output

pipe:filename
name pipe filename

COMn
[Windows only] Use host serial port n

udp:[remote_host]:remote_port[@[src_ip]:src_port]
This implements UDP Net Console. When remote_host or src_ip are not specified they default to `0.0.0.0`. When not using a specified src_port a random port is automatically chosen.

If you just want a simple readonly console you can use `netcat` or `nc`, by starting QEMU with: `-serial udp::4555` and nc as: `nc -u -l -p 4555`. Any time QEMU writes something to that port it will appear in the netconsole session.

If you plan to send characters back via netconsole or you want to stop and start QEMU a lot of times, you should have QEMU use the same source port each time by using something like `-serial udp::4555@:4556` to QEMU. Another approach is to use a patched version of netcat which can listen to a TCP port and send and receive characters via udp. If you have a patched version of netcat which activates telnet remote echo and single char transfer, then you can use the following options to set up a netcat redirector to allow telnet on port 5555 to access the QEMU port.

QEMU Options:
-serial udp::4555@:4556

netcat options:
-u -P 4555 -L 0.0.0.0:4556 -t -p 5555 -I -T

telnet options:
localhost 5555

tcp:[host]:port[,server=on|off][,wait=on|off][,nodelay=on|off][,reconnect=seconds]
The TCP Net Console has two modes of operation. It can send the serial I/O to a location or wait for a connection from a location. By default the TCP Net Console is sent to host at the port. If you use the `server=on` option QEMU will wait for a client socket application to connect to the port before continuing,

unless the `wait=on|off` option was specified. The `nodelay=on|off` option disables the Nagle buffering algorithm. The `reconnect=on` option only applies if `server=no` is set, if the connection goes down it will attempt to reconnect at the given interval. If host is omitted, 0.0.0.0 is assumed. Only one TCP connection at a time is accepted. You can use `telnet=on` to connect to the corresponding character device.

Example to send tcp console to 192.168.0.2 port 4444
-serial tcp:192.168.0.2:4444

Example to listen and wait on port 4444 for connection
-serial tcp::4444,server=on

Example to not wait and listen on ip 192.168.0.100 port 4444
-serial tcp:192.168.0.100:4444,server=on,wait=off

telnet:host:port[,server=on|off][,wait=on|off][,nodelay=on|off]
The telnet protocol is used instead of raw tcp sockets. The options work the same as if you had specified `-serial tcp`. The difference is that the port acts like a telnet server or client using telnet option negotiation. This will also allow you to send the MAGIC_SYSRQ sequence if you use a telnet that supports sending the break sequence. Typically in unix telnet you do it with Control-] and then type "send break" followed by pressing the enter key.

websocket:host:port,server=on[,wait=on|off][,nodelay=on|off]
The WebSocket protocol is used instead of raw tcp socket. The port acts as a WebSocket server. Client mode is not supported.

unix:path[,server=on|off][,wait=on|off][,reconnect=seconds]
A unix domain socket is used instead of a tcp socket. The option works the same as if you had specified `-serial tcp` except the unix domain socket path is used for connections.

mon:dev_string
This is a special option to allow the monitor to be multiplexed onto another serial port. The monitor is accessed with key sequence of Control-a and then pressing c. dev_string should be any one of the serial devices specified above. An example to multiplex the monitor onto a telnet server listening on port 4444 would be:

```
-serial mon:telnet::4444,server=on,wait=off
```

When the monitor is multiplexed to stdio in this way, Ctrl+C will not terminate QEMU any more but will be passed to the guest instead.

braille
Braille device. This will use BrlAPI to display the braille output on a real or fake device.

msmouse
Three button serial mouse. Configure the guest to use Microsoft protocol.

-parallel dev
Redirect the virtual parallel port to host device dev (same devices as the serial port). On Linux hosts, `/dev/parportN` can be used to use hardware devices connected on the corresponding host parallel port.

This option can be used several times to simulate up to 3 parallel ports.

Use `-parallel none` to disable all parallel ports.

-monitor dev
Redirect the monitor to host device dev (same devices as the serial port). The default device is `vc` in graphical mode and `stdio` in non graphical mode. Use `-monitor none` to disable the default monitor.

-qmp dev
Like `-monitor` but opens in 'control' mode. For example, to make QMP available on localhost port 4444:

```
-qmp tcp:localhost:4444,server=on,wait=off
```

Not all options are configurable via this syntax; for maximum flexibility use the `-mon` option and an accompanying `-chardev`.

-qmp-pretty dev
Like `-qmp` but uses pretty JSON formatting.

-mon [chardev=]name[,mode=readline|control][,pretty[=on|off]]
Set up a monitor connected to the chardev `name`. QEMU supports two monitors: the Human Monitor Protocol (HMP; for human interaction), and the QEMU Monitor Protocol (QMP; a JSON RPC-style protocol). The default is HMP; `mode=control` selects QMP instead. `pretty` is only valid when `mode=control`, turning on JSON pretty printing to ease human reading and debugging.

For example:

```
-chardev socket,id=mon1,host=localhost,port=4444,server=on,wait=off \
-mon chardev=mon1,mode=control,pretty=on
```

enables the QMP monitor on localhost port 4444 with pretty-printing.

-debugcon dev
Redirect the debug console to host device dev (same devices as the serial port). The debug console is an I/O port which is typically port 0xe9; writing to that I/O port sends output to this device. The default device is `vc` in graphical mode and `stdio` in non graphical mode.

-pidfile file
Store the QEMU process PID in file. It is useful if you launch QEMU from a script.

--preconfig
Pause QEMU for interactive configuration before the machine is created, which allows querying and configuring properties that will affect machine initialization. Use QMP command 'x-exit-preconfig' to exit the preconfig state and move to the next state (i.e. run guest if -S isn't used or pause the second time if -S is used). This option is experimental.

-S
Do not start CPU at startup (you must type 'c' in the monitor).

-overcommit mem-lock=on|off

-overcommit cpu-pm=on|off
Run qemu with hints about host resource overcommit. The default is to assume that host overcommits all resources.

Locking qemu and guest memory can be enabled via `mem-lock=on` (disabled by default). This works when host memory is not overcommitted and reduces the worst-case latency for guest.

Guest ability to manage power state of host cpus (increasing latency for other processes on the same host cpu, but decreasing latency for guest) can be enabled via `cpu-pm=on` (disabled by default). This works best when host CPU is not overcommitted. When used, host estimates of CPU cycle and power utilization will be incorrect, not taking into account guest idle time.

-gdb dev
Accept a gdb connection on device dev (see the *GDB usage* chapter in the System Emulation Users Guide). Note that this option does not pause QEMU execution – if you want QEMU to not start the guest until you connect with gdb and issue a `continue` command, you will need to also pass the `-S` option to QEMU.

The most usual configuration is to listen on a local TCP socket:

```
-gdb tcp::3117
```

but you can specify other backends; UDP, pseudo TTY, or even stdio are all reasonable use cases. For example, a stdio connection allows you to start QEMU from within gdb and establish the connection via a pipe:

```
(gdb) target remote | exec qemu-system-x86_64 -gdb stdio ...
```

-s
Shorthand for -gdb tcp::1234, i.e. open a gdbserver on TCP port 1234 (see the *GDB usage* chapter in the System Emulation Users Guide).

-d item1[,...]
Enable logging of specified items. Use '-d help' for a list of log items.

-D logfile
Output log in logfile instead of to stderr

-dfilter range1[,...]
Filter debug output to that relevant to a range of target addresses. The filter spec can be either start+size, start-size or start..end where start end and size are the addresses and sizes required. For example:

```
-dfilter 0x8000..0x8fff,0xffffffc000080000+0x200,0xffffffc000060000-0x1000
```

Will dump output for any code in the 0x1000 sized block starting at 0x8000 and the 0x200 sized block starting at 0xffffffc000080000 and another 0x1000 sized block starting at 0xffffffc00005f000.

-seed number
Force the guest to use a deterministic pseudo-random number generator, seeded with number. This does not affect crypto routines within the host.

-L path
Set the directory for the BIOS, VGA BIOS and keymaps.

To list all the data directories, use `-L help`.

-enable-kvm
Enable KVM full virtualization support. This option is only available if KVM support is enabled when compiling.

-xen-domid id
Specify xen guest domain id (XEN only).

-xen-attach
Attach to existing xen domain. libxl will use this when starting QEMU (XEN only). Restrict set of available xen operations to specified domain id (XEN only).

-no-reboot
Exit instead of rebooting.

-no-shutdown
Don't exit QEMU on guest shutdown, but instead only stop the emulation. This allows for instance switching to monitor to commit changes to the disk image.

-action event=action
The action parameter serves to modify QEMU's default behavior when certain guest events occur. It provides a generic method for specifying the same behaviors that are modified by the `-no-reboot` and `-no-shutdown` parameters.

Examples:

```
-action panic=none  -action reboot=shutdown,shutdown=pause  -device i6300esb -action
watchdog=pause
```

-loadvm file
Start right away with a saved state (loadvm in monitor)

-daemonize
Daemonize the QEMU process after initialization. QEMU will not detach from standard IO until it is ready to receive connections on any of its devices. This option is a useful way for external programs to launch QEMU without having to cope with initialization race conditions.

-option-rom file
Load the contents of file as an option ROM. This option is useful to load things like EtherBoot.

-rtc [base=utc|localtime|datetime][,clock=host|rt|vm][,driftfix=none|slew]
Specify base as utc or localtime to let the RTC start at the current UTC or local time, respectively. localtime is required for correct date in MS-DOS or Windows. To start at a specific point in time, provide datetime in the format 2006-06-17T16:01:21 or 2006-06-17. The default base is UTC.

By default the RTC is driven by the host system time. This allows using of the RTC as accurate reference clock inside the guest, specifically if the host time is smoothly following an accurate external reference clock, e.g. via NTP. If you want to isolate the guest time from the host, you can set clock to rt instead, which provides a host monotonic clock if host support it. To even prevent the RTC from progressing during suspension, you can set clock to vm (virtual clock). 'clock=vm' is recommended especially in icount mode in order to preserve determinism; however, note that in icount mode the speed of the virtual clock is variable and can in general differ from the host clock.

Enable driftfix (i386 targets only) if you experience time drift problems, specifically with Windows' ACPI HAL. This option will try to figure out how many timer interrupts were not processed by the Windows guest and will re-inject them.

-icount [shift=N|auto][,align=on|off][,sleep=on|off][,rr=record|replay,rrfile=filename[,rrsnapshot=snapshot]]
Enable virtual instruction counter. The virtual cpu will execute one instruction every 2^N ns of virtual time. If auto is specified then the virtual cpu speed will be automatically adjusted to keep virtual time within a few seconds of real time.

Note that while this option can give deterministic behavior, it does not provide cycle accurate emulation. Modern CPUs contain superscalar out of order cores with complex cache hierarchies. The number of instructions executed often has little or no correlation with actual performance.

When the virtual cpu is sleeping, the virtual time will advance at default speed unless sleep=on is specified. With sleep=on, the virtual time will jump to the next timer deadline instantly whenever the virtual cpu goes to sleep mode and will not advance if no timer is enabled. This behavior gives deterministic execution times from the guest point of view. The default if icount is enabled is sleep=off. sleep=on cannot be used together with either shift=auto or align=on.

align=on will activate the delay algorithm which will try to synchronise the host clock and the virtual clock. The goal is to have a guest running at the real frequency imposed by the shift option. Whenever the guest clock is behind the host clock and if align=on is specified then we print a message to the user to inform about the delay. Currently this option does not work when shift is auto. Note: The sync algorithm will work for those shift values for which the guest clock runs ahead of the host clock. Typically this happens when the shift value is high (how high depends on the host machine). The default if icount is enabled is align=off.

When the rr option is specified deterministic record/replay is enabled. The rrfile= option must also be provided to specify the path to the replay log. In record mode data is written to this file, and in replay mode it is read back. If the rrsnapshot option is given then it specifies a VM snapshot name. In record mode, a new VM snapshot with the given name is created at the start of execution recording. In replay mode this option specifies the snapshot name used to load the initial VM state.

-watchdog-action action
The action controls what QEMU will do when the watchdog timer expires. The default is reset (forcefully reset the guest). Other possible actions are: shutdown (attempt to gracefully shutdown the guest), poweroff

(forcefully poweroff the guest), `inject-nmi` (inject a NMI into the guest), `pause` (pause the guest), `debug` (print a debug message and continue), or `none` (do nothing).

Note that the `shutdown` action requires that the guest responds to ACPI signals, which it may not be able to do in the sort of situations where the watchdog would have expired, and thus `-watchdog-action shutdown` is not recommended for production use.

Examples:

```
-device i6300esb -watchdog-action pause
```

-echr numeric_ascii_value
Change the escape character used for switching to the monitor when using monitor and serial sharing. The default is `0x01` when using the `-nographic` option. `0x01` is equal to pressing `Control-a`. You can select a different character from the ascii control keys where 1 through 26 map to Control-a through Control-z. For instance you could use the either of the following to change the escape character to Control-t.

`-echr 0x14`; `-echr 20`

-incoming tcp:[host]:port[,to=maxport][,ipv4=on|off][,ipv6=on|off]

-incoming rdma:host:port[,ipv4=on|off][,ipv6=on|off]
Prepare for incoming migration, listen on a given tcp port.

-incoming unix:socketpath
Prepare for incoming migration, listen on a given unix socket.

-incoming fd:fd
Accept incoming migration from a given file descriptor.

-incoming file:filename[,offset=offset]
Accept incoming migration from a given file starting at offset. offset allows the common size suffixes, or a 0x prefix, but not both.

-incoming exec:cmdline
Accept incoming migration as an output from specified external command.

-incoming defer
Wait for the URI to be specified via migrate_incoming. The monitor can be used to change settings (such as migration parameters) prior to issuing the migrate_incoming to allow the migration to begin.

-only-migratable
Only allow migratable devices. Devices will not be allowed to enter an unmigratable state.

-nodefaults
Don't create default devices. Normally, QEMU sets the default devices like serial port, parallel port, virtual console, monitor device, VGA adapter, floppy and CD-ROM drive and others. The `-nodefaults` option will disable all those default devices.

-runas user
Immediately before starting guest execution, drop root privileges, switching to the specified user. This option is deprecated, use `-run-with user=...` instead.

-prom-env variable=value
Set OpenBIOS nvram variable to given value (PPC, SPARC only).

```
qemu-system-sparc -prom-env 'auto-boot?=false' \
 -prom-env 'boot-device=sd(0,2,0):d' -prom-env 'boot-args=linux single'
```

```
qemu-system-ppc -prom-env 'auto-boot?=false' \
 -prom-env 'boot-device=hd:2,\yaboot' \
 -prom-env 'boot-args=conf=hd:2,\yaboot.conf'
```

-semihosting
Enable *Semihosting* mode (ARM, M68K, Xtensa, MIPS, RISC-V only).

> **Warning:** Note that this allows guest direct access to the host filesystem, so should only be used with a trusted guest OS.

See the -semihosting-config option documentation for further information about the facilities this enables.

-semihosting-config
[enable=on|off][,target=native|gdb|auto][,chardev=id][,userspace=on|off][,arg=str[,...]]
Enable and configure *Semihosting* (ARM, M68K, Xtensa, MIPS, RISC-V only).

> **Warning:** Note that this allows guest direct access to the host filesystem, so should only be used with a trusted guest OS.

target=native|gdb|auto
Defines where the semihosting calls will be addressed, to QEMU (`native`) or to GDB (`gdb`). The default is `auto`, which means `gdb` during debug sessions and `native` otherwise.

chardev=str1
Send the output to a chardev backend output for native or auto output when not in gdb

userspace=on|off
Allows code running in guest userspace to access the semihosting interface. The default is that only privileged guest code can make semihosting calls. Note that setting `userspace=on` should only be used if all guest code is trusted (for example, in bare-metal test case code).

arg=str1,arg=str2,...
Allows the user to pass input arguments, and can be used multiple times to build up a list. The old-style `-kernel`/`-append` method of passing a command line is still supported for backward compatibility. If both the `--semihosting-config arg` and the `-kernel`/`-append` are specified, the former is passed to semihosting as it always takes precedence.

-old-param
Old param mode (ARM only).

-sandbox
arg[,obsolete=string][,elevateprivileges=string][,spawn=string][,resourcecontrol=string]
Enable Seccomp mode 2 system call filter. 'on' will enable syscall filtering and 'off' will disable it. The default is 'off'.

obsolete=string
Enable Obsolete system calls

elevateprivileges=string
Disable set*uid|gid system calls

spawn=string
Disable *fork and execve

resourcecontrol=string
Disable process affinity and schedular priority

-readconfig file
Read device configuration from file. This approach is useful when you want to spawn QEMU process with many command line options but you don't want to exceed the command line character limit.

-no-user-config
The `-no-user-config` option makes QEMU not load any of the user-provided config files on sysconfdir.

-trace [[enable=]pattern][,events=file][,file=file]
Specify tracing options.

`[enable=]PATTERN`

Immediately enable events matching *PATTERN* (either event name or a globbing pattern). This option is only available if QEMU has been compiled with the `simple`, `log` or `ftrace` tracing backend. To specify multiple events or patterns, specify the `-trace` option multiple times.

Use `-trace help` to print a list of names of trace points.

`events=FILE`

Immediately enable events listed in *FILE*. The file must contain one event name (as listed in the `trace-events-all` file) per line; globbing patterns are accepted too. This option is only available if QEMU has been compiled with the `simple`, `log` or `ftrace` tracing backend.

`file=FILE`

Log output traces to *FILE*. This option is only available if QEMU has been compiled with the `simple` tracing backend.

-plugin file=file[,argname=argvalue]
Load a plugin.

file=file
Load the given plugin from a shared library file.

argname=argvalue
Argument passed to the plugin. (Can be given multiple times.)

-run-with [async-teardown=on|off][,chroot=dir][user=username|uid:gid]
Set QEMU process lifecycle options.

`async-teardown=on` enables asynchronous teardown. A new process called "cleanup/<QEMU_PID>" will be created at startup sharing the address space with the main QEMU process, using clone. It will wait for the main QEMU process to terminate completely, and then exit. This allows QEMU to terminate very quickly even if the guest was huge, leaving the teardown of the address space to the cleanup process. Since the cleanup process shares the same cgroups as the main QEMU process, accounting is performed correctly. This only works if the cleanup process is not forcefully killed with SIGKILL before the main QEMU process has terminated completely.

`chroot=dir` can be used for doing a chroot to the specified directory immediately before starting the guest execution. This is especially useful in combination with -runas.

`user=username` or `user=uid:gid` can be used to drop root privileges by switching to the specified user (via username) or user and group (via uid:gid) immediately before starting guest execution.

-msg [timestamp[=on|off]][,guest-name[=on|off]]
Control error message format.

timestamp=on|off
Prefix messages with a timestamp. Default is off.

guest-name=on|off
Prefix messages with guest name but only if -name guest option is set otherwise the option is ignored. Default is off.

-dump-vmstate file
Dump json-encoded vmstate information for current machine type to file in file

-enable-sync-profile
Enable synchronization profiling.

-perfmap
Generate a map file for Linux perf tools that will allow basic profiling information to be broken down into basic blocks.

-jitdump
Generate a dump file for Linux perf tools that maps basic blocks to symbol names, line numbers and JITted code.

2.2.11 Generic object creation

-object typename[,prop1=value1,...]
Create a new object of type typename setting properties in the order they are specified. Note that the 'id' property must be set. These objects are placed in the '/objects' path.

**-object memory-backend-file,id=id,size=size,mem-path=dir,share=on|off,
discard-data=on|off,merge=on|off,dump=on|off,prealloc=on|off,host-nodes=host-nodes,
policy=default|preferred|bind|interleave,align=align,offset=offset,readonly=on|off,
rom=on|off|auto**
Creates a memory file backend object, which can be used to back the guest RAM with huge pages.

The `id` parameter is a unique ID that will be used to reference this memory region in other parameters, e.g. `-numa`, `-device nvdimm`, etc.

The `size` option provides the size of the memory region, and accepts common suffixes, e.g. `500M`.

The `mem-path` provides the path to either a shared memory or huge page filesystem mount.

The `share` boolean option determines whether the memory region is marked as private to QEMU, or shared. The latter allows a co-operating external process to access the QEMU memory region.

Setting share=on might affect the ability to configure NUMA bindings for the memory backend under some circumstances, see Documentation/vm/numa_memory_policy.txt on the Linux kernel source tree for additional details.

Setting the `discard-data` boolean option to on indicates that file contents can be destroyed when QEMU exits, to avoid unnecessarily flushing data to the backing file. Note that `discard-data` is only an optimization, and QEMU might not discard file contents if it aborts unexpectedly or is terminated using SIGKILL.

The `merge` boolean option enables memory merge, also known as MADV_MERGEABLE, so that Kernel Samepage Merging will consider the pages for memory deduplication.

Setting the `dump` boolean option to off excludes the memory from core dumps. This feature is also known as MADV_DONTDUMP.

The `prealloc` boolean option enables memory preallocation.

The `host-nodes` option binds the memory range to a list of NUMA host nodes.

The `policy` option sets the NUMA policy to one of the following values:

default
default host policy

preferred
prefer the given host node list for allocation

bind
restrict memory allocation to the given host node list

interleave
interleave memory allocations across the given host node list

The `align` option specifies the base address alignment when QEMU mmap(2) `mem-path`, and accepts common suffixes, eg `2M`. Some backend store specified by `mem-path` requires an alignment different than the default one used by QEMU, eg the device DAX /dev/dax0.0 requires 2M alignment rather than 4K. In such cases, users can specify the required alignment via this option.

The `offset` option specifies the offset into the target file that the region starts at. You can use this parameter to back multiple regions with a single file.

The `pmem` option specifies whether the backing file specified by `mem-path` is in host persistent memory that can be accessed using the SNIA NVM programming model (e.g. Intel NVDIMM). If `pmem` is set to 'on', QEMU will take necessary operations to guarantee the persistence of its own writes to `mem-path` (e.g. in vNVDIMM label emulation and live migration). Also, we will map the backend-file with MAP_SYNC flag, which ensures the file metadata is in sync for `mem-path` in case of host crash or a power failure. MAP_SYNC requires support from both the host kernel (since Linux kernel 4.15) and the filesystem of `mem-path` mounted with DAX option.

The `readonly` option specifies whether the backing file is opened read-only or read-write (default).

The `rom` option specifies whether to create Read Only Memory (ROM) that cannot be modified by the VM. Any write attempts to such ROM will be denied. Most use cases want proper RAM instead of ROM. However, selected use cases, like R/O NVDIMMs, can benefit from ROM. If set to `on`, create ROM; if set to `off`, create writable RAM; if set to `auto` (default), the value of the `readonly` option is used. This option is primarily helpful when we want to have writable RAM in configurations that would traditionally create ROM before the `rom` option was introduced: VM templating, where we want to open a file readonly (`readonly=on`) and mark the memory to be private for QEMU (`share=off`). For this use case, we need writable RAM instead of ROM, and want to also set `rom=off`.

-object
memory-backend-ram,id=id,merge=on|off,dump=on|off,share=on|off,prealloc=on|off,
size=size,host-nodes=host-nodes,policy=default|preferred|bind|interleave
Creates a memory backend object, which can be used to back the guest RAM. Memory backend objects offer more control than the `-m` option that is traditionally used to define guest RAM. Please refer to `memory-backend-file` for a description of the options.

-object memory-backend-memfd,id=id,merge=on|off,dump=on|off,share=on|off,
prealloc=on|off,size=size,host-nodes=host-nodes,
policy=default|preferred|bind|interleave,seal=on|off,hugetlb=on|off,hugetlbsize=size
Creates an anonymous memory file backend object, which allows QEMU to share the memory with an external process (e.g. when using vhost-user). The memory is allocated with memfd and optional sealing. (Linux only)

The `seal` option creates a sealed-file, that will block further resizing the memory ('on' by default).

The `hugetlb` option specify the file to be created resides in the hugetlbfs filesystem (since Linux 4.14). Used in conjunction with the `hugetlb` option, the `hugetlbsize` option specify the hugetlb page size on systems that support multiple hugetlb page sizes (it must be a power of 2 value supported by the system).

In some versions of Linux, the `hugetlb` option is incompatible with the `seal` option (requires at least Linux 4.16).

Please refer to `memory-backend-file` for a description of the other options.

The `share` boolean option is on by default with memfd.

-object iommufd,id=id[,fd=fd]
Creates an iommufd backend which allows control of DMA mapping through the /dev/iommu device.

The id parameter is a unique ID which frontends (such as vfio-pci of vdpa) will use to connect with the iommufd backend.

The fd parameter is an optional pre-opened file descriptor resulting from /dev/iommu opening. Usually the iommufd is shared across all subsystems, bringing the benefit of centralized reference counting.

-object rng-builtin,id=id
Creates a random number generator backend which obtains entropy from QEMU builtin functions. The id parameter is a unique ID that will be used to reference this entropy backend from the virtio-rng device. By default, the virtio-rng device uses this RNG backend.

-object rng-random,id=id,filename=/dev/random
Creates a random number generator backend which obtains entropy from a device on the host. The id parameter is a unique ID that will be used to reference this entropy backend from the virtio-rng device. The filename parameter specifies which file to obtain entropy from and if omitted defaults to /dev/urandom.

-object rng-egd,id=id,chardev=chardevid
Creates a random number generator backend which obtains entropy from an external daemon running on the host. The id parameter is a unique ID that will be used to reference this entropy backend from the virtio-rng device. The chardev parameter is the unique ID of a character device backend that provides the connection to the RNG daemon.

-object
tls-creds-anon,id=id,endpoint=endpoint,dir=/path/to/cred/dir,verify-peer=on|off
Creates a TLS anonymous credentials object, which can be used to provide TLS support on network backends. The id parameter is a unique ID which network backends will use to access the credentials. The endpoint is either server or client depending on whether the QEMU network backend that uses the credentials will be acting as a client or as a server. If verify-peer is enabled (the default) then once the handshake is completed, the peer credentials will be verified, though this is a no-op for anonymous credentials.

The dir parameter tells QEMU where to find the credential files. For server endpoints, this directory may contain a file dh-params.pem providing diffie-hellman parameters to use for the TLS server. If the file is missing, QEMU will generate a set of DH parameters at startup. This is a computationally expensive operation that consumes random pool entropy, so it is recommended that a persistent set of parameters be generated upfront and saved.

-object
tls-creds-psk,id=id,endpoint=endpoint,dir=/path/to/keys/dir[,username=username]
Creates a TLS Pre-Shared Keys (PSK) credentials object, which can be used to provide TLS support on network backends. The id parameter is a unique ID which network backends will use to access the credentials. The endpoint is either server or client depending on whether the QEMU network backend that uses the credentials will be acting as a client or as a server. For clients only, username is the username which will be sent to the server. If omitted it defaults to "qemu".

The dir parameter tells QEMU where to find the keys file. It is called "dir/keys.psk" and contains "username:key" pairs. This file can most easily be created using the GnuTLS psktool program.

For server endpoints, dir may also contain a file dh-params.pem providing diffie-hellman parameters to use for the TLS server. If the file is missing, QEMU will generate a set of DH parameters at startup. This is a computationally expensive operation that consumes random pool entropy, so it is recommended that a persistent set of parameters be generated up front and saved.

-object tls-creds-x509,id=id,endpoint=endpoint,dir=/path/to/cred/dir,
priority=priority,verify-peer=on|off,passwordid=id
Creates a TLS anonymous credentials object, which can be used to provide TLS support on network back-

ends. The `id` parameter is a unique ID which network backends will use to access the credentials. The `endpoint` is either `server` or `client` depending on whether the QEMU network backend that uses the credentials will be acting as a client or as a server. If `verify-peer` is enabled (the default) then once the handshake is completed, the peer credentials will be verified. With x509 certificates, this implies that the clients must be provided with valid client certificates too.

The dir parameter tells QEMU where to find the credential files. For server endpoints, this directory may contain a file dh-params.pem providing diffie-hellman parameters to use for the TLS server. If the file is missing, QEMU will generate a set of DH parameters at startup. This is a computationally expensive operation that consumes random pool entropy, so it is recommended that a persistent set of parameters be generated upfront and saved.

For x509 certificate credentials the directory will contain further files providing the x509 certificates. The certificates must be stored in PEM format, in filenames ca-cert.pem, ca-crl.pem (optional), server-cert.pem (only servers), server-key.pem (only servers), client-cert.pem (only clients), and client-key.pem (only clients).

For the server-key.pem and client-key.pem files which contain sensitive private keys, it is possible to use an encrypted version by providing the passwordid parameter. This provides the ID of a previously created `secret` object containing the password for decryption.

The priority parameter allows to override the global default priority used by gnutls. This can be useful if the system administrator needs to use a weaker set of crypto priorities for QEMU without potentially forcing the weakness onto all applications. Or conversely if one wants wants a stronger default for QEMU than for all other applications, they can do this through this parameter. Its format is a gnutls priority string as described at https://gnutls.org/manual/html_node/Priority-Strings.html.

-object tls-cipher-suites,id=id,priority=priority

Creates a TLS cipher suites object, which can be used to control the TLS cipher/protocol algorithms that applications are permitted to use.

The `id` parameter is a unique ID which frontends will use to access the ordered list of permitted TLS cipher suites from the host.

The `priority` parameter allows to override the global default priority used by gnutls. This can be useful if the system administrator needs to use a weaker set of crypto priorities for QEMU without potentially forcing the weakness onto all applications. Or conversely if one wants wants a stronger default for QEMU than for all other applications, they can do this through this parameter. Its format is a gnutls priority string as described at https://gnutls.org/manual/html_node/Priority-Strings.html.

An example of use of this object is to control UEFI HTTPS Boot. The tls-cipher-suites object exposes the ordered list of permitted TLS cipher suites from the host side to the guest firmware, via fw_cfg. The list is represented as an array of IANA_TLS_CIPHER objects. The firmware uses the IANA_TLS_CIPHER array for configuring guest-side TLS.

In the following example, the priority at which the host-side policy is retrieved is given by the `priority` property. Given that QEMU uses GNUTLS, `priority=@SYSTEM` may be used to refer to /etc/crypto-policies/back-ends/gnutls.config.

```
# qemu-system-x86_64 \
    -object tls-cipher-suites,id=mysuite0,priority=@SYSTEM \
    -fw_cfg name=etc/edk2/https/ciphers,gen_id=mysuite0
```

-object filter-buffer,id=id,netdev=netdevid,interval=t[,queue=all|rx|tx][,
status=on|off][,position=head|tail|id=<id>][,insert=behind|before]

Interval t can't be 0, this filter batches the packet delivery: all packets arriving in a given interval on netdev netdevid are delayed until the end of the interval. Interval is in microseconds. `status` is optional that indicate whether the netfilter is on (enabled) or off (disabled), the default status for netfilter will be 'on'.

queue all|rx|tx is an option that can be applied to any netfilter.

`all`: the filter is attached both to the receive and the transmit queue of the netdev (default).

`rx`: the filter is attached to the receive queue of the netdev, where it will receive packets sent to the netdev.

`tx`: the filter is attached to the transmit queue of the netdev, where it will receive packets sent by the netdev.

position head|tail|id=<id> is an option to specify where the filter should be inserted in the filter list. It can be applied to any netfilter.

`head`: the filter is inserted at the head of the filter list, before any existing filters.

`tail`: the filter is inserted at the tail of the filter list, behind any existing filters (default).

`id=<id>`: the filter is inserted before or behind the filter specified by <id>, see the insert option below.

insert behind|before is an option to specify where to insert the new filter relative to the one specified with position=id=<id>. It can be applied to any netfilter.

`before`: insert before the specified filter.

`behind`: insert behind the specified filter (default).

-object filter-mirror,id=id,netdev=netdevid,outdev=chardevid,queue=all|rx|tx[,vnet_hdr_support][,position=head|tail|id=<id>][,insert=behind|before]

filter-mirror on netdev netdevid,mirror net packet to chardevchardevid, if it has the vnet_hdr_support flag, filter-mirror will mirror packet with vnet_hdr_len.

-object filter-redirector,id=id,netdev=netdevid,indev=chardevid,outdev=chardevid,queue=all|rx|tx[,vnet_hdr_support][,position=head|tail|id=<id>][,insert=behind|before]

filter-redirector on netdev netdevid,redirect filter's net packet to chardev chardevid,and redirect indev's packet to filter.if it has the vnet_hdr_support flag, filter-redirector will redirect packet with vnet_hdr_len. Create a filter-redirector we need to differ outdev id from indev id, id can not be the same. we can just use indev or outdev, but at least one of indev or outdev need to be specified.

-object filter-rewriter,id=id,netdev=netdevid,queue=all|rx|tx,[vnet_hdr_support][,position=head|tail|id=<id>][,insert=behind|before]

Filter-rewriter is a part of COLO project.It will rewrite tcp packet to secondary from primary to keep secondary tcp connection,and rewrite tcp packet to primary from secondary make tcp packet can be handled by client.if it has the vnet_hdr_support flag, we can parse packet with vnet header.

usage: colo secondary: -object filter-redirector,id=f1,netdev=hn0,queue=tx,indev=red0 -object filter-redirector,id=f2,netdev=hn0,queue=rx,outdev=red1 -object filter-rewriter,id=rew0,netdev=hn0,queue=all

-object filter-dump,id=id,netdev=dev[,file=filename][,maxlen=len][,position=head|tail|id=<id>][,insert=behind|before]

Dump the network traffic on netdev dev to the file specified by filename. At most len bytes (64k by default) per packet are stored. The file format is libpcap, so it can be analyzed with tools such as tcpdump or Wireshark.

-object colo-compare,id=id,primary_in=chardevid,secondary_in=chardevid,outdev=chardevid,iothread=id[,vnet_hdr_support][,notify_dev=id][,compare_timeout=@var{ms}][,expired_scan_cycle=@var{ms}][,max_queue_size=@var{size}]

Colo-compare gets packet from primary_in chardevid and secondary_in, then compare whether the payload of primary packet and secondary packet are the same. If same, it will output primary packet to out_dev, else it will notify COLO-framework to do checkpoint and send primary packet to out_dev. In order to improve efficiency, we need to put the task of comparison in another iothread. If it has the vnet_hdr_support flag, colo compare will send/recv packet with vnet_hdr_len. The compare_timeout=@var{ms} determines the maximum time of the colo-compare hold the packet. The expired_scan_cycle=@var{ms} is to set the period of scanning expired primary node network packets. The max_queue_size=@var{size} is to set the max compare queue size depend on user environment. If user want to use Xen COLO, need to add the notify_dev to notify Xen colo-frame to do checkpoint.

COLO-compare must be used with the help of filter-mirror, filter-redirector and filter-rewriter.

```
KVM COLO

primary:
-netdev tap,id=hn0,vhost=off
-device e1000,id=e0,netdev=hn0,mac=52:a4:00:12:78:66
-chardev socket,id=mirror0,host=3.3.3.3,port=9003,server=on,wait=off
-chardev socket,id=compare1,host=3.3.3.3,port=9004,server=on,wait=off
-chardev socket,id=compare0,host=3.3.3.3,port=9001,server=on,wait=off
-chardev socket,id=compare0-0,host=3.3.3.3,port=9001
-chardev socket,id=compare_out,host=3.3.3.3,port=9005,server=on,wait=off
-chardev socket,id=compare_out0,host=3.3.3.3,port=9005
-object iothread,id=iothread1
-object filter-mirror,id=m0,netdev=hn0,queue=tx,outdev=mirror0
-object filter-redirector,netdev=hn0,id=redire0,queue=rx,indev=compare_out
-object filter-redirector,netdev=hn0,id=redire1,queue=rx,outdev=compare0
-object colo-compare,id=comp0,primary_in=compare0-0,secondary_in=compare1,
↪outdev=compare_out0,iothread=iothread1

secondary:
-netdev tap,id=hn0,vhost=off
-device e1000,netdev=hn0,mac=52:a4:00:12:78:66
-chardev socket,id=red0,host=3.3.3.3,port=9003
-chardev socket,id=red1,host=3.3.3.3,port=9004
-object filter-redirector,id=f1,netdev=hn0,queue=tx,indev=red0
-object filter-redirector,id=f2,netdev=hn0,queue=rx,outdev=red1


Xen COLO

primary:
-netdev tap,id=hn0,vhost=off
-device e1000,id=e0,netdev=hn0,mac=52:a4:00:12:78:66
-chardev socket,id=mirror0,host=3.3.3.3,port=9003,server=on,wait=off
-chardev socket,id=compare1,host=3.3.3.3,port=9004,server=on,wait=off
-chardev socket,id=compare0,host=3.3.3.3,port=9001,server=on,wait=off
-chardev socket,id=compare0-0,host=3.3.3.3,port=9001
-chardev socket,id=compare_out,host=3.3.3.3,port=9005,server=on,wait=off
-chardev socket,id=compare_out0,host=3.3.3.3,port=9005
-chardev socket,id=notify_way,host=3.3.3.3,port=9009,server=on,wait=off
-object filter-mirror,id=m0,netdev=hn0,queue=tx,outdev=mirror0
-object filter-redirector,netdev=hn0,id=redire0,queue=rx,indev=compare_out
-object filter-redirector,netdev=hn0,id=redire1,queue=rx,outdev=compare0
-object iothread,id=iothread1
-object colo-compare,id=comp0,primary_in=compare0-0,secondary_in=compare1,
↪outdev=compare_out0,notify_dev=notify_way,iothread=iothread1

secondary:
-netdev tap,id=hn0,vhost=off
-device e1000,netdev=hn0,mac=52:a4:00:12:78:66
-chardev socket,id=red0,host=3.3.3.3,port=9003
-chardev socket,id=red1,host=3.3.3.3,port=9004
```

(continues on next page)

(continued from previous page)

```
-object filter-redirector,id=f1,netdev=hn0,queue=tx,indev=red0
-object filter-redirector,id=f2,netdev=hn0,queue=rx,outdev=red1
```

If you want to know the detail of above command line, you can read the colo-compare git log.

-object cryptodev-backend-builtin,id=id[,queues=queues]

Creates a cryptodev backend which executes crypto operations from the QEMU cipher APIs. The id parameter is a unique ID that will be used to reference this cryptodev backend from the `virtio-crypto` device. The queues parameter is optional, which specify the queue number of cryptodev backend, the default of queues is 1.

```
# qemu-system-x86_64 \
  [...] \
      -object cryptodev-backend-builtin,id=cryptodev0 \
      -device virtio-crypto-pci,id=crypto0,cryptodev=cryptodev0 \
  [...]
```

-object cryptodev-vhost-user,id=id,chardev=chardevid[,queues=queues]

Creates a vhost-user cryptodev backend, backed by a chardev chardevid. The id parameter is a unique ID that will be used to reference this cryptodev backend from the `virtio-crypto` device. The chardev should be a unix domain socket backed one. The vhost-user uses a specifically defined protocol to pass vhost ioctl replacement messages to an application on the other end of the socket. The queues parameter is optional, which specify the queue number of cryptodev backend for multiqueue vhost-user, the default of queues is 1.

```
# qemu-system-x86_64 \
  [...] \
      -chardev socket,id=chardev0,path=/path/to/socket \
      -object cryptodev-vhost-user,id=cryptodev0,chardev=chardev0 \
      -device virtio-crypto-pci,id=crypto0,cryptodev=cryptodev0 \
  [...]
```

-object secret,id=id,data=string,format=raw|base64[,keyid=secretid,iv=string]

-object secret,id=id,file=filename,format=raw|base64[,keyid=secretid,iv=string]

Defines a secret to store a password, encryption key, or some other sensitive data. The sensitive data can either be passed directly via the data parameter, or indirectly via the file parameter. Using the data parameter is insecure unless the sensitive data is encrypted.

The sensitive data can be provided in raw format (the default), or base64. When encoded as JSON, the raw format only supports valid UTF-8 characters, so base64 is recommended for sending binary data. QEMU will convert from which ever format is provided to the format it needs internally. eg, an RBD password can be provided in raw format, even though it will be base64 encoded when passed onto the RBD sever.

For added protection, it is possible to encrypt the data associated with a secret using the AES-256-CBC cipher. Use of encryption is indicated by providing the keyid and iv parameters. The keyid parameter provides the ID of a previously defined secret that contains the AES-256 decryption key. This key should be 32-bytes long and be base64 encoded. The iv parameter provides the random initialization vector used for encryption of this particular secret and should be a base64 encrypted string of the 16-byte IV.

The simplest (insecure) usage is to provide the secret inline

```
# qemu-system-x86_64 -object secret,id=sec0,data=letmein,format=raw
```

The simplest secure usage is to provide the secret via a file

printf "letmein" > mypasswd.txt # QEMU_SYSTEM_MACRO -object secret,id=sec0,file=mypasswd.txt,format=raw

For greater security, AES-256-CBC should be used. To illustrate usage, consider the openssl command line tool which can encrypt the data. Note that when encrypting, the plaintext must be padded to the cipher block size (32 bytes) using the standard PKCS#5/6 compatible padding algorithm.

First a master key needs to be created in base64 encoding:

```
# openssl rand -base64 32 > key.b64
# KEY=$(base64 -d key.b64 | hexdump  -v -e '/1 "%02X"')
```

Each secret to be encrypted needs to have a random initialization vector generated. These do not need to be kept secret

```
# openssl rand -base64 16 > iv.b64
# IV=$(base64 -d iv.b64 | hexdump  -v -e '/1 "%02X"')
```

The secret to be defined can now be encrypted, in this case we're telling openssl to base64 encode the result, but it could be left as raw bytes if desired.

```
# SECRET=$(printf "letmein" |
           openssl enc -aes-256-cbc -a -K $KEY -iv $IV)
```

When launching QEMU, create a master secret pointing to `key.b64` and specify that to be used to decrypt the user password. Pass the contents of `iv.b64` to the second secret

```
# qemu-system-x86_64 \
    -object secret,id=secmaster0,format=base64,file=key.b64 \
    -object secret,id=sec0,keyid=secmaster0,format=base64,\
        data=$SECRET,iv=$(<iv.b64)
```

-object sev-guest,id=id,cbitpos=cbitpos,reduced-phys-bits=val,[sev-device=string,policy=policy,handle=handle,dh-cert-file=file,session-file=file,kernel-hashes=on|off]

Create a Secure Encrypted Virtualization (SEV) guest object, which can be used to provide the guest memory encryption support on AMD processors.

When memory encryption is enabled, one of the physical address bit (aka the C-bit) is utilized to mark if a memory page is protected. The `cbitpos` is used to provide the C-bit position. The C-bit position is Host family dependent hence user must provide this value. On EPYC, the value should be 47.

When memory encryption is enabled, we loose certain bits in physical address space. The `reduced-phys-bits` is used to provide the number of bits we loose in physical address space. Similar to C-bit, the value is Host family dependent. On EPYC, a guest will lose a maximum of 1 bit, so the value should be 1.

The `sev-device` provides the device file to use for communicating with the SEV firmware running inside AMD Secure Processor. The default device is '/dev/sev'. If hardware supports memory encryption then /dev/sev devices are created by CCP driver.

The `policy` provides the guest policy to be enforced by the SEV firmware and restrict what configuration and operational commands can be performed on this guest by the hypervisor. The policy should be provided by the guest owner and is bound to the guest and cannot be changed throughout the lifetime of the guest. The default is 0.

If guest `policy` allows sharing the key with another SEV guest then `handle` can be use to provide handle of the guest from which to share the key.

The `dh-cert-file` and `session-file` provides the guest owner's Public Diffie-Hillman key defined in SEV spec. The PDH and session parameters are used for establishing a cryptographic session with the guest owner to negotiate keys used for attestation. The file must be encoded in base64.

The `kernel-hashes` adds the hashes of given kernel/initrd/ cmdline to a designated guest firmware page for measured Linux boot with -kernel. The default is off. (Since 6.2)

e.g to launch a SEV guest

```
# qemu-system-x86_64 \
    ..... \
    -object sev-guest,id=sev0,cbitpos=47,reduced-phys-bits=1 \
    -machine ...,memory-encryption=sev0 \
    .....
```

-object authz-simple,id=id,identity=string

Create an authorization object that will control access to network services.

The `identity` parameter is identifies the user and its format depends on the network service that authorization object is associated with. For authorizing based on TLS x509 certificates, the identity must be the x509 distinguished name. Note that care must be taken to escape any commas in the distinguished name.

An example authorization object to validate a x509 distinguished name would look like:

```
# qemu-system-x86_64 \
    ... \
    -object 'authz-simple,id=auth0,identity=CN=laptop.example.com,,O=Example␣
↪Org,,L=London,,ST=London,,C=GB' \
    ...
```

Note the use of quotes due to the x509 distinguished name containing whitespace, and escaping of ‘,’.

-object authz-listfile,id=id,filename=path,refresh=on|off

Create an authorization object that will control access to network services.

The `filename` parameter is the fully qualified path to a file containing the access control list rules in JSON format.

An example set of rules that match against SASL usernames might look like:

```
{
  "rules": [
     { "match": "fred", "policy": "allow", "format": "exact" },
     { "match": "bob", "policy": "allow", "format": "exact" },
     { "match": "danb", "policy": "deny", "format": "glob" },
     { "match": "dan*", "policy": "allow", "format": "exact" },
  ],
  "policy": "deny"
}
```

When checking access the object will iterate over all the rules and the first rule to match will have its `policy` value returned as the result. If no rules match, then the default `policy` value is returned.

The rules can either be an exact string match, or they can use the simple UNIX glob pattern matching to allow wildcards to be used.

If `refresh` is set to true the file will be monitored and automatically reloaded whenever its content changes.

As with the `authz-simple` object, the format of the identity strings being matched depends on the network service, but is usually a TLS x509 distinguished name, or a SASL username.

An example authorization object to validate a SASL username would look like:

```
# qemu-system-x86_64 \
    ... \
    -object authz-simple,id=auth0,filename=/etc/qemu/vnc-sasl.acl,refresh=on \
```

```
    ...
```

-object authz-pam,id=id,service=string

Create an authorization object that will control access to network services.

The `service` parameter provides the name of a PAM service to use for authorization. It requires that a file `/etc/pam.d/service` exist to provide the configuration for the `account` subsystem.

An example authorization object to validate a TLS x509 distinguished name would look like:

```
# qemu-system-x86_64 \
    ... \
    -object authz-pam,id=auth0,service=qemu-vnc \
    ...
```

There would then be a corresponding config file for PAM at `/etc/pam.d/qemu-vnc` that contains:

```
account requisite  pam_listfile.so item=user sense=allow \
           file=/etc/qemu/vnc.allow
```

Finally the `/etc/qemu/vnc.allow` file would contain the list of x509 distinguished names that are permitted access

```
CN=laptop.example.com,O=Example Home,L=London,ST=London,C=GB
```

-object iothread,id=id,poll-max-ns=poll-max-ns,poll-grow=poll-grow, poll-shrink=poll-shrink,aio-max-batch=aio-max-batch

Creates a dedicated event loop thread that devices can be assigned to. This is known as an IOThread. By default device emulation happens in vCPU threads or the main event loop thread. This can become a scalability bottleneck. IOThreads allow device emulation and I/O to run on other host CPUs.

The `id` parameter is a unique ID that will be used to reference this IOThread from `-device ..., iothread=id`. Multiple devices can be assigned to an IOThread. Note that not all devices support an `iothread` parameter.

The `query-iothreads` QMP command lists IOThreads and reports their thread IDs so that the user can configure host CPU pinning/affinity.

IOThreads use an adaptive polling algorithm to reduce event loop latency. Instead of entering a blocking system call to monitor file descriptors and then pay the cost of being woken up when an event occurs, the polling algorithm spins waiting for events for a short time. The algorithm's default parameters are suitable for many cases but can be adjusted based on knowledge of the workload and/or host device latency.

The `poll-max-ns` parameter is the maximum number of nanoseconds to busy wait for events. Polling can be disabled by setting this value to 0.

The `poll-grow` parameter is the multiplier used to increase the polling time when the algorithm detects it is missing events due to not polling long enough.

The `poll-shrink` parameter is the divisor used to decrease the polling time when the algorithm detects it is spending too long polling without encountering events.

The `aio-max-batch` parameter is the maximum number of requests in a batch for the AIO engine, 0 means that the engine will use its default.

The IOThread parameters can be modified at run-time using the `qom-set` command (where `iothread1` is the IOThread's `id`):

```
(qemu) qom-set /objects/iothread1 poll-max-ns 100000
```

2.2.12 Device URL Syntax

In addition to using normal file images for the emulated storage devices, QEMU can also use networked resources such as iSCSI devices. These are specified using a special URL syntax.

iSCSI

iSCSI support allows QEMU to access iSCSI resources directly and use as images for the guest storage. Both disk and cdrom images are supported.

Syntax for specifying iSCSI LUNs is "iscsi://<target-ip>[:<port>]/<target-iqn>/<lun>"

By default qemu will use the iSCSI initiator-name 'iqn.2008-11.org.linux-kvm[:<name>]' but this can also be set from the command line or a configuration file.

Since version QEMU 2.4 it is possible to specify a iSCSI request timeout to detect stalled requests and force a reestablishment of the session. The timeout is specified in seconds. The default is 0 which means no timeout. Libiscsi 1.15.0 or greater is required for this feature.

Example (without authentication):

```
qemu-system-x86_64 -iscsi initiator-name=iqn.2001-04.com.example:my-initiator \
                   -cdrom iscsi://192.0.2.1/iqn.2001-04.com.example/2 \
                   -drive file=iscsi://192.0.2.1/iqn.2001-04.com.example/1
```

Example (CHAP username/password via URL):

```
qemu-system-x86_64 -drive file=iscsi://user%password@192.0.2.1/iqn.2001-04.com.
↪example/1
```

Example (CHAP username/password via environment variables):

```
LIBISCSI_CHAP_USERNAME="user" \
LIBISCSI_CHAP_PASSWORD="password" \
qemu-system-x86_64 -drive file=iscsi://192.0.2.1/iqn.2001-04.com.example/1
```

NBD

QEMU supports NBD (Network Block Devices) both using TCP protocol as well as Unix Domain Sockets. With TCP, the default port is 10809.

Syntax for specifying a NBD device using TCP, in preferred URI form: "nbd://<server-ip>[:<port>]/[<export>]"

Syntax for specifying a NBD device using Unix Domain Sockets; remember that '?' is a shell glob character and may need quoting: "nbd+unix:///[<export>]?socket=<domain-socket>"

Older syntax that is also recognized: "nbd:<server-ip>:<port>[:exportname=<export>]"

Syntax for specifying a NBD device using Unix Domain Sockets "nbd:unix:<domain-socket>[:exportname=<export>]"

Example for TCP

```
qemu-system-x86_64 --drive file=nbd:192.0.2.1:30000
```

Example for Unix Domain Sockets

```
qemu-system-x86_64 --drive file=nbd:unix:/tmp/nbd-socket
```

SSH

QEMU supports SSH (Secure Shell) access to remote disks.

Examples:

```
qemu-system-x86_64 -drive file=ssh://user@host/path/to/disk.img
qemu-system-x86_64 -drive file.driver=ssh,file.user=user,file.host=host,file.
↪port=22,file.path=/path/to/disk.img
```

Currently authentication must be done using ssh-agent. Other authentication methods may be supported in future.

GlusterFS

GlusterFS is a user space distributed file system. QEMU supports the use of GlusterFS volumes for hosting VM disk images using TCP and Unix Domain Sockets transport protocols.

Syntax for specifying a VM disk image on GlusterFS volume is

```
URI:
gluster[+type]://[host[:port]]/volume/path[?socket=...][,debug=N][,logfile=...]

JSON:
'json:{"driver":"qcow2","file":{"driver":"gluster","volume":"testvol","path":"a.img
↪","debug":N,"logfile":"...",
                                  "server":[{"type":"tcp","host":"...","port":"..."},
                                            {"type":"unix","socket":"..."}]}}'
```

Example

```
URI:
qemu-system-x86_64 --drive file=gluster://192.0.2.1/testvol/a.img,
                                file.debug=9,file.logfile=/var/log/qemu-gluster.log

JSON:
qemu-system-x86_64 'json:{"driver":"qcow2",
                          "file":{"driver":"gluster",
                                  "volume":"testvol","path":"a.img",
                                  "debug":9,"logfile":"/var/log/qemu-gluster.log",
                                  "server":[{"type":"tcp","host":"1.2.3.4",
↪"port":24007},
                                            {"type":"unix","socket":"/var/run/
↪glusterd.socket"}]}}'
qemu-system-x86_64 -drive driver=qcow2,file.driver=gluster,file.volume=testvol,file.
↪path=/path/a.img,
                                    file.debug=9,file.logfile=/var/log/
↪qemu-gluster.log,
                                    file.server.0.type=tcp,file.server.0.host=1.2.
↪3.4,file.server.0.port=24007,
                                    file.server.1.type=unix,file.server.1.socket=/
↪var/run/glusterd.socket
```

See also http://www.gluster.org.

HTTP/HTTPS/FTP/FTPS

QEMU supports read-only access to files accessed over http(s) and ftp(s).

Syntax using a single filename:

```
<protocol>://[<username>[:<password>]@]<host>/<path>
```

where:

protocol
: 'http', 'https', 'ftp', or 'ftps'.

username
: Optional username for authentication to the remote server.

password
Optional password for authentication to the remote server.

host
Address of the remote server.

path
Path on the remote server, including any query string.

The following options are also supported:

url
The full URL when passing options to the driver explicitly.

readahead
The amount of data to read ahead with each range request to the remote server. This value may optionally have the suffix 'T', 'G', 'M', 'K', 'k' or 'b'. If it does not have a suffix, it will be assumed to be in bytes. The value must be a multiple of 512 bytes. It defaults to 256k.

sslverify
Whether to verify the remote server's certificate when connecting over SSL. It can have the value 'on' or 'off'. It defaults to 'on'.

cookie
Send this cookie (it can also be a list of cookies separated by ';') with each outgoing request. Only supported when using protocols such as HTTP which support cookies, otherwise ignored.

timeout
Set the timeout in seconds of the CURL connection. This timeout is the time that CURL waits for a response from the remote server to get the size of the image to be downloaded. If not set, the default timeout of 5 seconds is used.

Note that when passing options to qemu explicitly, `driver` is the value of <protocol>.

Example: boot from a remote Fedora 20 live ISO image

```
qemu-system-x86_64 --drive media=cdrom,file=https://archives.fedoraproject.org/pub/
↪archive/fedora/linux/releases/20/Live/x86_64/Fedora-Live-Desktop-x86_64-20-1.iso,
↪readonly

qemu-system-x86_64 --drive media=cdrom,file.driver=http,file.url=http:/
↪/archives.fedoraproject.org/pub/fedora/linux/releases/20/Live/x86_64/
↪Fedora-Live-Desktop-x86_64-20-1.iso,readonly
```

Example: boot from a remote Fedora 20 cloud image using a local overlay for writes, copy-on-read, and a readahead of 64k

```
qemu-img create -f qcow2 -o backing_file='json:{"file.driver":"http",,␣
↪"file.url":"http://archives.fedoraproject.org/pub/archive/fedora/linux/
releases/20/Images/x86_64/Fedora\unhbox\voidb@x\kern\z@\char‘\protect\
discretionary{\char\defaulthyphenchar}{}{}x86_64\unhbox\voidb@x\kern\z@\char‘\
protect\discretionary{\char\defaulthyphenchar}{}{}20\unhbox\voidb@x\kern\z@\char‘\
protect\discretionary{\char\defaulthyphenchar}{}{}20131211.1\unhbox\voidb@x\kern\
z@\char‘\protect\discretionary{\char\defaulthyphenchar}{}{}sda.qcow2",, "file.
↪readahead":"64k"}' /tmp/Fedora-x86_64-20-20131211.1-sda.qcow2

qemu-system-x86_64 -drive file=/tmp/Fedora-x86_64-20-20131211.1-sda.qcow2,
↪copy-on-read=on
```

Example: boot from an image stored on a VMware vSphere server with a self-signed certificate using a local overlay for writes, a readahead of 64k and a timeout of 10 seconds.

```
qemu-img create -f qcow2 -o backing_file='json:{"file.driver":"https",, "file.
↪url":"https://user:password@vsphere.example.com/folder/test/test\unhbox\voidb@
x\kern\z@\char`\protect\discretionary{\char\defaulthyphenchar}{}{}flat.vmdk?dcPath=
Datacenter&dsName=datastore1",, "file.sslverify":"off",, "file.readahead":"64k",,␣
↪"file.timeout":10}' /tmp/test.qcow2

qemu-system-x86_64 -drive file=/tmp/test.qcow2
```

2.3 Device Emulation

QEMU supports the emulation of a large number of devices from peripherals such network cards and USB devices to integrated systems on a chip (SoCs). Configuration of these is often a source of confusion so it helps to have an understanding of some of the terms used to describes devices within QEMU.

2.3.1 Common Terms

Device Front End

A device front end is how a device is presented to the guest. The type of device presented should match the hardware that the guest operating system is expecting to see. All devices can be specified with the `--device` command line option. Running QEMU with the command line options `--device help` will list all devices it is aware of. Using the command line `--device foo,help` will list the additional configuration options available for that device.

A front end is often paired with a back end, which describes how the host's resources are used in the emulation.

Device Buses

Most devices will exist on a BUS of some sort. Depending on the machine model you choose (`-M foo`) a number of buses will have been automatically created. In most cases the BUS a device is attached to can be inferred, for example PCI devices are generally automatically allocated to the next free address of first PCI bus found. However in complicated configurations you can explicitly specify what bus (`bus=ID`) a device is attached to along with its address (`addr=N`).

Some devices, for example a PCI SCSI host controller, will add an additional buses to the system that other devices can be attached to. A hypothetical chain of devices might look like:

–device foo,bus=pci.0,addr=0,id=foo –device bar,bus=foo.0,addr=1,id=baz

which would be a bar device (with the ID of baz) which is attached to the first foo bus (foo.0) at address 1. The foo device which provides that bus is itself is attached to the first PCI bus (pci.0).

Device Back End

The back end describes how the data from the emulated device will be processed by QEMU. The configuration of the back end is usually specific to the class of device being emulated. For example serial devices will be backed by a `--chardev` which can redirect the data to a file or socket or some other system. Storage devices are handled by `--blockdev` which will specify how blocks are handled, for example being stored in a qcow2 file or accessing a raw host disk partition. Back ends can sometimes be stacked to implement features like snapshots.

While the choice of back end is generally transparent to the guest, there are cases where features will not be reported to the guest if the back end is unable to support it.

Device Pass Through

Device pass through is where the device is actually given access to the underlying hardware. This can be as simple as exposing a single USB device on the host system to the guest or dedicating a video card in a PCI slot to the exclusive use of the guest.

2.3.2 Emulated Devices

CAN Bus Emulation Support

The CAN bus emulation provides mechanism to connect multiple emulated CAN controller chips together by one or multiple CAN buses (the controller device "canbus" parameter). The individual buses can be connected to host system CAN API (at this time only Linux SocketCAN is supported).

The concept of buses is generic and different CAN controllers can be implemented.

The initial submission implemented SJA1000 controller which is common and well supported by by drivers for the most operating systems.

The PCI addon card hardware has been selected as the first CAN interface to implement because such device can be easily connected to systems with different CPU architectures (x86, PowerPC, Arm, etc.).

In 2020, CTU CAN FD controller model has been added as part of the bachelor thesis of Jan Charvat. This controller is complete open-source/design/hardware solution. The core designer of the project is Ondrej Ille, the financial support has been provided by CTU, and more companies including Volkswagen subsidiaries.

The project has been initially started in frame of RTEMS GSoC 2013 slot by Jin Yang under our mentoring The initial idea was to provide generic CAN subsystem for RTEMS. But lack of common environment for code and RTEMS testing lead to goal change to provide environment which provides complete emulated environment for testing and RTEMS GSoC slot has been donated to work on CAN hardware emulation on QEMU.

Examples how to use CAN emulation for SJA1000 based boards

When QEMU with CAN PCI support is compiled then one of the next CAN boards can be selected

(1) CAN bus Kvaser PCI CAN-S (single SJA1000 channel) board. QEMU startup options:

```
-object can-bus,id=canbus0
-device kvaser_pci,canbus=canbus0
```

Add "can-host-socketcan" object to connect device to host system CAN bus:

```
-object can-host-socketcan,id=canhost0,if=can0,canbus=canbus0
```

(2) CAN bus PCM-3680I PCI (dual SJA1000 channel) emulation:

```
-object can-bus,id=canbus0
-device pcm3680_pci,canbus0=canbus0,canbus1=canbus0
```

Another example:

```
-object can-bus,id=canbus0
-object can-bus,id=canbus1
-device pcm3680_pci,canbus0=canbus0,canbus1=canbus1
```

(3) CAN bus MIOe-3680 PCI (dual SJA1000 channel) emulation:

```
-device mioe3680_pci,canbus0=canbus0
```

The ''kvaser_pci" board/device model is compatible with and has been tested with the ''kvaser_pci" driver included in mainline Linux kernel. The tested setup was Linux 4.9 kernel on the host and guest side.

Example for qemu-system-x86_64:

```
qemu-system-x86_64 -accel kvm -kernel /boot/vmlinuz-4.9.0-4-amd64 \
  -initrd ramdisk.cpio \
  -virtfs local,path=shareddir,security_model=none,mount_tag=shareddir \
  -object can-bus,id=canbus0 \
  -object can-host-socketcan,id=canhost0,if=can0,canbus=canbus0 \
  -device kvaser_pci,canbus=canbus0 \
  -nographic -append "console=ttyS0"
```

Example for qemu-system-arm:

```
qemu-system-arm -cpu arm1176 -m 256 -M versatilepb \
  -kernel kernel-qemu-arm1176-versatilepb \
  -hda rpi-wheezy-overlay \
  -append "console=ttyAMA0 root=/dev/sda2 ro init=/sbin/init-overlay" \
  -nographic \
  -virtfs local,path=shareddir,security_model=none,mount_tag=shareddir \
  -object can-bus,id=canbus0 \
  -object can-host-socketcan,id=canhost0,if=can0,canbus=canbus0 \
  -device kvaser_pci,canbus=canbus0,host=can0 \
```

The CAN interface of the host system has to be configured for proper bitrate and set up. Configuration is not propagated from emulated devices through bus to the physical host device. Example configuration for 1 Mbit/s:

```
ip link set can0 type can bitrate 1000000
ip link set can0 up
```

Virtual (host local only) can interface can be used on the host side instead of physical interface:

```
ip link add dev can0 type vcan
```

The CAN interface on the host side can be used to analyze CAN traffic with "candump" command which is included in "can-utils":

```
candump can0
```

CTU CAN FD support examples

This open-source core provides CAN FD support. CAN FD drames are delivered even to the host systems when SocketCAN interface is found CAN FD capable.

The PCIe board emulation is provided for now (the device identifier is ctucan_pci). The default build defines two CTU CAN FD cores on the board.

Example how to connect the canbus0-bus (virtual wire) to the host Linux system (SocketCAN used) and to both CTU CAN FD cores emulated on the corresponding PCI card expects that host system CAN bus is setup according to the previous SJA1000 section:

```
qemu-system-x86_64 -enable-kvm -kernel /boot/vmlinuz-4.19.52+ \
    -initrd ramdisk.cpio \
    -virtfs local,path=shareddir,security_model=none,mount_tag=shareddir \
    -vga cirrus \
    -append "console=ttyS0" \
    -object can-bus,id=canbus0-bus \
    -object can-host-socketcan,if=can0,canbus=canbus0-bus,id=canbus0-socketcan \
    -device ctucan_pci,canbus0=canbus0-bus,canbus1=canbus0-bus \
    -nographic
```

Setup of CTU CAN FD controller in a guest Linux system:

```
insmod ctucanfd.ko || modprobe ctucanfd
insmod ctucanfd_pci.ko || modprobe ctucanfd_pci

for ifc in /sys/class/net/can* ; do
  if [ -e  $ifc/device/vendor ] ; then
    if ! grep -q 0x1760 $ifc/device/vendor ; then
      continue;
    fi
  else
    continue;
  fi
  if [ -e  $ifc/device/device ] ; then
     if ! grep -q 0xff00 $ifc/device/device ; then
       continue;
     fi
  else
    continue;
  fi
  ifc=$(basename $ifc)
  /bin/ip link set $ifc type can bitrate 1000000 dbitrate 10000000 fd on
  /bin/ip link set $ifc up
done
```

The test can run for example:

```
candump can1
```

in the guest system and next commands in the host system for basic CAN:

```
cangen can0
```

for CAN FD without bitrate switch:

```
cangen can0 -f
```

and with bitrate switch:

```
cangen can0 -b
```

The test can also be run the other way around, generating messages in the guest system and capturing them in the host system. Other combinations are also possible.

Links to other resources

(1) CAN related projects at Czech Technical University, Faculty of Electrical Engineering

(2) Repository with development can-pci branch at Czech Technical University

(3) RTEMS page describing project

(4) RTLWS 2015 article about the project and its use with CANopen emulation

(5) GNU/Linux, CAN and CANopen in Real-time Control Applications Slides from LinuxDays 2017 (include updated RTLWS 2015 content)

(6) Linux SocketCAN utilities

(7) CTU CAN FD project including core VHDL design, Linux driver, test utilities etc.

(8) CTU CAN FD Core Datasheet Documentation

(9) CTU CAN FD Core System Architecture Documentation

(10) CTU CAN FD Driver Documentation

(11) Integration with PCIe interfacing for Intel/Altera Cyclone IV based board

Chip Card Interface Device (CCID)

USB CCID device

The USB CCID device is a USB device implementing the CCID specification, which lets one connect smart card readers that implement the same spec. For more information see the specification:

```
Universal Serial Bus
Device Class: Smart Card
CCID
Specification for
Integrated Circuit(s) Cards Interface Devices
Revision 1.1
April 22rd, 2005
```

Smartcards are used for authentication, single sign on, decryption in public/private schemes and digital signatures. A smartcard reader on the client cannot be used on a guest with simple usb passthrough since it will then not be available on the client, possibly locking the computer when it is "removed". On the other hand this device can let you use the smartcard on both the client and the guest machine. It is also possible to have a completely virtual smart card reader and smart card (i.e. not backed by a physical device) using this device.

Building

The cryptographic functions and access to the physical card is done via the libcacard library, whose development package must be installed prior to building QEMU:

In redhat/fedora:

```
yum install libcacard-devel
```

In ubuntu:

```
apt-get install libcacard-dev
```

Configuring and building:

```
./configure --enable-smartcard && make
```

Using ccid-card-emulated with hardware

Assuming you have a working smartcard on the host with the current user, using libcacard, QEMU acts as another client using ccid-card-emulated:

```
qemu -usb -device usb-ccid -device ccid-card-emulated
```

Using ccid-card-emulated with certificates stored in files

You must create the CA and card certificates. This is a one time process. We use NSS certificates:

```
mkdir fake-smartcard
cd fake-smartcard
certutil -N -d sql:$PWD
certutil -S -d sql:$PWD -s "CN=Fake Smart Card CA" -x -t TC,TC,TC -n fake-smartcard-ca
certutil -S -d sql:$PWD -t ,, -s "CN=John Doe" -n id-cert -c fake-smartcard-ca
certutil -S -d sql:$PWD -t ,, -s "CN=John Doe (signing)" --nsCertType smime -n signing-
↪cert -c fake-smartcard-ca
certutil -S -d sql:$PWD -t ,, -s "CN=John Doe (encryption)" --nsCertType sslClient -n␣
↪encryption-cert -c fake-smartcard-ca
```

Note: you must have exactly three certificates.

You can use the emulated card type with the certificates backend:

```
qemu -usb -device usb-ccid -device ccid-card-emulated,backend=certificates,db=sql:$PWD,
↪cert1=id-cert,cert2=signing-cert,cert3=encryption-cert
```

To use the certificates in the guest, export the CA certificate:

```
certutil -L -r -d sql:$PWD -o fake-smartcard-ca.cer -n fake-smartcard-ca
```

and import it in the guest:

```
certutil -A -d /etc/pki/nssdb -i fake-smartcard-ca.cer -t TC,TC,TC -n fake-smartcard-ca
```

In a Linux guest you can then use the CoolKey PKCS #11 module to access the card:

```
certutil -d /etc/pki/nssdb -L -h all
```

It will prompt you for the PIN (which is the password you assigned to the certificate database early on), and then show you all three certificates together with the manually imported CA cert:

```
Certificate Nickname                          Trust Attributes
fake-smartcard-ca                             CT,C,C
John Doe:CAC ID Certificate                   u,u,u
```

(continues on next page)

(continued from previous page)

```
John Doe:CAC Email Signature Certificate    u,u,u
John Doe:CAC Email Encryption Certificate   u,u,u
```

If this does not happen, CoolKey is not installed or not registered with NSS. Registration can be done from Firefox or the command line:

```
modutil -dbdir /etc/pki/nssdb -add "CAC Module" -libfile /usr/lib64/pkcs11/
↪libcoolkeypk11.so
modutil -dbdir /etc/pki/nssdb -list
```

Using ccid-card-passthru with client side hardware

On the host specify the ccid-card-passthru device with a suitable chardev:

```
qemu -chardev socket,server=on,host=0.0.0.0,port=2001,id=ccid,wait=off \
     -usb -device usb-ccid -device ccid-card-passthru,chardev=ccid
```

On the client run vscclient, built when you built QEMU:

```
vscclient <qemu-host> 2001
```

Using ccid-card-passthru with client side certificates

This case is not particularly useful, but you can use it to debug your setup.

Follow instructions above, except run QEMU and vscclient as follows.

Run qemu as per above, and run vscclient from the "fake-smartcard" directory as follows:

```
qemu -chardev socket,server=on,host=0.0.0.0,port=2001,id=ccid,wait=off \
     -usb -device usb-ccid -device ccid-card-passthru,chardev=ccid
vscclient -e "db=\"sql:$PWD\" use_hw=no soft=(,Test,CAC,,id-cert,signing-cert,encryption-
↪cert)" <qemu-host> 2001
```

Passthrough protocol scenario

This is a typical interchange of messages when using the passthru card device. usb-ccid is a usb device. It defaults to an unattached usb device on startup. usb-ccid expects a chardev and expects the protocol defined in cac_card/vscard_common.h to be passed over that. The usb-ccid device can be in one of three modes:

- detached
- attached with no card
- attached with card

A typical interchange is (the arrow shows who started each exchange, it can be client originated or guest originated):

```
client event        |    vscclient          |    passthru    |    usb-ccid  |  guest␣
↪event
-------------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
↪-------
                    |    VSC_Init             |                 |               |
                    |    VSC_ReaderAdd        |                 |    attach     |
                    |                         |                 |               |  sees new␣
↪usb device.
  card inserted ->  |                         |                 |               |
                    |    VSC_ATR              |   insert        |    insert     |  see new␣
↪card
                    |                         |                 |               |
                    |    VSC_APDU             |   VSC_APDU      |               | <- guest␣
↪sends APDU
client <-> physical |                         |                 |               |
 card APDU exchange |                         |                 |               |
 client response -> |    VSC_APDU             |   VSC_APDU      |               |  receive␣
↪APDU response
                                                      ...
                                    [APDU<->APDU repeats several times]
                                                      ...
   card removed  -> |                         |                 |               |
                    |    VSC_CardRemove       |   remove        |   remove      |   card␣
↪removed
                                                      ...
                                    [(card insert, apdu's, card remove) repeat]
                                                      ...
  kill/quit         |                         |                 |               |
    vscclient       |                         |                 |               |
                    |    VSC_ReaderRemove     |                 |   detach      |
                    |                         |                 |               |   usb␣
↪device removed.
```

libcacard

Both ccid-card-emulated and vscclient use libcacard as the card emulator. libcacard implements a completely virtual CAC (DoD standard for smart cards) compliant card and uses NSS to retrieve certificates and do any encryption. The backend can then be a real reader and card, or certificates stored in files.

Compute Express Link (CXL)

From the view of a single host, CXL is an interconnect standard that targets accelerators and memory devices attached to a CXL host. This description will focus on those aspects visible either to software running on a QEMU emulated host or to the internals of functional emulation. As such, it will skip over many of the electrical and protocol elements that would be more of interest for real hardware and will dominate more general introductions to CXL. It will also completely ignore the fabric management aspects of CXL by considering only a single host and a static configuration.

CXL shares many concepts and much of the infrastructure of PCI Express, with CXL Host Bridges, which have CXL Root Ports which may be directly attached to CXL or PCI End Points. Alternatively there may be CXL Switches with CXL and PCI Endpoints attached below them. In many cases additional control and capabilities are exposed via PCI Express interfaces. This sharing of interfaces and hence emulation code is reflected in how the devices are emulated in QEMU. In most cases the various CXL elements are built upon an equivalent PCIe devices.

CXL devices support the following interfaces:

- Most conventional PCIe interfaces
  - Configuration space access
  - BAR mapped memory accesses used for registers and mailboxes.
  - MSI/MSI-X
  - AER
  - DOE mailboxes
  - IDE
  - Many other PCI express defined interfaces..
- Memory operations
  - Equivalent of accessing DRAM / NVDIMMs. Any access / feature supported by the host for normal memory should also work for CXL attached memory devices.
- Cache operations. The are mostly irrelevant to QEMU emulation as QEMU is not emulating a coherency protocol. Any emulation related to these will be device specific and is out of the scope of this document.

CXL 2.0 Device Types

CXL 2.0 End Points are often categorized into three types.

Type 1: These support coherent caching of host memory. Example might be a crypto accelerators. May also have device private memory accessible via means such as PCI memory reads and writes to BARs.

Type 2: These support coherent caching of host memory and host managed device memory (HDM) for which the coherency protocol is managed by the host. This is a complex topic, so for more information on CXL coherency see the CXL 2.0 specification.

Type 3 Memory devices: These devices act as a means of attaching additional memory (HDM) to a CXL host including both volatile and persistent memory. The CXL topology may support interleaving across a number of Type 3 memory devices using HDM Decoders in the host, host bridge, switch upstream port and endpoints.

Scope of CXL emulation in QEMU

The focus of CXL emulation is CXL revision 2.0 and later. Earlier CXL revisions defined a smaller set of features, leaving much of the control interface as implementation defined or device specific, making generic emulation challenging with host specific firmware being responsible for setup and the Endpoints being presented to operating systems as Root Complex Integrated End Points. CXL rev 2.0 looks a lot more like PCI Express, with fully specified discoverability of the CXL topology.

CXL System components

A CXL system is made up a Host with a number of 'standard components' the control and capabilities of which are discoverable by system software using means described in the CXL 2.0 specification.

CXL Fixed Memory Windows (CFMW)

A CFMW consists of a particular range of Host Physical Address space which is routed to particular CXL Host Bridges. At time of generic software initialization it will have a particularly interleaving configuration and associated Quality of Service Throttling Group (QTG). This information is available to system software, when making decisions about how to configure interleave across available CXL memory devices. It is provide as CFMW Structures (CFMWS) in the CXL Early Discovery Table, an ACPI table.

Note: QTG 0 is the only one currently supported in QEMU.

CXL Host Bridge (CXL HB)

A CXL host bridge is similar to the PCIe equivalent, but with a specification defined register interface called CXL Host Bridge Component Registers (CHBCR). The location of this CHBCR MMIO space is described to system software via a CXL Host Bridge Structure (CHBS) in the CEDT ACPI table. The actual interfaces are identical to those used for other parts of the CXL hierarchy as CXL Component Registers in PCI BARs.

Interfaces provided include:

- Configuration of HDM Decoders to route CXL Memory accesses with a particularly Host Physical Address range to the target port below which the CXL device servicing that address lies. This may be a mapping to a single Root Port (RP) or across a set of target RPs.

CXL Root Ports (CXL RP)

A CXL Root Port serves the same purpose as a PCIe Root Port. There are a number of CXL specific Designated Vendor Specific Extended Capabilities (DVSEC) in PCIe Configuration Space and associated component register access via PCI bars.

CXL Switch

Here we consider a simple CXL switch with only a single virtual hierarchy. Whilst more complex devices exist, their visibility to a particular host is generally the same as for a simple switch design. Hosts often have no awareness of complex rerouting and device pooling, they simply see devices being hot added or hot removed.

A CXL switch has a similar architecture to those in PCIe, with a single upstream port, internal PCI bus and multiple downstream ports.

Both the CXL upstream and downstream ports have CXL specific DVSECs in configuration space, and component registers in PCI BARs. The Upstream Port has the configuration interfaces for the HDM decoders which route incoming memory accesses to the appropriate downstream port.

A CXL switch is created in a similar fashion to PCI switches by creating an upstream port (cxl-upstream) and a number of downstream ports on the internal switch bus (cxl-downstream).

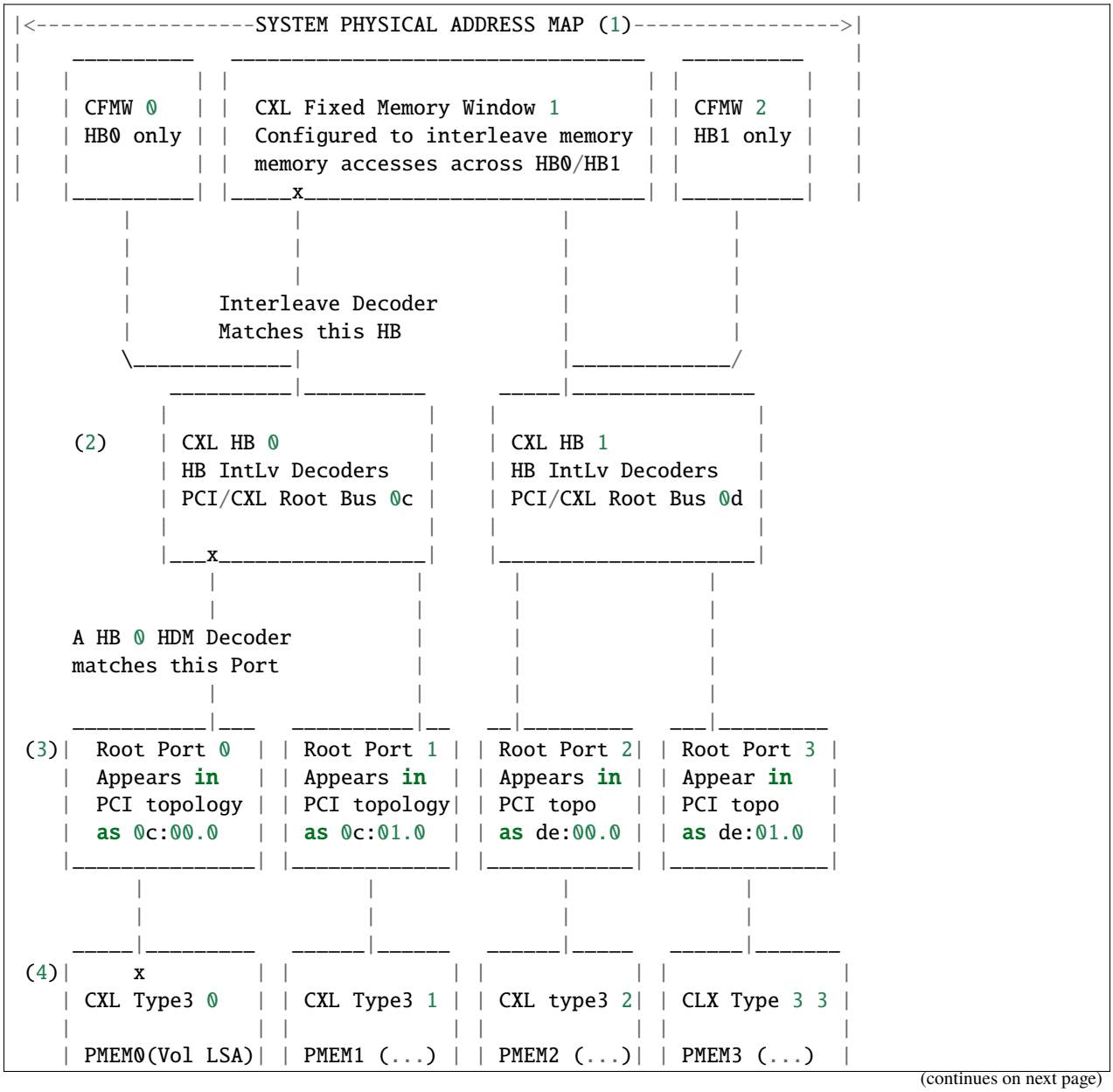
CXL Memory Devices - Type 3

CXL type 3 devices use a PCI class code and are intended to be supported by a generic operating system driver. They have HDM decoders though in these EP devices, the decoder is responsible not for routing but for translation of the incoming host physical address (HPA) into a Device Physical Address (DPA).

CXL Memory Interleave

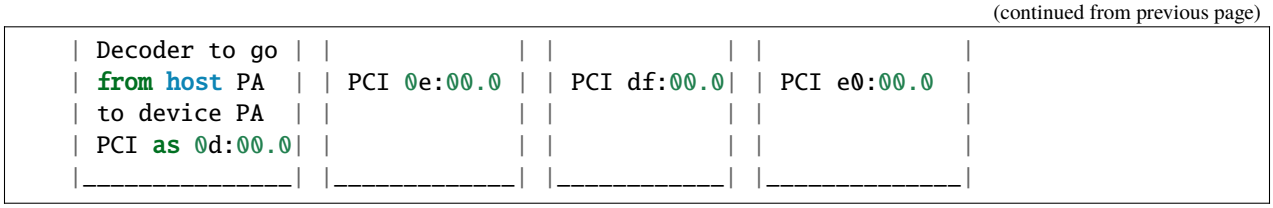
To understand the interaction of different CXL hardware components which are emulated in QEMU, let us consider a memory read in a fully configured CXL topology. Note that system software is responsible for configuration of all components with the exception of the CFMWs. System software is responsible for allocating appropriate ranges from within the CFMWs and exposing those via normal memory configurations as would be done for system RAM.

Example system topology. x marks the match in each decoder level:



(continues on next page)

(continued from previous page)



Notes:

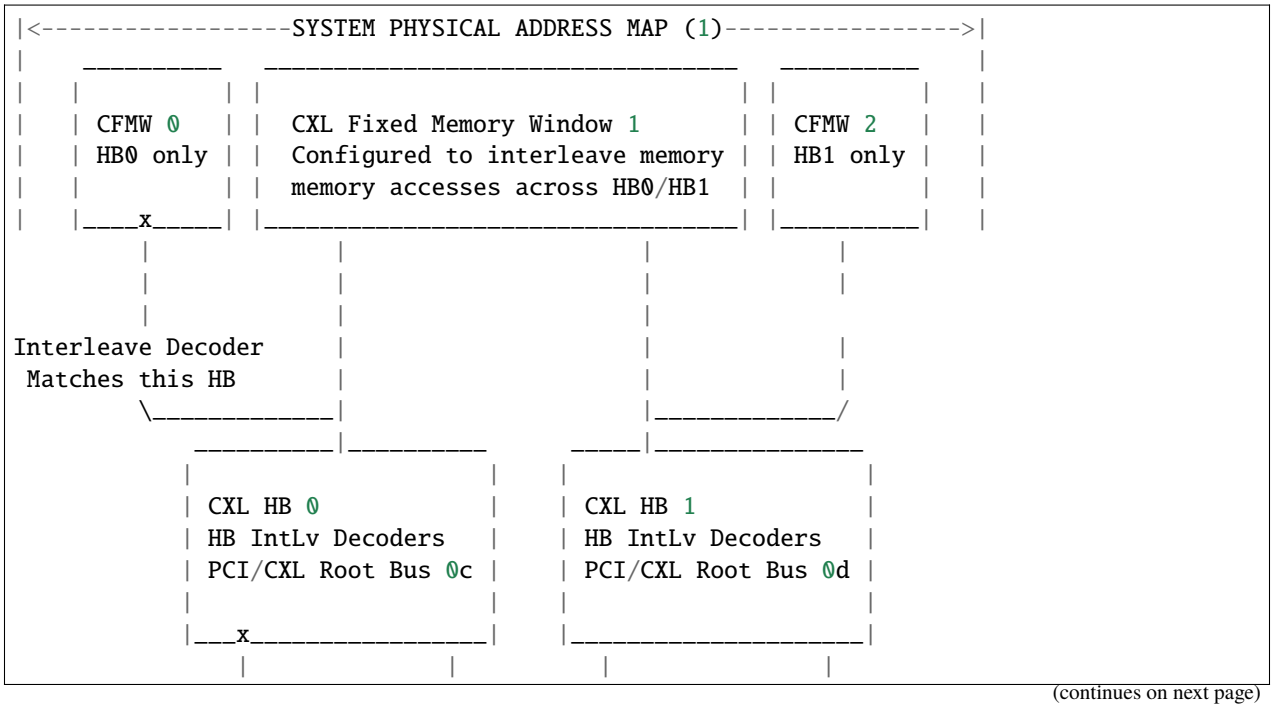
(1) **3 CXL Fixed Memory Windows (CFMW)** corresponding to different ranges of the system physical address map. Each CFMW has particular interleave setup across the CXL Host Bridges (HB) CFMW0 provides uninterleaved access to HB0, CFMW2 provides uninterleaved access to HB1. CFMW1 provides interleaved memory access across HB0 and HB1.

(2) **Two CXL Host Bridges**. Each of these has 2 CXL Root Ports and programmable HDM decoders to route memory accesses either to a single port or interleave them across multiple ports. A complex configuration here, might be to use the following HDM decoders in HB0. HDM0 routes CFMW0 requests to RP0 and hence part of CXL Type3 0. HDM1 routes CFMW0 requests from a different region of the CFMW0 PA range to RP2 and hence part of CXL Type 3 1. HDM2 routes yet another PA range from within CFMW0 to be interleaved across RP0 and RP1, providing 2 way interleave of part of the memory provided by CXL Type3 0 and CXL Type 3 1. HDM3 routes those interleaved accesses from CFMW1 that target HB0 to RP 0 and another part of the memory of CXL Type 3 0 (as part of a 2 way interleave at the system level across for example CXL Type3 0 and CXL Type3 2. HDM4 is used to enable system wide 4 way interleave across all the present CXL type3 devices, by interleaving those (interleaved) requests that HB0 receives from from CFMW1 across RP 0 and RP 1 and hence to yet more regions of the memory of the attached Type3 devices. Note this is a representative subset of the full range of possible HDM decoder configurations in this topology.

(3) **Four CXL Root Ports.** In this case the CXL Type 3 devices are directly attached to these ports.

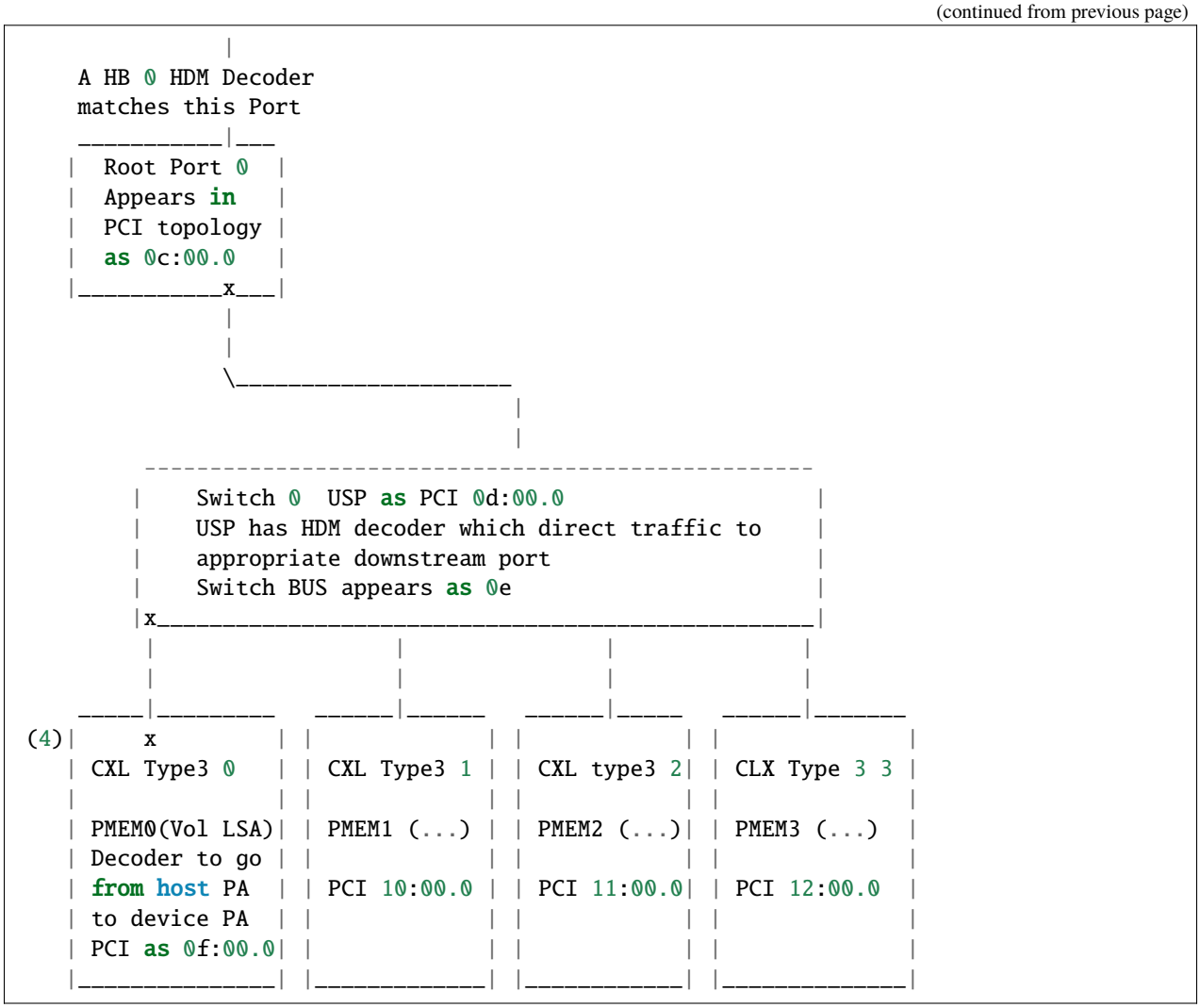
(4) **Four CXL Type3 memory expansion devices.** These will each have HDM decoders, but in this case rather than performing interleave they will take the Host Physical Addresses of accesses and map them to their own local Device Physical Address Space (DPA).

Example topology involving a switch:



(continues on next page)

(continued from previous page)



Example command lines

A very simple setup with just one directly attached CXL Type 3 Persistent Memory device:

```
qemu-system-x86_64 -M q35,cxl=on -m 4G,maxmem=8G,slots=8 -smp 4 \
...
-object memory-backend-file,id=cxl-mem1,share=on,mem-path=/tmp/cxltest.raw,size=256M \
-object memory-backend-file,id=cxl-lsa1,share=on,mem-path=/tmp/lsa.raw,size=256M \
-device pxb-cxl,bus_nr=12,bus=pcie.0,id=cxl.1 \
-device cxl-rp,port=0,bus=cxl.1,id=root_port13,chassis=0,slot=2 \
-device cxl-type3,bus=root_port13,persistent-memdev=cxl-mem1,lsa=cxl-lsa1,id=cxl-pmem0 \
-M cxl-fmw.0.targets.0=cxl.1,cxl-fmw.0.size=4G
```

A very simple setup with just one directly attached CXL Type 3 Volatile Memory device:

```
qemu-system-x86_64 -M q35,cxl=on -m 4G,maxmem=8G,slots=8 -smp 4 \
...
-object memory-backend-ram,id=vmem0,share=on,size=256M \
```

(continues on next page)

(continued from previous page)

```
-device pxb-cxl,bus_nr=12,bus=pcie.0,id=cxl.1 \
-device cxl-rp,port=0,bus=cxl.1,id=root_port13,chassis=0,slot=2 \
-device cxl-type3,bus=root_port13,volatile-memdev=vmem0,id=cxl-vmem0 \
-M cxl-fmw.0.targets.0=cxl.1,cxl-fmw.0.size=4G
```

The same volatile setup may optionally include an LSA region:

```
qemu-system-x86_64 -M q35,cxl=on -m 4G,maxmem=8G,slots=8 -smp 4 \
...
-object memory-backend-ram,id=vmem0,share=on,size=256M \
-object memory-backend-file,id=cxl-lsa0,share=on,mem-path=/tmp/lsa.raw,size=256M \
-device pxb-cxl,bus_nr=12,bus=pcie.0,id=cxl.1 \
-device cxl-rp,port=0,bus=cxl.1,id=root_port13,chassis=0,slot=2 \
-device cxl-type3,bus=root_port13,volatile-memdev=vmem0,lsa=cxl-lsa0,id=cxl-vmem0 \
-M cxl-fmw.0.targets.0=cxl.1,cxl-fmw.0.size=4G
```

A setup suitable for 4 way interleave. Only one fixed window provided, to enable 2 way interleave across 2 CXL host bridges. Each host bridge has 2 CXL Root Ports, with the CXL Type3 device directly attached (no switches).:

```
qemu-system-x86_64 -M q35,cxl=on -m 4G,maxmem=8G,slots=8 -smp 4 \
...
-object memory-backend-file,id=cxl-mem1,share=on,mem-path=/tmp/cxltest.raw,size=256M \
-object memory-backend-file,id=cxl-mem2,share=on,mem-path=/tmp/cxltest2.raw,size=256M \
-object memory-backend-file,id=cxl-mem3,share=on,mem-path=/tmp/cxltest3.raw,size=256M \
-object memory-backend-file,id=cxl-mem4,share=on,mem-path=/tmp/cxltest4.raw,size=256M \
-object memory-backend-file,id=cxl-lsa1,share=on,mem-path=/tmp/lsa.raw,size=256M \
-object memory-backend-file,id=cxl-lsa2,share=on,mem-path=/tmp/lsa2.raw,size=256M \
-object memory-backend-file,id=cxl-lsa3,share=on,mem-path=/tmp/lsa3.raw,size=256M \
-object memory-backend-file,id=cxl-lsa4,share=on,mem-path=/tmp/lsa4.raw,size=256M \
-device pxb-cxl,bus_nr=12,bus=pcie.0,id=cxl.1 \
-device pxb-cxl,bus_nr=222,bus=pcie.0,id=cxl.2 \
-device cxl-rp,port=0,bus=cxl.1,id=root_port13,chassis=0,slot=2 \
-device cxl-type3,bus=root_port13,persistent-memdev=cxl-mem1,lsa=cxl-lsa1,id=cxl-pmem0 \
-device cxl-rp,port=1,bus=cxl.1,id=root_port14,chassis=0,slot=3 \
-device cxl-type3,bus=root_port14,persistent-memdev=cxl-mem2,lsa=cxl-lsa2,id=cxl-pmem1 \
-device cxl-rp,port=0,bus=cxl.2,id=root_port15,chassis=0,slot=5 \
-device cxl-type3,bus=root_port15,persistent-memdev=cxl-mem3,lsa=cxl-lsa3,id=cxl-pmem2 \
-device cxl-rp,port=1,bus=cxl.2,id=root_port16,chassis=0,slot=6 \
-device cxl-type3,bus=root_port16,persistent-memdev=cxl-mem4,lsa=cxl-lsa4,id=cxl-pmem3 \
-M cxl-fmw.0.targets.0=cxl.1,cxl-fmw.0.targets.1=cxl.2,cxl-fmw.0.size=4G,cxl-fmw.0.
↪interleave-granularity=8k
```

An example of 4 devices below a switch suitable for 1, 2 or 4 way interleave:

```
qemu-system-x86_64 -M q35,cxl=on -m 4G,maxmem=8G,slots=8 -smp 4 \
...
-object memory-backend-file,id=cxl-mem0,share=on,mem-path=/tmp/cxltest.raw,size=256M \
-object memory-backend-file,id=cxl-mem1,share=on,mem-path=/tmp/cxltest1.raw,size=256M \
-object memory-backend-file,id=cxl-mem2,share=on,mem-path=/tmp/cxltest2.raw,size=256M \
-object memory-backend-file,id=cxl-mem3,share=on,mem-path=/tmp/cxltest3.raw,size=256M \
-object memory-backend-file,id=cxl-lsa0,share=on,mem-path=/tmp/lsa0.raw,size=256M \
-object memory-backend-file,id=cxl-lsa1,share=on,mem-path=/tmp/lsa1.raw,size=256M \
-object memory-backend-file,id=cxl-lsa2,share=on,mem-path=/tmp/lsa2.raw,size=256M \
```

(continues on next page)

(continued from previous page)

```
-object memory-backend-file,id=cxl-lsa3,share=on,mem-path=/tmp/lsa3.raw,size=256M \
-device pxb-cxl,bus_nr=12,bus=pcie.0,id=cxl.1 \
-device cxl-rp,port=0,bus=cxl.1,id=root_port0,chassis=0,slot=0 \
-device cxl-rp,port=1,bus=cxl.1,id=root_port1,chassis=0,slot=1 \
-device cxl-upstream,bus=root_port0,id=us0 \
-device cxl-downstream,port=0,bus=us0,id=swport0,chassis=0,slot=4 \
-device cxl-type3,bus=swport0,persistent-memdev=cxl-mem0,lsa=cxl-lsa0,id=cxl-pmem0 \
-device cxl-downstream,port=1,bus=us0,id=swport1,chassis=0,slot=5 \
-device cxl-type3,bus=swport1,persistent-memdev=cxl-mem1,lsa=cxl-lsa1,id=cxl-pmem1 \
-device cxl-downstream,port=2,bus=us0,id=swport2,chassis=0,slot=6 \
-device cxl-type3,bus=swport2,persistent-memdev=cxl-mem2,lsa=cxl-lsa2,id=cxl-pmem2 \
-device cxl-downstream,port=3,bus=us0,id=swport3,chassis=0,slot=7 \
-device cxl-type3,bus=swport3,persistent-memdev=cxl-mem3,lsa=cxl-lsa3,id=cxl-pmem3 \
-M cxl-fmw.0.targets.0=cxl.1,cxl-fmw.0.size=4G,cxl-fmw.0.interleave-granularity=4k
```

Deprecations

The Type 3 device [memdev] attribute has been deprecated in favor of the [persistent-memdev] attributes. [memdev] will default to a persistent memory device for backward compatibility and is incapable of being used in combination with [persistent-memdev].

Kernel Configuration Options

In Linux 5.18 the following options are necessary to make use of OS management of CXL memory devices as described here.

- CONFIG_CXL_BUS
- CONFIG_CXL_PCI
- CONFIG_CXL_ACPI
- CONFIG_CXL_PMEM
- CONFIG_CXL_MEM
- CONFIG_CXL_PORT
- CONFIG_CXL_REGION

References

- Consortium website for specifications etc: http://www.computeexpresslink.org
- Compute Express Link (CXL) Specification, Revision 3.1, August 2023

Inter-VM Shared Memory device

On Linux hosts, a shared memory device is available. The basic syntax is:

```
qemu-system-x86_64 -device ivshmem-plain,memdev=hostmem
```

where hostmem names a host memory backend. For a POSIX shared memory backend, use something like

```
-object memory-backend-file,size=1M,share,mem-path=/dev/shm/ivshmem,id=hostmem
```

If desired, interrupts can be sent between guest VMs accessing the same shared memory region. Interrupt support requires using a shared memory server and using a chardev socket to connect to it. The code for the shared memory server is qemu.git/contrib/ivshmem-server. An example syntax when using the shared memory server is:

```
# First start the ivshmem server once and for all
ivshmem-server -p pidfile -S path -m shm-name -l shm-size -n vectors

# Then start your qemu instances with matching arguments
qemu-system-x86_64 -device ivshmem-doorbell,vectors=vectors,chardev=id
                   -chardev socket,path=path,id=id
```

When using the server, the guest will be assigned a VM ID (>=0) that allows guests using the same server to communicate via interrupts. Guests can read their VM ID from a device register (see *Device Specification for Inter-VM shared memory device*).

Migration with ivshmem

With device property `master=on`, the guest will copy the shared memory on migration to the destination host. With `master=off`, the guest will not be able to migrate with the device attached. In the latter case, the device should be detached and then reattached after migration using the PCI hotplug support.

At most one of the devices sharing the same memory can be master. The master must complete migration before you plug back the other devices.

ivshmem and hugepages

Instead of specifying the <shm size> using POSIX shm, you may specify a memory backend that has hugepage support:

```
qemu-system-x86_64 -object memory-backend-file,size=1G,mem-path=/dev/hugepages/
↪my-shmem-file,share,id=mb1
                   -device ivshmem-plain,memdev=mb1
```

ivshmem-server also supports hugepages mount points with the `-m` memory path argument.

Sparc32 keyboard

SUN Type 4, 5 and 5c keyboards have dip switches to choose the language layout of the keyboard. Solaris makes an ioctl to query the value of the dipswitches and uses that value to select keyboard layout. Also the SUN bios like the one in the file ss5.bin uses this value to support at least some keyboard layouts. However, the OpenBIOS provided with qemu is hardcoded to always use an US keyboard layout.

With the escc.chnA-sunkbd-layout driver property it is possible to select keyboard layout. Example:

-global escc.chnA-sunkbd-layout=de

Depending on type of keyboard, the keyboard can have 6 or 5 dip-switches to select keyboard layout, giving up to 64 different layouts. Not all combinations are supported by Solaris and even less by Sun OpenBoot BIOS.

The dip switch settings can be given as hexadecimal number, decimal number or in some cases as a language string. Examples:

-global escc.chnA-sunkbd-layout=0x2b

-global escc.chnA-sunkbd-layout=43

-global escc.chnA-sunkbd-layout=sv

The above 3 examples all select a swedish keyboard layout. Table 3-15 at https://docs.oracle.com/cd/E19683-01/806-6642/new-43/index.html explains which keytable file is used for different dip switch settings. The information in that table can be summarized in this table:

Table 3: Language selection values for escc.chnA-sunkbd-layout

Hexadecimal value	Decimal value	Language code
0x21	33	en-us
0x23	35	fr
0x24	36	da
0x25	37	de
0x26	38	it
0x27	39	nl
0x28	40	no
0x29	41	pt
0x2a	42	es
0x2b	43	sv
0x2c	44	fr-ch
0x2d	45	de-ch
0x2e	46	en-gb
0x2f	47	ko
0x30	48	tw
0x31	49	ja
0x32	50	fr-ca
0x33	51	hu
0x34	52	pl
0x35	53	cz
0x36	54	ru
0x37	55	lv
0x38	56	tr
0x39	57	gr
0x3a	58	ar
0x3b	59	lt
0x3c	60	nl-be
0x3c	60	be

Not all dip switch values have a corresponding language code and both “be” and “nl-be” correspond to the same dip switch value. By default, if no value is given to escc.chnA-sunkbd-layout 0x21 (en-us) will be used.

Network emulation

QEMU can simulate several network cards (e.g. PCI or ISA cards on the PC target) and can connect them to a network backend on the host or an emulated hub. The various host network backends can either be used to connect the NIC of the guest to a real network (e.g. by using a TAP devices or the non-privileged user mode network stack), or to other guest instances running in another QEMU process (e.g. by using the socket host network backend).

Using TAP network interfaces

This is the standard way to connect QEMU to a real network. QEMU adds a virtual network device on your host (called `tapN`), and you can then configure it as if it was a real ethernet card.

Linux host

As an example, you can download the `linux-test-xxx.tar.gz` archive and copy the script `qemu-ifup` in `/etc` and configure properly `sudo` so that the command `ifconfig` contained in `qemu-ifup` can be executed as root. You must verify that your host kernel supports the TAP network interfaces: the device `/dev/net/tun` must be present.

See *Invocation* to have examples of command lines using the TAP network interfaces.

Windows host

There is a virtual ethernet driver for Windows 2000/XP systems, called TAP-Win32. But it is not included in standard QEMU for Windows, so you will need to get it separately. It is part of OpenVPN package, so download OpenVPN from : https://openvpn.net/.

Using the user mode network stack

By using the option `-net user` (default configuration if no `-net` option is specified), QEMU uses a completely user mode network stack (you don't need root privilege to use the virtual network). The virtual network configuration is the following:

```
guest (10.0.2.15)  <------>  Firewall/DHCP server <-----> Internet
                        |          (10.0.2.2)
                        |
                        ---->  DNS server (10.0.2.3)
                        |
                        ---->  SMB server (10.0.2.4)
```

The QEMU VM behaves as if it was behind a firewall which blocks all incoming connections. You can use a DHCP client to automatically configure the network in the QEMU VM. The DHCP server assign addresses to the hosts starting from 10.0.2.15.

In order to check that the user mode network is working, you can ping the address 10.0.2.2 and verify that you got an address in the range 10.0.2.x from the QEMU virtual DHCP server.

Note that ICMP traffic in general does not work with user mode networking. `ping`, aka. ICMP echo, to the local router (10.0.2.2) shall work, however. If you're using QEMU on Linux >= 3.0, it can use unprivileged ICMP ping sockets to allow `ping` to the Internet. The host admin has to set the ping_group_range in order to grant access to those sockets. To allow ping for GID 100 (usually users group):

```
echo 100 100 > /proc/sys/net/ipv4/ping_group_range
```

When using the built-in TFTP server, the router is also the TFTP server.

When using the `'-netdev user,hostfwd=...'` option, TCP or UDP connections can be redirected from the host to the guest. It allows for example to redirect X11, telnet or SSH connections.

Hubs

QEMU can simulate several hubs. A hub can be thought of as a virtual connection between several network devices. These devices can be for example QEMU virtual ethernet cards or virtual Host ethernet devices (TAP devices). You can connect guest NICs or host network backends to such a hub using the `-netdev hubport` or `-nic hubport` options. The legacy `-net` option also connects the given device to the emulated hub with ID 0 (i.e. the default hub) unless you specify a netdev with `-net nic,netdev=xxx` here.

Connecting emulated networks between QEMU instances

Using the `-netdev socket` (or `-nic socket` or `-net socket`) option, it is possible to create emulated networks that span several QEMU instances. See the description of the `-netdev socket` option in *Invocation* to have a basic example.

NVMe Emulation

QEMU provides NVMe emulation through the `nvme`, `nvme-ns` and `nvme-subsys` devices.

See the following sections for specific information on

- *Adding NVMe Devices*, *additional namespaces* and *NVM subsystems*.
- Configuration of *Optional Features* such as *Controller Memory Buffer*, *Simple Copy*, *Zoned Namespaces*, *metadata* and *End-to-End Data Protection*,

Adding NVMe Devices

Controller Emulation

The QEMU emulated NVMe controller implements version 1.4 of the NVM Express specification. All mandatory features are implement with a couple of exceptions and limitations:

- Accounting numbers in the SMART/Health log page are reset when the device is power cycled.
- Interrupt Coalescing is not supported and is disabled by default.

The simplest way to attach an NVMe controller on the QEMU PCI bus is to add the following parameters:

```
-drive file=nvm.img,if=none,id=nvm
-device nvme,serial=deadbeef,drive=nvm
```

There are a number of optional general parameters for the `nvme` device. Some are mentioned here, but see `-device nvme,help` to list all possible parameters.

`max_ioqpairs=UINT32` (default: `64`)
: Set the maximum number of allowed I/O queue pairs. This replaces the deprecated `num_queues` parameter.

msix_qsize=UINT16 (default: 65)
The number of MSI-X vectors that the device should support.

mdts=UINT8 (default: 7)
Set the Maximum Data Transfer Size of the device.

use-intel-id (default: off)
Since QEMU 5.2, the device uses a QEMU allocated "Red Hat" PCI Device and Vendor ID. Set this to on to revert to the unallocated Intel ID previously used.

Additional Namespaces

In the simplest possible invocation sketched above, the device only support a single namespace with the namespace identifier 1. To support multiple namespaces and additional features, the nvme-ns device must be used.

```
-device nvme,id=nvme-ctrl-0,serial=deadbeef
-drive file=nvm-1.img,if=none,id=nvm-1
-device nvme-ns,drive=nvm-1
-drive file=nvm-2.img,if=none,id=nvm-2
-device nvme-ns,drive=nvm-2
```

The namespaces defined by the nvme-ns device will attach to the most recently defined nvme-bus that is created by the nvme device. Namespace identifiers are allocated automatically, starting from 1.

There are a number of parameters available:

nsid (default: 0)
Explicitly set the namespace identifier.

uuid (default: *autogenerated*)
Set the UUID of the namespace. This will be reported as a "Namespace UUID" descriptor in the Namespace Identification Descriptor List.

nguid
Set the NGUID of the namespace. This will be reported as a "Namespace Globally Unique Identifier" descriptor in the Namespace Identification Descriptor List. It is specified as a string of hexadecimal digits containing exactly 16 bytes or "auto" for a random value. An optional '-' separator could be used to group bytes. If not specified the NGUID will remain all zeros.

eui64
Set the EUI-64 of the namespace. This will be reported as a "IEEE Extended Unique Identifier" descriptor in the Namespace Identification Descriptor List. Since machine type 6.1 a non-zero default value is used if the parameter is not provided. For earlier machine types the field defaults to 0.

bus
If there are more nvme devices defined, this parameter may be used to attach the namespace to a specific nvme device (identified by an id parameter on the controller device).

NVM Subsystems

Additional features becomes available if the controller device (`nvme`) is linked to an NVM Subsystem device (`nvme-subsys`).

The NVM Subsystem emulation allows features such as shared namespaces and multipath I/O.

```
-device nvme-subsys,id=nvme-subsys-0,nqn=subsys0
-device nvme,serial=deadbeef,subsys=nvme-subsys-0
-device nvme,serial=deadbeef,subsys=nvme-subsys-0
```

This will create an NVM subsystem with two controllers. Having controllers linked to an `nvme-subsys` device allows additional `nvme-ns` parameters:

`shared` (default: `on` since 6.2)
: Specifies that the namespace will be attached to all controllers in the subsystem. If set to `off`, the namespace will remain a private namespace and may only be attached to a single controller at a time. Shared namespaces are always automatically attached to all controllers (also when controllers are hotplugged).

`detached` (default: `off`)
: If set to `on`, the namespace will be be available in the subsystem, but not attached to any controllers initially. A shared namespace with this set to `on` will never be automatically attached to controllers.

Thus, adding

```
-drive file=nvm-1.img,if=none,id=nvm-1
-device nvme-ns,drive=nvm-1,nsid=1
-drive file=nvm-2.img,if=none,id=nvm-2
-device nvme-ns,drive=nvm-2,nsid=3,shared=off,detached=on
```

will cause NSID 1 will be a shared namespace that is initially attached to both controllers. NSID 3 will be a private namespace due to `shared=off` and only attachable to a single controller at a time. Additionally it will not be attached to any controller initially (due to `detached=on`) or to hotplugged controllers.

Optional Features

Controller Memory Buffer

`nvme` device parameters related to the Controller Memory Buffer support:

`cmb_size_mb=UINT32` (default: `0`)
: This adds a Controller Memory Buffer of the given size at offset zero in BAR 2.

`legacy-cmb` (default: `off`)
: By default, the device uses the "v1.4 scheme" for the Controller Memory Buffer support (i.e, the CMB is initially disabled and must be explicitly enabled by the host). Set this to `on` to behave as a v1.3 device wrt. the CMB.

Simple Copy

The device includes support for TP 4065 ("Simple Copy Command"). A number of additional `nvme-ns` device parameters may be used to control the Copy command limits:

`mssrl=UINT16` (default: `128`)
Set the Maximum Single Source Range Length (`MSSRL`). This is the maximum number of logical blocks that may be specified in each source range.

`mcl=UINT32` (default: `128`)
Set the Maximum Copy Length (`MCL`). This is the maximum number of logical blocks that may be specified in a Copy command (the total for all source ranges).

`msrc=UINT8` (default: `127`)
Set the Maximum Source Range Count (`MSRC`). This is the maximum number of source ranges that may be used in a Copy command. This is a 0's based value.

Zoned Namespaces

A namespaces may be "Zoned" as defined by TP 4053 ("Zoned Namespaces"). Set `zoned=on` on an `nvme-ns` device to configure it as a zoned namespace.

The namespace may be configured with additional parameters

`zoned.zone_size=SIZE` (default: `128MiB`)
Define the zone size (`ZSZE`).

`zoned.zone_capacity=SIZE` (default: `0`)
Define the zone capacity (`ZCAP`). If left at the default (`0`), the zone capacity will equal the zone size.

`zoned.descr_ext_size=UINT32` (default: `0`)
Set the Zone Descriptor Extension Size (`ZDES`). Must be a multiple of 64 bytes.

`zoned.cross_read=BOOL` (default: `off`)
Set to `on` to allow reads to cross zone boundaries.

`zoned.max_active=UINT32` (default: `0`)
Set the maximum number of active resources (`MAR`). The default (`0`) allows all zones to be active.

`zoned.max_open=UINT32` (default: `0`)
Set the maximum number of open resources (`MOR`). The default (`0`) allows all zones to be open. If `zoned.max_active` is specified, this value must be less than or equal to that.

`zoned.zasl=UINT8` (default: `0`)
Set the maximum data transfer size for the Zone Append command. Like `mdts`, the value is specified as a power of two (2^n) and is in units of the minimum memory page size (CAP.MPSMIN). The default value (`0`) has this property inherit the `mdts` value.

Flexible Data Placement

The device may be configured to support TP4146 ("Flexible Data Placement") by configuring it (`fdp=on`) on the subsystem:

```
-device nvme-subsys,id=nvme-subsys-0,nqn=subsys0,fdp=on,fdp.nruh=16
```

The subsystem emulates a single Endurance Group, on which Flexible Data Placement will be supported. Also note that the device emulation deviates slightly from the specification, by always enabling the "FDP Mode" feature on the controller if the subsystems is configured for Flexible Data Placement.

Enabling Flexible Data Placement on the subsyste enables the following parameters:

`fdp.nrg` (default: `1`)
Set the number of Reclaim Groups.

`fdp.nruh` (default: `0`)
Set the number of Reclaim Unit Handles. This is a mandatory parameter and must be non-zero.

`fdp.runs` (default: `96M`)
Set the Reclaim Unit Nominal Size. Defaults to 96 MiB.

Namespaces within this subsystem may requests Reclaim Unit Handles:

```
-device nvme-ns,drive=nvm-1,fdp.ruhs=RUHLIST
```

The `RUHLIST` is a semicolon separated list (i.e. `0;1;2;3`) and may include ranges (i.e. `0;8-15`). If no reclaim unit handle list is specified, the controller will assign the controller-specified reclaim unit handle to placement handle identifier 0.

Metadata

The virtual namespace device supports LBA metadata in the form separate metadata (`MPTR`-based) and extended LBAs.

`ms=UINT16` (default: `0`)
Defines the number of metadata bytes per LBA.

`mset=UINT8` (default: `0`)
Set to `1` to enable extended LBAs.

End-to-End Data Protection

The virtual namespace device supports DIF- and DIX-based protection information (depending on `mset`).

`pi=UINT8` (default: `0`)
Enable protection information of the specified type (type `1`, `2` or `3`).

`pil=UINT8` (default: `0`)
Controls the location of the protection information within the metadata. Set to `1` to transfer protection information as the first bytes of metadata. Otherwise, the protection information is transferred as the last bytes of metadata.

`pif=UINT8` (default: `0`)
By default, the namespace device uses 16 bit guard protection information format (`pif=0`). Set to `2` to enable 64 bit guard protection information format. This requires at least 16 bytes of metadata. Note that `pif=1` (32 bit guards) is currently not supported.

Virtualization Enhancements and SR-IOV (Experimental Support)

The `nvme` device supports Single Root I/O Virtualization and Sharing along with Virtualization Enhancements. The controller has to be linked to an NVM Subsystem device (`nvme-subsys`) for use with SR-IOV.

A number of parameters are present (**please note, that they may be subject to change**):

`sriov_max_vfs` (default: `0`)
: Indicates the maximum number of PCIe virtual functions supported by the controller. Specifying a non-zero value enables reporting of both SR-IOV and ARI (Alternative Routing-ID Interpretation) capabilities by the NVMe device. Virtual function controllers will not report SR-IOV.

`sriov_vq_flexible`
: Indicates the total number of flexible queue resources assignable to all the secondary controllers. Implicitly sets the number of primary controller's private resources to `(max_ioqpairs - sriov_vq_flexible)`.

`sriov_vi_flexible`
: Indicates the total number of flexible interrupt resources assignable to all the secondary controllers. Implicitly sets the number of primary controller's private resources to `(msix_qsize - sriov_vi_flexible)`.

`sriov_max_vi_per_vf` (default: `0`)
: Indicates the maximum number of virtual interrupt resources assignable to a secondary controller. The default `0` resolves to `(sriov_vi_flexible / sriov_max_vfs)`

`sriov_max_vq_per_vf` (default: `0`)
: Indicates the maximum number of virtual queue resources assignable to a secondary controller. The default `0` resolves to `(sriov_vq_flexible / sriov_max_vfs)`

The simplest possible invocation enables the capability to set up one VF controller and assign an admin queue, an IO queue, and a MSI-X interrupt.

```
-device nvme-subsys,id=subsys0
-device nvme,serial=deadbeef,subsys=subsys0,sriov_max_vfs=1,
 sriov_vq_flexible=2,sriov_vi_flexible=1
```

The minimum steps required to configure a functional NVMe secondary controller are:

- unbind flexible resources from the primary controller

```
 nvme virt-mgmt /dev/nvme0 -c 0 -r 1 -a 1 -n 0
 nvme virt-mgmt /dev/nvme0 -c 0 -r 0 -a 1 -n 0

* perform a Function Level Reset on the primary controller to actually
  release the resources
```

```
 echo 1 > /sys/bus/pci/devices/0000:01:00.0/reset

* enable VF
```

```
 echo 1 > /sys/bus/pci/devices/0000:01:00.0/sriov_numvfs

* assign the flexible resources to the VF and set it ONLINE
```

```
 nvme virt-mgmt /dev/nvme0 -c 1 -r 1 -a 8 -n 1
 nvme virt-mgmt /dev/nvme0 -c 1 -r 0 -a 8 -n 2
 nvme virt-mgmt /dev/nvme0 -c 1 -r 0 -a 9 -n 0
```

(continues on next page)

(continued from previous page)

```
* bind the NVMe driver to the VF
```

```
echo 0000:01:00.1 > /sys/bus/pci/drivers/nvme/bind
```

USB emulation

QEMU can emulate a PCI UHCI, OHCI, EHCI or XHCI USB controller. You can plug virtual USB devices or real host USB devices (only works with certain host operating systems). QEMU will automatically create and connect virtual USB hubs as necessary to connect multiple USB devices.

USB controllers

XHCI controller support

QEMU has XHCI host adapter support. The XHCI hardware design is much more virtualization-friendly when compared to EHCI and UHCI, thus XHCI emulation uses less resources (especially CPU). So if your guest supports XHCI (which should be the case for any operating system released around 2010 or later) we recommend using it:

qemu -device qemu-xhci

XHCI supports USB 1.1, USB 2.0 and USB 3.0 devices, so this is the only controller you need. With only a single USB controller (and therefore only a single USB bus) present in the system there is no need to use the bus= parameter when adding USB devices.

EHCI controller support

The QEMU EHCI Adapter supports USB 2.0 devices. It can be used either standalone or with companion controllers (UHCI, OHCI) for USB 1.1 devices. The companion controller setup is more convenient to use because it provides a single USB bus supporting both USB 2.0 and USB 1.1 devices. See next section for details.

When running EHCI in standalone mode you can add UHCI or OHCI controllers for USB 1.1 devices too. Each controller creates its own bus though, so there are two completely separate USB buses: One USB 1.1 bus driven by the UHCI controller and one USB 2.0 bus driven by the EHCI controller. Devices must be attached to the correct controller manually.

The easiest way to add a UHCI controller to a `pc` machine is the `-usb` switch. QEMU will create the UHCI controller as function of the PIIX3 chipset. The USB 1.1 bus will carry the name `usb-bus.0`.

You can use the standard `-device` switch to add a EHCI controller to your virtual machine. It is strongly recommended to specify an ID for the controller so the USB 2.0 bus gets an individual name, for example `-device usb-ehci,id=ehci`. This will give you a USB 2.0 bus named `ehci.0`.

When adding USB devices using the `-device` switch you can specify the bus they should be attached to. Here is a complete example:

```
qemu-system-x86_64 -M pc ${otheroptions}                        \
    -drive if=none,id=usbstick,format=raw,file=/path/to/image   \
    -usb                                                        \
    -device usb-ehci,id=ehci                                    \
    -device usb-tablet,bus=usb-bus.0                            \
    -device usb-storage,bus=ehci.0,drive=usbstick
```

This attaches a USB tablet to the UHCI adapter and a USB mass storage device to the EHCI adapter.

Companion controller support

The UHCI and OHCI controllers can attach to a USB bus created by EHCI as companion controllers. This is done by specifying the `masterbus` and `firstport` properties. `masterbus` specifies the bus name the controller should attach to. `firstport` specifies the first port the controller should attach to, which is needed as usually one EHCI controller with six ports has three UHCI companion controllers with two ports each.

There is a config file in docs which will do all this for you, which you can use like this:

```
qemu-system-x86_64 -readconfig docs/config/ich9-ehci-uhci.cfg
```

Then use `bus=ehci.0` to assign your USB devices to that bus.

Using the `-usb` switch for `q35` machines will create a similar USB controller configuration.

Connecting USB devices

USB devices can be connected with the `-device usb-...` command line option or the `device_add` monitor command. Available devices are:

usb-mouse
Virtual Mouse. This will override the PS/2 mouse emulation when activated.

usb-tablet
Pointer device that uses absolute coordinates (like a touchscreen). This means QEMU is able to report the mouse position without having to grab the mouse. Also overrides the PS/2 mouse emulation when activated.

usb-storage,drive=drive_id
Mass storage device backed by drive_id (see the *Disk Images* chapter in the System Emulation Users Guide). This is the classic bulk-only transport protocol used by 99% of USB sticks. This example shows it connected to an XHCI USB controller and with a drive backed by a raw format disk image:

```
qemu-system-x86_64 [...]                                  \
 -drive if=none,id=stick,format=raw,file=/path/to/file.img \
 -device nec-usb-xhci,id=xhci                             \
 -device usb-storage,bus=xhci.0,drive=stick
```

usb-uas
USB attached SCSI device. This does not create a SCSI disk, so you need to explicitly create a `scsi-hd` or `scsi-cd` device on the command line, as well as using the `-drive` option to specify what those disks are backed by. One `usb-uas` device can handle multiple logical units (disks). This example creates three logical units: two disks and one cdrom drive:

```
qemu-system-x86_64 [...]                                          \
 -drive if=none,id=uas-disk1,format=raw,file=/path/to/file1.img   \
 -drive if=none,id=uas-disk2,format=raw,file=/path/to/file2.img   \
 -drive if=none,id=uas-cdrom,media=cdrom,format=raw,file=/path/to/image.iso \
 -device nec-usb-xhci,id=xhci                                     \
 -device usb-uas,id=uas,bus=xhci.0                                \
 -device scsi-hd,bus=uas.0,scsi-id=0,lun=0,drive=uas-disk1        \
 -device scsi-hd,bus=uas.0,scsi-id=0,lun=1,drive=uas-disk2        \
 -device scsi-cd,bus=uas.0,scsi-id=0,lun=5,drive=uas-cdrom
```

usb-bot
Bulk-only transport storage device. This presents the guest with the same USB bulk-only transport protocol

interface as `usb-storage`, but the QEMU command line option works like `usb-uas` and does not automatically create SCSI disks for you. `usb-bot` supports up to 16 LUNs. Unlike `usb-uas`, the LUN numbers must be continuous, i.e. for three devices you must use 0+1+2. The 0+1+5 numbering from the `usb-uas` example above won't work with `usb-bot`.

usb-mtp,rootdir=dir
Media transfer protocol device, using dir as root of the file tree that is presented to the guest.

usb-host,hostbus=bus,hostaddr=addr
Pass through the host device identified by bus and addr

usb-host,vendorid=vendor,productid=product
Pass through the host device identified by vendor and product ID

usb-wacom-tablet
Virtual Wacom PenPartner tablet. This device is similar to the `tablet` above but it can be used with the tslib library because in addition to touch coordinates it reports touch pressure.

usb-kbd
Standard USB keyboard. Will override the PS/2 keyboard (if present).

usb-serial,chardev=id
Serial converter. This emulates an FTDI FT232BM chip connected to host character device id.

usb-braille,chardev=id
Braille device. This emulates a Baum Braille device USB port. id has to specify a character device defined with `-chardev ...,id=id`. One will normally use BrlAPI to display the braille output on a BRLTTY-supported device with

```
qemu-system-x86_64 [...] -chardev braille,id=brl -device usb-braille,chardev=brl
```

or alternatively, use the following equivalent shortcut:

```
qemu-system-x86_64 [...] -usbdevice braille
```

usb-net[,netdev=id]
Network adapter that supports CDC ethernet and RNDIS protocols. id specifies a netdev defined with `-netdev ...,id=id`. For instance, user-mode networking can be used with

```
qemu-system-x86_64 [...] -netdev user,id=net0 -device usb-net,netdev=net0
```

usb-ccid
Smartcard reader device

usb-audio
USB audio device

u2f-{emulated,passthru}
Universal Second Factor (U2F) USB Key Device

canokey
An Open-source Secure Key implementing FIDO2, OpenPGP, PIV and more. For more information, see *CanoKey QEMU*.

Physical port addressing

For all the above USB devices, by default QEMU will plug the device into the next available port on the specified USB bus, or onto some available USB bus if you didn't specify one explicitly. If you need to, you can also specify the physical port where the device will show up in the guest. This can be done using the `port` property. UHCI has two root ports (1,2). EHCI has six root ports (1-6), and the emulated (1.1) USB hub has eight ports.

Plugging a tablet into UHCI port 1 works like this:

```
-device usb-tablet,bus=usb-bus.0,port=1
```

Plugging a hub into UHCI port 2 works like this:

```
-device usb-hub,bus=usb-bus.0,port=2
```

Plugging a virtual USB stick into port 4 of the hub just plugged works this way:

```
-device usb-storage,bus=usb-bus.0,port=2.4,drive=...
```

In the monitor, the `device_add` command also accepts a `port` property specification. If you want to unplug devices too you should specify some unique id which you can use to refer to the device. You can then use `device_del` to unplug the device later. For example:

```
(qemu) device_add usb-tablet,bus=usb-bus.0,port=1,id=my-tablet
(qemu) device_del my-tablet
```

Hotplugging USB storage

The `usb-bot` and `usb-uas` devices can be hotplugged. In the hotplug case they are added with `attached = false` so the guest will not see the device until the `attached` property is explicitly set to true. That allows you to attach one or more scsi devices before making the device visible to the guest. The workflow looks like this:

1. `device-add usb-bot,id=foo`
2. `device-add scsi-{hd,cd},bus=foo.0,lun=0`
3. optionally add more devices (luns 1 ... 15)
4. `scripts/qmp/qom-set foo.attached = true`

Using host USB devices on a Linux host

WARNING: this is an experimental feature. QEMU will slow down when using it. USB devices requiring real time streaming (i.e. USB Video Cameras) are not supported yet.

1. If you use an early Linux 2.4 kernel, verify that no Linux driver is actually using the USB device. A simple way to do that is simply to disable the corresponding kernel module by renaming it from `mydriver.o` to `mydriver.o.disabled`.
2. Verify that `/proc/bus/usb` is working (most Linux distributions should enable it by default). You should see something like that:

   ```
   ls /proc/bus/usb
   001  devices  drivers
   ```

3. Since only root can access to the USB devices directly, you can either launch QEMU as root or change the permissions of the USB devices you want to use. For testing, the following suffices:

```
chown -R myuid /proc/bus/usb
```

4. Launch QEMU and do in the monitor:

```
info usbhost
  Device 1.2, speed 480 Mb/s
    Class 00: USB device 1234:5678, USB DISK
```

   You should see the list of the devices you can use (Never try to use hubs, it won't work).

5. Add the device in QEMU by using:

```
device_add usb-host,vendorid=0x1234,productid=0x5678
```

   Normally the guest OS should report that a new USB device is plugged. You can use the option `-device usb-host,...` to do the same.

6. Now you can try to use the host USB device in QEMU.

When relaunching QEMU, you may have to unplug and plug again the USB device to make it work again (this is a bug).

`usb-host` properties for specifying the host device

The example above uses the `vendorid` and `productid` to specify which host device to pass through, but this is not the only way to specify the host device. `usb-host` supports the following properties:

hostbus=<nr>
Specifies the bus number the device must be attached to

hostaddr=<nr>
Specifies the device address the device got assigned by the guest os

hostport=<str>
Specifies the physical port the device is attached to

vendorid=<hexnr>
Specifies the vendor ID of the device

productid=<hexnr>
Specifies the product ID of the device.

In theory you can combine all these properties as you like. In practice only a few combinations are useful:

- `vendorid` and `productid` – match for a specific device, pass it to the guest when it shows up somewhere in the host.
- `hostbus` and `hostport` – match for a specific physical port in the host, any device which is plugged in there gets passed to the guest.
- `hostbus` and `hostaddr` – most useful for ad-hoc pass through as the hostaddr isn't stable. The next time you plug the device into the host it will get a new hostaddr.

Note that on the host USB 1.1 devices are handled by UHCI/OHCI and USB 2.0 by EHCI. That means different USB devices plugged into the very same physical port on the host may show up on different host buses depending on the speed. Supposing that devices plugged into a given physical port appear as bus 1 + port 1 for 2.0 devices and bus 3 +

port 1 for 1.1 devices, you can pass through any device plugged into that port and also assign it to the correct USB bus in QEMU like this:

```
qemu-system-x86_64 -M pc [...]                                \
     -usb                                                     \
     -device usb-ehci,id=ehci                                 \
     -device usb-host,bus=usb-bus.0,hostbus=3,hostport=1      \
     -device usb-host,bus=ehci.0,hostbus=1,hostport=1
```

usb-host properties for reset behavior

The guest-reset and guest-reset-all properties control whenever the guest is allowed to reset the physical usb device on the host. There are three cases:

guest-reset=false
: The guest is not allowed to reset the (physical) usb device.

guest-reset=true,guest-resets-all=false
: The guest is allowed to reset the device when it is not yet initialized (aka no usb bus address assigned). Usually this results in one guest reset being allowed. This is the default behavior.

guest-reset=true,guest-resets-all=true
: The guest is allowed to reset the device as it pleases.

The reason for this existing are broken usb devices. In theory one should be able to reset (and re-initialize) usb devices at any time. In practice that may result in shitty usb device firmware crashing and the device not responding any more until you power-cycle (aka un-plug and re-plug) it.

What works best pretty much depends on the behavior of the specific usb device at hand, so it's a trial-and-error game. If the default doesn't work, try another option and see whenever the situation improves.

record usb transfers

All usb devices have support for recording the usb traffic. This can be enabled using the pcap=<file> property, for example:

```
-device usb-mouse,pcap=mouse.pcap
```

The pcap files are compatible with the linux kernels usbmon. Many tools, including wireshark, can decode and inspect these trace files.

vhost-user back ends

vhost-user back ends are way to service the request of VirtIO devices outside of QEMU itself. To do this there are a number of things required.

vhost-user device

These are simple stub devices that ensure the VirtIO device is visible to the guest. The code is mostly boilerplate although each device has a `chardev` option which specifies the ID of the `--chardev` device that connects via a socket to the vhost-user *daemon*.

Each device will have an virtio-mmio and virtio-pci variant. See your platform details for what sort of virtio bus to use.

Table 4: vhost-user devices

Device	Type	Notes
vhost-user-blk	Block storage	See contrib/vhost-user-blk
vhost-user-fs	File based storage driver	See https://gitlab.com/virtio-fs/virtiofsd
vhost-user-gpio	Proxy gpio pins to host	See https://github.com/rust-vmm/vhost-device
vhost-user-gpu	GPU driver	See contrib/vhost-user-gpu
vhost-user-i2c	Proxy i2c devices to host	See https://github.com/rust-vmm/vhost-device
vhost-user-input	Generic input driver	*QEMU vhost-user-input - Input emulation*
vhost-user-rng	Entropy driver	*QEMU vhost-user-rng - RNG emulation*
vhost-user-scmi	System Control and Management Interface	See https://github.com/rust-vmm/vhost-device
vhost-user-snd	Audio device	See https://github.com/rust-vmm/vhost-device/staging
vhost-user-scsi	SCSI based storage	See contrib/vhost-user-scsi
vhost-user-vsock	Socket based communication	See https://github.com/rust-vmm/vhost-device

The referenced *daemons* are not exhaustive, any conforming backend implementing the device and using the vhost-user protocol should work.

vhost-user-device

The vhost-user-device is a generic development device intended for expert use while developing new backends. The user needs to specify all the required parameters including:

- Device `virtio-id`
- The `num_vqs` it needs and their `vq_size`
- The `config_size` if needed

Note: To prevent user confusion you cannot currently instantiate vhost-user-device without first patching out:

```
/* Reason: stop inexperienced users confusing themselves */
dc->user_creatable = false;
```

in `vhost-user-device.c` and `vhost-user-device-pci.c` file and rebuilding.

---

vhost-user daemon

This is a separate process that is connected to by QEMU via a socket following the *Vhost-user Protocol*. There are a number of daemons that can be built when enabled by the project although any daemon that meets the specification for a given device can be used.

Shared memory object

In order for the daemon to access the VirtIO queues to process the requests it needs access to the guest's address space. This is achieved via the `memory-backend-file` or `memory-backend-memfd` objects. A reference to a file-descriptor which can access this object will be passed via the socket as part of the protocol negotiation.

Currently the shared memory object needs to match the size of the main system memory as defined by the `-m` argument.

Example

First start your daemon.

```
$ virtio-foo --socket-path=/var/run/foo.sock $OTHER_ARGS
```

Then you start your QEMU instance specifying the device, chardev and memory objects.

```
$ qemu-system-x86_64 \
    -m 4096 \
    -chardev socket,id=ba1,path=/var/run/foo.sock \
    -device vhost-user-foo,chardev=ba1,$OTHER_ARGS \
    -object memory-backend-memfd,id=mem,size=4G,share=on \
    -numa node,memdev=mem \
      ...
```

virtio-gpu

This document explains the setup and usage of the virtio-gpu device. The virtio-gpu device paravirtualizes the GPU and display controller.

Linux kernel support

virtio-gpu requires a guest Linux kernel built with the `CONFIG_DRM_VIRTIO_GPU` option.

QEMU virtio-gpu variants

QEMU virtio-gpu device variants come in the following form:

- `virtio-vga[-BACKEND]`
- `virtio-gpu[-BACKEND][-INTERFACE]`
- `vhost-user-vga`
- `vhost-user-pci`

Backends: QEMU provides a 2D virtio-gpu backend, and two accelerated backends: virglrenderer ('gl' device label) and rutabaga_gfx ('rutabaga' device label). There is a vhost-user backend that runs the graphics stack in a separate process for improved isolation.

Interfaces: QEMU further categorizes virtio-gpu device variants based on the interface exposed to the guest. The interfaces can be classified into VGA and non-VGA variants. The VGA ones are prefixed with virtio-vga or vhost-user-vga while the non-VGA ones are prefixed with virtio-gpu or vhost-user-gpu.

The VGA ones always use the PCI interface, but for the non-VGA ones, the user can further pick between MMIO or PCI. For MMIO, the user can suffix the device name with -device, though vhost-user-gpu does not support MMIO. For PCI, the user can suffix it with -pci. Without these suffixes, the platform default will be chosen.

virtio-gpu 2d

The default 2D backend only performs 2D operations. The guest needs to employ a software renderer for 3D graphics.

Typically, the software renderer is provided by Mesa or SwiftShader. Mesa's implementations (LLVMpipe, Lavapipe and virgl below) work out of box on typical modern Linux distributions.

```
-device virtio-gpu
```

virtio-gpu virglrenderer

When using virgl accelerated graphics mode in the guest, OpenGL API calls are translated into an intermediate representation (see Gallium3D). The intermediate representation is communicated to the host and the virglrenderer library on the host translates the intermediate representation back to OpenGL API calls.

```
-device virtio-gpu-gl
```

virtio-gpu rutabaga

virtio-gpu can also leverage rutabaga_gfx to provide gfxstream rendering and Wayland display passthrough. With the gfxstream rendering mode, GLES and Vulkan calls are forwarded to the host with minimal modification.

The crosvm book provides directions on how to build a gfxstream-enabled rutabaga and launch a guest Wayland proxy.

This device does require host blob support (`hostmem` field below). The `hostmem` field specifies the size of virtio-gpu host memory window. This is typically between 256M and 8G.

At least one virtio-gpu capability set ("capset") must be specified when starting the device. The currently capsets supported are `gfxstream-vulkan` and `cross-domain` for Linux guests. For Android guests, the experimental `x-gfxstream-gles` and `x-gfxstream-composer` capsets are also supported.

The device will try to auto-detect the wayland socket path if the `cross-domain` capset name is set. The user may optionally specify `wayland-socket-path` for non-standard paths.

The `wsi` option can be set to `surfaceless` or `headless`. Surfaceless doesn't create a native window surface, but does copy from the render target to the Pixman buffer if a virtio-gpu 2D hypercall is issued. Headless is like surfaceless, but doesn't copy to the Pixman buffer. Surfaceless is the default if `wsi` is not specified.

```
-device virtio-gpu-rutabaga,gfxstream-vulkan=on,cross-domain=on,
   hostmem=8G,wayland-socket-path=/tmp/nonstandard/mock_wayland.sock,
   wsi=headless
```

virtio pmem

This document explains the setup and usage of the virtio pmem device. The virtio pmem device is a paravirtualized persistent memory device on regular (i.e non-NVDIMM) storage.

Usecase

Virtio pmem allows to bypass the guest page cache and directly use host page cache. This reduces guest memory footprint as the host can make efficient memory reclaim decisions under memory pressure.

How does virtio-pmem compare to the nvdimm emulation?

NVDIMM emulation on regular (i.e. non-NVDIMM) host storage does not persist the guest writes as there are no defined semantics in the device specification. The virtio pmem device provides guest write persistence on non-NVDIMM host storage.

virtio pmem usage

A virtio pmem device backed by a memory-backend-file can be created on the QEMU command line as in the following example:

```
-object memory-backend-file,id=mem1,share,mem-path=./virtio_pmem.img,size=4G
-device virtio-pmem-pci,memdev=mem1,id=nv1
```

where:

- "object memory-backend-file,id=mem1,share,mem-path=<image>, size=<image size>" creates a backend file with the specified size.
- "device virtio-pmem-pci,id=nvdimm1,memdev=mem1" creates a virtio pmem pci device whose storage is provided by above memory backend device.

Multiple virtio pmem devices can be created if multiple pairs of "-object" and "-device" are provided.

Hotplug

Virtio pmem devices can be hotplugged via the QEMU monitor. First, the memory backing has to be added via 'object_add'; afterwards, the virtio pmem device can be added via 'device_add'.

For example, the following commands add another 4GB virtio pmem device to the guest:

```
(qemu) object_add memory-backend-file,id=mem2,share=on,mem-path=virtio_pmem2.img,size=4G
(qemu) device_add virtio-pmem-pci,id=virtio_pmem2,memdev=mem2
```

Guest Data Persistence

Guest data persistence on non-NVDIMM requires guest userspace applications to perform fsync/msync. This is different from a real nvdimm backend where no additional fsync/msync is required. This is to persist guest writes in host backing file which otherwise remains in host page cache and there is risk of losing the data in case of power failure.

With virtio pmem device, MAP_SYNC mmap flag is not supported. This provides a hint to application to perform fsync for write persistence.

Limitations

- Real nvdimm device backend is not supported.
- virtio pmem hotunplug is not supported.
- ACPI NVDIMM features like regions/namespaces are not supported.
- ndctl command is not supported.

virtio sound

This document explains the setup and usage of the Virtio sound device. The Virtio sound device is a paravirtualized sound card device.

Linux kernel support

Virtio sound requires a guest Linux kernel built with the `CONFIG_SND_VIRTIO` option.

Description

Virtio sound implements capture and playback from inside a guest using the configured audio backend of the host machine.

Device properties

The Virtio sound device can be configured with the following properties:

- `jacks` number of physical jacks (Unimplemented).
- `streams` number of PCM streams. At the moment, no stream configuration is supported: the first one will always be a playback stream, an optional second will always be a capture stream. Adding more will cycle stream directions from playback to capture.
- `chmaps` number of channel maps (Unimplemented).

All streams are stereo and have the default channel positions `Front left, right`.

Examples

Add an audio device and an audio backend at once with `-audio` and `model=virtio`:

- pulseaudio: `-audio driver=pa,model=virtio` or `-audio driver=pa,model=virtio,server=/run/user/1000/pulse/native`
- sdl: `-audio driver=sdl,model=virtio`
- coreaudio: `-audio driver=coreaudio,model=virtio`

etc.

To specifically add virtualized sound devices, you have to specify a PCI device and an audio backend listed with `-audio driver=help` that works on your host machine, e.g.:

```
-device virtio-sound-pci,audiodev=my_audiodev \
-audiodev alsa,id=my_audiodev
```

QEMU vhost-user-input - Input emulation

This document describes the setup and usage of the Virtio input device. The Virtio input device is a paravirtualized device for input events.

Description

The vhost-user-input device implementation was designed to work with a daemon polling on input devices and passes input events to the guest.

QEMU provides a backend implementation in contrib/vhost-user-input.

Linux kernel support

Virtio input requires a guest Linux kernel built with the `CONFIG_VIRTIO_INPUT` option.

Examples

The backend daemon should be started first:

```
host# vhost-user-input --socket-path=input.sock       \
    --evdev-path=/dev/input/event17
```

The QEMU invocation needs to create a chardev socket to communicate with the backend daemon and access the VirtIO queues with the guest over the *shared memory*.

```
host# qemu-system                                                    \
    -chardev socket,path=/tmp/input.sock,id=mouse0                   \
    -device vhost-user-input-pci,chardev=mouse0                      \
    -m 4096                                                          \
    -object memory-backend-file,id=mem,size=4G,mem-path=/dev/shm,share=on  \
    -numa node,memdev=mem                                            \
    ...
```

QEMU vhost-user-rng - RNG emulation

Background

What follows builds on the material presented in vhost-user.rst - it should be reviewed before moving forward with the content in this file.

Description

The vhost-user-rng device implementation was designed to work with a random number generator daemon such as the one found in the vhost-device crate of the rust-vmm project available on github [1].

[1]. https://github.com/rust-vmm/vhost-device

Examples

The daemon should be started first:

```
host# vhost-device-rng --socket-path=rng.sock -c 1 -m 512 -p 1000
```

The QEMU invocation needs to create a chardev socket the device can use to communicate as well as share the guests memory over a memfd.

```
host# qemu-system                                                          \
    -chardev socket,path=$(PATH)/rng.sock,id=rng0                          \
    -device vhost-user-rng-pci,chardev=rng0                                \
    -m 4096                                                                \
    -object memory-backend-file,id=mem,size=4G,mem-path=/dev/shm,share=on  \
    -numa node,memdev=mem                                                  \
    ...
```

CanoKey QEMU

CanoKey[1] is an open-source secure key with supports of

- U2F / FIDO2 with Ed25519 and HMAC-secret
- OpenPGP Card V3.4 with RSA4096, Ed25519 and more[2]
- PIV (NIST SP 800-73-4)
- HOTP / TOTP
- NDEF

All these platform-independent features are in canokey-core[3].

For different platforms, CanoKey has different implementations, including both hardware implementations and virtual cards:

- CanoKey STM32[4]

[1] https://canokeys.org
[2] https://docs.canokeys.org/userguide/openpgp/#supported-algorithm
[3] https://github.com/canokeys/canokey-core
[4] https://github.com/canokeys/canokey-stm32

- CanoKey Pigeon[5]
- (virt-card) CanoKey USB/IP
- (virt-card) CanoKey FunctionFS

In QEMU, yet another CanoKey virt-card is implemented. CanoKey QEMU exposes itself as a USB device to the guest OS.

With the same software configuration as a hardware key, the guest OS can use all the functionalities of a secure key as if there was actually an hardware key plugged in.

CanoKey QEMU provides much convenience for debugging:

- libcanokey-qemu supports debugging output thus developers can inspect what happens inside a secure key
- CanoKey QEMU supports trace event thus event
- QEMU USB stack supports pcap thus USB packet between the guest and key can be captured and analysed

Then for developers:

- For developers on software with secure key support (e.g. FIDO2, OpenPGP), they can see what happens inside the secure key
- For secure key developers, USB packets between guest OS and CanoKey can be easily captured and analysed

Also since this is a virtual card, it can be easily used in CI for testing on code coping with secure key.

Building

libcanokey-qemu is required to use CanoKey QEMU.

```
git clone https://github.com/canokeys/canokey-qemu
mkdir canokey-qemu/build
pushd canokey-qemu/build
```

If you want to install libcanokey-qemu in a different place, add `-DCMAKE_INSTALL_PREFIX=/path/to/your/place` to cmake below.

```
cmake ..
make
make install # may need sudo
popd
```

Then configuring and building:

```
# depending on your env, lib/pkgconfig can be lib64/pkgconfig
export PKG_CONFIG_PATH=/path/to/your/place/lib/pkgconfig:$PKG_CONFIG_PATH
./configure --enable-canokey && make
```

[5] https://github.com/canokeys/canokey-pigeon

Using CanoKey QEMU

CanoKey QEMU stores all its data on a file of the host specified by the argument when invoking qemu.

```
qemu-system-x86_64 -usb -device canokey,file=$HOME/.canokey-file
```

Note: you should keep this file carefully as it may contain your private key!

The first time when the file is used, it is created and initialized by CanoKey, afterwards CanoKey QEMU would just read this file.

After the guest OS boots, you can check that there is a USB device.

For example, If the guest OS is an Linux machine. You may invoke lsusb and find CanoKey QEMU there:

```
$ lsusb
Bus 001 Device 002: ID 20a0:42d4 Clay Logic CanoKey QEMU
```

You may setup the key as guided in[6]. The console for the key is at[7].

Debugging

CanoKey QEMU consists of two parts, `libcanokey-qemu.so` and `canokey.c`, the latter of which resides in QEMU. The former provides core functionality of a secure key while the latter provides platform-dependent functions: USB packet handling.

If you want to trace what happens inside the secure key, when compiling libcanokey-qemu, you should add `-DQEMU_DEBUG_OUTPUT=ON` in cmake command line:

```
cmake .. -DQEMU_DEBUG_OUTPUT=ON
```

If you want to trace events happened in canokey.c, use

```
qemu-system-x86_64 --trace "canokey_*" \
    -usb -device canokey,file=$HOME/.canokey-file
```

If you want to capture USB packets between the guest and the host, you can:

```
qemu-system-x86_64 -usb -device canokey,file=$HOME/.canokey-file,pcap=key.pcap
```

Limitations

Currently libcanokey-qemu.so has dozens of global variables as it was originally designed for embedded systems. Thus one qemu instance can not have multiple CanoKey QEMU running, namely you can not

```
qemu-system-x86_64 -usb -device canokey,file=$HOME/.canokey-file \
     -device canokey,file=$HOME/.canokey-file2
```

Also, there is no lock on canokey-file, thus two CanoKey QEMU instance can not read one canokey-file at the same time.

[6] https://docs.canokeys.org/
[7] https://console.canokeys.org/

References

Universal Second Factor (U2F) USB Key Device

U2F is an open authentication standard that enables relying parties exposed to the internet to offer a strong second factor option for end user authentication.

The second factor is provided by a device implementing the U2F protocol. In case of a USB U2F security key, it is a USB HID device that implements the U2F protocol.

QEMU supports both pass-through of a host U2F key device to a VM, and software emulation of a U2F key.

u2f-passthru

The `u2f-passthru` device allows you to connect a real hardware U2F key on your host to a guest VM. All requests made from the guest are passed through to the physical security key connected to the host machine and vice versa.

In addition, the dedicated pass-through allows you to share a single U2F security key with several guest VMs, which is not possible with a simple host device assignment pass-through.

You can specify the host U2F key to use with the `hidraw` option, which takes the host path to a Linux `/dev/hidrawN` device:

```
qemu-system-x86_64 -usb -device u2f-passthru,hidraw=/dev/hidraw0
```

If you don't specify the device, the `u2f-passthru` device will autoscan to take the first U2F device it finds on the host (this requires a working libudev):

```
qemu-system-x86_64 -usb -device u2f-passthru
```

u2f-emulated

`u2f-emulated` is a completely software emulated U2F device. It uses libu2f-emu for the U2F key emulation. libu2f-emu provides a complete implementation of the U2F protocol device part for all specified transports given by the FIDO Alliance.

To work, an emulated U2F device must have four elements:

- ec x509 certificate
- ec private key
- counter (four bytes value)
- 48 bytes of entropy (random bits)

To use this type of device, these have to be configured, and these four elements must be passed one way or another.

Assuming that you have a working libu2f-emu installed on the host, there are three possible ways to configure the `u2f-emulated` device:

- ephemeral
- setup directory
- manual

Ephemeral is the simplest way to configure; it lets the device generate all the elements it needs for a single use of the lifetime of the device. It is the default if you do not pass any other options to the device.

```
qemu-system-x86_64 -usb -device u2f-emulated
```

You can pass the device the path of a setup directory on the host using the `dir` option; the directory must contain these four files:

- `certificate.pem`: ec x509 certificate
- `private-key.pem`: ec private key
- `counter`: counter value
- `entropy`: 48 bytes of entropy

```
qemu-system-x86_64 -usb -device u2f-emulated,dir=$dir
```

You can also manually pass the device the paths to each of these files, if you don't want them all to be in the same directory, using the options

- `cert`
- `priv`
- `counter`
- `entropy`

```
qemu-system-x86_64 -usb -device u2f-emulated,cert=$DIR1/$FILE1,priv=$DIR2/$FILE2,counter=
↪$DIR3/$FILE3,entropy=$DIR4/$FILE4
```

igb

igb is a family of Intel's gigabit ethernet controllers. In QEMU, 82576 emulation is implemented in particular. Its datasheet is available at[1].

This implementation is expected to be useful to test SR-IOV networking without requiring physical hardware.

Limitations

This igb implementation was tested with Linux Test Project[2] and Windows HLK[3] during the initial development. Later it was also tested with DPDK Test Suite[4]. The command used when testing with LTP is:

```
network.sh -6mta
```

Be aware that this implementation lacks many functionalities available with the actual hardware, and you may experience various failures if you try to use it with a different operating system other than DPDK, Linux, and Windows or if you try functionalities not covered by the tests.

[1] https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82576eb-gigabit-ethernet-controller-datasheet.pdf
[2] https://github.com/linux-test-project/ltp
[3] https://learn.microsoft.com/en-us/windows-hardware/test/hlk/
[4] https://doc.dpdk.org/dts/gsg/

Using igb

Using igb should be nothing different from using another network device. See *Network emulation* in general.

However, you may also need to perform additional steps to activate SR-IOV feature on your guest. For Linux, refer to[5].

Developing igb

igb is the successor of e1000e, and e1000e is the successor of e1000 in turn. As these devices are very similar, if you make a change for igb and the same change can be applied to e1000e and e1000, please do so.

Please do not forget to run tests before submitting a change. As tests included in QEMU is very minimal, run some application which is likely to be affected by the change to confirm it works in an integrated system.

Testing igb

A qtest of the basic functionality is available. Run the below at the build directory:

```
meson test qtest-x86_64/qos-test
```

ethtool can test register accesses, interrupts, etc. It is automated as an Avocado test and can be ran with the following command:

```
make check-avocado AVOCADO_TESTS=tests/avocado/netdev-ethtool.py
```

References

2.4 Keys in the graphical frontends

During the graphical emulation, you can use special key combinations from the following table to change modes. By default the modifier is Ctrl-Alt (used in the table below) which can be changed with `-display` suboption `mod=` where appropriate. For example, `-display sdl, grab-mod=lshift-lctrl-lalt` changes the modifier key to Ctrl-Alt-Shift, while `-display sdl,grab-mod=rctrl` changes it to the right Ctrl key.

Ctrl-Alt-f
: Toggle full screen

Ctrl-Alt-+
: Enlarge the screen

Ctrl-Alt--
: Shrink the screen

Ctrl-Alt-u
: Restore the screen's un-scaled dimensions

Ctrl-Alt-n
: Switch to virtual console 'n'. Standard console mappings are:

  1
  : Target system display

[5] https://docs.kernel.org/PCI/pci-iov-howto.html

2
: Monitor

3
: Serial port

Ctrl-Alt-g
: Toggle mouse and keyboard grab.

In the virtual consoles, you can use Ctrl-Up, Ctrl-Down, Ctrl-PageUp and Ctrl-PageDown to move in the back log.

2.5 Keys in the character backend multiplexer

During emulation, if you are using a character backend multiplexer (which is the default if you are using `-nographic`) then several commands are available via an escape sequence. These key sequences all start with an escape character, which is Ctrl-a by default, but can be changed with `-echr`. The list below assumes you're using the default.

Ctrl-a h
: Print this help

Ctrl-a x
: Exit emulator

Ctrl-a s
: Save disk data back to file (if -snapshot)

Ctrl-a t
: Toggle console timestamps

Ctrl-a b
: Send break (magic sysrq in Linux)

Ctrl-a c
: Rotate between the frontends connected to the multiplexer (usually this switches between the monitor and the console)

Ctrl-a Ctrl-a
: Send the escape character to the frontend

2.6 QEMU Monitor

The QEMU monitor is used to give complex commands to the QEMU emulator. You can use it to:

- Remove or insert removable media images (such as CD-ROM or floppies).
- Freeze/unfreeze the Virtual Machine (VM) and save or restore its state from a disk file.
- Inspect the VM state without an external debugger.

2.6.1 Commands

The following commands are available:

help or ? [*cmd*]
Show the help for all commands or just for command *cmd*.

commit
Commit changes to the disk images (if -snapshot is used) or backing files. If the backing file is smaller than the snapshot, then the backing file will be resized to be the same size as the snapshot. If the snapshot is smaller than the backing file, the backing file will not be truncated. If you want the backing file to match the size of the smaller snapshot, you can safely truncate it yourself once the commit operation successfully completes.

quit or q
Quit the emulator.

exit_preconfig
This command makes QEMU exit the preconfig state and proceed with VM initialization using configuration data provided on the command line and via the QMP monitor during the preconfig state. The command is only available during the preconfig state (i.e. when the –preconfig command line option was in use).

block_resize
Resize a block image while a guest is running. Usually requires guest action to see the updated size. Resize to a lower size is supported, but should be used with extreme caution. Note that this command only resizes image files, it can not resize block devices like LVM volumes.

block_stream
Copy data from a backing file into a block device.

block_job_set_speed
Set maximum speed for a background block operation.

block_job_cancel
Stop an active background block operation (streaming, mirroring).

block_job_complete
Manually trigger completion of an active background block operation. For mirroring, this will switch the device to the destination path.

block_job_pause
Pause an active block streaming operation.

block_job_resume
Resume a paused block streaming operation.

eject [-f] ***device***
Eject a removable medium (use -f to force it).

drive_del ***device***
Remove host block device. The result is that guest generated IO is no longer submitted against the host device underlying the disk. Once a drive has been deleted, the QEMU Block layer returns -EIO which results in IO errors in the guest for applications that are reading/writing to the device. These errors are always reported to the guest, regardless of the drive's error actions (drive options rerror, werror).

change ***device setting***
Change the configuration of a device.

change ***diskdevice*** **[-f]** ***filename*** **[*format* [*read-only-mode*]]**
Change the medium for a removable disk device to point to *filename*. eg:

```
(qemu) change ide1-cd0 /path/to/some.iso
```

-f
forces the operation even if the guest has locked the tray.

format is optional.

read-only-mode may be used to change the read-only status of the device. It accepts the following values:

retain
Retains the current status; this is the default.

read-only
Makes the device read-only.

read-write
Makes the device writable.

`change vnc password` [*password*]

Change the password associated with the VNC server. If the new password is not supplied, the monitor will prompt for it to be entered. VNC passwords are only significant up to 8 letters. eg:

```
(qemu) change vnc password
Password: ********
```

screendump *filename*
Save screen into PPM image *filename*.

logfile *filename*
Output logs to *filename*.

trace-event
changes status of a trace event

trace-file on|off|flush
Open, close, or flush the trace file. If no argument is given, the status of the trace file is displayed.

log *item1*[,...]
Activate logging of the specified items.

savevm *tag*
Create a snapshot of the whole virtual machine. If *tag* is provided, it is used as human readable identifier. If there is already a snapshot with the same tag, it is replaced. More info at *VM snapshots*.

Since 4.0, savevm stopped allowing the snapshot id to be set, accepting only *tag* as parameter.

loadvm *tag*
Set the whole virtual machine to the snapshot identified by the tag *tag*.

Since 4.0, loadvm stopped accepting snapshot id as parameter.

delvm *tag*
Delete the snapshot identified by *tag*.

Since 4.0, delvm stopped deleting snapshots by snapshot id, accepting only *tag* as parameter.

one-insn-per-tb [off]
Run the emulation with one guest instruction per translation block. This slows down emulation a lot, but can be useful in some situations, such as when trying to analyse the logs produced by the `-d` option. This only has an effect when using TCG, not with KVM or other accelerators.

If called with option off, the emulation returns to normal mode.

stop or s
Stop emulation.

cont or c
Resume emulation.

system_wakeup
Wakeup guest from suspend.

gdbserver [*port*]
Start gdbserver session (default *port*=1234)

x/*fmt addr*
Virtual memory dump starting at *addr*.

xp /*fmt addr*
Physical memory dump starting at *addr*.

fmt is a format which tells the command how to format the data. Its syntax is: `/{count}{format}{size}`

count
is the number of items to be dumped.

format
can be x (hex), d (signed decimal), u (unsigned decimal), o (octal), c (char) or i (asm instruction).

size
can be b (8 bits), h (16 bits), w (32 bits) or g (64 bits). On x86, `h` or `w` can be specified with the `i` format to respectively select 16 or 32 bit code instruction size.

Examples:

Dump 10 instructions at the current instruction pointer:

```
(qemu) x/10i $eip
0x90107063:  ret
0x90107064:  sti
0x90107065:  lea    0x0(%esi,1),%esi
0x90107069:  lea    0x0(%edi,1),%edi
0x90107070:  ret
0x90107071:  jmp    0x90107080
0x90107073:  nop
0x90107074:  nop
0x90107075:  nop
0x90107076:  nop
```

Dump 80 16 bit values at the start of the video memory:

```
(qemu) xp/80hx 0xb8000
0x000b8000: 0x0b50 0x0b6c 0x0b65 0x0b78 0x0b38 0x0b36 0x0b2f 0x0b42
0x000b8010: 0x0b6f 0x0b63 0x0b68 0x0b73 0x0b20 0x0b56 0x0b47 0x0b41
0x000b8020: 0x0b42 0x0b69 0x0b6f 0x0b73 0x0b20 0x0b63 0x0b75 0x0b72
0x000b8030: 0x0b72 0x0b65 0x0b6e 0x0b74 0x0b2d 0x0b63 0x0b76 0x0b73
0x000b8040: 0x0b20 0x0b30 0x0b35 0x0b20 0x0b4e 0x0b6f 0x0b76 0x0b20
0x000b8050: 0x0b32 0x0b30 0x0b30 0x0b33 0x0720 0x0720 0x0720 0x0720
0x000b8060: 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720
0x000b8070: 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720
0x000b8080: 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720
0x000b8090: 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720
```

gpa2hva *addr*
Print the host virtual address at which the guest's physical address *addr* is mapped.

gpa2hpa *addr*
Print the host physical address at which the guest's physical address *addr* is mapped.

gva2gpa *addr*
Print the guest physical address at which the guest's virtual address *addr* is mapped based on the mapping for the current CPU.

print or p/*fmt expr*
Print expression value. Only the *format* part of *fmt* is used.

i/*fmt addr* [.*index*]
Read I/O port.

o/*fmt addr val*
Write to I/O port.

sendkey *keys*
Send *keys* to the guest. *keys* could be the name of the key or the raw value in hexadecimal format. Use - to press several keys simultaneously. Example:

```
sendkey ctrl-alt-f1
```

This command is useful to send keys that your graphical user interface intercepts at low level, such as `ctrl-alt-f1` in X Window.

sync-profile [on|off|reset]
Enable, disable or reset synchronization profiling. With no arguments, prints whether profiling is on or off.

system_reset
Reset the system.

system_powerdown
Power down the system (if supported).

sum *addr size*
Compute the checksum of a memory region.

device_add *config*
Add device.

device_del *id*
Remove device *id*. *id* may be a short ID or a QOM object path.

cpu *index*
Set the default CPU.

mouse_move *dx dy* [*dz*]
Move the active mouse to the specified coordinates *dx dy* with optional scroll axis *dz*.

mouse_button *val*
Change the active mouse button state *val* (1=L, 2=M, 4=R).

mouse_set *index*
Set which mouse device receives events at given *index*, index can be obtained with:

```
info mice
```

wavcapture *filename audiodev* [*frequency* [*bits* [*channels*]]]
Capture audio into *filename* from *audiodev*, using sample rate *frequency* bits per sample *bits* and number of channels *channels*.

Defaults:

- Sample rate = 44100 Hz - CD quality
- Bits = 16
- Number of channels = 2 - Stereo

stopcapture *index*
Stop capture with a given *index*, index can be obtained with:

```
info capture
```

memsave *addr size file*
save to disk virtual memory dump starting at *addr* of size *size*.

pmemsave *addr size file*
save to disk physical memory dump starting at *addr* of size *size*.

boot_set *bootdevicelist*
Define new values for the boot device list. Those values will override the values specified on the command line through the `-boot` option.

The values that can be specified here depend on the machine type, but are the same that can be specified in the `-boot` command line option.

nmi *cpu*
Inject an NMI on the default CPU (x86/s390) or all CPUs (ppc64).

ringbuf_write *device data*
Write *data* to ring buffer character device *device*. *data* must be a UTF-8 string.

ringbuf_read *device*
Read and print up to *size* bytes from ring buffer character device *device*. Certain non-printable characters are printed `\uXXXX`, where `XXXX` is the character code in hexadecimal. Character \ is printed \\. Bug: can screw up when the buffer contains invalid UTF-8 sequences, NUL characters, after the ring buffer lost data, and when reading stops because the size limit is reached.

announce_self
Trigger a round of GARP/RARP broadcasts; this is useful for explicitly updating the network infrastructure after a reconfiguration or some forms of migration. The timings of the round are set by the migration announce parameters. An optional comma separated *interfaces* list restricts the announce to the named set of interfaces. An optional *id* can be used to start a separate announce timer and to change the parameters of it later.

migrate [-d] [-r] *uri*
Migrate the VM to *uri*.

-d
Start the migration process, but do not wait for its completion. To query an ongoing migration process, use "info migrate".

-r
Resume a paused postcopy migration.

migrate_cancel
Cancel the current VM migration.

migrate_continue *state*
Continue migration from the paused state *state*

migrate_incoming *uri*
Continue an incoming migration using the *uri* (that has the same syntax as the `-incoming` option).

migrate_recover *uri*
Continue a paused incoming postcopy migration using the *uri*.

migrate_pause
Pause an ongoing migration. Currently it only supports postcopy.

migrate_set_capability *capability state*
Enable/Disable the usage of a capability *capability* for migration.

migrate_set_parameter *parameter value*
Set the parameter *parameter* for migration.

migrate_start_postcopy
Switch in-progress migration to postcopy mode. Ignored after the end of migration (or once already in postcopy).

x_colo_lost_heartbeat
Tell COLO that heartbeat is lost, a failover or takeover is needed.

client_migrate_info *protocol hostname port tls-port cert-subject*
Set migration information for remote display. This makes the server ask the client to automatically reconnect using the new parameters once migration finished successfully. Only implemented for SPICE.

dump-guest-memory [-p] *filename begin length*

dump-guest-memory [-z|-l|-s|-w] *filename*
Dump guest memory to *protocol*. The file can be processed with crash or gdb. Without `-z|-l|-s|-w`, the dump format is ELF.

-p
do paging to get guest's memory mapping.

-z
dump in kdump-compressed format, with zlib compression.

-l
dump in kdump-compressed format, with lzo compression.

-s
dump in kdump-compressed format, with snappy compression.

-R
when using kdump (-z, -l, -s), use raw rather than makedumpfile-flattened format

-w
dump in Windows crashdump format (can be used instead of ELF-dump converting), for Windows x64 guests with vmcoreinfo driver only

filename
dump file name.

begin
the starting physical address. It's optional, and should be specified together with *length*.

length
the memory size, in bytes. It's optional, and should be specified together with *begin*.

dump-skeys *filename*
Save guest storage keys to a file.

migration_mode *mode*
Enables or disables migration mode.

snapshot_blkdev
Snapshot device, using snapshot file as target if provided

snapshot_blkdev_internal
Take an internal snapshot on device if it support

snapshot_delete_blkdev_internal
Delete an internal snapshot on device if it support

drive_mirror
Start mirroring a block device's writes to a new destination, using the specified target.

drive_backup
Start a point-in-time copy of a block device to a specified target.

drive_add
Add drive to PCI storage controller.

pcie_aer_inject_error
Inject PCIe AER error

netdev_add
Add host network device.

netdev_del
Remove host network device.

object_add
Create QOM object.

object_del
Destroy QOM object.

hostfwd_add
Redirect TCP or UDP connections from host to guest (requires -net user).

hostfwd_remove
Remove host-to-guest TCP or UDP redirection.

balloon *value*
Request VM to change its memory allocation to *value* (in MB).

set_link *name* **[on|off]**
Switch link *name* on (i.e. up) or off (i.e. down).

watchdog_action
Change watchdog action.

nbd_server_start *host*:*port*
Start an NBD server on the given host and/or port. If the `-a` option is included, all of the virtual machine's block devices that have an inserted media on them are automatically exported; in this case, the `-w` option makes the devices writable too.

nbd_server_add *device* **[** *name* **]**
Export a block device through QEMU's NBD server, which must be started beforehand with `nbd_server_start`. The `-w` option makes the exported device writable too. The export name is controlled by *name*, defaulting to *device*.

nbd_server_remove [-f] *name*
Stop exporting a block device through QEMU's NBD server, which was previously started with `nbd_server_add`. The `-f` option forces the server to drop the export immediately even if clients are connected; otherwise the command fails unless there are no clients.

nbd_server_stop
Stop the QEMU embedded NBD server.

mce ***cpu bank status mcgstatus addr misc***
Inject an MCE on the given CPU (x86 only).

getfd ***fdname***
If a file descriptor is passed alongside this command using the SCM_RIGHTS mechanism on unix sockets, it is stored using the name *fdname* for later use by other monitor commands.

closefd ***fdname***
Close the file descriptor previously assigned to *fdname* using the `getfd` command. This is only needed if the file descriptor was never used by another monitor command.

block_set_io_throttle ***device bps bps_rd bps_wr iops iops_rd iops_wr***
Change I/O throttle limits for a block drive to *bps bps_rd bps_wr iops iops_rd iops_wr*. *device* can be a block device name, a qdev ID or a QOM path.

set_password [vnc | spice] password [-d display] [action-if-connected]
Change spice/vnc password. *display* can be used with 'vnc' to specify which display to set the password on. *action-if-connected* specifies what should happen in case a connection is established: *fail* makes the password change fail. *disconnect* changes the password and disconnects the client. *keep* changes the password and keeps the connection up. *keep* is the default.

expire_password [vnc | spice] expire-time [-d display]
Specify when a password for spice/vnc becomes invalid. *display* behaves the same as in `set_password`. *expire-time* accepts:

now
Invalidate password instantly.

never
Password stays valid forever.

+nsec
Password stays valid for *nsec* seconds starting now.

nsec
Password is invalidated at the given time. *nsec* are the seconds passed since 1970, i.e. unix epoch.

chardev-add ***args***
chardev-add accepts the same parameters as the -chardev command line switch.

chardev-change ***args***
chardev-change accepts existing chardev *id* and then the same arguments as the -chardev command line switch (except for "id").

chardev-remove ***id***
Removes the chardev *id*.

chardev-send-break ***id***
Send a break on the chardev *id*.

qemu-io ***device command***
Executes a qemu-io command on the given block device.

qom-list [***path***]
Print QOM properties of object at location *path*

qom-get ***path property***
Print QOM property *property* of object at location *path*

qom-set *path property value*
Set QOM property *property* of object at location *path* to value *value*

replay_break *icount*
Set replay breakpoint at instruction count *icount*. Execution stops when the specified instruction is reached. There can be at most one breakpoint. When breakpoint is set, any prior one is removed. The breakpoint may be set only in replay mode and only "in the future", i.e. at instruction counts greater than the current one. The current instruction count can be observed with `info replay`.

replay_delete_break
Remove replay breakpoint which was previously set with `replay_break`. The command is ignored when there are no replay breakpoints.

replay_seek *icount*
Automatically proceed to the instruction count *icount*, when replaying the execution. The command automatically loads nearest snapshot and replays the execution to find the desired instruction. When there is no preceding snapshot or the execution is not replayed, then the command fails. *icount* for the reference may be observed with `info replay` command.

calc_dirty_rate *second*
Start a round of dirty rate measurement with the period specified in *second*. The result of the dirty rate measurement may be observed with `info dirty_rate` command.

set_vcpu_dirty_limit
Set dirty page rate limit on virtual CPU, the information about all the virtual CPU dirty limit status can be observed with `info vcpu_dirty_limit` command.

cancel_vcpu_dirty_limit
Cancel dirty page rate limit on virtual CPU, the information about all the virtual CPU dirty limit status can be observed with `info vcpu_dirty_limit` command.

dumpdtb *filename*
Dump the FDT in dtb format to *filename*.

xen-event-inject *port*
Notify guest via event channel on port *port*.

xen-event-list
List event channels in the guest

info *subcommand*
Show various information about the system state.

info version
Show the version of QEMU.

info network
Show the network state.

info chardev
Show the character devices.

info block
Show info of one block device or all block devices.

info blockstats
Show block device statistics.

info block-jobs
Show progress of ongoing block device operations.

info registers
Show the cpu registers.

info lapic
Show local APIC state

info cpus
Show infos for each CPU.

info history
Show the command line history.

info irq
Show the interrupts statistics (if available).

info pic
Show PIC state.

info pci
Show PCI information.

info tlb
Show virtual to physical memory mappings.

info mem
Show the active virtual memory mappings.

info mtree
Show memory tree.

info jit
Show dynamic compiler info.

info opcount
Show dynamic compiler opcode counters

info sync-profile [-m|-n] [*max*]
Show synchronization profiling info, up to *max* entries (default: 10), sorted by total wait time.

-m
sort by mean wait time

-n
do not coalesce objects with the same call site

When different objects that share the same call site are coalesced, the "Object" field shows—enclosed in brackets—the number of objects being coalesced.

info kvm
Show KVM information.

info numa
Show NUMA information.

info usb
Show guest USB devices.

info usbhost
Show host USB devices.

info capture
Show capture information.

info snapshots
Show the currently saved VM snapshots.

info status
Show the current VM status (running|paused).

info mice
Show which guest mouse is receiving events.

info vnc
Show the vnc server status.

info spice
Show the spice server status.

info name
Show the current VM name.

info uuid
Show the current VM UUID.

info usernet
Show user network stack connection states.

info migrate
Show migration status.

info migrate_capabilities
Show current migration capabilities.

info migrate_parameters
Show current migration parameters.

info balloon
Show balloon information.

info qtree
Show device tree.

info qdm
Show qdev device model list.

info qom-tree
Show QOM composition tree.

info roms
Show roms.

info trace-events
Show available trace-events & their state.

info tpm
Show the TPM device.

info memdev
Show memory backends

info memory-devices
Show memory devices.

info iothreads
Show iothread's identifiers.

info rocker *name*
Show rocker switch.

info rocker-ports *name*-**ports**
Show rocker ports.

info rocker-of-dpa-flows *name* [*tbl_id*]
Show rocker OF-DPA flow tables.

info rocker-of-dpa-groups *name* [*type*]
Show rocker OF-DPA groups.

info skeys *address*
Display the value of a storage key (s390 only)

info cmma *address*
Display the values of the CMMA storage attributes for a range of pages (s390 only)

info dump
Display the latest dump status.

info ramblock
Dump all the ramblocks of the system.

info hotpluggable-cpus
Show information about hotpluggable CPUs

info vm-generation-id
Show Virtual Machine Generation ID

info memory_size_summary
Display the amount of initially allocated and present hotpluggable (if enabled) memory in bytes.

info sev
Show SEV information.

info replay
Display the record/replay information: mode and the current icount.

info dirty_rate
Display the vcpu dirty rate information.

info vcpu_dirty_limit
Display the vcpu dirty page limit information.

info sgx
Show intel SGX information.

info via
Show guest mos6522 VIA devices.

stats
Show runtime-collected statistics

info virtio
List all available virtio devices

info virtio-status *path*
Display status of a given virtio device

info virtio-queue-status *path queue*
Display status of a given virtio queue

info virtio-vhost-queue-status *path queue*
Display status of a given vhost queue

info virtio-queue-element *path queue* [*index*]
Display element of a given virtio queue

info cryptodev
Show the crypto devices.

2.6.2 Integer expressions

The monitor understands integers expressions for every integer argument. You can use register names to get the value of specifics CPU registers by prefixing them with *$*.

2.7 Disk Images

QEMU supports many disk image formats, including growable disk images (their size increase as non empty sectors are written), compressed and encrypted disk images.

2.7.1 Quick start for disk image creation

You can create a disk image with the command:

```
qemu-img create myimage.img mysize
```

where myimage.img is the disk image filename and mysize is its size in kilobytes. You can add an `M` suffix to give the size in megabytes and a `G` suffix for gigabytes.

See the `qemu-img` invocation documentation for more information.

2.7.2 Snapshot mode

If you use the option `-snapshot`, all disk images are considered as read only. When sectors in written, they are written in a temporary file created in `/tmp`. You can however force the write back to the raw disk images by using the `commit` monitor command (or C-a s in the serial console).

2.7.3 VM snapshots

VM snapshots are snapshots of the complete virtual machine including CPU state, RAM, device state and the content of all the writable disks. In order to use VM snapshots, you must have at least one non removable and writable block device using the `qcow2` disk image format. Normally this device is the first virtual hard drive.

Use the monitor command `savevm` to create a new VM snapshot or replace an existing one. A human readable name can be assigned to each snapshot in addition to its numerical ID.

Use `loadvm` to restore a VM snapshot and `delvm` to remove a VM snapshot. `info snapshots` lists the available snapshots with their associated information:

```
(qemu) info snapshots
Snapshot devices: hda
Snapshot list (from hda):
ID        TAG                 VM SIZE                DATE       VM CLOCK
1         start                   41M 2006-08-06 12:38:02   00:00:14.954
2                                 40M 2006-08-06 12:43:29   00:00:18.633
3         msys                    40M 2006-08-06 12:44:04   00:00:23.514
```

A VM snapshot is made of a VM state info (its size is shown in `info snapshots`) and a snapshot of every writable disk image. The VM state info is stored in the first `qcow2` non removable and writable block device. The disk image snapshots are stored in every disk image. The size of a snapshot in a disk image is difficult to evaluate and is not shown by `info snapshots` because the associated disk sectors are shared among all the snapshots to save disk space (otherwise each snapshot would need a full copy of all the disk images).

When using the (unrelated) `-snapshot` option (*Snapshot mode*), you can always make VM snapshots, but they are deleted as soon as you exit QEMU.

VM snapshots currently have the following known limitations:

- They cannot cope with removable devices if they are removed or inserted after a snapshot is done.
- A few device drivers still have incomplete snapshot support so their state is not saved or restored properly (in particular USB).

2.7.4 Disk image file formats

QEMU supports many image file formats that can be used with VMs as well as with any of the tools (like `qemu-img`). This includes the preferred formats raw and qcow2 as well as formats that are supported for compatibility with older QEMU versions or other hypervisors.

Depending on the image format, different options can be passed to `qemu-img create` and `qemu-img convert` using the `-o` option. This section describes each format and the options that are supported for it.

raw

Raw disk image format. This format has the advantage of being simple and easily exportable to all other emulators. If your file system supports *holes* (for example in ext2 or ext3 on Linux or NTFS on Windows), then only the written sectors will reserve space. Use `qemu-img info` to know the real size used by the image or `ls -ls` on Unix/Linux.

Supported options:

preallocation

Preallocation mode (allowed values: `off`, `falloc`, `full`). `falloc` mode preallocates space for image by calling `posix_fallocate()`. `full` mode preallocates space for image by writing data to underlying storage. This data may or may not be zero, depending on the storage location.

qcow2

QEMU image format, the most versatile format. Use it to have smaller images (useful if your filesystem does not supports holes, for example on Windows), zlib based compression and support of multiple VM snapshots.

Supported options:

compat

Determines the qcow2 version to use. `compat=0.10` uses the traditional image format that can be read by any QEMU since 0.10. `compat=1.1` enables image format extensions that only QEMU 1.1 and newer understand (this is the default). Amongst others, this includes zero clusters, which allow efficient copy-on-read for sparse images.

backing_file

File name of a base image (see `create` subcommand)

backing_fmt

Image format of the base image

encryption

This option is deprecated and equivalent to `encrypt.format=aes`

encrypt.format

If this is set to `luks`, it requests that the qcow2 payload (not qcow2 header) be encrypted using the LUKS format. The passphrase to use to unlock the LUKS key slot is given by the `encrypt.key-secret` parameter. LUKS encryption parameters can be tuned with the other `encrypt.*` parameters.

If this is set to `aes`, the image is encrypted with 128-bit AES-CBC. The encryption key is given by the `encrypt.key-secret` parameter. This encryption format is considered to be flawed by modern cryptography standards, suffering from a number of design problems:

- The AES-CBC cipher is used with predictable initialization vectors based on the sector number. This makes it vulnerable to chosen plaintext attacks which can reveal the existence of encrypted data.
- The user passphrase is directly used as the encryption key. A poorly chosen or short passphrase will compromise the security of the encryption.
- In the event of the passphrase being compromised there is no way to change the passphrase to protect data in any qcow images. The files must be cloned, using a different encryption passphrase in the new file. The original file must then be securely erased using a program like shred, though even this is ineffective with many modern storage technologies.

The use of this is no longer supported in system emulators. Support only remains in the command line utilities, for the purposes of data liberation and interoperability with old versions of QEMU. The `luks` format should be used instead.

encrypt.key-secret

Provides the ID of a `secret` object that contains the passphrase (`encrypt.format=luks`) or encryption key (`encrypt.format=aes`).

encrypt.cipher-alg

Name of the cipher algorithm and key length. Currently defaults to `aes-256`. Only used when `encrypt.format=luks`.

encrypt.cipher-mode

Name of the encryption mode to use. Currently defaults to `xts`. Only used when `encrypt.format=luks`.

encrypt.ivgen-alg

Name of the initialization vector generator algorithm. Currently defaults to `plain64`. Only used when `encrypt.format=luks`.

encrypt.ivgen-hash-alg

Name of the hash algorithm to use with the initialization vector generator (if required). Defaults to `sha256`. Only used when `encrypt.format=luks`.

encrypt.hash-alg

Name of the hash algorithm to use for PBKDF algorithm Defaults to `sha256`. Only used when `encrypt.format=luks`.

encrypt.iter-time

Amount of time, in milliseconds, to use for PBKDF algorithm per key slot. Defaults to `2000`. Only used when `encrypt.format=luks`.

cluster_size

Changes the qcow2 cluster size (must be between 512 and 2M). Smaller cluster sizes can improve the image file size whereas larger cluster sizes generally provide better performance.

preallocation

Preallocation mode (allowed values: `off`, `metadata`, `falloc`, `full`). An image with preallocated metadata is initially larger but can improve performance when the image needs to grow. `falloc` and `full` preallocations are like the same options of `raw` format, but sets up metadata also.

lazy_refcounts

If this option is set to `on`, reference count updates are postponed with the goal of avoiding metadata I/O and improving performance. This is particularly interesting with `cache=writethrough` which doesn't batch metadata updates. The tradeoff is that after a host crash, the reference count tables must be rebuilt, i.e. on the next open an (automatic) `qemu-img check -r all` is required, which may take some time.

This option can only be enabled if `compat=1.1` is specified.

nocow

If this option is set to `on`, it will turn off COW of the file. It's only valid on btrfs, no effect on other file systems.

Btrfs has low performance when hosting a VM image file, even more when the guest on the VM also using btrfs as file system. Turning off COW is a way to mitigate this bad performance. Generally there are two ways to turn off COW on btrfs:

- Disable it by mounting with nodatacow, then all newly created files will be NOCOW.
- For an empty file, add the NOCOW file attribute. That's what this option does.

Note: this option is only valid to new or empty files. If there is an existing file which is COW and has data blocks already, it couldn't be changed to NOCOW by setting `nocow=on`. One can issue `lsattr filename` to check if the NOCOW flag is set or not (Capital 'C' is NOCOW flag).

qed

Old QEMU image format with support for backing files and compact image files (when your filesystem or transport medium does not support holes).

When converting QED images to qcow2, you might want to consider using the `lazy_refcounts=on` option to get a more QED-like behaviour.

Supported options:

backing_file

File name of a base image (see `create` subcommand).

backing_fmt

Image file format of backing file (optional). Useful if the format cannot be autodetected because it has no header, like some vhd/vpc files.

cluster_size

Changes the cluster size (must be power-of-2 between 4K and 64K). Smaller cluster sizes can improve the image file size whereas larger cluster sizes generally provide better performance.

table_size

Changes the number of clusters per L1/L2 table (must be power-of-2 between 1 and 16). There is normally no need to change this value but this option can between used for performance benchmarking.

qcow

Old QEMU image format with support for backing files, compact image files, encryption and compression.

Supported options:

backing_file

File name of a base image (see `create` subcommand)

encryption

This option is deprecated and equivalent to `encrypt.format=aes`

encrypt.format

If this is set to `aes`, the image is encrypted with 128-bit AES-CBC. The encryption key is given by the `encrypt.key-secret` parameter. This encryption format is considered to be flawed by modern cryptography standards, suffering from a number of design problems enumerated previously against the `qcow2` image format.

The use of this is no longer supported in system emulators. Support only remains in the command line utilities, for the purposes of data liberation and interoperability with old versions of QEMU.

Users requiring native encryption should use the `qcow2` format instead with `encrypt.format=luks`.

encrypt.key-secret

Provides the ID of a `secret` object that contains the encryption key (`encrypt.format=aes`).

luks

LUKS v1 encryption format, compatible with Linux dm-crypt/cryptsetup

Supported options:

key-secret

Provides the ID of a `secret` object that contains the passphrase.

cipher-alg

Name of the cipher algorithm and key length. Currently defaults to `aes-256`.

cipher-mode

Name of the encryption mode to use. Currently defaults to `xts`.

ivgen-alg

Name of the initialization vector generator algorithm. Currently defaults to `plain64`.

ivgen-hash-alg

Name of the hash algorithm to use with the initialization vector generator (if required). Defaults to `sha256`.

hash-alg

Name of the hash algorithm to use for PBKDF algorithm Defaults to `sha256`.

iter-time

Amount of time, in milliseconds, to use for PBKDF algorithm per key slot. Defaults to `2000`.

vdi

VirtualBox 1.1 compatible image format.

Supported options:

static

If this option is set to `on`, the image is created with metadata preallocation.

vmdk

VMware 3 and 4 compatible image format.

Supported options:

backing_file

File name of a base image (see `create` subcommand).

compat6

Create a VMDK version 6 image (instead of version 4)

hwversion

Specify vmdk virtual hardware version. Compat6 flag cannot be enabled if hwversion is specified.

subformat

Specifies which VMDK subformat to use. Valid options are `monolithicSparse` (default), `monolithicFlat`, `twoGbMaxExtentSparse`, `twoGbMaxExtentFlat` and `streamOptimized`.

vpc

VirtualPC compatible image format (VHD).

Supported options:

subformat

Specifies which VHD subformat to use. Valid options are `dynamic` (default) and `fixed`.

VHDX

Hyper-V compatible image format (VHDX).

Supported options:

subformat

Specifies which VHDX subformat to use. Valid options are `dynamic` (default) and `fixed`.

block_state_zero

Force use of payload blocks of type 'ZERO'. Can be set to `on` (default) or `off`. When set to `off`, new blocks will be created as `PAYLOAD_BLOCK_NOT_PRESENT`, which means parsers are free to return arbitrary data for those blocks. Do not set to `off` when using `qemu-img convert` with `subformat=dynamic`.

block_size

Block size; min 1 MB, max 256 MB. 0 means auto-calculate based on image size.

log_size

Log size; min 1 MB.

2.7.5 Read-only formats

More disk image file formats are supported in a read-only mode.

bochs

Bochs images of `growing` type.

cloop

Linux Compressed Loop image, useful only to reuse directly compressed CD-ROM images present for example in the Knoppix CD-ROMs.

dmg
: Apple disk image.

parallels
: Parallels disk image format.

2.7.6 Using host drives

In addition to disk image files, QEMU can directly access host devices. We describe here the usage for QEMU version >= 0.8.3.

Linux

On Linux, you can directly use the host device filename instead of a disk image filename provided you have enough privileges to access it. For example, use `/dev/cdrom` to access to the CDROM.

CD
: You can specify a CDROM device even if no CDROM is loaded. QEMU has specific code to detect CDROM insertion or removal. CDROM ejection by the guest OS is supported. Currently only data CDs are supported.

Floppy
: You can specify a floppy device even if no floppy is loaded. Floppy removal is currently not detected accurately (if you change floppy without doing floppy access while the floppy is not loaded, the guest OS will think that the same floppy is loaded). Use of the host's floppy device is deprecated, and support for it will be removed in a future release.

Hard disks
: Hard disks can be used. Normally you must specify the whole disk (`/dev/hdb` instead of `/dev/hdb1`) so that the guest OS can see it as a partitioned disk. WARNING: unless you know what you do, it is better to only make READ-ONLY accesses to the hard disk otherwise you may corrupt your host data (use the `-snapshot` command line option or modify the device permissions accordingly).

Zoned block devices
: Zoned block devices can be passed through to the guest if the emulated storage controller supports zoned storage. Use `--blockdev host_device, node-name=drive0,filename=/dev/nullb0,cache.direct=on` to pass through `/dev/nullb0` as `drive0`.

Windows

CD
: The preferred syntax is the drive letter (e.g. `d:`). The alternate syntax `\\.\d:` is supported. `/dev/cdrom` is supported as an alias to the first CDROM drive.

  Currently there is no specific code to handle removable media, so it is better to use the `change` or `eject` monitor commands to change or eject media.

Hard disks
: Hard disks can be used with the syntax: `\\.\PhysicalDriveN` where *N* is the drive number (0 is the first hard disk).

  WARNING: unless you know what you do, it is better to only make READ-ONLY accesses to the hard disk otherwise you may corrupt your host data (use the `-snapshot` command line so that the modifications are written in a temporary file).

Mac OS X

`/dev/cdrom` is an alias to the first CDROM.

Currently there is no specific code to handle removable media, so it is better to use the `change` or `eject` monitor commands to change or eject media.

2.7.7 Virtual FAT disk images

QEMU can automatically create a virtual FAT disk image from a directory tree. In order to use it, just type:

```
qemu-system-x86_64 linux.img -hdb fat:/my_directory
```

Then you access access to all the files in the `/my_directory` directory without having to copy them in a disk image or to export them via SAMBA or NFS. The default access is *read-only*.

Floppies can be emulated with the `:floppy:` option:

```
qemu-system-x86_64 linux.img -fda fat:floppy:/my_directory
```

A read/write support is available for testing (beta stage) with the `:rw:` option:

```
qemu-system-x86_64 linux.img -fda fat:floppy:rw:/my_directory
```

What you should *never* do:

- use non-ASCII filenames
- use "-snapshot" together with ":rw:"
- expect it to work when loadvm'ing
- write to the FAT directory on the host system while accessing it with the guest system

2.7.8 NBD access

QEMU can access directly to block device exported using the Network Block Device protocol.

```
qemu-system-x86_64 linux.img -hdb nbd://my_nbd_server.mydomain.org:1024/
```

If the NBD server is located on the same host, you can use an unix socket instead of an inet socket:

```
qemu-system-x86_64 linux.img -hdb nbd+unix://?socket=/tmp/my_socket
```

In this case, the block device must be exported using `qemu-nbd`:

```
qemu-nbd --socket=/tmp/my_socket my_disk.qcow2
```

The use of `qemu-nbd` allows sharing of a disk between several guests:

```
qemu-nbd --socket=/tmp/my_socket --share=2 my_disk.qcow2
```

and then you can use it with two guests:

```
qemu-system-x86_64 linux1.img -hdb nbd+unix://?socket=/tmp/my_socket
qemu-system-x86_64 linux2.img -hdb nbd+unix://?socket=/tmp/my_socket
```

If the `nbd-server` uses named exports (supported since NBD 2.9.18, or with QEMU's own embedded NBD server), you must specify an export name in the URI:

```
qemu-system-x86_64 -cdrom nbd://localhost/debian-500-ppc-netinst
qemu-system-x86_64 -cdrom nbd://localhost/openSUSE-11.1-ppc-netinst
```

The URI syntax for NBD is supported since QEMU 1.3. An alternative syntax is also available. Here are some example of the older syntax:

```
qemu-system-x86_64 linux.img -hdb nbd:my_nbd_server.mydomain.org:1024
qemu-system-x86_64 linux2.img -hdb nbd:unix:/tmp/my_socket
qemu-system-x86_64 -cdrom nbd:localhost:10809:exportname=debian-500-ppc-netinst
```

2.7.9 iSCSI LUNs

iSCSI is a popular protocol used to access SCSI devices across a computer network.

There are two different ways iSCSI devices can be used by QEMU.

The first method is to mount the iSCSI LUN on the host, and make it appear as any other ordinary SCSI device on the host and then to access this device as a /dev/sd device from QEMU. How to do this differs between host OSes.

The second method involves using the iSCSI initiator that is built into QEMU. This provides a mechanism that works the same way regardless of which host OS you are running QEMU on. This section will describe this second method of using iSCSI together with QEMU.

In QEMU, iSCSI devices are described using special iSCSI URLs. URL syntax:

```
iscsi://[<username>[%<password>]@]<host>[:<port>]/<target-iqn-name>/<lun>
```

Username and password are optional and only used if your target is set up using CHAP authentication for access control. Alternatively the username and password can also be set via environment variables to have these not show up in the process list:

```
export LIBISCSI_CHAP_USERNAME=<username>
export LIBISCSI_CHAP_PASSWORD=<password>
iscsi://<host>/<target-iqn-name>/<lun>
```

Various session related parameters can be set via special options, either in a configuration file provided via '-readconfig' or directly on the command line.

If the initiator-name is not specified qemu will use a default name of 'iqn.2008-11.org.linux-kvm[:<uuid>'] where <uuid> is the UUID of the virtual machine. If the UUID is not specified qemu will use 'iqn.2008-11.org.linux-kvm[:<name>'] where <name> is the name of the virtual machine.

Setting a specific initiator name to use when logging in to the target:

```
-iscsi initiator-name=iqn.qemu.test:my-initiator
```

Controlling which type of header digest to negotiate with the target:

```
-iscsi header-digest=CRC32C|CRC32C-NONE|NONE-CRC32C|NONE
```

These can also be set via a configuration file:

```
[iscsi]
  user = "CHAP username"
  password = "CHAP password"
  initiator-name = "iqn.qemu.test:my-initiator"
  # header digest is one of CRC32C|CRC32C-NONE|NONE-CRC32C|NONE
  header-digest = "CRC32C"
```

Setting the target name allows different options for different targets:

```
[iscsi "iqn.target.name"]
  user = "CHAP username"
  password = "CHAP password"
  initiator-name = "iqn.qemu.test:my-initiator"
  # header digest is one of CRC32C|CRC32C-NONE|NONE-CRC32C|NONE
  header-digest = "CRC32C"
```

How to use a configuration file to set iSCSI configuration options:

```
cat >iscsi.conf <<EOF
[iscsi]
  user = "me"
  password = "my password"
  initiator-name = "iqn.qemu.test:my-initiator"
  header-digest = "CRC32C"
EOF

qemu-system-x86_64 -drive file=iscsi://127.0.0.1/iqn.qemu.test/1 \
  -readconfig iscsi.conf
```

How to set up a simple iSCSI target on loopback and access it via QEMU: this example shows how to set up an iSCSI target with one CDROM and one DISK using the Linux STGT software target. This target is available on Red Hat based systems as the package 'scsi-target-utils'.

```
tgtd --iscsi portal=127.0.0.1:3260
tgtadm --lld iscsi --op new --mode target --tid 1 -T iqn.qemu.test
tgtadm --lld iscsi --mode logicalunit --op new --tid 1 --lun 1 \
    -b /IMAGES/disk.img --device-type=disk
tgtadm --lld iscsi --mode logicalunit --op new --tid 1 --lun 2 \
    -b /IMAGES/cd.iso --device-type=cd
tgtadm --lld iscsi --op bind --mode target --tid 1 -I ALL

qemu-system-x86_64 -iscsi initiator-name=iqn.qemu.test:my-initiator \
  -boot d -drive file=iscsi://127.0.0.1/iqn.qemu.test/1 \
  -cdrom iscsi://127.0.0.1/iqn.qemu.test/2
```

2.7.10 GlusterFS disk images

GlusterFS is a user space distributed file system.

You can boot from the GlusterFS disk image with the command:

URI:

```
qemu-system-x86_64 -drive file=gluster[+TYPE]://[HOST}[:PORT]]/VOLUME/PATH
                           [?socket=...][,file.debug=9][,file.logfile=...]
```

JSON:

```
qemu-system-x86_64 'json:{"driver":"qcow2",
                       "file":{"driver":"gluster",
                               "volume":"testvol","path":"a.img","debug":9,
↪"logfile":"...",
                               "server":[{"type":"tcp","host":"...","port":"..."},
                                         {"type":"unix","socket":"..."}]}}'
```

gluster is the protocol.

TYPE specifies the transport type used to connect to gluster management daemon (glusterd). Valid transport types are tcp and unix. In the URI form, if a transport type isn't specified, then tcp type is assumed.

HOST specifies the server where the volume file specification for the given volume resides. This can be either a hostname or an ipv4 address. If transport type is unix, then *HOST* field should not be specified. Instead *socket* field needs to be populated with the path to unix domain socket.

PORT is the port number on which glusterd is listening. This is optional and if not specified, it defaults to port 24007. If the transport type is unix, then *PORT* should not be specified.

VOLUME is the name of the gluster volume which contains the disk image.

PATH is the path to the actual disk image that resides on gluster volume.

debug is the logging level of the gluster protocol driver. Debug levels are 0-9, with 9 being the most verbose, and 0 representing no debugging output. The default level is 4. The current logging levels defined in the gluster source are 0 - None, 1 - Emergency, 2 - Alert, 3 - Critical, 4 - Error, 5 - Warning, 6 - Notice, 7 - Info, 8 - Debug, 9 - Trace

logfile is a commandline option to mention log file path which helps in logging to the specified file and also help in persisting the gfapi logs. The default is stderr.

You can create a GlusterFS disk image with the command:

```
qemu-img create gluster://HOST/VOLUME/PATH SIZE
```

Examples

```
qemu-system-x86_64 -drive file=gluster://1.2.3.4/testvol/a.img
qemu-system-x86_64 -drive file=gluster+tcp://1.2.3.4/testvol/a.img
qemu-system-x86_64 -drive file=gluster+tcp://1.2.3.4:24007/testvol/dir/a.img
qemu-system-x86_64 -drive file=gluster+tcp://[1:2:3:4:5:6:7:8]/testvol/dir/a.img
qemu-system-x86_64 -drive file=gluster+tcp://[1:2:3:4:5:6:7:8]:24007/testvol/dir/a.img
qemu-system-x86_64 -drive file=gluster+tcp://server.domain.com:24007/testvol/dir/a.img
qemu-system-x86_64 -drive file=gluster+unix:///testvol/dir/a.img?socket=/tmp/glusterd.
↪socket
qemu-system-x86_64 -drive file=gluster://1.2.3.4/testvol/a.img,file.debug=9,file.
↪logfile=/var/log/qemu-gluster.log
qemu-system-x86_64 'json:{"driver":"qcow2",
                        "file":{"driver":"gluster",
                                "volume":"testvol","path":"a.img",
                                "debug":9,"logfile":"/var/log/qemu-gluster.log",
                                "server":[{"type":"tcp","host":"1.2.3.4","port":24007},
                                          {"type":"unix","socket":"/var/run/glusterd.
↪socket"}]}}'
qemu-system-x86_64 -drive driver=qcow2,file.driver=gluster,file.volume=testvol,file.
↪path=/path/a.img,
                                   file.debug=9,file.logfile=/var/log/qemu-gluster.log,
                                   file.server.0.type=tcp,file.server.0.host=1.2.3.4,
↪file.server.0.port=24007,
                                   file.server.1.type=unix,file.server.1.socket=/var/
↪run/glusterd.socket
```

2.7.11 Secure Shell (ssh) disk images

You can access disk images located on a remote ssh server by using the ssh protocol:

```
qemu-system-x86_64 -drive file=ssh://[USER@]SERVER[:PORT]/PATH[?host_key_check=HOST_KEY_
↪CHECK]
```

Alternative syntax using properties:

```
qemu-system-x86_64 -drive file.driver=ssh[,file.user=USER],file.host=SERVER[,file.
↪port=PORT],file.path=PATH[,file.host_key_check=HOST_KEY_CHECK]
```

ssh is the protocol.

USER is the remote user. If not specified, then the local username is tried.

SERVER specifies the remote ssh server. Any ssh server can be used, but it must implement the sftp-server protocol. Most Unix/Linux systems should work without requiring any extra configuration.

PORT is the port number on which sshd is listening. By default the standard ssh port (22) is used.

PATH is the path to the disk image.

The optional *HOST_KEY_CHECK* parameter controls how the remote host's key is checked. The default is `yes` which means to use the local `.ssh/known_hosts` file. Setting this to `no` turns off known-hosts checking. Or you can check that the host key matches a specific fingerprint. The fingerprint can be provided in `md5`, `sha1`, or `sha256` format, however, it is strongly recommended to only use `sha256`, since the other options are considered insecure by modern standards. The fingerprint value must be given as a hex encoded string:

```
host_key_check=sha256:04ce2ae89ff4295a6b9c4111640bdcb3297858ee55cb434d9dd88796e93aa795
```

The key string may optionally contain ":" separators between each pair of hex digits.

The `$HOME/.ssh/known_hosts` file contains the base64 encoded host keys. These can be converted into the format needed for QEMU using a command such as:

```
$ for key in `grep 10.33.8.112 known_hosts | awk '{print $3}'`
  do
    echo $key | base64 -d | sha256sum
  done
  6c3aa525beda9dc83eadfbd7e5ba7d976ecb59575d1633c87cd06ed2ed6e366f  -
  12214fd9ea5b408086f98ecccd9958609bd9ac7c0ea316734006bc7818b45dc8  -
  d36420137bcbd101209ef70c3b15dc07362fbe0fa53c5b135eba6e6afa82f0ce  -
```

Note that there can be multiple keys present per host, each with different key ciphers. Care is needed to pick the key fingerprint that matches the cipher QEMU will negotiate with the remote server.

Currently authentication must be done using ssh-agent. Other authentication methods may be supported in future.

Note: Many ssh servers do not support an `fsync`-style operation. The ssh driver cannot guarantee that disk flush requests are obeyed, and this causes a risk of disk corruption if the remote server or network goes down during writes. The driver will print a warning when `fsync` is not supported:

```
warning: ssh server ssh.example.com:22 does not support fsync
```

With sufficiently new versions of libssh and OpenSSH, `fsync` is supported.

2.7.12 NVMe disk images

NVM Express (NVMe) storage controllers can be accessed directly by a userspace driver in QEMU. This bypasses the host kernel file system and block layers while retaining QEMU block layer functionalities, such as block jobs, I/O throttling, image formats, etc. Disk I/O performance is typically higher than with `-drive file=/dev/sda` using either thread pool or linux-aio.

The controller will be exclusively used by the QEMU process once started. To be able to share storage between multiple VMs and other applications on the host, please use the file based protocols.

Before starting QEMU, bind the host NVMe controller to the host vfio-pci driver. For example:

```
# modprobe vfio-pci
# lspci -n -s 0000:06:0d.0
06:0d.0 0401: 1102:0002 (rev 08)
# echo 0000:06:0d.0 > /sys/bus/pci/devices/0000:06:0d.0/driver/unbind
# echo 1102 0002 > /sys/bus/pci/drivers/vfio-pci/new_id

# qemu-system-x86_64 -drive file=nvme://HOST:BUS:SLOT.FUNC/NAMESPACE
```

Alternative syntax using properties:

```
qemu-system-x86_64 -drive file.driver=nvme,file.device=HOST:BUS:SLOT.FUNC,file.
↪namespace=NAMESPACE
```

HOST:*BUS*:*SLOT.FUNC* is the NVMe controller's PCI device address on the host.

NAMESPACE is the NVMe namespace number, starting from 1.

2.7.13 Disk image file locking

By default, QEMU tries to protect image files from unexpected concurrent access, as long as it's supported by the block protocol driver and host operating system. If multiple QEMU processes (including QEMU emulators and utilities) try to open the same image with conflicting accessing modes, all but the first one will get an error.

This feature is currently supported by the file protocol on Linux with the Open File Descriptor (OFD) locking API, and can be configured to fall back to POSIX locking if the POSIX host doesn't support Linux OFD locking.

To explicitly enable image locking, specify "locking=on" in the file protocol driver options. If OFD locking is not possible, a warning will be printed and the POSIX locking API will be used. In this case there is a risk that the lock will get silently lost when doing hot plugging and block jobs, due to the shortcomings of the POSIX locking API.

QEMU transparently handles lock handover during shared storage migration. For shared virtual disk images between multiple VMs, the "share-rw" device option should be used.

By default, the guest has exclusive write access to its disk image. If the guest can safely share the disk image with other writers the `-device ...,share-rw=on` parameter can be used. This is only safe if the guest is running software, such as a cluster file system, that coordinates disk accesses to avoid corruption.

Note that share-rw=on only declares the guest's ability to share the disk. Some QEMU features, such as image file formats, require exclusive write access to the disk image and this is unaffected by the share-rw=on option.

Alternatively, locking can be fully disabled by "locking=off" block device option. In the command line, the option is usually in the form of "file.locking=off" as the protocol driver is normally placed as a "file" child under a format driver. For example:

```
-blockdev driver=qcow2,file.filename=/path/to/image,file.locking=off,file.driver=file
```

To check if image locking is active, check the output of the "lslocks" command on host and see if there are locks held by the QEMU process on the image file. More than one byte could be locked by the QEMU instance, each byte of which reflects a particular permission that is acquired or protected by the running block driver.

2.7.14 Filter drivers

QEMU supports several filter drivers, which don't store any data, but perform some additional tasks, hooking io requests.

preallocate

The preallocate filter driver is intended to be inserted between format and protocol nodes and preallocates some additional space (expanding the protocol file) when writing past the file's end. This can be useful for file-systems with slow allocation.

Supported options:

prealloc-align

On preallocation, align the file length to this value (in bytes), default 1M.

prealloc-size

How much to preallocate (in bytes), default 128M.

2.8 QEMU virtio-net standby (net_failover)

This document explains the setup and usage of virtio-net standby feature which is used to create a net_failover pair of devices.

The general idea is that we have a pair of devices, a (vfio-)pci and a virtio-net device. Before migration the vfio device is unplugged and data flows through the virtio-net device, on the target side another vfio-pci device is plugged in to take over the data-path. In the guest the net_failover kernel module will pair net devices with the same MAC address.

The two devices are called primary and standby device. The fast hardware based networking device is called the primary device and the virtio-net device is the standby device.

2.8.1 Restrictions

Currently only PCIe devices are allowed as primary devices, this restriction can be lifted in the future with enhanced QEMU support. Also, only networking devices are allowed as primary device. The user needs to ensure that primary and standby devices are not plugged into the same PCIe slot.

2.8.2 Usecase

Virtio-net standby allows easy migration while using a passed-through fast networking device by falling back to a virtio-net device for the duration of the migration. It is like a simple version of a bond, the difference is that it requires no configuration in the guest. When a guest is live-migrated to another host QEMU will unplug the primary device via the PCIe based hotplug handler and traffic will go through the virtio-net device. On the target system the primary device will be automatically plugged back and the net_failover module registers it again as the primary device.

2.8.3 Usage

The primary device can be hotplugged or be part of the startup configuration

-device virtio-net-pci,netdev=hostnet1,id=net1,mac=52:54:00:6f:55:cc,
bus=root2,failover=on

With the parameter failover=on the VIRTIO_NET_F_STANDBY feature will be enabled.

-device vfio-pci,host=5e:00.2,id=hostdev0,bus=root1,failover_pair_id=net1

failover_pair_id references the id of the virtio-net standby device. This is only for pairing the devices within QEMU. The guest kernel module net_failover will match devices with identical MAC addresses.

2.8.4 Hotplug

Both primary and standby device can be hotplugged via the QEMU monitor. Note that if the virtio-net device is plugged first a warning will be issued that it couldn't find the primary device.

2.8.5 Migration

A new migration state wait-unplug was added for this feature. If failover primary devices are present in the configuration, migration will go into this state. It will wait until the device unplug is completed in the guest and then move into active state. On the target system the primary devices will be automatically hotplugged when the feature bit was negotiated for the virtio-net standby device.

2.9 Direct Linux Boot

This section explains how to launch a Linux kernel inside QEMU without having to make a full bootable image. It is very useful for fast Linux kernel testing.

The syntax is:

```
qemu-system-x86_64 -kernel bzImage -hda rootdisk.img -append "root=/dev/hda"
```

Use `-kernel` to provide the Linux kernel image and `-append` to give the kernel command line arguments. The `-initrd` option can be used to provide an INITRD image.

If you do not need graphical output, you can disable it and redirect the virtual serial port and the QEMU monitor to the console with the `-nographic` option. The typical command line is:

```
qemu-system-x86_64 -kernel bzImage -hda rootdisk.img                    -append "root=/dev/
↪hda console=ttyS0" -nographic
```

Use Ctrl-a c to switch between the serial console and the monitor (see *Keys in the graphical frontends*).

2.10 Generic Loader

The 'loader' device allows the user to load multiple images or values into QEMU at startup.

2.10.1 Loading Data into Memory Values

The loader device allows memory values to be set from the command line. This can be done by following the syntax below:

```
-device loader,addr=<addr>,data=<data>,data-len=<data-len> \
                [,data-be=<data-be>][,cpu-num=<cpu-num>]
```

<addr>
The address to store the data in.

<data>
The value to be written to the address. The maximum size of the data is 8 bytes.

<data-len>
The length of the data in bytes. This argument must be included if the data argument is.

<data-be>
Set to true if the data to be stored on the guest should be written as big endian data. The default is to write little endian data.

<cpu-num>
The number of the CPU's address space where the data should be loaded. If not specified the address space of the first CPU is used.

All values are parsed using the standard QemuOps parsing. This allows the user to specify any values in any format supported. By default the values will be parsed as decimal. To use hex values the user should prefix the number with a '0x'.

An example of loading value 0x8000000e to address 0xfd1a0104 is:

```
-device loader,addr=0xfd1a0104,data=0x8000000e,data-len=4
```

2.10.2 Setting a CPU's Program Counter

The loader device allows the CPU's PC to be set from the command line. This can be done by following the syntax below:

```
-device loader,addr=<addr>,cpu-num=<cpu-num>
```

<addr>
The value to use as the CPU's PC.

<cpu-num>
The number of the CPU whose PC should be set to the specified value.

All values are parsed using the standard QemuOpts parsing. This allows the user to specify any values in any format supported. By default the values will be parsed as decimal. To use hex values the user should prefix the number with a '0x'.

An example of setting CPU 0's PC to 0x8000 is:

```
-device loader,addr=0x8000,cpu-num=0
```

2.10.3 Loading Files

The loader device also allows files to be loaded into memory. It can load ELF, U-Boot, and Intel HEX executable formats as well as raw images. The syntax is shown below:

-device loader,file=<file>[,addr=<addr>][,cpu-num=<cpu-num>][,force-raw=<raw>]

<file>
A file to be loaded into memory

<addr>
The memory address where the file should be loaded. This is required for raw images and ignored for non-raw files.

<cpu-num>
This specifies the CPU that should be used. This is an optional argument and will cause the CPU's PC to be set to the memory address where the raw file is loaded or the entry point specified in the executable format header. This option should only be used for the boot image. This will also cause the image to be written to the specified CPU's address space. If not specified, the default is CPU 0.

<force-raw>
Setting 'force-raw=on' forces the file to be treated as a raw image. This can be used to load supported executable formats as if they were raw.

All values are parsed using the standard QemuOpts parsing. This allows the user to specify any values in any format supported. By default the values will be parsed as decimal. To use hex values the user should prefix the number with a '0x'.

An example of loading an ELF file which CPU0 will boot is shown below:

```
-device loader,file=./images/boot.elf,cpu-num=0
```

2.10.4 Restrictions and ToDos

At the moment it is just assumed that if you specify a cpu-num then you want to set the PC as well. This might not always be the case. In future the internal state 'set_pc' (which exists in the generic loader now) should be exposed to the user so that they can choose if the PC is set or not.

2.11 Guest Loader

The guest loader is similar to the `generic-loader` although it is aimed at a particular use case of loading hypervisor guests. This is useful for debugging hypervisors without having to jump through the hoops of firmware and boot-loaders.

The guest loader does two things:

- load blobs (kernels and initial ram disks) into memory
- sets platform FDT data so hypervisors can find and boot them

This is what is typically done by a boot-loader like grub using its multi-boot capability. A typical example would look like:

```
qemu-system-x86_64 -kernel ~/xen.git/xen/xen   -append "dom0_mem=1G,max:1G loglvl=all␣
↪guest_loglvl=all"   -device guest-loader,addr=0x42000000,kernel=Image,bootargs="root=/
↪dev/sda2 ro console=hvc0 earlyprintk=xen"   -device guest-loader,addr=0x47000000,
↪initrd=rootfs.cpio
```

In the above example the Xen hypervisor is loaded by the -kernel parameter and passed its boot arguments via -append. The Dom0 guest is loaded into the areas of memory. Each blob will get `/chosen/module@<addr>` entry in the FDT to indicate its location and size. Additional information can be passed with by using additional arguments.

Currently the only supported machines which use FDT data to boot are the ARM and RiscV `virt` machines.

2.11.1 Arguments

The full syntax of the guest-loader is:

```
-device guest-loader,addr=<addr>[,kernel=<file>,[bootargs=<args>]][,initrd=<file>]
```

addr=<addr>
: This is mandatory and indicates the start address of the blob.

kernel|initrd=<file>
: Indicates the filename of the kernel or initrd blob. Both blobs will have the "multiboot,module" compatibility string as well as "multiboot,kernel" or "multiboot,ramdisk" as appropriate.

bootargs=<args>
: This is an optional field for kernel blobs which will pass command like via the `/chosen/module@<addr>/bootargs` node.

2.12 QEMU Barrier Client

Generally, mouse and keyboard are grabbed through the QEMU video interface emulation.

But when we want to use a video graphic adapter via a PCI passthrough there is no way to provide the keyboard and mouse inputs to the VM except by plugging a second set of mouse and keyboard to the host or by installing a KVM software in the guest OS.

The QEMU Barrier client avoids this by implementing directly the Barrier protocol into QEMU.

Barrier is a KVM (Keyboard-Video-Mouse) software forked from Symless's synergy 1.9 codebase.

This protocol is enabled by adding an input-barrier object to QEMU.

Syntax:

```
input-barrier,id=<object-id>,name=<guest display name>
[,server=<barrier server address>][,port=<barrier server port>]
[,x-origin=<x-origin>][,y-origin=<y-origin>]
[,width=<width>][,height=<height>]
```

The object can be added on the QEMU command line, for instance with:

```
-object input-barrier,id=barrier0,name=VM-1
```

where VM-1 is the name the display configured in the Barrier server on the host providing the mouse and the keyboard events.

by default <barrier server address> is localhost, <port> is 24800, <x-origin> and <y-origin> are set to 0, <width> and <height> to 1920 and 1080.

If the Barrier server is stopped QEMU needs to be reconnected manually, by removing and re-adding the input-barrier object, for instance with the help of the HMP monitor:

```
(qemu) object_del barrier0
(qemu) object_add input-barrier,id=barrier0,name=VM-1
```

2.13 VNC security

The VNC server capability provides access to the graphical console of the guest VM across the network. This has a number of security considerations depending on the deployment scenarios.

2.13.1 Without passwords

The simplest VNC server setup does not include any form of authentication. For this setup it is recommended to restrict it to listen on a UNIX domain socket only. For example

```
qemu-system-x86_64 [...OPTIONS...] -vnc unix:/home/joebloggs/.qemu-myvm-vnc
```

This ensures that only users on local box with read/write access to that path can access the VNC server. To securely access the VNC server from a remote machine, a combination of netcat+ssh can be used to provide a secure tunnel.

2.13.2 With passwords

The VNC protocol has limited support for password based authentication. Since the protocol limits passwords to 8 characters it should not be considered to provide high security. The password can be fairly easily brute-forced by a client making repeat connections. For this reason, a VNC server using password authentication should be restricted to only listen on the loopback interface or UNIX domain sockets. Password authentication is not supported when operating in FIPS 140-2 compliance mode as it requires the use of the DES cipher. Password authentication is requested with the password option, and then once QEMU is running the password is set with the monitor. Until the monitor is used to set the password all clients will be rejected.

```
qemu-system-x86_64 [...OPTIONS...] -vnc :1,password=on -monitor stdio
(qemu) change vnc password
Password: ****
(qemu)
```

2.13.3 With x509 certificates

The QEMU VNC server also implements the VeNCrypt extension allowing use of TLS for encryption of the session, and x509 certificates for authentication. The use of x509 certificates is strongly recommended, because TLS on its own is susceptible to man-in-the-middle attacks. Basic x509 certificate support provides a secure session, but no authentication. This allows any client to connect, and provides an encrypted session.

```
qemu-system-x86_64 [...OPTIONS...]   -object tls-creds-x509,id=tls0,dir=/etc/pki/qemu,
↪endpoint=server,verify-peer=off   -vnc :1,tls-creds=tls0 -monitor stdio
```

In the above example /etc/pki/qemu should contain at least three files, ca-cert.pem, server-cert.pem and server-key.pem. Unprivileged users will want to use a private directory, for example $HOME/.pki/qemu. NB the server-key.pem file should be protected with file mode 0600 to only be readable by the user owning it.

2.13.4 With x509 certificates and client verification

Certificates can also provide a means to authenticate the client connecting. The server will request that the client provide a certificate, which it will then validate against the CA certificate. This is a good choice if deploying in an environment with a private internal certificate authority. It uses the same syntax as previously, but with `verify-peer` set to `on` instead.

```
qemu-system-x86_64 [...OPTIONS...]   -object tls-creds-x509,id=tls0,dir=/etc/pki/qemu,
↪endpoint=server,verify-peer=on   -vnc :1,tls-creds=tls0 -monitor stdio
```

2.13.5 With x509 certificates, client verification and passwords

Finally, the previous method can be combined with VNC password authentication to provide two layers of authentication for clients.

```
qemu-system-x86_64 [...OPTIONS...]   -object tls-creds-x509,id=tls0,dir=/etc/pki/qemu,
↪endpoint=server,verify-peer=on   -vnc :1,tls-creds=tls0,password=on -monitor stdio
(qemu) change vnc password
Password: ****
(qemu)
```

2.13.6 With SASL authentication

The SASL authentication method is a VNC extension, that provides an easily extendable, pluggable authentication method. This allows for integration with a wide range of authentication mechanisms, such as PAM, GSSAPI/Kerberos, LDAP, SQL databases, one-time keys and more. The strength of the authentication depends on the exact mechanism configured. If the chosen mechanism also provides a SSF layer, then it will encrypt the datastream as well.

Refer to the later docs on how to choose the exact SASL mechanism used for authentication, but assuming use of one supporting SSF, then QEMU can be launched with:

```
qemu-system-x86_64 [...OPTIONS...] -vnc :1,sasl=on -monitor stdio
```

2.13.7 With x509 certificates and SASL authentication

If the desired SASL authentication mechanism does not supported SSF layers, then it is strongly advised to run it in combination with TLS and x509 certificates. This provides securely encrypted data stream, avoiding risk of compromising of the security credentials. This can be enabled, by combining the 'sasl' option with the aforementioned TLS + x509 options:

```
qemu-system-x86_64 [...OPTIONS...]   -object tls-creds-x509,id=tls0,dir=/etc/pki/qemu,
↪endpoint=server,verify-peer=on   -vnc :1,tls-creds=tls0,sasl=on -monitor stdio
```

2.13.8 Configuring SASL mechanisms

The following documentation assumes use of the Cyrus SASL implementation on a Linux host, but the principles should apply to any other SASL implementation or host. When SASL is enabled, the mechanism configuration will be loaded from system default SASL service config /etc/sasl2/qemu.conf. If running QEMU as an unprivileged user, an environment variable SASL_CONF_PATH can be used to make it search alternate locations for the service config file.

If the TLS option is enabled for VNC, then it will provide session encryption, otherwise the SASL mechanism will have to provide encryption. In the latter case the list of possible plugins that can be used is drastically reduced. In fact only the GSSAPI SASL mechanism provides an acceptable level of security by modern standards. Previous versions of QEMU

referred to the DIGEST-MD5 mechanism, however, it has multiple serious flaws described in detail in RFC 6331 and thus should never be used any more. The SCRAM-SHA-256 mechanism provides a simple username/password auth facility similar to DIGEST-MD5, but does not support session encryption, so can only be used in combination with TLS.

When not using TLS the recommended configuration is

```
mech_list: gssapi
keytab: /etc/qemu/krb5.tab
```

This says to use the 'GSSAPI' mechanism with the Kerberos v5 protocol, with the server principal stored in /etc/qemu/krb5.tab. For this to work the administrator of your KDC must generate a Kerberos principal for the server, with a name of 'qemu/somehost.example.com@EXAMPLE.COM' replacing 'somehost.example.com' with the fully qualified host name of the machine running QEMU, and 'EXAMPLE.COM' with the Kerberos Realm.

When using TLS, if username+password authentication is desired, then a reasonable configuration is

```
mech_list: scram-sha-256
sasldb_path: /etc/qemu/passwd.db
```

The `saslpasswd2` program can be used to populate the `passwd.db` file with accounts. Note that the `passwd.db` file stores passwords in clear text.

Other SASL configurations will be left as an exercise for the reader. Note that all mechanisms, except GSSAPI, should be combined with use of TLS to ensure a secure data channel.

2.14 TLS setup for network services

Almost all network services in QEMU have the ability to use TLS for session data encryption, along with x509 certificates for simple client authentication. What follows is a description of how to generate certificates suitable for usage with QEMU, and applies to the VNC server, character devices with the TCP backend, NBD server and client, and migration server and client.

At a high level, QEMU requires certificates and private keys to be provided in PEM format. Aside from the core fields, the certificates should include various extension data sets, including v3 basic constraints data, key purpose, key usage and subject alt name.

The GnuTLS package includes a command called `certtool` which can be used to easily generate certificates and keys in the required format with expected data present. Alternatively a certificate management service may be used.

At a minimum it is necessary to setup a certificate authority, and issue certificates to each server. If using x509 certificates for authentication, then each client will also need to be issued a certificate.

Assuming that the QEMU network services will only ever be exposed to clients on a private intranet, there is no need to use a commercial certificate authority to create certificates. A self-signed CA is sufficient, and in fact likely to be more secure since it removes the ability of malicious 3rd parties to trick the CA into mis-issuing certs for impersonating your services. The only likely exception where a commercial CA might be desirable is if enabling the VNC websockets server and exposing it directly to remote browser clients. In such a case it might be useful to use a commercial CA to avoid needing to install custom CA certs in the web browsers.

The recommendation is for the server to keep its certificates in either `/etc/pki/qemu` or for unprivileged users in `$HOME/.pki/qemu`.

2.14.1 Setup the Certificate Authority

This step only needs to be performed once per organization / organizational unit. First the CA needs a private key. This key must be kept VERY secret and secure. If this key is compromised the entire trust chain of the certificates issued with it is lost.

```
# certtool --generate-privkey > ca-key.pem
```

To generate a self-signed certificate requires one core piece of information, the name of the organization. A template file `ca.info` should be populated with the desired data to avoid having to deal with interactive prompts from certtool:

```
# cat > ca.info <<EOF
cn = Name of your organization
ca
cert_signing_key
EOF
# certtool --generate-self-signed \
           --load-privkey ca-key.pem \
           --template ca.info \
           --outfile ca-cert.pem
```

The `ca` keyword in the template sets the v3 basic constraints extension to indicate this certificate is for a CA, while `cert_signing_key` sets the key usage extension to indicate this will be used for signing other keys. The generated `ca-cert.pem` file should be copied to all servers and clients wishing to utilize TLS support in the VNC server. The `ca-key.pem` must not be disclosed/copied anywhere except the host responsible for issuing certificates.

2.14.2 Issuing server certificates

Each server (or host) needs to be issued with a key and certificate. When connecting the certificate is sent to the client which validates it against the CA certificate. The core pieces of information for a server certificate are the hostnames and/or IP addresses that will be used by clients when connecting. The hostname / IP address that the client specifies when connecting will be validated against the hostname(s) and IP address(es) recorded in the server certificate, and if no match is found the client will close the connection.

Thus it is recommended that the server certificate include both the fully qualified and unqualified hostnames. If the server will have permanently assigned IP address(es), and clients are likely to use them when connecting, they may also be included in the certificate. Both IPv4 and IPv6 addresses are supported. Historically certificates only included 1 hostname in the `CN` field, however, usage of this field for validation is now deprecated. Instead modern TLS clients will validate against the Subject Alt Name extension data, which allows for multiple entries. In the future usage of the `CN` field may be discontinued entirely, so providing SAN extension data is strongly recommended.

On the host holding the CA, create template files containing the information for each server, and use it to issue server certificates.

```
# cat > server-hostNNN.info <<EOF
organization = Name  of your organization
cn = hostNNN.foo.example.com
dns_name = hostNNN
dns_name = hostNNN.foo.example.com
ip_address = 10.0.1.87
ip_address = 192.8.0.92
ip_address = 2620:0:cafe::87
ip_address = 2001:24::92
tls_www_server
```

(continues on next page)

(continued from previous page)

```
encryption_key
signing_key
EOF
# certtool --generate-privkey > server-hostNNN-key.pem
# certtool --generate-certificate \
           --load-ca-certificate ca-cert.pem \
           --load-ca-privkey ca-key.pem \
           --load-privkey server-hostNNN-key.pem \
           --template server-hostNNN.info \
           --outfile server-hostNNN-cert.pem
```

The `dns_name` and `ip_address` fields in the template are setting the subject alt name extension data. The `tls_www_server` keyword is the key purpose extension to indicate this certificate is intended for usage in a web server. Although QEMU network services are not in fact HTTP servers (except for VNC websockets), setting this key purpose is still recommended. The `encryption_key` and `signing_key` keyword is the key usage extension to indicate this certificate is intended for usage in the data session.

The `server-hostNNN-key.pem` and `server-hostNNN-cert.pem` files should now be securely copied to the server for which they were generated, and renamed to `server-key.pem` and `server-cert.pem` when added to the `/etc/pki/qemu` directory on the target host. The `server-key.pem` file is security sensitive and should be kept protected with file mode 0600 to prevent disclosure.

2.14.3 Issuing client certificates

The QEMU x509 TLS credential setup defaults to enabling client verification using certificates, providing a simple authentication mechanism. If this default is used, each client also needs to be issued a certificate. The client certificate contains enough metadata to uniquely identify the client with the scope of the certificate authority. The client certificate would typically include fields for organization, state, city, building, etc.

Once again on the host holding the CA, create template files containing the information for each client, and use it to issue client certificates.

```
# cat > client-hostNNN.info <<EOF
country = GB
state = London
locality = City Of London
organization = Name of your organization
cn = hostNNN.foo.example.com
tls_www_client
encryption_key
signing_key
EOF
# certtool --generate-privkey > client-hostNNN-key.pem
# certtool --generate-certificate \
           --load-ca-certificate ca-cert.pem \
           --load-ca-privkey ca-key.pem \
           --load-privkey client-hostNNN-key.pem \
           --template client-hostNNN.info \
           --outfile client-hostNNN-cert.pem
```

The subject alt name extension data is not required for clients, so the `dns_name` and `ip_address` fields are not included. The `tls_www_client` keyword is the key purpose extension to indicate this certificate is intended for usage in a web client. Although QEMU network clients are not in fact HTTP clients, setting this key purpose is still recommended.

The encryption_key and signing_key keyword is the key usage extension to indicate this certificate is intended for usage in the data session.

The client-hostNNN-key.pem and client-hostNNN-cert.pem files should now be securely copied to the client for which they were generated, and renamed to client-key.pem and client-cert.pem when added to the /etc/pki/qemu directory on the target host. The client-key.pem file is security sensitive and should be kept protected with file mode 0600 to prevent disclosure.

If a single host is going to be using TLS in both a client and server role, it is possible to create a single certificate to cover both roles. This would be quite common for the migration and NBD services, where a QEMU process will be started by accepting a TLS protected incoming migration, and later itself be migrated out to another host. To generate a single certificate, simply include the template data from both the client and server instructions in one.

```
# cat > both-hostNNN.info <<EOF
country = GB
state = London
locality = City Of London
organization = Name of your organization
cn = hostNNN.foo.example.com
dns_name = hostNNN
dns_name = hostNNN.foo.example.com
ip_address = 10.0.1.87
ip_address = 192.8.0.92
ip_address = 2620:0:cafe::87
ip_address = 2001:24::92
tls_www_server
tls_www_client
encryption_key
signing_key
EOF
# certtool --generate-privkey > both-hostNNN-key.pem
# certtool --generate-certificate \
           --load-ca-certificate ca-cert.pem \
           --load-ca-privkey ca-key.pem \
           --load-privkey both-hostNNN-key.pem \
           --template both-hostNNN.info \
           --outfile both-hostNNN-cert.pem
```

When copying the PEM files to the target host, save them twice, once as server-cert.pem and server-key.pem, and again as client-cert.pem and client-key.pem.

2.14.4 TLS x509 credential configuration

QEMU has a standard mechanism for loading x509 credentials that will be used for network services and clients. It requires specifying the tls-creds-x509 class name to the --object command line argument for the system emulators. Each set of credentials loaded should be given a unique string identifier via the id parameter. A single set of TLS credentials can be used for multiple network backends, so VNC, migration, NBD, character devices can all share the same credentials. Note, however, that credentials for use in a client endpoint must be loaded separately from those used in a server endpoint.

When specifying the object, the dir parameters specifies which directory contains the credential files. This directory is expected to contain files with the names mentioned previously, ca-cert.pem, server-key.pem, server-cert.pem, client-key.pem and client-cert.pem as appropriate. It is also possible to include a set of pre-generated Diffie-Hellman (DH) parameters in a file dh-params.pem, which can be created using the certtool

`--generate-dh-params` command. If omitted, QEMU will dynamically generate DH parameters when loading the credentials.

The `endpoint` parameter indicates whether the credentials will be used for a network client or server, and determines which PEM files are loaded.

The `verify` parameter determines whether x509 certificate validation should be performed. This defaults to enabled, meaning clients will always validate the server hostname against the certificate subject alt name fields and/or CN field. It also means that servers will request that clients provide a certificate and validate them. Verification should never be turned off for client endpoints, however, it may be turned off for server endpoints if an alternative mechanism is used to authenticate clients. For example, the VNC server can use SASL to authenticate clients instead.

To load server credentials with client certificate validation enabled

```
qemu-system-x86_64 -object tls-creds-x509,id=tls0,dir=/etc/pki/qemu,endpoint=server
```

while to load client credentials use

```
qemu-system-x86_64 -object tls-creds-x509,id=tls0,dir=/etc/pki/qemu,endpoint=client
```

Network services which support TLS will all have a `tls-creds` parameter which expects the ID of the TLS credentials object. For example with VNC:

```
qemu-system-x86_64 -vnc 0.0.0.0:0,tls-creds=tls0
```

2.14.5 TLS Pre-Shared Keys (PSK)

Instead of using certificates, you may also use TLS Pre-Shared Keys (TLS-PSK). This can be simpler to set up than certificates but is less scalable.

Use the GnuTLS `psktool` program to generate a `keys.psk` file containing one or more usernames and random keys:

```
mkdir -m 0700 /tmp/keys
psktool -u rich -p /tmp/keys/keys.psk
```

TLS-enabled servers such as `qemu-nbd` can use this directory like so:

```
qemu-nbd \
  -t -x / \
  --object tls-creds-psk,id=tls0,endpoint=server,dir=/tmp/keys \
  --tls-creds tls0 \
  image.qcow2
```

When connecting from a qemu-based client you must specify the directory containing `keys.psk` and an optional username (defaults to "qemu"):

```
qemu-img info \
  --object tls-creds-psk,id=tls0,dir=/tmp/keys,username=rich,endpoint=client \
  --image-opts \
  file.driver=nbd,file.host=localhost,file.port=10809,file.tls-creds=tls0,file.export=/
```

2.15 Providing secret data to QEMU

There are a variety of objects in QEMU which require secret data to be provided by the administrator or management application. For example, network block devices often require a password, LUKS block devices require a passphrase to unlock key material, remote desktop services require an access password. QEMU has a general purpose mechanism for providing secret data to QEMU in a secure manner, using the `secret` object type.

At startup this can be done using the `-object secret,...` command line argument. At runtime this can be done using the `object_add` QMP / HMP monitor commands. The examples that follow will illustrate use of `-object` command lines, but they all apply equivalentely in QMP / HMP. When creating a `secret` object it must be given a unique ID string. This ID is then used to identify the object when configuring the thing which need the data.

2.15.1 INSECURE: Passing secrets as clear text inline

The following should never be done in a production environment or on a multi-user host. Command line arguments are usually visible in the process listings and are often collected in log files by system monitoring agents or bug reporting tools. QMP/HMP commands and their arguments are also often logged and attached to bug reports. This all risks compromising secrets that are passed inline.

For the convenience of people debugging / developing with QEMU, it is possible to pass secret data inline on the command line.

```
-object secret,id=secvnc0,data=87539319
```

Again it is possible to provide the data in base64 encoded format, which is particularly useful if the data contains binary characters that would clash with argument parsing.

```
-object secret,id=secvnc0,data=ODc1MzkzMTk=,format=base64
```

Note: base64 encoding does not provide any security benefit.

2.15.2 Passing secrets as clear text via a file

The simplest approach to providing data securely is to use a file to store the secret:

```
-object secret,id=secvnc0,file=vnc-password.txt
```

In this example the file `vnc-password.txt` contains the plain text secret data. It is important to note that the contents of the file are treated as an opaque blob. The entire raw file contents is used as the value, thus it is important not to mistakenly add any trailing newline character in the file if this newline is not intended to be part of the secret data.

In some cases it might be more convenient to pass the secret data in base64 format and have QEMU decode to get the raw bytes before use:

```
-object secret,id=sec0,file=vnc-password.txt,format=base64
```

The file should generally be given mode `0600` or `0400` permissions, and have its user/group ownership set to the same account that the QEMU process will be launched under. If using mandatory access control such as SELinux, then the file should be labelled to only grant access to the specific QEMU process that needs access. This will prevent other processes/users from compromising the secret data.

2.15.3 Passing secrets as cipher text inline

To address the insecurity of passing secrets inline as clear text, it is possible to configure a second secret as an AES key to use for decrypting the data.

The secret used as the AES key must always be configured using the file based storage mechanism:

```
-object secret,id=secmaster,file=masterkey.data,format=base64
```

In this case the `masterkey.data` file would be initialized with 32 cryptographically secure random bytes, which are then base64 encoded. The contents of this file will by used as an AES-256 key to encrypt the real secret that can now be safely passed to QEMU inline as cipher text

```
-object secret,id=secvnc0,keyid=secmaster,data=BASE64-CIPHERTEXT,iv=BASE64-IV,
↪format=base64
```

In this example `BASE64-CIPHERTEXT` is the result of AES-256-CBC encrypting the secret with `masterkey.data` and then base64 encoding the ciphertext. The `BASE64-IV` data is 16 random bytes which have been base64 encrypted. These bytes are used as the initialization vector for the AES-256-CBC value.

A single master key can be used to encrypt all subsequent secrets, **but it is critical that a different initialization vector is used for every secret**.

2.15.4 Passing secrets via the Linux keyring

The earlier mechanisms described are platform agnostic. If using QEMU on a Linux host, it is further possible to pass secrets to QEMU using the Linux keyring:

```
-object secret_keyring,id=secvnc0,serial=1729
```

This instructs QEMU to load data from the Linux keyring secret identified by the serial number `1729`. It is possible to combine use of the keyring with other features mentioned earlier such as base64 encoding:

```
-object secret_keyring,id=secvnc0,serial=1729,format=base64
```

and also encryption with a master key:

```
-object secret_keyring,id=secvnc0,keyid=secmaster,serial=1729,iv=BASE64-IV
```

2.15.5 Best practice

It is recommended for production deployments to use a master key secret, and then pass all subsequent inline secrets encrypted with the master key.

Each QEMU instance must have a distinct master key, and that must be generated from a cryptographically secure random data source. The master key should be deleted immediately upon QEMU shutdown. If passing the master key as a file, the key file must have access control rules applied that restrict access to just the one QEMU process that is intended to use it. Alternatively the Linux keyring can be used to pass the master key to QEMU.

The secrets for individual QEMU device backends must all then be encrypted with this master key.

This procedure helps ensure that the individual secrets for QEMU backends will not be compromised, even if `-object` CLI args or `object_add` monitor commands are collected in log files and attached to public bug support tickets. The only item that needs strongly protecting is the master key file.

2.16 Client authorization

When configuring a QEMU network backend with either TLS certificates or SASL authentication, access will be granted if the client successfully proves their identity. If the authorization identity database is scoped to the QEMU client this may be sufficient. It is common, however, for the identity database to be much broader and thus authentication alone does not enable sufficient access control. In this case QEMU provides a flexible system for enforcing finer grained authorization on clients post-authentication.

2.16.1 Identity providers

At the time of writing there are two authentication frameworks used by QEMU that emit an identity upon completion.

- TLS x509 certificate distinguished name.

  When configuring the QEMU backend as a network server with TLS, there are a choice of credentials to use. The most common scenario is to utilize x509 certificates. The simplest configuration only involves issuing certificates to the servers, allowing the client to avoid a MITM attack against their intended server.

  It is possible, however, to enable mutual verification by requiring that the client provide a certificate to the server to prove its own identity. This is done by setting the property `verify-peer=yes` on the `tls-creds-x509` object, which is in fact the default.

  When peer verification is enabled, client will need to be issued with a certificate by the same certificate authority as the server. If this is still not sufficiently strong access control the Distinguished Name of the certificate can be used as an identity in the QEMU authorization framework.

- SASL username.

  When configuring the QEMU backend as a network server with SASL, upon completion of the SASL authentication mechanism, a username will be provided. The format of this username will vary depending on the choice of mechanism configured for SASL. It might be a simple UNIX style user `joebloggs`, while if using Kerberos/GSSAPI it can have a realm attached `joebloggs@QEMU.ORG`. Whatever format the username is presented in, it can be used with the QEMU authorization framework.

2.16.2 Authorization drivers

The QEMU authorization framework is a general purpose design with choice of user customizable drivers. These are provided as objects that can be created at startup using the `-object` argument, or at runtime using the `object_add` monitor command.

Simple

This authorization driver provides a simple mechanism for granting access based on an exact match against a single identity. This is useful when it is known that only a single client is to be allowed access.

A possible use case would be when configuring QEMU for an incoming live migration. It is known exactly which source QEMU the migration is expected to arrive from. The x509 certificate associated with this source QEMU would thus be used as the identity to match against. Alternatively if the virtual machine is dedicated to a specific tenant, then the VNC server would be configured with SASL and the username of only that tenant listed.

To create an instance of this driver via QMP:

```
{
  "execute": "object-add",
  "arguments": {
    "qom-type": "authz-simple",
    "id": "authz0",
    "identity": "fred"
  }
}
```

Or via the command line

```
-object authz-simple,id=authz0,identity=fred
```

List

In some network backends it will be desirable to grant access to a range of clients. This authorization driver provides a list mechanism for granting access by matching identities against a list of permitted one. Each match rule has an associated policy and a catch all policy applies if no rule matches. The match can either be done as an exact string comparison, or can use the shell-like glob syntax, which allows for use of wildcards.

To create an instance of this class via QMP:

```
{
  "execute": "object-add",
  "arguments": {
    "qom-type": "authz-list",
    "id": "authz0",
    "rules": [
       { "match": "fred", "policy": "allow", "format": "exact" },
       { "match": "bob", "policy": "allow", "format": "exact" },
       { "match": "danb", "policy": "deny", "format": "exact" },
       { "match": "dan*", "policy": "allow", "format": "glob" }
    ],
    "policy": "deny"
  }
}
```

Due to the way this driver requires setting nested properties, creating it on the command line will require use of the JSON syntax for `-object`. In most cases, however, the next driver will be more suitable.

List file

This is a variant on the previous driver that allows for a more dynamic access control policy by storing the match rules in a standalone file that can be reloaded automatically upon change.

To create an instance of this class via QMP:

```
{
  "execute": "object-add",
  "arguments": {
    "qom-type": "authz-list-file",
    "id": "authz0",
```

(continues on next page)

(continued from previous page)

```
    "filename": "/etc/qemu/myvm-vnc.acl",
    "refresh": true
  }
}
```

If `refresh` is `yes`, inotify is used to monitor for changes to the file and auto-reload the rules.

The `myvm-vnc.acl` file should contain the match rules in a format that closely matches the previous driver:

```
{
  "rules": [
    { "match": "fred", "policy": "allow", "format": "exact" },
    { "match": "bob", "policy": "allow", "format": "exact" },
    { "match": "danb", "policy": "deny", "format": "exact" },
    { "match": "dan*", "policy": "allow", "format": "glob" }
  ],
  "policy": "deny"
}
```

The object can be created on the command line using

```
-object authz-list-file,id=authz0,\
        filename=/etc/qemu/myvm-vnc.acl,refresh=on
```

PAM

In some scenarios it might be desirable to integrate with authorization mechanisms that are implemented outside of QEMU. In order to allow maximum flexibility, QEMU provides a driver that uses the `PAM` framework.

To create an instance of this class via QMP:

```
{
  "execute": "object-add",
  "arguments": {
    "qom-type": "authz-pam",
    "id": "authz0",
    "parameters": {
      "service": "qemu-vnc-tls"
    }
  }
}
```

The driver only uses the PAM "account" verification subsystem. The above config would require a config file /etc/pam.d/qemu-vnc-tls. For a simple file lookup it would contain

```
account requisite  pam_listfile.so item=user sense=allow \
        file=/etc/qemu/vnc.allow
```

The external file would then contain a list of usernames. If x509 cert was being used as the username, a suitable entry would match the distinguished name:

```
CN=laptop.berrange.com,O=Berrange Home,L=London,ST=London,C=GB
```

On the command line it can be created using

```
-object authz-pam,id=authz0,service=qemu-vnc-tls
```

There are a variety of PAM plugins that can be used which are not illustrated here, and it is possible to implement brand new plugins using the PAM API.

2.16.3 Connecting backends

The authorization driver is created using the `-object` argument and then needs to be associated with a network service. The authorization driver object will be given a unique ID that needs to be referenced.

The property to set in the network service will vary depending on the type of identity to verify. By convention, any network server backend that uses TLS will provide `tls-authz` property, while any server using SASL will provide a `sasl-authz` property.

Thus an example using SASL and authorization for the VNC server would look like:

```
$QEMU --object authz-simple,id=authz0,identity=fred \
      --vnc 0.0.0.0:1,sasl,sasl-authz=authz0
```

While to validate both the x509 certificate and SASL username:

```
echo "CN=laptop.qemu.org,O=QEMU Project,L=London,ST=London,C=GB" >> tls.acl
$QEMU --object authz-simple,id=authz0,identity=fred \
      --object authz-list-file,id=authz1,filename=tls.acl \
      --object tls-creds-x509,id=tls0,dir=/etc/qemu/tls,verify-peer=yes \
      --vnc 0.0.0.0:1,sasl,sasl-authz=auth0,tls-creds=tls0,tls-authz=authz1
```

2.17 GDB usage

QEMU supports working with gdb via gdb's remote-connection facility (the "gdbstub"). This allows you to debug guest code in the same way that you might with a low-level debug facility like JTAG on real hardware. You can stop and start the virtual machine, examine state like registers and memory, and set breakpoints and watchpoints.

In order to use gdb, launch QEMU with the `-s` and `-S` options. The `-s` option will make QEMU listen for an incoming connection from gdb on TCP port 1234, and `-S` will make QEMU not start the guest until you tell it to from gdb. (If you want to specify which TCP port to use or to use something other than TCP for the gdbstub connection, use the `-gdb dev` option instead of `-s`. See *Using unix sockets* for an example.)

```
qemu-system-x86_64 -s -S -kernel bzImage -hda rootdisk.img -append "root=/dev/hda"
```

QEMU will launch but will silently wait for gdb to connect.

Then launch gdb on the 'vmlinux' executable:

```
> gdb vmlinux
```

In gdb, connect to QEMU:

```
(gdb) target remote localhost:1234
```

Then you can use gdb normally. For example, type 'c' to launch the kernel:

```
(gdb) c
```

Here are some useful tips in order to use gdb on system code:

1. Use `info reg` to display all the CPU registers.
2. Use `x/10i $eip` to display the code at the PC position.
3. Use `set architecture i8086` to dump 16 bit code. Then use `x/10i $cs*16+$eip` to dump the code at the PC position.

2.17.1 Breakpoint and Watchpoint support

While GDB can always fall back to inserting breakpoints into memory (if writable) other features are very much dependent on support of the accelerator. For TCG system emulation we advertise an infinite number of hardware assisted breakpoints and watchpoints. For other accelerators it will depend on if support has been added (see supports_guest_debug and related hooks in AccelOpsClass).

As TCG cannot track all memory accesses in user-mode there is no support for watchpoints.

2.17.2 Relocating code

On modern kernels confusion can be caused by code being relocated by features such as address space layout randomisation. To avoid confusion when debugging such things you either need to update gdb's view of where things are in memory or perhaps more trivially disable ASLR when booting the system.

2.17.3 Debugging user-space in system emulation

While it is technically possible to debug a user-space program running inside a system image, it does present challenges. Kernel preemption and execution mode changes between kernel and user mode can make it hard to follow what's going on. Unless you are specifically trying to debug some interaction between kernel and user-space you are better off running your guest program with gdb either in the guest or using a gdbserver exposed via a port to the outside world.

2.17.4 Debugging multicore machines

GDB's abstraction for debugging targets with multiple possible parallel flows of execution is a two layer one: it supports multiple "inferiors", each of which can have multiple "threads". When the QEMU machine has more than one CPU, QEMU exposes each CPU cluster as a separate "inferior", where each CPU within the cluster is a separate "thread". Most QEMU machine types have identical CPUs, so there is a single cluster which has all the CPUs in it. A few machine types are heterogeneous and have multiple clusters: for example the `sifive_u` machine has a cluster with one E51 core and a second cluster with four U54 cores. Here the E51 is the only thread in the first inferior, and the U54 cores are all threads in the second inferior.

When you connect gdb to the gdbstub, it will automatically connect to the first inferior; you can display the CPUs in this cluster using the gdb `info thread` command, and switch between them using gdb's usual thread-management commands.

For multi-cluster machines, unfortunately gdb does not by default handle multiple inferiors, and so you have to explicitly connect to them. First, you must connect with the `extended-remote` protocol, not `remote`:

```
(gdb) target extended-remote localhost:1234
```

Once connected, gdb will have a single inferior, for the first cluster. You need to create inferiors for the other clusters and attach to them, like this:

```
(gdb) add-inferior
Added inferior 2
(gdb) inferior 2
[Switching to inferior 2 [<null>] (<noexec>)]
(gdb) attach 2
Attaching to process 2
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x00000000 in ?? ()
```

Once you've done this, `info threads` will show CPUs in all the clusters you have attached to:

```
(gdb) info threads
  Id   Target Id         Frame
  1.1  Thread 1.1 (cortex-m33-arm-cpu cpu [running]) 0x00000000 in ?? ()
* 2.1  Thread 2.2 (cortex-m33-arm-cpu cpu [halted ]) 0x00000000 in ?? ()
```

You probably also want to set gdb to `schedule-multiple` mode, so that when you tell gdb to `continue` it resumes all CPUs, not just those in the cluster you are currently working on:

```
(gdb) set schedule-multiple on
```

2.17.5 Using unix sockets

An alternate method for connecting gdb to the QEMU gdbstub is to use a unix socket (if supported by your operating system). This is useful when running several tests in parallel, or if you do not have a known free TCP port (e.g. when running automated tests).

First create a chardev with the appropriate options, then instruct the gdbserver to use that device:

```
qemu-system-x86_64 -chardev socket,path=/tmp/gdb-socket,server=on,wait=off,id=gdb0 -gdb␣
↪chardev:gdb0 -S ...
```

Start gdb as before, but this time connect using the path to the socket:

```
(gdb) target remote /tmp/gdb-socket
```

Note that to use a unix socket for the connection you will need gdb version 9.0 or newer.

2.17.6 Advanced debugging options

Changing single-stepping behaviour

The default single stepping behavior is step with the IRQs and timer service routines off. It is set this way because when gdb executes a single step it expects to advance beyond the current instruction. With the IRQs and timer service routines on, a single step might jump into the one of the interrupt or exception vectors instead of executing the current instruction. This means you may hit the same breakpoint a number of times before executing the instruction gdb wants to have executed. Because there are rare circumstances where you want to single step into an interrupt vector the behavior can be controlled from GDB. There are three commands you can query and set the single step behavior:

`maintenance packet qqemu.sstepbits`
: This will display the MASK bits used to control the single stepping IE:

```
(gdb) maintenance packet qqemu.sstepbits
sending: "qqemu.sstepbits"
received: "ENABLE=1,NOIRQ=2,NOTIMER=4"
```

maintenance packet qqemu.sstep
This will display the current value of the mask used when single stepping IE:

```
(gdb) maintenance packet qqemu.sstep
sending: "qqemu.sstep"
received: "0x7"
```

maintenance packet Qqemu.sstep=HEX_VALUE
This will change the single step mask, so if wanted to enable IRQs on the single step, but not timers, you would use:

```
(gdb) maintenance packet Qqemu.sstep=0x5
sending: "qemu.sstep=0x5"
received: "OK"
```

Examining physical memory

Another feature that QEMU gdbstub provides is to toggle the memory GDB works with, by default GDB will show the current process memory respecting the virtual address translation.

If you want to examine/change the physical memory you can set the gdbstub to work with the physical memory rather with the virtual one.

The memory mode can be checked by sending the following command:

maintenance packet qqemu.PhyMemMode
This will return either 0 or 1, 1 indicates you are currently in the physical memory mode.

maintenance packet Qqemu.PhyMemMode:1
This will change the memory mode to physical memory.

maintenance packet Qqemu.PhyMemMode:0
This will change it back to normal memory mode.

2.17.7 Security considerations

Connecting to the GDB socket allows running arbitrary code inside the guest; in case of the TCG emulation, which is not considered a security boundary, this also means running arbitrary code on the host. Additionally, when debugging qemu-user, it allows directly downloading any file readable by QEMU from the host.

The GDB socket is not protected by authentication, authorization or encryption. It is therefore a responsibility of the user to make sure that only authorized clients can connect to it, e.g., by using a unix socket with proper permissions, or by opening a TCP socket only on interfaces that are not reachable by potential attackers.

2.18 Record/replay

Record/replay functions are used for the deterministic replay of qemu execution. Execution recording writes a non-deterministic events log, which can be later used for replaying the execution anywhere and for unlimited number of times. It also supports checkpointing for faster rewind to the specific replay moment. Execution replaying reads the log and replays all non-deterministic events including external input, hardware clocks, and interrupts.

Deterministic replay has the following features:

- Deterministically replays whole system execution and all contents of the memory, state of the hardware devices, clocks, and screen of the VM.
- Writes execution log into the file for later replaying for multiple times on different machines.
- Supports i386, x86_64, ARM, AArch64, Risc-V, MIPS, MIPS64, S390X, Alpha, PowerPC, PowerPC64, M68000, Microblaze, OpenRISC, SPARC, and Xtensa hardware platforms.
- Performs deterministic replay of all operations with keyboard and mouse input devices, serial ports, and network.

Usage of the record/replay:

- First, record the execution with the following command line:

```
qemu-system-x86_64 \
-icount shift=auto,rr=record,rrfile=replay.bin \
-drive file=disk.qcow2,if=none,snapshot,id=img-direct \
-drive driver=blkreplay,if=none,image=img-direct,id=img-blkreplay \
-device ide-hd,drive=img-blkreplay \
-netdev user,id=net1 -device rtl8139,netdev=net1 \
-object filter-replay,id=replay,netdev=net1
```

- After recording, you can replay it by using another command line:

```
qemu-system-x86_64 \
-icount shift=auto,rr=replay,rrfile=replay.bin \
-drive file=disk.qcow2,if=none,snapshot,id=img-direct \
-drive driver=blkreplay,if=none,image=img-direct,id=img-blkreplay \
-device ide-hd,drive=img-blkreplay \
-netdev user,id=net1 -device rtl8139,netdev=net1 \
-object filter-replay,id=replay,netdev=net1
```

  The only difference with recording is changing the rr option from record to replay.

- Block device images are not actually changed in the recording mode, because all of the changes are written to the temporary overlay file. This behavior is enabled by using blkreplay driver. It should be used for every enabled block device, as described in *Block devices* section.
- `-net none` option should be specified when network is not used, because QEMU adds network card by default. When network is needed, it should be configured explicitly with replay filter, as described in *Network devices* section.
- Interaction with audio devices and serial ports are recorded and replayed automatically when such devices are enabled.

2.18.1 Core idea

Record/replay system is based on saving and replaying non-deterministic events (e.g. keyboard input) and simulating deterministic ones (e.g. reading from HDD or memory of the VM). Saving only non-deterministic events makes log file smaller and simulation faster.

The following non-deterministic data from peripheral devices is saved into the log: mouse and keyboard input, network packets, audio controller input, serial port input, and hardware clocks (they are non-deterministic too, because their values are taken from the host machine). Inputs from simulated hardware, memory of VM, software interrupts, and execution of instructions are not saved into the log, because they are deterministic and can be replayed by simulating the behavior of virtual machine starting from initial state.

2.18.2 Instruction counting

QEMU should work in icount mode to use record/replay feature. icount was designed to allow deterministic execution in absence of external inputs of the virtual machine. Record/replay feature is enabled through `-icount` command-line option, making possible deterministic execution of the machine, interacting with user or network.

2.18.3 Block devices

Block devices record/replay module intercepts calls of bdrv coroutine functions at the top of block drivers stack. To record and replay block operations the drive must be configured as following:

```
-drive file=disk.qcow2,if=none,snapshot,id=img-direct
-drive driver=blkreplay,if=none,image=img-direct,id=img-blkreplay
-device ide-hd,drive=img-blkreplay
```

blkreplay driver should be inserted between disk image and virtual driver controller. Therefore all disk requests may be recorded and replayed.

2.18.4 Snapshotting

New VM snapshots may be created in replay mode. They can be used later to recover the desired VM state. All VM states created in replay mode are associated with the moment of time in the replay scenario. After recovering the VM state replay will start from that position.

Default starting snapshot name may be specified with icount field rrsnapshot as follows:

```
-icount shift=auto,rr=record,rrfile=replay.bin,rrsnapshot=snapshot_name
```

This snapshot is created at start of recording and restored at start of replaying. It also can be loaded while replaying to roll back the execution.

`snapshot` flag of the disk image must be removed to save the snapshots in the overlay (or original image) instead of using the temporary overlay.

```
-drive file=disk.ovl,if=none,id=img-direct
-drive driver=blkreplay,if=none,image=img-direct,id=img-blkreplay
-device ide-hd,drive=img-blkreplay
```

Use QEMU monitor to create additional snapshots. `savevm <name>` command created the snapshot and `loadvm <name>` restores it. To prevent corruption of the original disk image, use overlay files linked to the original images.

Therefore all new snapshots (including the starting one) will be saved in overlays and the original image remains unchanged.

When you need to use snapshots with diskless virtual machine, it must be started with "orphan" qcow2 image. This image will be used for storing VM snapshots. Here is the example of the command line for this:

```
qemu-system-x86_64 \
  -icount shift=auto,rr=replay,rrfile=record.bin,rrsnapshot=init \
  -net none -drive file=empty.qcow2,if=none,id=rr
```

`empty.qcow2` drive does not connected to any virtual block device and used for VM snapshots only.

2.18.5 Network devices

Record and replay for network interactions is performed with the network filter. Each backend must have its own instance of the replay filter as follows:

```
-netdev user,id=net1 -device rtl8139,netdev=net1
-object filter-replay,id=replay,netdev=net1
```

Replay network filter is used to record and replay network packets. While recording the virtual machine this filter puts all packets coming from the outer world into the log. In replay mode packets from the log are injected into the network device. All interactions with network backend in replay mode are disabled.

2.18.6 Audio devices

Audio data is recorded and replay automatically. The command line for recording and replaying must contain identical specifications of audio hardware, e.g.:

```
-audio pa,model=ac97
```

2.18.7 Serial ports

Serial ports input is recorded and replay automatically. The command lines for recording and replaying must contain identical number of ports in record and replay modes, but their backends may differ. E.g., `-serial stdio` in record mode, and `-serial null` in replay mode.

2.18.8 Reverse debugging

Reverse debugging allows "executing" the program in reverse direction. GDB remote protocol supports "reverse step" and "reverse continue" commands. The first one steps single instruction backwards in time, and the second one finds the last breakpoint in the past.

Recorded executions may be used to enable reverse debugging. QEMU can't execute the code in backwards direction, but can load a snapshot and replay forward to find the desired position or breakpoint.

The following GDB commands are supported:

- `reverse-stepi` (or `rsi`) - step one instruction backwards
- `reverse-continue` (or `rc`) - find last breakpoint in the past

Reverse step loads the nearest snapshot and replays the execution until the required instruction is met.

Reverse continue may include several passes of examining the execution between the snapshots. Each of the passes include the following steps:

1. loading the snapshot
2. replaying to examine the breakpoints
3. if breakpoint or watchpoint was met
    - loading the snapshot again
    - replaying to the required breakpoint
4. else
    - proceeding to the p.1 with the earlier snapshot

Therefore usage of the reverse debugging requires at least one snapshot created. This can be done by omitting `snapshot` option for the block drives and adding `rrsnapshot` for both record and replay command lines. See the *Snapshotting* section to learn more about running record/replay and creating the snapshot in these modes.

When `rrsnapshot` is not used, then snapshot named `start_debugging` created in temporary overlay. This allows using reverse debugging, but with temporary snapshots (existing within the session).

2.19 Managed start up options

In system mode emulation, it's possible to create a VM in a paused state using the `-S` command line option. In this state the machine is completely initialized according to command line options and ready to execute VM code but VCPU threads are not executing any code. The VM state in this paused state depends on the way QEMU was started. It could be in:

- initial state (after reset/power on state)
- with direct kernel loading, the initial state could be amended to execute code loaded by QEMU in the VM's RAM and with incoming migration
- with incoming migration, initial state will be amended with the migrated machine state after migration completes

This paused state is typically used by users to query machine state and/or additionally configure the machine (by hotplugging devices) in runtime before allowing VM code to run.

However, at the `-S` pause point, it's impossible to configure options that affect initial VM creation (like: `-smp`/`-m`/`-numa` ...) or cold plug devices. The experimental `--preconfig` command line option allows pausing QEMU before the initial VM creation, in a "preconfig" state, where additional queries and configuration can be performed via QMP before moving on to the resulting configuration startup. In the preconfig state, QEMU only allows a limited set of commands over the QMP monitor, where the commands do not depend on an initialized machine, including but not limited to:

- `qmp_capabilities`
- `query-qmp-schema`
- `query-commands`
- `query-status`
- `x-exit-preconfig`

2.20 Managing device boot order with bootindex properties

QEMU can tell QEMU-aware guest firmware (like the x86 PC BIOS) which order it should look for a bootable OS on which devices. A simple way to set this order is to use the `-boot order=` option, but you can also do this more flexibly, by setting a `bootindex` property on the individual block or net devices you specify on the QEMU command line.

The `bootindex` properties are used to determine the order in which firmware will consider devices for booting the guest OS. If the `bootindex` property is not set for a device, it gets the lowest boot priority. There is no particular order in which devices with no `bootindex` property set will be considered for booting, but they will still be bootable.

Some guest machine types (for instance the s390x machines) do not support `-boot order=`; on those machines you must always use `bootindex` properties.

There is no way to set a `bootindex` property if you are using a short-form option like `-hda` or `-cdrom`, so to use `bootindex` properties you will need to expand out those options into long-form `-drive` and `-device` option pairs.

2.20.1 Example

Let's assume we have a QEMU machine with two NICs (virtio, e1000) and two disks (IDE, virtio):

```
qemu-system-x86_64 -drive file=disk1.img,if=none,id=disk1 \
               -device ide-hd,drive=disk1,bootindex=4 \
               -drive file=disk2.img,if=none,id=disk2 \
               -device virtio-blk-pci,drive=disk2,bootindex=3 \
               -netdev type=user,id=net0 \
               -device virtio-net-pci,netdev=net0,bootindex=2 \
               -netdev type=user,id=net1 \
               -device e1000,netdev=net1,bootindex=1
```

Given the command above, firmware should try to boot from the e1000 NIC first. If this fails, it should try the virtio NIC next; if this fails too, it should try the virtio disk, and then the IDE disk.

2.20.2 Limitations

Some firmware has limitations on which devices can be considered for booting. For instance, the PC BIOS boot specification allows only one disk to be bootable. If boot from disk fails for some reason, the BIOS won't retry booting from other disk. It can still try to boot from floppy or net, though.

Sometimes, firmware cannot map the device path QEMU wants firmware to boot from to a boot method. It doesn't happen for devices the firmware can natively boot from, but if firmware relies on an option ROM for booting, and the same option ROM is used for booting from more then one device, the firmware may not be able to ask the option ROM to boot from a particular device reliably. For instance with the PC BIOS, if a SCSI HBA has three bootable devices target1, target3, target5 connected to it, the option ROM will have a boot method for each of them, but it is not possible to map from boot method back to a specific target. This is a shortcoming of the PC BIOS boot specification.

2.20.3 Mixing bootindex and boot order parameters

Note that it does not make sense to use the bootindex property together with the `-boot order=...` (or `-boot once=...`) parameter. The guest firmware implementations normally either support the one or the other, but not both parameters at the same time. Mixing them will result in undefined behavior, and thus the guest firmware will likely not boot from the expected devices.

2.21 Virtual CPU hotplug

A complete example of vCPU hotplug (and hot-unplug) using QMP `device_add` and `device_del`.

2.21.1 vCPU hotplug

(1) Launch QEMU as follows (note that the "maxcpus" is mandatory to allow vCPU hotplug):

```
$ qemu-system-x86_64 -display none -no-user-config -m 2048 \
    -nodefaults -monitor stdio -machine pc,accel=kvm,usb=off \
    -smp 1,maxcpus=2 -cpu IvyBridge-IBRS \
    -qmp unix:/tmp/qmp-sock,server=on,wait=off
```

(2) Run 'qmp-shell' (located in the source tree, under: "scripts/qmp/) to connect to the just-launched QEMU:

```
$> ./qmp-shell -p -v /tmp/qmp-sock
[...]
(QEMU)
```

(3) Find out which CPU types could be plugged, and into which sockets:

```
(QEMU) query-hotpluggable-cpus
{
    "execute": "query-hotpluggable-cpus",
    "arguments": {}
}
{
    "return": [
        {
            "type": "IvyBridge-IBRS-x86_64-cpu",
            "vcpus-count": 1,
            "props": {
                "socket-id": 1,
                "core-id": 0,
                "thread-id": 0
            }
        },
        {
            "qom-path": "/machine/unattached/device[0]",
            "type": "IvyBridge-IBRS-x86_64-cpu",
            "vcpus-count": 1,
            "props": {
                "socket-id": 0,
                "core-id": 0,
```

(continues on next page)

(continued from previous page)

```
                "thread-id": 0
            }
        }
    ]
}
(QEMU)
```

(4) The `query-hotpluggable-cpus` command returns an object for CPUs that are present (containing a "qom-path" member) or which may be hot-plugged (no "qom-path" member). From its output in step (3), we can see that `IvyBridge-IBRS-x86_64-cpu` is present in socket 0, while hot-plugging a CPU into socket 1 requires passing the listed properties to QMP `device_add`:

```
(QEMU) device_add id=cpu-2 driver=IvyBridge-IBRS-x86_64-cpu socket-id=1 core-id=0␣
↪thread-id=0
{
    "execute": "device_add",
    "arguments": {
        "socket-id": 1,
        "driver": "IvyBridge-IBRS-x86_64-cpu",
        "id": "cpu-2",
        "core-id": 0,
        "thread-id": 0
    }
}
{
    "return": {}
}
(QEMU)
```

(5) Optionally, run QMP `query-cpus-fast` for some details about the vCPUs:

```
(QEMU) query-cpus-fast
{
    "execute": "query-cpus-fast",
    "arguments": {}
}
{
    "return": [
        {
            "qom-path": "/machine/unattached/device[0]",
            "target": "x86_64",
            "thread-id": 11534,
            "cpu-index": 0,
            "props": {
                "socket-id": 0,
                "core-id": 0,
                "thread-id": 0
            },
            "arch": "x86"
        },
        {
            "qom-path": "/machine/peripheral/cpu-2",
```

(continues on next page)

(continued from previous page)

```
            "target": "x86_64",
            "thread-id": 12106,
            "cpu-index": 1,
            "props": {
                "socket-id": 1,
                "core-id": 0,
                "thread-id": 0
            },
            "arch": "x86"
        }
    ]
}
(QEMU)
```

2.21.2 vCPU hot-unplug

From the 'qmp-shell', invoke the QMP `device_del` command:

```
(QEMU) device_del id=cpu-2
{
    "execute": "device_del",
    "arguments": {
        "id": "cpu-2"
    }
}
{
    "return": {}
}
(QEMU)
```

Note: vCPU hot-unplug requires guest cooperation; so the `device_del` command above does not guarantee vCPU removal – it's a "request to unplug". At this point, the guest will get a System Control Interrupt (SCI) and calls the ACPI handler for the affected vCPU device. Then the guest kernel will bring the vCPU offline and tell QEMU to unplug it.

2.22 Persistent reservation managers

SCSI persistent reservations allow restricting access to block devices to specific initiators in a shared storage setup. When implementing clustering of virtual machines, it is a common requirement for virtual machines to send persistent reservation SCSI commands. However, the operating system restricts sending these commands to unprivileged programs because incorrect usage can disrupt regular operation of the storage fabric.

For this reason, QEMU's SCSI passthrough devices, `scsi-block` and `scsi-generic` (both are only available on Linux) can delegate implementation of persistent reservations to a separate object, the "persistent reservation manager". Only PERSISTENT RESERVE OUT and PERSISTENT RESERVE IN commands are passed to the persistent reservation manager object; other commands are processed by QEMU as usual.

2.22.1 Defining a persistent reservation manager

A persistent reservation manager is an instance of a subclass of the "pr-manager" QOM class.

Right now only one subclass is defined, `pr-manager-helper`, which forwards the commands to an external privileged helper program over Unix sockets. The helper program only allows sending persistent reservation commands to devices for which QEMU has a file descriptor, so that QEMU will not be able to effect persistent reservations unless it has access to both the socket and the device.

`pr-manager-helper` has a single string property, `path`, which accepts the path to the helper program's Unix socket. For example, the following command line defines a `pr-manager-helper` object and attaches it to a SCSI passthrough device:

```
$ qemu-system-x86_64
    -device virtio-scsi \
    -object pr-manager-helper,id=helper0,path=/var/run/qemu-pr-helper.sock
    -drive if=none,id=hd,driver=raw,file.filename=/dev/sdb,file.pr-manager=helper0
    -device scsi-block,drive=hd
```

Alternatively, using `-blockdev`:

```
$ qemu-system-x86_64
    -device virtio-scsi \
    -object pr-manager-helper,id=helper0,path=/var/run/qemu-pr-helper.sock
    -blockdev node-name=hd,driver=raw,file.driver=host_device,file.filename=/dev/sdb,
↪file.pr-manager=helper0
    -device scsi-block,drive=hd
```

You will also need to ensure that the helper program **qemu-pr-helper** is running, and that it has been set up to use the same socket filename as your QEMU commandline specifies. See the qemu-pr-helper documentation or manpage for further details.

2.22.2 Multipath devices and persistent reservations

Proper support of persistent reservation for multipath devices requires communication with the multipath daemon, so that the reservation is registered and applied when a path is newly discovered or becomes online again. **qemu-pr-helper** can do this if the `libmpathpersist` library was available on the system at build time.

As of August 2017, a reservation key must be specified in `multipath.conf` for `multipathd` to check for persistent reservation for newly discovered paths or reinstated paths. The attribute can be added to the `defaults` section or the `multipaths` section; for example:

```
multipaths {
    multipath {
        wwid   XXXXXXXXXXXXXXXX
        alias      yellow
        reservation_key  0x123abc
    }
}
```

Linking **qemu-pr-helper** to `libmpathpersist` does not impede its usage on regular SCSI devices.

2.23 QEMU System Emulator Targets

QEMU is a generic emulator and it emulates many machines. Most of the options are similar for all machines. Specific information about the various targets are mentioned in the following sections.

Contents:

2.23.1 Arm System emulator

QEMU can emulate both 32-bit and 64-bit Arm CPUs. Use the `qemu-system-aarch64` executable to simulate a 64-bit Arm machine. You can use either `qemu-system-arm` or `qemu-system-aarch64` to simulate a 32-bit Arm machine: in general, command lines that work for `qemu-system-arm` will behave the same when used with `qemu-system-aarch64`.

QEMU has generally good support for Arm guests. It has support for nearly fifty different machines. The reason we support so many is that Arm hardware is much more widely varying than x86 hardware. Arm CPUs are generally built into "system-on-chip" (SoC) designs created by many different companies with different devices, and these SoCs are then built into machines which can vary still further even if they use the same SoC. Even with fifty boards QEMU does not cover more than a small fraction of the Arm hardware ecosystem.

The situation for 64-bit Arm is fairly similar, except that we don't implement so many different machines.

As well as the more common "A-profile" CPUs (which have MMUs and will run Linux) QEMU also supports "M-profile" CPUs such as the Cortex-M0, Cortex-M4 and Cortex-M33 (which are microcontrollers used in very embedded boards). For most boards the CPU type is fixed (matching what the hardware has), so typically you don't need to specify the CPU type by hand, except for special cases like the `virt` board.

Choosing a board model

For QEMU's Arm system emulation, you must specify which board model you want to use with the `-M` or `--machine` option; there is no default.

Because Arm systems differ so much and in fundamental ways, typically operating system or firmware images intended to run on one machine will not run at all on any other. This is often surprising for new users who are used to the x86 world where every system looks like a standard PC. (Once the kernel has booted, most userspace software cares much less about the detail of the hardware.)

If you already have a system image or a kernel that works on hardware and you want to boot with QEMU, check whether QEMU lists that machine in its `-machine help` output. If it is listed, then you can probably use that board model. If it is not listed, then unfortunately your image will almost certainly not boot on QEMU. (You might be able to extract the filesystem and use that with a different kernel which boots on a system that QEMU does emulate.)

If you don't care about reproducing the idiosyncrasies of a particular bit of hardware, such as small amount of RAM, no PCI or other hard disk, etc., and just want to run Linux, the best option is to use the `virt` board. This is a platform which doesn't correspond to any real hardware and is designed for use in virtual machines. You'll need to compile Linux with a suitable configuration for running on the `virt` board. `virt` supports PCI, virtio, recent CPUs and large amounts of RAM. It also supports 64-bit CPUs.

Board-specific documentation

Unfortunately many of the Arm boards QEMU supports are currently undocumented; you can get a complete list by running `qemu-system-aarch64 --machine help`.

Arm Integrator/CP (`integratorcp`)

The Arm Integrator/CP board is emulated with the following devices:

- ARM926E, ARM1026E, ARM946E, ARM1136 or Cortex-A8 CPU
- Two PL011 UARTs
- SMC 91c111 Ethernet adapter
- PL110 LCD controller
- PL050 KMI with PS/2 keyboard and mouse.
- PL181 MultiMedia Card Interface with SD card.

Arm MPS2 and MPS3 boards (`mps2-an385`, `mps2-an386`, `mps2-an500`, `mps2-an505`, `mps2-an511`, `mps2-an521`, `mps3-an524`, `mps3-an536`, `mps3-an547`)

These board models use Arm M-profile or R-profile CPUs.

The Arm MPS2, MPS2+ and MPS3 dev boards are FPGA based (the 2+ has a bigger FPGA but is otherwise the same as the 2; the 3 has a bigger FPGA again, can handle 4GB of RAM and has a USB controller and QSPI flash).

Since the CPU itself and most of the devices are in the FPGA, the details of the board as seen by the guest depend significantly on the FPGA image.

QEMU models the following FPGA images:

FPGA images using M-profile CPUs:

`mps2-an385`
Cortex-M3 as documented in Arm Application Note AN385

`mps2-an386`
Cortex-M4 as documented in Arm Application Note AN386

`mps2-an500`
Cortex-M7 as documented in Arm Application Note AN500

`mps2-an505`
Cortex-M33 as documented in Arm Application Note AN505

`mps2-an511`
Cortex-M3 'DesignStart' as documented in Arm Application Note AN511

`mps2-an521`
Dual Cortex-M33 as documented in Arm Application Note AN521

`mps3-an524`
Dual Cortex-M33 on an MPS3, as documented in Arm Application Note AN524

`mps3-an547`
Cortex-M55 on an MPS3, as documented in Arm Application Note AN547

FPGA images using R-profile CPUs:

`mps3-an536`
: Dual Cortex-R52 on an MPS3, as documented in Arm Application Note AN536

Differences between QEMU and real hardware:

- AN385/AN386 remapping of low 16K of memory to either ZBT SSRAM1 or to block RAM is unimplemented (QEMU always maps this to ZBT SSRAM1, as if zbt_boot_ctrl is always zero)
- AN524 remapping of low memory to either BRAM or to QSPI flash is unimplemented (QEMU always maps this to BRAM, ignoring the SCC CFG_REG0 memory-remap bit)
- QEMU provides a LAN9118 ethernet rather than LAN9220; the only guest visible difference is that the LAN9118 doesn't support checksum offloading
- QEMU does not model the QSPI flash in MPS3 boards as real QSPI flash, but only as simple ROM, so attempting to rewrite the flash from the guest will fail
- QEMU does not model the USB controller in MPS3 boards
- AN536 does not support runtime control of CPU reset and halt via the SCC CFG_REG0 register.
- AN536 does not support enabling or disabling the flash and ATCM interfaces via the SCC CFG_REG1 register.
- AN536 does not support setting of the initial vector table base address via the SCC CFG_REG6 and CFG_REG7 register config, and does not provide a mechanism for specifying these values at startup, so all guest images must be built to start from TCM (i.e. to expect the interrupt vector base at 0 from reset).
- AN536 defaults to only creating a single CPU; this is the equivalent of the way the real FPGA image usually runs with the second Cortex-R52 held in halt via the initial SCC CFG_REG0 register setting. You can create the second CPU with `-smp 2`; both CPUs will then start execution immediately on startup.

Note that for the AN536 the first UART is accessible only by CPU0, and the second UART is accessible only by CPU1. The first UART accessible shared between both CPUs is the third UART. Guest software might therefore be built to use either the first UART or the third UART; if you don't see any output from the UART you are looking at, try one of the others. (Even if the AN536 machine is started with a single CPU and so no "CPU1-only UART", the UART numbering remains the same, with the third UART being the first of the shared ones.)

Machine-specific options

The following machine-specific options are supported:

remap
: Supported for `mps3-an524` only. Set `BRAM/QSPI` to select the initial memory mapping. The default is `BRAM`.

Arm Musca boards (`musca-a`, `musca-b1`)

The Arm Musca development boards are a reference implementation of a system using the SSE-200 Subsystem for Embedded. They are dual Cortex-M33 systems.

QEMU provides models of the A and B1 variants of this board.

Unimplemented devices:

- SPI
- I^2C
- I^2S
- PWM

- QSPI
- Timer
- SCC
- GPIO
- eFlash
- MHU
- PVT
- SDIO
- CryptoCell

Note that (like the real hardware) the Musca-A machine is asymmetric: CPU 0 does not have the FPU or DSP extensions, but CPU 1 does. Also like the real hardware, the memory maps for the A and B1 variants differ significantly, so guest software must be built for the right variant.

Arm Realview boards (`realview-eb`, `realview-eb-mpcore`, `realview-pb-a8`, `realview-pbx-a9`)

Several variants of the Arm RealView baseboard are emulated, including the EB, PB-A8 and PBX-A9. Due to interactions with the bootloader, only certain Linux kernel configurations work out of the box on these boards.

Kernels for the PB-A8 board should have CONFIG_REALVIEW_HIGH_PHYS_OFFSET enabled in the kernel, and expect 512M RAM. Kernels for The PBX-A9 board should have CONFIG_SPARSEMEM enabled, CONFIG_REALVIEW_HIGH_PHYS_OFFSET disabled and expect 1024M RAM.

The following devices are emulated:

- ARM926E, ARM1136, ARM11MPCore, Cortex-A8 or Cortex-A9 MPCore CPU
- Arm AMBA Generic/Distributed Interrupt Controller
- Four PL011 UARTs
- SMC 91c111 or SMSC LAN9118 Ethernet adapter
- PL110 LCD controller
- PL050 KMI with PS/2 keyboard and mouse
- PCI host bridge
- PCI OHCI USB controller
- LSI53C895A PCI SCSI Host Bus Adapter with hard disk and CD-ROM devices
- PL181 MultiMedia Card Interface with SD card.

Arm Server Base System Architecture Reference board (`sbsa-ref`)

The `sbsa-ref` board intends to look like real hardware (while the `virt` board is a generic board platform that doesn't match any real hardware).

The hardware part is defined by two specifications:

- Base System Architecture (BSA)
- Server Base System Architecture (SBSA)

The Arm Base Boot Requirements (BBR) specification defines how the firmware reports that to any operating system.

It is intended to be a machine for developing firmware and testing standards compliance with operating systems.

Supported devices

The `sbsa-ref` board supports:

- A configurable number of AArch64 CPUs
- GIC version 3
- System bus AHCI controller
- System bus XHCI controller
- CDROM and hard disc on AHCI bus
- E1000E ethernet card on PCIe bus
- Bochs display adapter on PCIe bus
- A generic SBSA watchdog device

Board to firmware interface

`sbsa-ref` is a static system that reports a very minimal devicetree to the firmware for non-discoverable information about system components. This includes both internal hardware and parts affected by the qemu command line (i.e. CPUs and memory). As a result it must have a firmware specifically built to expect a certain hardware layout (as you would in a real machine).

Note

QEMU provides the guest EL3 firmware with minimal information about hardware platform using minimalistic devicetree. This is not a Linux devicetree. It is not even a firmware devicetree.

It is information passed from QEMU to describe the information a hardware platform would have other mechanisms to discover at runtime, that are affected by the QEMU command line.

Ultimately this devicetree may be replaced by IPC calls to an emulated SCP.

DeviceTree information

The devicetree reports:

- CPUs
- memory
- platform version
- GIC addresses
- NUMA node id for CPUs and memory

Platform version

The platform version is only for informing platform firmware about what kind of `sbsa-ref` board it is running on. It is neither a QEMU versioned machine type nor a reflection of the level of the SBSA/SystemReady SR support provided.

The `machine-version-major` value is updated when changes breaking fw compatibility are introduced. The `machine-version-minor` value is updated when features are added that don't break fw compatibility.

Platform version changes:

0.0
Devicetree holds information about CPUs, memory and platform version.

0.1
GIC information is present in devicetree.

0.2
GIC ITS information is present in devicetree.

0.3
The USB controller is an XHCI device, not EHCI.

Arm Versatile boards (`versatileab`, `versatilepb`)

The Arm Versatile baseboard is emulated with the following devices:

- ARM926E, ARM1136 or Cortex-A8 CPU
- PL190 Vectored Interrupt Controller
- Four PL011 UARTs
- SMC 91c111 Ethernet adapter
- PL110 LCD controller
- PL050 KMI with PS/2 keyboard and mouse.
- PCI host bridge. Note the emulated PCI bridge only provides access to PCI memory space. It does not provide access to PCI IO space. This means some devices (eg. ne2k_pci NIC) are not usable, and others (eg. rtl8139 NIC) are only usable when the guest drivers use the memory mapped control registers.
- PCI OHCI USB controller.
- LSI53C895A PCI SCSI Host Bus Adapter with hard disk and CD-ROM devices.
- PL181 MultiMedia Card Interface with SD card.

Booting a Linux kernel

Building a current Linux kernel with `versatile_defconfig` should be enough to get something running. Nowadays an out-of-tree build is recommended (and also useful if you build a lot of different targets). In the following example $BLD points to the build directory and $SRC points to the root of the Linux source tree. You can drop $SRC if you are running from there.

```
$ make O=$BLD -C $SRC ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- versatile_defconfig
$ make O=$BLD -C $SRC ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
```

You may want to enable some additional modules if you want to boot something from the SCSI interface:

```
CONFIG_PCI=y
CONFIG_PCI_VERSATILE=y
CONFIG_SCSI=y
CONFIG_SCSI_SYM53C8XX_2=y
```

You can then boot with a command line like:

```
$ qemu-system-arm -machine type=versatilepb \
    -serial mon:stdio \
    -drive if=scsi,driver=file,filename=debian-buster-armel-rootfs.ext4 \
    -kernel zImage \
    -dtb versatile-pb.dtb  \
    -append "console=ttyAMA0 ro root=/dev/sda"
```

Arm Versatile Express boards (`vexpress-a9`, `vexpress-a15`)

QEMU models two variants of the Arm Versatile Express development board family:

- `vexpress-a9` models the combination of the Versatile Express motherboard and the CoreTile Express A9x4 daughterboard
- `vexpress-a15` models the combination of the Versatile Express motherboard and the CoreTile Express A15x2 daughterboard

Note that as this hardware does not have PCI, IDE or SCSI, the only available storage option is emulated SD card.

Implemented devices:

- PL041 audio
- PL181 SD controller
- PL050 keyboard and mouse
- PL011 UARTs
- SP804 timers
- I2C controller
- PL031 RTC
- PL111 LCD display controller
- Flash memory
- LAN9118 ethernet

Unimplemented devices:

- SP810 system control block
- PCI-express
- USB controller (Philips ISP1761)
- Local DAP ROM
- CoreSight interfaces
- PL301 AXI interconnect
- SCC

- System counter
- HDLCD controller (`vexpress-a15`)
- SP805 watchdog
- PL341 dynamic memory controller
- DMA330 DMA controller
- PL354 static memory controller
- BP147 TrustZone Protection Controller
- TrustZone Address Space Controller

Other differences between the hardware and the QEMU model:

- QEMU will default to creating one CPU unless you pass a different `-smp` argument
- QEMU allows the amount of RAM provided to be specified with the `-m` argument
- QEMU defaults to providing a CPU which does not provide either TrustZone or the Virtualization Extensions: if you want these you must enable them with `-machine secure=on` and `-machine virtualization=on`
- QEMU provides 4 virtio-mmio virtio transports; these start at address `0x10013000` for `vexpress-a9` and at `0x1c130000` for `vexpress-a15`, and have IRQs from 40 upwards. If a dtb is provided on the command line then QEMU will edit it to include suitable entries describing these transports for the guest.
- QEMU does not currently support either dynamic or static remapping of the area of memory at address 0: it is always mapped to alias the first flash bank

Booting a Linux kernel

Building a current Linux kernel with `multi_v7_defconfig` should be enough to get something running. Nowadays an out-of-tree build is recommended (and also useful if you build a lot of different targets). In the following example $BLD points to the build directory and $SRC points to the root of the Linux source tree. You can drop $SRC if you are running from there.

```
$ make O=$BLD -C $SRC ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- multi_v7_defconfig
$ make O=$BLD -C $SRC ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
```

By default you will want to boot your rootfs off the sdcard interface. Your rootfs will need to be padded to the right size. With a suitable DTB you could also add devices to the virtio-mmio bus.

```
$ qemu-system-arm -cpu cortex-a15 -smp 4 -m 4096 \
    -machine type=vexpress-a15 -serial mon:stdio \
    -drive if=sd,driver=file,filename=armel-rootfs.ext4 \
    -kernel zImage  \
    -dtb vexpress-v2p-ca15-tc1.dtb \
    -append "console=ttyAMA0 root=/dev/mmcblk0 ro"
```

Aspeed family boards (*-bmc, ast2500-evb, ast2600-evb)

The QEMU Aspeed machines model BMCs of various OpenPOWER systems and Aspeed evaluation boards. They are based on different releases of the Aspeed SoC : the AST2400 integrating an ARM926EJ-S CPU (400MHz), the AST2500 with an ARM1176JZS CPU (800MHz) and more recently the AST2600 with dual cores ARM Cortex-A7 CPUs (1.2GHz).

The SoC comes with RAM, Gigabit ethernet, USB, SD/MMC, USB, SPI, I2C, etc.

AST2400 SoC based machines :

- `palmetto-bmc` OpenPOWER Palmetto POWER8 BMC
- `quanta-q71l-bmc` OpenBMC Quanta BMC
- `supermicrox11-bmc` Supermicro X11 BMC

AST2500 SoC based machines :

- `ast2500-evb` Aspeed AST2500 Evaluation board
- `romulus-bmc` OpenPOWER Romulus POWER9 BMC
- `witherspoon-bmc` OpenPOWER Witherspoon POWER9 BMC
- `sonorapass-bmc` OCP SonoraPass BMC
- `fp5280g2-bmc` Inspur FP5280G2 BMC
- `g220a-bmc` Bytedance G220A BMC
- `yosemitev2-bmc` Facebook YosemiteV2 BMC
- `tiogapass-bmc` Facebook Tiogapass BMC

AST2600 SoC based machines :

- `ast2600-evb` Aspeed AST2600 Evaluation board (Cortex-A7)
- `tacoma-bmc` OpenPOWER Witherspoon POWER9 AST2600 BMC
- `rainier-bmc` IBM Rainier POWER10 BMC
- `fuji-bmc` Facebook Fuji BMC
- `bletchley-bmc` Facebook Bletchley BMC
- `fby35-bmc` Facebook fby35 BMC
- `qcom-dc-scm-v1-bmc` Qualcomm DC-SCM V1 BMC
- `qcom-firework-bmc` Qualcomm Firework BMC

Supported devices

- SMP (for the AST2600 Cortex-A7)
- Interrupt Controller (VIC)
- Timer Controller
- RTC Controller
- I2C Controller, including the new register interface of the AST2600
- System Control Unit (SCU)

- SRAM mapping
- X-DMA Controller (basic interface)
- Static Memory Controller (SMC or FMC) - Only SPI Flash support
- SPI Memory Controller
- USB 2.0 Controller
- SD/MMC storage controllers
- SDRAM controller (dummy interface for basic settings and training)
- Watchdog Controller
- GPIO Controller (Master only)
- UART
- Ethernet controllers
- Front LEDs (PCA9552 on I2C bus)
- LPC Peripheral Controller (a subset of subdevices are supported)
- Hash/Crypto Engine (HACE) - Hash support only. TODO: HMAC and RSA
- ADC
- Secure Boot Controller (AST2600)
- eMMC Boot Controller (dummy)
- PECI Controller (minimal)
- I3C Controller

Missing devices

- Coprocessor support
- PWM and Fan Controller
- Slave GPIO Controller
- Super I/O Controller
- PCI-Express 1 Controller
- Graphic Display Controller
- MCTP Controller
- Mailbox Controller
- Virtual UART
- eSPI Controller

Boot options

The Aspeed machines can be started using the `-kernel` and `-dtb` options to load a Linux kernel or from a firmware. Images can be downloaded from the OpenBMC jenkins :

https://jenkins.openbmc.org/job/ci-openbmc/lastSuccessfulBuild/

or directly from the OpenBMC GitHub release repository :

https://github.com/openbmc/openbmc/releases

To boot a kernel directly from a Linux build tree:

```
$ qemu-system-arm -M ast2600-evb -nographic \
      -kernel arch/arm/boot/zImage \
      -dtb arch/arm/boot/dts/aspeed-ast2600-evb.dtb \
      -initrd rootfs.cpio
```

To boot the machine from the flash image, use an MTD drive :

```
$ qemu-system-arm -M romulus-bmc -nic user \
      -drive file=obmc-phosphor-image-romulus.static.mtd,format=raw,if=mtd -nographic
```

Options specific to Aspeed machines are :

- `execute-in-place` which emulates the boot from the CE0 flash device by using the FMC controller to load the instructions, and not simply from RAM. This takes a little longer.
- `fmc-model` to change the default FMC Flash model. FW needs support for the chip model to boot.
- `spi-model` to change the default SPI Flash model.
- `bmc-console` to change the default console device. Most of the machines use the `UART5` device for a boot console, which is mapped on `/dev/ttyS4` under Linux, but it is not always the case.

To use other flash models, for instance a different FMC chip and a bigger (64M) SPI for the `ast2500-evb` machine, run :

```
-M ast2500-evb,fmc-model=mx25l25635e,spi-model=mx66u51235f
```

When more flexibility is needed to define the flash devices, to use different flash models or define all flash devices (up to 8), the `-nodefaults` QEMU option can be used to avoid creating the default flash devices.

Flash devices should then be created from the command line and attached to a block device :

```
$ qemu-system-arm -M ast2600-evb \
      -blockdev node-name=fmc0,driver=file,filename=/path/to/fmc0.img \
      -device mx66u51235f,bus=ssi.0,cs=0x0,drive=fmc0 \
      -blockdev node-name=fmc1,driver=file,filename=/path/to/fmc1.img \
      -device mx66u51235f,bus=ssi.0,cs=0x1,drive=fmc1 \
      -blockdev node-name=spi1,driver=file,filename=/path/to/spi1.img \
      -device mx66u51235f,cs=0x0,bus=ssi.1,drive=spi1 \
      -nographic -nodefaults
```

In that case, the machine boots fetching instructions from the FMC0 device. It is slower to start but closer to what HW does. Using the machine option `execute-in-place` has a similar effect.

To change the boot console and use device `UART3` (`/dev/ttyS2` under Linux), use :

```
-M ast2500-evb,bmc-console=uart3
```

Aspeed minibmc family boards (`ast1030-evb`)

The QEMU Aspeed machines model mini BMCs of various Aspeed evaluation boards. They are based on different releases of the Aspeed SoC : the AST1030 integrating an ARM Cortex M4F CPU (200MHz).

The SoC comes with SRAM, SPI, I2C, etc.

AST1030 SoC based machines :

- `ast1030-evb` Aspeed AST1030 Evaluation board (Cortex-M4F)

Supported devices

- SMP (for the AST1030 Cortex-M4F)
- Interrupt Controller (VIC)
- Timer Controller
- I2C Controller
- System Control Unit (SCU)
- SRAM mapping
- Static Memory Controller (SMC or FMC) - Only SPI Flash support
- SPI Memory Controller
- USB 2.0 Controller
- Watchdog Controller
- GPIO Controller (Master only)
- UART
- LPC Peripheral Controller (a subset of subdevices are supported)
- Hash/Crypto Engine (HACE) - Hash support only. TODO: HMAC and RSA
- ADC
- Secure Boot Controller
- PECI Controller (minimal)

Missing devices

- PWM and Fan Controller
- Slave GPIO Controller
- Mailbox Controller
- Virtual UART
- eSPI Controller
- I3C Controller

Boot options

The Aspeed machines can be started using the `-kernel` to load a Zephyr OS or from a firmware. Images can be downloaded from the ASPEED GitHub release repository :

https://github.com/AspeedTech-BMC/zephyr/releases

To boot a kernel directly from a Zephyr build tree:

```
$ qemu-system-arm -M ast1030-evb -nographic \
      -kernel zephyr.elf
```

Facebook Yosemite v3.5 Platform and CraterLake Server (`fby35`)

Facebook has a series of multi-node compute server designs named Yosemite. The most recent version released was Yosemite v3.

Yosemite v3.5 is an iteration on this design, and is very similar: there's a baseboard with a BMC, and 4 server slots. The new server board design termed "CraterLake" includes a Bridge IC (BIC), with room for expansion boards to include various compute accelerators (video, inferencing, etc). At the moment, only the first server slot's BIC is included.

Yosemite v3.5 is itself a sled which fits into a 40U chassis, and 3 sleds can be fit into a chassis. See here for an example.

In this generation, the BMC is an AST2600 and each BIC is an AST1030. The BMC runs OpenBMC, and the BIC runs OpenBIC.

Firmware images can be retrieved from the Github releases or built from the source code, see the README's for instructions on that. This image uses the "fby35" machine recipe from OpenBMC, and the "yv35-cl" target from OpenBIC. Some reference images can also be found here:

```
$ wget https://github.com/facebook/openbmc/releases/download/openbmc-e2294ff5d31d/fby35.
↪mtd
$ wget https://github.com/peterdelevoryas/OpenBIC/releases/download/oby35-cl-2022.13.01/
↪Y35BCL.elf
```

Since this machine has multiple SoC's, each with their own serial console, the recommended way to run it is to allocate a pseudoterminal for each serial console and let the monitor use stdio. Also, starting in a paused state is useful because it allows you to attach to the pseudoterminals before the boot process starts.

```
$ qemu-system-arm -machine fby35 \
    -drive file=fby35.mtd,format=raw,if=mtd \
    -device loader,file=Y35BCL.elf,addr=0,cpu-num=2 \
    -serial pty -serial pty -serial mon:stdio \
    -display none -S
$ screen /dev/tty0 # In a separate TMUX pane, terminal window, etc.
$ screen /dev/tty1
$ (qemu) c             # Start the boot process once screen is setup.
```

Banana Pi BPI-M2U (bpim2u)

Banana Pi BPI-M2 Ultra is a quad-core mini single board computer built with Allwinner A40i/R40/V40 SoC. It features 2GB of RAM and 8GB eMMC. It also has onboard WiFi and BT. On the ports side, the BPI-M2 Ultra has 2 USB A 2.0 ports, 1 USB OTG port, 1 HDMI port, 1 audio jack, a DC power port, and last but not least, a SATA port.

Supported devices

The Banana Pi M2U machine supports the following devices:

- SMP (Quad Core Cortex-A7)
- Generic Interrupt Controller configuration
- SRAM mappings
- SDRAM controller
- Timer device (re-used from Allwinner A10)
- UART
- SD/MMC storage controller
- EMAC ethernet
- GMAC ethernet
- Clock Control Unit
- SATA
- TWI (I2C)
- USB 2.0
- Hardware Watchdog

Limitations

Currently, Banana Pi M2U does *not* support the following features:

- Graphical output via HDMI, GPU and/or the Display Engine
- Audio output
- Real Time Clock

Also see the 'unimplemented' array in the Allwinner R40 SoC module for a complete list of unimplemented I/O devices: `./hw/arm/allwinner-r40.c`

Boot options

The Banana Pi M2U machine can start using the standard -kernel functionality for loading a Linux kernel or ELF executable. Additionally, the Banana Pi M2U machine can also emulate the BootROM which is present on an actual Allwinner R40 based SoC, which loads the bootloader from a SD card, specified via the -sd argument to qemu-system-arm.

Running mainline Linux

To build a Linux mainline kernel that can be booted by the Banana Pi M2U machine, simply configure the kernel using the sunxi_defconfig configuration:

```
$ ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- make mrproper
$ ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- make sunxi_defconfig
```

To boot the newly build linux kernel in QEMU with the Banana Pi M2U machine, use:

```
$ qemu-system-arm -M bpim2u -nographic \
    -kernel /path/to/linux/arch/arm/boot/zImage \
    -append 'console=ttyS0,115200' \
    -dtb /path/to/linux/arch/arm/boot/dts/sun8i-r40-bananapi-m2-ultra.dtb
```

Banana Pi M2U images

Note that the mainline kernel does not have a root filesystem. You can choose to build you own image with buildroot using the bananapi_m2_ultra_defconfig. Also see https://buildroot.org for more information.

Another possibility is to run an OpenWrt image for Banana Pi M2U which can be downloaded from:

https://downloads.openwrt.org/releases/22.03.3/targets/sunxi/cortexa7/

When using an image as an SD card, it must be resized to a power of two. This can be done with the `qemu-img` command. It is recommended to only increase the image size instead of shrinking it to a power of two, to avoid loss of data. For example, to prepare a downloaded Armbian image, first extract it and then increase its size to one gigabyte as follows:

```
$ qemu-img resize \
  openwrt-22.03.3-sunxi-cortexa7-sinovoip_bananapi-m2-ultra-ext4-sdcard.img \
  1G
```

Instead of providing a custom Linux kernel via the -kernel command you may also choose to let the Banana Pi M2U machine load the bootloader from SD card, just like a real board would do using the BootROM. Simply pass the selected image via the -sd argument and remove the -kernel, -append, -dbt and -initrd arguments:

```
$ qemu-system-arm -M bpim2u -nic user -nographic \
  -sd openwrt-22.03.3-sunxi-cortexa7-sinovoip_bananapi-m2-ultra-ext4-sdcard.img
```

Running U-Boot

U-Boot mainline can be build and configured using the Bananapi_M2_Ultra_defconfig using similar commands as describe above for Linux. Note that it is recommended for development/testing to select the following configuration setting in U-Boot:

> Device Tree Control > Provider for DTB for DT Control > Embedded DTB

The BootROM of allwinner R40 loading u-boot from the 8KiB offset of sdcard. Let's create an bootable disk image:

```
$ dd if=/dev/zero of=sd.img bs=32M count=1
$ dd if=u-boot-sunxi-with-spl.bin of=sd.img bs=1k seek=8 conv=notrunc
```

And then boot it.

```
$ qemu-system-arm -M bpim2u -nographic -sd sd.img
```

Banana Pi M2U integration tests

The Banana Pi M2U machine has several integration tests included. To run the whole set of tests, build QEMU from source and simply provide the following command:

```
$ cd qemu-build-dir
$ AVOCADO_ALLOW_LARGE_STORAGE=yes tests/venv/bin/avocado \
  --verbose --show=app,console run -t machine:bpim2u \
  ../tests/avocado/boot_linux_console.py
```

B-L475E-IOT01A IoT Node (`b-l475e-iot01a`)

The B-L475E-IOT01A IoT Node uses the STM32L475VG SoC which is based on ARM Cortex-M4F core. It is part of STMicroelectronics *STM32 boards* and more specifically the STM32L4 ultra-low power series. The STM32L4x5 chip runs at up to 80 MHz and integrates 128 KiB of SRAM and up to 1MiB of Flash. The B-L475E-IOT01A board namely features 64 Mibit QSPI Flash, BT, WiFi and RF connectivity, USART, I2C, SPI, CAN and USB OTG, as well as a variety of sensors.

Supported devices

Currently B-L475E-IOT01A machines support the following devices:

- Cortex-M4F based STM32L4x5 SoC
- STM32L4x5 EXTI (Extended interrupts and events controller)
- STM32L4x5 SYSCFG (System configuration controller)
- STM32L4x5 RCC (Reset and clock control)
- STM32L4x5 GPIOs (General-purpose I/Os)
- STM32L4x5 USARTs, UARTs and LPUART (Serial ports)
- optional 8x8 led display (based on DM163 driver)

Missing devices

The B-L475E-IOT01A does *not* support the following devices:

- Analog to Digital Converter (ADC)
- SPI controller
- Timer controller (TIMER)

See the complete list of unimplemented peripheral devices in the STM32L4x5 module : `./hw/arm/stm32l4x5_soc.c`

Boot options

The B-L475E-IOT01A machine can be started using the `-kernel` option to load a firmware. Example:

```
$ qemu-system-arm -M b-l475e-iot01a -kernel firmware.bin
```

Boundary Devices SABRE Lite (`sabrelite`)

Boundary Devices SABRE Lite i.MX6 Development Board is a low-cost development platform featuring the powerful Freescale / NXP Semiconductor's i.MX 6 Quad Applications Processor.

Supported devices

The SABRE Lite machine supports the following devices:

- Up to 4 Cortex-A9 cores
- Generic Interrupt Controller
- 1 Clock Controller Module
- 1 System Reset Controller
- 5 UARTs
- 2 EPIC timers
- 1 GPT timer
- 2 Watchdog timers
- 1 FEC Ethernet controller
- 3 I2C controllers
- 7 GPIO controllers
- 4 SDHC storage controllers
- 4 USB 2.0 host controllers
- 5 ECSPI controllers
- 1 SST 25VF016B flash

Please note above list is a complete superset the QEMU SABRE Lite machine can support. For a normal use case, a device tree blob that represents a real world SABRE Lite board, only exposes a subset of devices to the guest software.

Boot options

The SABRE Lite machine can start using the standard -kernel functionality for loading a Linux kernel, U-Boot bootloader or ELF executable.

Running Linux kernel

Linux mainline v5.10 release is tested at the time of writing. To build a Linux mainline kernel that can be booted by the SABRE Lite machine, simply configure the kernel using the imx_v6_v7_defconfig configuration:

```
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabihf-
$ make imx_v6_v7_defconfig
$ make
```

To boot the newly built Linux kernel in QEMU with the SABRE Lite machine, use:

```
$ qemu-system-arm -M sabrelite -smp 4 -m 1G \
    -display none -serial null -serial stdio \
    -kernel arch/arm/boot/zImage \
    -dtb arch/arm/boot/dts/imx6q-sabrelite.dtb \
    -initrd /path/to/rootfs.ext4 \
    -append "root=/dev/ram"
```

Running U-Boot

U-Boot mainline v2020.10 release is tested at the time of writing. To build a U-Boot mainline bootloader that can be booted by the SABRE Lite machine, use the mx6qsabrelite_defconfig with similar commands as described above for Linux:

```
$ export CROSS_COMPILE=arm-linux-gnueabihf-
$ make mx6qsabrelite_defconfig
```

Note we need to adjust settings by:

```
$ make menuconfig
```

then manually select the following configuration in U-Boot:

> Device Tree Control > Provider of DTB for DT Control > Embedded DTB

To start U-Boot using the SABRE Lite machine, provide the u-boot binary to the -kernel argument, along with an SD card image with rootfs:

```
$ qemu-system-arm -M sabrelite -smp 4 -m 1G \
    -display none -serial null -serial stdio \
    -kernel u-boot
```

The following example shows booting Linux kernel from dhcp, and uses the rootfs on an SD card. This requires some additional command line parameters for QEMU:

```
-nic user,tftp=/path/to/kernel/zImage \
-drive file=sdcard.img,id=rootfs -device sd-card,drive=rootfs
```

The directory for the built-in TFTP server should also contain the device tree blob of the SABRE Lite board. The sample SD card image was populated with the root file system with one single partition. You may adjust the kernel "root=" boot parameter accordingly.

After U-Boot boots, type the following commands in the U-Boot command shell to boot the Linux kernel:

```
=> setenv ethaddr 00:11:22:33:44:55
=> setenv bootfile zImage
=> dhcp
=> tftpboot 14000000 imx6q-sabrelite.dtb
=> setenv bootargs root=/dev/mmcblk3p1
=> bootz 12000000 - 14000000
```

Canon A1100 (canon-a1100)

This machine is a model of the Canon PowerShot A1100 camera, which uses the DIGIC SoC. This model is based on reverse engineering efforts by the contributors to the CHDK and Magic Lantern projects.

The emulation is incomplete. In particular it can't be used to run the original camera firmware, but it can successfully run an experimental version of the barebox bootloader.

Cubietech Cubieboard (cubieboard)

The `cubieboard` model emulates the Cubietech Cubieboard, which is a Cortex-A8 based single-board computer using the AllWinner A10 SoC.

Emulated devices:

- Timer
- UART
- RTC
- EMAC
- SDHCI
- USB controller
- SATA controller
- TWI (I2C) controller
- Watchdog timer

Emcraft SmartFusion2 SOM kit (emcraft-sf2)

The `emcraft-sf2` board emulates the SmartFusion2 SOM kit from Emcraft (M2S010). This is a System-on-Module from EmCraft systems, based on the SmartFusion2 SoC FPGA from Microsemi Corporation. The SoC is based on a Cortex-M4 processor.

Emulated devices:

- System timer
- System registers
- SPI controller

- UART
- EMAC

Calxeda Highbank and Midway (`highbank`, `midway`)

`highbank` is a model of the Calxeda Highbank (ECX-1000) system, which has four Cortex-A9 cores.

`midway` is a model of the Calxeda Midway (ECX-2000) system, which has four Cortex-A15 cores.

Emulated devices:

- L2x0 cache controller
- SP804 dual timer
- PL011 UART
- PL061 GPIOs
- PL031 RTC
- PL022 synchronous serial port controller
- AHCI
- XGMAC ethernet controllers

Freecom MusicPal (`musicpal`)

The Freecom MusicPal internet radio emulation includes the following elements:

- Marvell MV88W8618 Arm core.
- 32 MB RAM, 256 KB SRAM, 8 MB flash.
- Up to 2 16550 UARTs
- MV88W8xx8 Ethernet controller
- MV88W8618 audio controller, WM8750 CODEC and mixer
- 128x64 display with brightness control
- 2 buttons, 2 navigation wheels with button function

Gumstix Connex and Verdex (`connex`, `verdex`)

These machines model the Gumstix Connex and Verdex boards. The Connex has a PXA255 CPU and the Verdex has a PXA270.

Implemented devices:

- NOR flash
- SMC91C111 ethernet
- Interrupt controller
- DMA
- Timer

- GPIO
- MMC/SD card
- Fast infra-red communications port (FIR)
- LCD controller
- Synchronous serial ports (SPI)
- PCMCIA interface
- I2C
- I2S

Intel Mainstone II board (`mainstone`)

The `mainstone` board emulates the Intel Mainstone II development board, which uses a PXA270 CPU.

Emulated devices:

- Flash memory
- Keypad
- MMC controller
- 91C111 ethernet
- PIC
- Timer
- DMA
- GPIO
- FIR
- Serial
- LCD controller
- SSP
- USB controller
- RTC
- PCMCIA
- I2C
- I2S

Kyoto Microcomputer KZM-ARM11-01 (`kzm`)

The `kzm` board emulates the Kyoto Microcomputer KZM-ARM11-01 evaluation board, which is based on an NXP i.MX32 SoC which uses an ARM1136 CPU.

Emulated devices:

- UARTs
- LAN9118 ethernet
- AVIC
- CCM
- GPT
- EPIT timers
- I2C
- GPIO controllers
- Watchdog timer

Nordic nRF boards (`microbit`)

The Nordic nRF chips are a family of ARM-based System-on-Chip that are designed to be used for low-power and short-range wireless solutions.

The nRF51 series is the first series for short range wireless applications. It is superseded by the nRF52 series. The following machines are based on this chip :

- `microbit` BBC micro:bit board with nRF51822 SoC

There are other series such as nRF52, nRF53 and nRF91 which are currently not supported by QEMU.

Supported devices

- ARM Cortex-M0 (ARMv6-M)
- Serial ports (UART)
- Clock controller
- Timers
- Random Number Generator (RNG)
- GPIO controller
- NVMC
- SWI

Missing devices

- Watchdog
- Real-Time Clock (RTC) controller
- TWI (i2c)
- SPI controller
- Analog to Digital Converter (ADC)
- Quadrature decoder
- Radio

Boot options

The Micro:bit machine can be started using the `-device` option to load a firmware in ihex format. Example:

```
$ qemu-system-arm -M microbit -device loader,file=test.hex
```

Nokia N800 and N810 tablets (`n800`, `n810`)

Nokia N800 and N810 internet tablets (known also as RX-34 and RX-44 / 48) emulation supports the following elements:

- Texas Instruments OMAP2420 System-on-chip (ARM1136 core)
- RAM and non-volatile OneNAND Flash memories
- Display connected to EPSON remote framebuffer chip and OMAP on-chip display controller and a LS041y3 MIPI DBI-C controller
- TI TSC2301 (in N800) and TI TSC2005 (in N810) touchscreen controllers driven through SPI bus
- National Semiconductor LM8323-controlled qwerty keyboard driven through I^2C bus
- Secure Digital card connected to OMAP MMC/SD host
- Three OMAP on-chip UARTs and on-chip STI debugging console
- Mentor Graphics "Inventra" dual-role USB controller embedded in a TI TUSB6010 chip - only USB host mode is supported
- TI TMP105 temperature sensor driven through I^2C bus
- TI TWL92230C power management companion with an RTC on I^2C bus
- Nokia RETU and TAHVO multi-purpose chips with an RTC, connected through CBUS

Nuvoton iBMC boards (*-bmc, npcm750-evb, quanta-gsj)

The Nuvoton iBMC chips (NPCM7xx) are a family of ARM-based SoCs that are designed to be used as Baseboard Management Controllers (BMCs) in various servers. They all feature one or two ARM Cortex-A9 CPU cores, as well as an assortment of peripherals targeted for either Enterprise or Data Center / Hyperscale applications. The former is a superset of the latter, so NPCM750 has all the peripherals of NPCM730 and more.

The NPCM750 SoC has two Cortex-A9 cores and is targeted for the Enterprise segment. The following machines are based on this chip :

- `npcm750-evb` Nuvoton NPCM750 Evaluation board

The NPCM730 SoC has two Cortex-A9 cores and is targeted for Data Center and Hyperscale applications. The following machines are based on this chip :

- `quanta-gbs-bmc` Quanta GBS server BMC
- `quanta-gsj` Quanta GSJ server BMC
- `kudo-bmc` Fii USA Kudo server BMC
- `mori-bmc` Fii USA Mori server BMC

There are also two more SoCs, NPCM710 and NPCM705, which are single-core variants of NPCM750 and NPCM730, respectively. These are currently not supported by QEMU.

Supported devices

- SMP (Dual Core Cortex-A9)
- Cortex-A9MPCore built-in peripherals: SCU, GIC, Global Timer, Private Timer and Watchdog.
- SRAM, ROM and DRAM mappings
- System Global Control Registers (GCR)
- Clock and reset controller (CLK)
- Timer controller (TIM)
- Serial ports (16550-based)
- DDR4 memory controller (dummy interface indicating memory training is done)
- OTP controllers (no protection features)
- Flash Interface Unit (FIU; no protection features)
- Random Number Generator (RNG)
- USB host (USBH)
- GPIO controller
- Analog to Digital Converter (ADC)
- Pulse Width Modulation (PWM)
- SMBus controller (SMBF)
- Ethernet controller (EMC)
- Tachometer
- Peripheral SPI controller (PSPI)

Missing devices

- LPC/eSPI host-to-BMC interface, including
  - Keyboard and mouse controller interface (KBCI)
  - Keyboard Controller Style (KCS) channels
  - BIOS POST code FIFO
  - System Wake-up Control (SWC)
  - Shared memory (SHM)
  - eSPI slave interface
- Ethernet controller (GMAC)
- USB device (USBD)
- SD/MMC host
- PECI interface
- PCI and PCIe root complex and bridges
- VDM and MCTP support
- Serial I/O expansion
- LPC/eSPI host
- Coprocessor
- Graphics
- Video capture
- Encoding compression engine
- Security features

Boot options

The Nuvoton machines can boot from an OpenBMC firmware image, or directly into a kernel using the `-kernel` option. OpenBMC images for `quanta-gsj` and possibly others can be downloaded from the OpenBMC jenkins :

https://jenkins.openbmc.org/

The firmware image should be attached as an MTD drive. Example :

```
$ qemu-system-arm -machine quanta-gsj -nographic \
    -drive file=image-bmc,if=mtd,bus=0,unit=0,format=raw
```

The default root password for test images is usually `0penBmc`.

NXP i.MX25 PDK board (`imx25-pdk`)

The `imx25-pdk` board emulates the NXP i.MX25 Product Development Kit board, which is based on an i.MX25 SoC which uses an ARM926 CPU.

Emulated devices:

- SD controller
- AVIC
- CCM
- GPT
- EPIT timers
- FEC
- RNGC
- I2C
- GPIO controllers
- Watchdog timer
- USB controllers

Orange Pi PC (`orangepi-pc`)

The Xunlong Orange Pi PC is an Allwinner H3 System on Chip based embedded computer with mainline support in both U-Boot and Linux. The board comes with a Quad Core Cortex-A7 @ 1.3GHz, 1GiB RAM, 100Mbit ethernet, USB, SD/MMC, USB, HDMI and various other I/O.

Supported devices

The Orange Pi PC machine supports the following devices:

- SMP (Quad Core Cortex-A7)
- Generic Interrupt Controller configuration
- SRAM mappings
- SDRAM controller
- Real Time Clock
- Timer device (re-used from Allwinner A10)
- UART
- SD/MMC storage controller
- EMAC ethernet
- USB 2.0 interfaces
- Clock Control Unit
- System Control module
- Security Identifier device

- TWI (I2C)
- Watchdog timer

Limitations

Currently, Orange Pi PC does *not* support the following features:

- Graphical output via HDMI, GPU and/or the Display Engine
- Audio output
- Hardware Watchdog

Also see the 'unimplemented' array in the Allwinner H3 SoC module for a complete list of unimplemented I/O devices: `./hw/arm/allwinner-h3.c`

Boot options

The Orange Pi PC machine can start using the standard -kernel functionality for loading a Linux kernel or ELF executable. Additionally, the Orange Pi PC machine can also emulate the BootROM which is present on an actual Allwinner H3 based SoC, which loads the bootloader from a SD card, specified via the -sd argument to qemu-system-arm.

Machine-specific options

The following machine-specific options are supported:

- allwinner-rtc.base-year=YYYY

  The Allwinner RTC device is automatically created by the Orange Pi PC machine and uses a default base year value which can be overridden using the 'base-year' property. The base year is the actual represented year when the RTC year value is zero. This option can be used in case the target operating system driver uses a different base year value. The minimum value for the base year is 1900.

- allwinner-sid.identifier=abcd1122-a000-b000-c000-12345678ffff

  The Security Identifier value can be read by the guest. For example, U-Boot uses it to determine a unique MAC address.

The above machine-specific options can be specified in qemu-system-arm via the '-global' argument, for example:

```
$ qemu-system-arm -M orangepi-pc -sd mycard.img \
     -global allwinner-rtc.base-year=2000
```

Running mainline Linux

Mainline Linux kernels from 4.19 up to latest master are known to work. To build a Linux mainline kernel that can be booted by the Orange Pi PC machine, simply configure the kernel using the sunxi_defconfig configuration:

```
$ ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- make mrproper
$ ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- make sunxi_defconfig
```

To be able to use USB storage, you need to manually enable the corresponding configuration item. Start the kconfig configuration tool:

```
$ ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- make menuconfig
```

Navigate to the following item, enable it and save your configuration:

> Device Drivers > USB support > USB Mass Storage support

Build the Linux kernel with:

```
$ ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- make
```

To boot the newly build linux kernel in QEMU with the Orange Pi PC machine, use:

```
$ qemu-system-arm -M orangepi-pc -nic user -nographic \
    -kernel /path/to/linux/arch/arm/boot/zImage \
    -append 'console=ttyS0,115200' \
    -dtb /path/to/linux/arch/arm/boot/dts/sun8i-h3-orangepi-pc.dtb
```

Orange Pi PC images

Note that the mainline kernel does not have a root filesystem. You may provide it with an official Orange Pi PC image from the official website:

> http://www.orangepi.org/downloadresources/

Another possibility is to run an Armbian image for Orange Pi PC which can be downloaded from:

> https://www.armbian.com/orange-pi-pc/

Alternatively, you can also choose to build you own image with buildroot using the orangepi_pc_defconfig. Also see https://buildroot.org for more information.

When using an image as an SD card, it must be resized to a power of two. This can be done with the `qemu-img` command. It is recommended to only increase the image size instead of shrinking it to a power of two, to avoid loss of data. For example, to prepare a downloaded Armbian image, first extract it and then increase its size to one gigabyte as follows:

```
$ qemu-img resize Armbian_19.11.3_Orangepipc_bionic_current_5.3.9.img 1G
```

You can choose to attach the selected image either as an SD card or as USB mass storage. For example, to boot using the Orange Pi PC Debian image on SD card, simply add the -sd argument and provide the proper root= kernel parameter:

```
$ qemu-system-arm -M orangepi-pc -nic user -nographic \
    -kernel /path/to/linux/arch/arm/boot/zImage \
    -append 'console=ttyS0,115200 root=/dev/mmcblk0p2' \
    -dtb /path/to/linux/arch/arm/boot/dts/sun8i-h3-orangepi-pc.dtb \
    -sd OrangePi_pc_debian_stretch_server_linux5.3.5_v1.0.img
```

To attach the image as an USB mass storage device to the machine, simply append to the command:

```
-drive if=none,id=stick,file=myimage.img \
-device usb-storage,bus=usb-bus.0,drive=stick
```

Instead of providing a custom Linux kernel via the -kernel command you may also choose to let the Orange Pi PC machine load the bootloader from SD card, just like a real board would do using the BootROM. Simply pass the selected image via the -sd argument and remove the -kernel, -append, -dbt and -initrd arguments:

```
$ qemu-system-arm -M orangepi-pc -nic user -nographic \
     -sd Armbian_19.11.3_Orangepipc_buster_current_5.3.9.img
```

Note that both the official Orange Pi PC images and Armbian images start a lot of userland programs via systemd. Depending on the host hardware and OS, they may be slow to emulate, especially due to emulating the 4 cores. To help reduce the performance slow down due to emulating the 4 cores, you can give the following kernel parameters via U-Boot (or via -append):

```
=> setenv extraargs 'systemd.default_timeout_start_sec=9000 loglevel=7 nosmp␣
↪console=ttyS0,115200'
```

Running U-Boot

U-Boot mainline can be build and configured using the orangepi_pc_defconfig using similar commands as describe above for Linux. Note that it is recommended for development/testing to select the following configuration setting in U-Boot:

> Device Tree Control > Provider for DTB for DT Control > Embedded DTB

To start U-Boot using the Orange Pi PC machine, provide the u-boot binary to the -kernel argument:

```
$ qemu-system-arm -M orangepi-pc -nic user -nographic \
    -kernel /path/to/uboot/u-boot -sd disk.img
```

Use the following U-boot commands to load and boot a Linux kernel from SD card:

```
=> setenv bootargs console=ttyS0,115200
=> ext2load mmc 0 0x42000000 zImage
=> ext2load mmc 0 0x43000000 sun8i-h3-orangepi-pc.dtb
=> bootz 0x42000000 - 0x43000000
```

Running NetBSD

The NetBSD operating system also includes support for Allwinner H3 based boards, including the Orange Pi PC. NetBSD 9.0 is known to work best for the Orange Pi PC board and provides a fully working system with serial console, networking and storage. For the Orange Pi PC machine, get the 'evbarm-earmv7hf' based image from:

> https://cdn.netbsd.org/pub/NetBSD/NetBSD-9.0/evbarm-earmv7hf/binary/gzimg/armv7.img.gz

The image requires manually installing U-Boot in the image. Build U-Boot with the orangepi_pc_defconfig configuration as described in the previous section. Next, unzip the NetBSD image and write the U-Boot binary including SPL using:

```
$ gunzip armv7.img.gz
$ dd if=/path/to/u-boot-sunxi-with-spl.bin of=armv7.img bs=1024 seek=8 conv=notrunc
```

Finally, before starting the machine the SD image must be extended such that the size of the SD image is a power of two and that the NetBSD kernel will not conclude the NetBSD partition is larger than the emulated SD card:

```
$ qemu-img resize armv7.img 2G
```

Start the machine using the following command:

```
$ qemu-system-arm -M orangepi-pc -nic user -nographic \
      -sd armv7.img -global allwinner-rtc.base-year=2000
```

At the U-Boot stage, interrupt the automatic boot process by pressing a key and set the following environment variables before booting:

```
=> setenv bootargs root=ld0a
=> setenv kernel netbsd-GENERIC.ub
=> setenv fdtfile dtb/sun8i-h3-orangepi-pc.dtb
=> setenv bootcmd 'fatload mmc 0:1 ${kernel_addr_r} ${kernel}; fatload mmc 0:1 ${fdt_
↪addr_r} ${fdtfile}; fdt addr ${fdt_addr_r}; bootm ${kernel_addr_r} - ${fdt_addr_r}'
```

Optionally you may save the environment variables to SD card with 'saveenv'. To continue booting simply give the 'boot' command and NetBSD boots.

Orange Pi PC integration tests

The Orange Pi PC machine has several integration tests included. To run the whole set of tests, build QEMU from source and simply provide the following command:

```
$ AVOCADO_ALLOW_LARGE_STORAGE=yes avocado --show=app,console run \
   -t machine:orangepi-pc tests/avocado/boot_linux_console.py
```

Palm Tungsten|E PDA (cheetah)

The Palm Tungsten|E PDA (codename "Cheetah") emulation includes the following elements:

- Texas Instruments OMAP310 System-on-chip (ARM925T core)
- ROM and RAM memories (ROM firmware image can be loaded with -option-rom)
- On-chip LCD controller
- On-chip Real Time Clock
- TI TSC2102i touchscreen controller / analog-digital converter / Audio CODEC, connected through MicroWire and I^2S buses
- GPIO-connected matrix keypad
- Secure Digital card connected to OMAP MMC/SD host
- Three on-chip UARTs

Raspberry Pi boards (raspi0, raspi1ap, raspi2b, raspi3ap, raspi3b, raspi4b)

QEMU provides models of the following Raspberry Pi boards:

raspi0 and raspi1ap
: ARM1176JZF-S core, 512 MiB of RAM

raspi2b
: Cortex-A7 (4 cores), 1 GiB of RAM

raspi3ap
: Cortex-A53 (4 cores), 512 MiB of RAM

raspi3b
: Cortex-A53 (4 cores), 1 GiB of RAM

raspi4b
: Cortex-A72 (4 cores), 2 GiB of RAM

Implemented devices

- ARM1176JZF-S, Cortex-A7, Cortex-A53 or Cortex-A72 CPU
- Interrupt controller
- DMA controller
- Clock and reset controller (CPRMAN)
- System Timer
- GPIO controller
- Serial ports (BCM2835 AUX - 16550 based - and PL011)
- Random Number Generator (RNG)
- Frame Buffer
- USB host (USBH)
- GPIO controller
- SD/MMC host controller
- SoC thermal sensor
- USB2 host controller (DWC2 and MPHI)
- MailBox controller (MBOX)
- VideoCore firmware (property)
- Peripheral SPI controller (SPI)
- Broadcom Serial Controller (I2C)

Missing devices

- Analog to Digital Converter (ADC)
- Pulse Width Modulation (PWM)
- PCIE Root Port (raspi4b)
- GENET Ethernet Controller (raspi4b)

Sharp XScale-based PDA models (`akita`, `borzoi`, `spitz`, `terrier`, `tosa`)

The Sharp Zaurus are PDAs based on XScale, able to run Linux ('SL series').

The SL-6000 ("Tosa"), released in 2005, uses a PXA255 System-on-chip.

The SL-C3000 ("Spitz"), SL-C1000 ("Akita"), SL-C3100 ("Borzoi") and SL-C3200 ("Terrier") use a PXA270.

The clamshell PDA models emulation includes the following peripherals:

- Intel PXA255/PXA270 System-on-chip (ARMv5TE core)
- NAND Flash memory - not in "Tosa"
- IBM/Hitachi DSCM microdrive in a PXA PCMCIA slot - not in "Akita"
- On-chip OHCI USB controller - not in "Tosa"
- On-chip LCD controller
- On-chip Real Time Clock
- TI ADS7846 touchscreen controller on SSP bus
- Maxim MAX1111 analog-digital converter on I^2C bus
- GPIO-connected keyboard controller and LEDs
- Secure Digital card connected to PXA MMC/SD host
- Three on-chip UARTs
- WM8750 audio CODEC on I^2C and I^2S buses

Sharp Zaurus SL-5500 (`collie`)

This machine is a model of the Sharp Zaurus SL-5500, which was a 1990s PDA based on the StrongARM SA1110.

Implemented devices:

- NOR flash
- Interrupt controller
- Timer
- RTC
- GPIO
- Peripheral Pin Controller (PPC)
- UARTs
- Synchronous Serial Ports (SSP)

Siemens SX1 (`sx1`, `sx1-v1`)

The Siemens SX1 models v1 and v2 (default) basic emulation. The emulation includes the following elements:

- Texas Instruments OMAP310 System-on-chip (ARM925T core)
- ROM and RAM memories (ROM firmware image can be loaded with -pflash) V1 1 Flash of 16MB and 1 Flash of 8MB V2 1 Flash of 32MB
- On-chip LCD controller
- On-chip Real Time Clock
- Secure Digital card connected to OMAP MMC/SD host
- Three on-chip UARTs

Stellaris boards (`lm3s6965evb`, `lm3s811evb`)

The Luminary Micro Stellaris LM3S811EVB emulation includes the following devices:

- Cortex-M3 CPU core.
- 64k Flash and 8k SRAM.
- Timers, UARTs, ADC and I^2C interface.
- OSRAM Pictiva 96x16 OLED with SSD0303 controller on I^2C bus.

The Luminary Micro Stellaris LM3S6965EVB emulation includes the following devices:

- Cortex-M3 CPU core.
- 256k Flash and 64k SRAM.
- Timers, UARTs, ADC, I^2C and SSI interfaces.
- OSRAM Pictiva 128x64 OLED with SSD0323 controller connected via SSI.

STMicroelectronics STM32 boards (`netduino2`, `netduinoplus2`, `stm32vldiscovery`)

The STM32 chips are a family of 32-bit ARM-based microcontroller by STMicroelectronics.

The STM32F1 series is based on ARM Cortex-M3 core. The following machines are based on this chip :

- `stm32vldiscovery` STM32VLDISCOVERY board with STM32F100RBT6 microcontroller

The STM32F2 series is based on ARM Cortex-M3 core. The following machines are based on this chip :

- `netduino2` Netduino 2 board with STM32F205RFT6 microcontroller

The STM32F4 series is based on ARM Cortex-M4F core, as well as the STM32L4 ultra-low-power series. The STM32F4 series is pin-to-pin compatible with STM32F2 series. The following machines are based on this ARM Cortex-M4F chip :

- `netduinoplus2` Netduino Plus 2 board with STM32F405RGT6 microcontroller
- `olimex-stm32-h405` Olimex STM32 H405 board with STM32F405RGT6 microcontroller
- `b-l475e-iot01a` *B-L475E-IOT01A IoT Node* board with STM32L475VG microcontroller

There are many other STM32 series that are currently not supported by QEMU.

Supported devices

- ARM Cortex-M3, Cortex M4F
- Analog to Digital Converter (ADC)
- EXTI interrupt
- Serial ports (USART)
- SPI controller
- System configuration (SYSCFG)
- Timer controller (TIMER)

Missing devices

- Camera interface (DCMI)
- Controller Area Network (CAN)
- Cycle Redundancy Check (CRC) calculation unit
- Digital to Analog Converter (DAC)
- DMA controller
- Ethernet controller
- Flash Interface Unit
- GPIO controller
- I2C controller
- Inter-Integrated Sound (I2S) controller
- Power supply configuration (PWR)
- Random Number Generator (RNG)
- Real-Time Clock (RTC) controller
- Reset and Clock Controller (RCC)
- Secure Digital Input/Output (SDIO) interface
- USB OTG
- Watchdog controller (IWDG, WWDG)

Boot options

The STM32 machines can be started using the `-kernel` option to load a firmware. Example:

```
$ qemu-system-arm -M stm32vldiscovery -kernel firmware.bin
```

'virt' generic virtual platform (`virt`)

The `virt` board is a platform which does not correspond to any real hardware; it is designed for use in virtual machines. It is the recommended board type if you simply want to run a guest such as Linux and do not care about reproducing the idiosyncrasies and limitations of a particular bit of real-world hardware.

This is a "versioned" board model, so as well as the `virt` machine type itself (which may have improvements, bugfixes and other minor changes between QEMU versions) a version is provided that guarantees to have the same behaviour as that of previous QEMU releases, so that VM migration will work between QEMU versions. For instance the `virt-5.0` machine type will behave like the `virt` machine from the QEMU 5.0 release, and migration should work between `virt-5.0` of the 5.0 release and `virt-5.0` of the 5.1 release. Migration is not guaranteed to work between different QEMU releases for the non-versioned `virt` machine type.

Supported devices

The virt board supports:

- PCI/PCIe devices
- Flash memory
- One PL011 UART
- An RTC
- The fw_cfg device that allows a guest to obtain data from QEMU
- A PL061 GPIO controller
- An optional SMMUv3 IOMMU
- hotpluggable DIMMs
- hotpluggable NVDIMMs
- An MSI controller (GICv2M or ITS). GICv2M is selected by default along with GICv2. ITS is selected by default with GICv3 (>= virt-2.7). Note that ITS is not modeled in TCG mode.
- 32 virtio-mmio transport devices
- running guests using the KVM accelerator on aarch64 hardware
- large amounts of RAM (at least 255GB, and more if using highmem)
- many CPUs (up to 512 if using a GICv3 and highmem)
- Secure-World-only devices if the CPU has TrustZone:
  - A second PL011 UART
  - A second PL061 GPIO controller, with GPIO lines for triggering a system reset or system poweroff
  - A secure flash memory
  - 16MB of secure RAM

Supported guest CPU types:

- `cortex-a7` (32-bit)
- `cortex-a15` (32-bit; the default)
- `cortex-a35` (64-bit)
- `cortex-a53` (64-bit)

- cortex-a55 (64-bit)
- cortex-a57 (64-bit)
- cortex-a72 (64-bit)
- cortex-a76 (64-bit)
- cortex-a710 (64-bit)
- a64fx (64-bit)
- host (with KVM only)
- neoverse-n1 (64-bit)
- neoverse-v1 (64-bit)
- neoverse-n2 (64-bit)
- max (same as host for KVM; best possible emulation with TCG)

Note that the default is cortex-a15, so for an AArch64 guest you must specify a CPU type.

Also, please note that passing max CPU (i.e. -cpu max) won't enable all the CPU features for a given virt machine. Where a CPU architectural feature requires support in both the CPU itself and in the wider system (e.g. the MTE feature), it may not be enabled by default, but instead requires a machine option to enable it.

For example, MTE support must be enabled with -machine virt,mte=on, as well as by selecting an MTE-capable CPU (e.g., max) with the -cpu option.

See the machine-specific options below, or check them for a given machine by passing the help suboption, like: -machine virt-9.0,help.

Graphics output is available, but unlike the x86 PC machine types there is no default display device enabled: you should select one from the Display devices section of "-device help". The recommended option is virtio-gpu-pci; this is the only one which will work correctly with KVM. You may also need to ensure your guest kernel is configured with support for this; see below.

Machine-specific options

The following machine-specific options are supported:

secure
Set on/off to enable/disable emulating a guest CPU which implements the Arm Security Extensions (TrustZone). The default is off.

virtualization
Set on/off to enable/disable emulating a guest CPU which implements the Arm Virtualization Extensions. The default is off.

mte
Set on/off to enable/disable emulating a guest CPU which implements the Arm Memory Tagging Extensions. The default is off.

highmem
Set on/off to enable/disable placing devices and RAM in physical address space above 32 bits. The default is on for machine types later than virt-2.12 when the CPU supports an address space bigger than 32 bits (i.e. 64-bit CPUs, and 32-bit CPUs with the Large Physical Address Extension (LPAE) feature). If you want to boot a 32-bit kernel which does not have CONFIG_LPAE enabled on a CPU type which implements LPAE, you will need to manually set this to off; otherwise some devices, such as the PCI controller, will not be accessible.

compact-highmem
Set on/off to enable/disable the compact layout for high memory regions. The default is on for machine types later than virt-7.2.

highmem-redists
Set on/off to enable/disable the high memory region for GICv3 or GICv4 redistributor. The default is on. Setting this to off will limit the maximum number of CPUs when GICv3 or GICv4 is used.

highmem-ecam
Set on/off to enable/disable the high memory region for PCI ECAM. The default is on for machine types later than virt-3.0.

highmem-mmio
Set on/off to enable/disable the high memory region for PCI MMIO. The default is on.

gic-version
Specify the version of the Generic Interrupt Controller (GIC) to provide. Valid values are:

2
GICv2. Note that this limits the number of CPUs to 8.

3
GICv3. This allows up to 512 CPUs.

4
GICv4. Requires virtualization to be on; allows up to 317 CPUs.

host
Use the same GIC version the host provides, when using KVM

max
Use the best GIC version possible (same as host when using KVM; with TCG this is currently 3 if virtualization is off and 4 if virtualization is on, but this may change in future)

its
Set on/off to enable/disable ITS instantiation. The default is on for machine types later than virt-2.7.

iommu
Set the IOMMU type to create for the guest. Valid values are:

none
Don't create an IOMMU (the default)

smmuv3
Create an SMMUv3

ras
Set on/off to enable/disable reporting host memory errors to a guest using ACPI and guest external abort exceptions. The default is off.

dtb-randomness
Set on/off to pass random seeds via the guest DTB rng-seed and kaslr-seed nodes (in both "/chosen" and "/secure-chosen") to use for features like the random number generator and address space randomisation. The default is on. You will want to disable it if your trusted boot chain will verify the DTB it is passed, since this option causes the DTB to be non-deterministic. It would be the responsibility of the firmware to come up with a seed and pass it on if it wants to.

dtb-kaslr-seed
A deprecated synonym for dtb-randomness.

Linux guest kernel configuration

The 'defconfig' for Linux arm and arm64 kernels should include the right device drivers for virtio and the PCI controller; however some older kernel versions, especially for 32-bit Arm, did not have everything enabled by default. If you're not seeing PCI devices that you expect, then check that your guest config has:

```
CONFIG_PCI=y
CONFIG_VIRTIO_PCI=y
CONFIG_PCI_HOST_GENERIC=y
```

If you want to use the `virtio-gpu-pci` graphics device you will also need:

```
CONFIG_DRM=y
CONFIG_DRM_VIRTIO_GPU=y
```

Hardware configuration information for bare-metal programming

The `virt` board automatically generates a device tree blob ("dtb") which it passes to the guest. This provides information about the addresses, interrupt lines and other configuration of the various devices in the system. Guest code can rely on and hard-code the following addresses:

- Flash memory starts at address 0x0000_0000
- RAM starts at 0x4000_0000

All other information about device locations may change between QEMU versions, so guest code must look in the DTB.

QEMU supports two types of guest image boot for `virt`, and the way for the guest code to locate the dtb binary differs:

- For guests using the Linux kernel boot protocol (this means any non-ELF file passed to the QEMU `-kernel` option) the address of the DTB is passed in a register (`r2` for 32-bit guests, or `x0` for 64-bit guests)
- For guests booting as "bare-metal" (any other kind of boot), the DTB is at the start of RAM (0x4000_0000)

Xilinx Versal Virt (`xlnx-versal-virt`)

Xilinx Versal is a family of heterogeneous multi-core SoCs (System on Chip) that combine traditional hardened CPUs and I/O peripherals in a Processing System (PS) with runtime programmable FPGA logic (PL) and an Artificial Intelligence Engine (AIE).

More details here: https://www.xilinx.com/products/silicon-devices/acap/versal.html

The family of Versal SoCs share a single architecture but come in different parts with different speed grades, amounts of PL and other differences.

The Xilinx Versal Virt board in QEMU is a model of a virtual board (does not exist in reality) with a virtual Versal SoC without I/O limitations. Currently, we support the following cores and devices:

Implemented CPU cores:

- 2 ACPUs (ARM Cortex-A72)

Implemented devices:

- Interrupt controller (ARM GICv3)
- 2 UARTs (ARM PL011)

- An RTC (Versal built-in)
- 2 GEMs (Cadence MACB Ethernet MACs)
- 8 ADMA (Xilinx zDMA) channels
- 2 SD Controllers
- OCM (256KB of On Chip Memory)
- XRAM (4MB of on chip Accelerator RAM)
- DDR memory
- BBRAM (36 bytes of Battery-backed RAM)
- eFUSE (3072 bytes of one-time field-programmable bit array)
- 2 CANFDs

QEMU does not yet model any other devices, including the PL and the AI Engine.

Other differences between the hardware and the QEMU model:

- QEMU allows the amount of DDR memory provided to be specified with the `-m` argument. If a DTB is provided on the command line then QEMU will edit it to include suitable entries describing the Versal DDR memory ranges.
- QEMU provides 8 virtio-mmio virtio transports; these start at address `0xa0000000` and have IRQs from 111 and upwards.

Running

If the user provides an Operating System to be loaded, we expect users to use the `-kernel` command line option.

Users can load firmware or boot-loaders with the `-device loader` options.

When loading an OS, QEMU generates a DTB and selects an appropriate address where it gets loaded. This DTB will be passed to the kernel in register x0.

If there's no `-kernel` option, we generate a DTB and place it at 0x1000 for boot-loaders or firmware to pick it up.

If users want to provide their own DTB, they can use the `-dtb` option. These DTBs will have their memory nodes modified to match QEMU's selected ram_size option before they get passed to the kernel or FW.

When loading an OS, we turn on QEMU's PSCI implementation with SMC as the PSCI conduit. When there's no `-kernel` option, we assume the user provides EL3 firmware to handle PSCI.

A few examples:

Direct Linux boot of a generic ARM64 upstream Linux kernel:

```
$ qemu-system-aarch64 -M xlnx-versal-virt -m 2G \
    -serial mon:stdio -display none \
    -kernel arch/arm64/boot/Image \
    -nic user -nic user \
    -device virtio-rng-device,bus=virtio-mmio-bus.0 \
    -drive if=none,index=0,file=hd0.qcow2,id=hd0,snapshot \
    -drive file=qemu_sd.qcow2,if=sd,index=0,snapshot \
    -device virtio-blk-device,drive=hd0 -append root=/dev/vda
```

Direct Linux boot of PetaLinux 2019.2:

```
$ qemu-system-aarch64  -M xlnx-versal-virt -m 2G \
    -serial mon:stdio -display none \
    -kernel petalinux-v2019.2/Image \
    -append "rdinit=/sbin/init console=ttyAMA0,115200n8 earlycon=pl011,mmio,0xFF000000,
↪115200n8" \
    -net nic,model=cadence_gem,netdev=net0 -netdev user,id=net0 \
    -device virtio-rng-device,bus=virtio-mmio-bus.0,rng=rng0 \
    -object rng-random,filename=/dev/urandom,id=rng0
```

Boot PetaLinux 2019.2 via ARM Trusted Firmware (2018.3 because the 2019.2 version of ATF tries to configure the CCI which we don't model) and U-boot:

```
$ qemu-system-aarch64 -M xlnx-versal-virt -m 2G \
    -serial stdio -display none \
    -device loader,file=petalinux-v2018.3/bl31.elf,cpu-num=0 \
    -device loader,file=petalinux-v2019.2/u-boot.elf \
    -device loader,addr=0x20000000,file=petalinux-v2019.2/Image \
    -nic user -nic user \
    -device virtio-rng-device,bus=virtio-mmio-bus.0,rng=rng0 \
    -object rng-random,filename=/dev/urandom,id=rng0
```

Run the following at the U-Boot prompt:

```
Versal>
fdt addr $fdtcontroladdr
fdt move $fdtcontroladdr 0x40000000
fdt set /timer clock-frequency <0x3dfd240>
setenv bootargs "rdinit=/sbin/init maxcpus=1 console=ttyAMA0,115200n8 earlycon=pl011,
↪mmio,0xFF000000,115200n8"
booti 20000000 - 40000000
fdt addr $fdtcontroladdr
```

Boot Linux as DOM0 on Xen via U-Boot:

```
$ qemu-system-aarch64 -M xlnx-versal-virt -m 4G \
    -serial stdio -display none \
    -device loader,file=petalinux-v2019.2/u-boot.elf,cpu-num=0 \
    -device loader,addr=0x30000000,file=linux/2018-04-24/xen \
    -device loader,addr=0x40000000,file=petalinux-v2019.2/Image \
    -nic user -nic user \
    -device virtio-rng-device,bus=virtio-mmio-bus.0,rng=rng0 \
    -object rng-random,filename=/dev/urandom,id=rng0
```

Run the following at the U-Boot prompt:

```
Versal>
fdt addr $fdtcontroladdr
fdt move $fdtcontroladdr 0x20000000
fdt set /timer clock-frequency <0x3dfd240>
fdt set /chosen xen,xen-bootargs "console=dtuart dtuart=/uart@ff000000 dom0_mem=640M␣
↪bootscrub=0 maxcpus=1 timer_slop=0"
fdt set /chosen xen,dom0-bootargs "rdinit=/sbin/init clk_ignore_unused console=hvc0␣
↪maxcpus=1"
```

(continues on next page)

(continued from previous page)

```
fdt mknode /chosen dom0
fdt set /chosen/dom0 compatible "xen,multiboot-module"
fdt set /chosen/dom0 reg <0x00000000 0x40000000 0x0 0x03100000>
booti 30000000 - 20000000
```

Boot Linux as Dom0 on Xen via ARM Trusted Firmware and U-Boot:

```
$ qemu-system-aarch64 -M xlnx-versal-virt -m 4G \
    -serial stdio -display none \
    -device loader,file=petalinux-v2018.3/bl31.elf,cpu-num=0 \
    -device loader,file=petalinux-v2019.2/u-boot.elf \
    -device loader,addr=0x30000000,file=linux/2018-04-24/xen \
    -device loader,addr=0x40000000,file=petalinux-v2019.2/Image \
    -nic user -nic user \
    -device virtio-rng-device,bus=virtio-mmio-bus.0,rng=rng0 \
    -object rng-random,filename=/dev/urandom,id=rng0
```

Run the following at the U-Boot prompt:

```
Versal>
fdt addr $fdtcontroladdr
fdt move $fdtcontroladdr 0x20000000
fdt set /timer clock-frequency <0x3dfd240>
fdt set /chosen xen,xen-bootargs "console=dtuart dtuart=/uart@ff000000 dom0_mem=640M␣
↪bootscrub=0 maxcpus=1 timer_slop=0"
fdt set /chosen xen,dom0-bootargs "rdinit=/sbin/init clk_ignore_unused console=hvc0␣
↪maxcpus=1"
fdt mknode /chosen dom0
fdt set /chosen/dom0 compatible "xen,multiboot-module"
fdt set /chosen/dom0 reg <0x00000000 0x40000000 0x0 0x03100000>
booti 30000000 - 20000000
```

BBRAM File Backend

BBRAM can have an optional file backend, which must be a seekable binary file with a size of 36 bytes or larger. A file with all binary 0s is a 'blank'.

To add a file-backend for the BBRAM:

```
-drive if=pflash,index=0,file=versal-bbram.bin,format=raw
```

To use a different index value, N, from default of 0, add:

```
-global driver=xlnx.bbram-ctrl,property=drive-index,value=N
```

eFUSE File Backend

eFUSE can have an optional file backend, which must be a seekable binary file with a size of 3072 bytes or larger. A file with all binary 0s is a 'blank'.

To add a file-backend for the eFUSE:

```
-drive if=pflash,index=1,file=versal-efuse.bin,format=raw
```

To use a different index value, N, from default of 1, add:

```
-global xlnx-efuse.drive-index=N
```

Warning: In actual physical Versal, BBRAM and eFUSE contain sensitive data. The QEMU device models do **not** encrypt nor obfuscate any data when holding them in models' memory or when writing them to their file backends.

Thus, a file backend should be used with caution, and 'format=luks' is highly recommended (albeit with usage complexity).

Better yet, do not use actual product data when running guest image on this Xilinx Versal Virt board.

Using CANFDs for Versal Virt

Versal CANFD controller is developed based on SocketCAN and QEMU CAN bus implementation. Bus connection and socketCAN connection for each CAN module can be set through command lines.

To connect both CANFD0 and CANFD1 on the same bus:

```
-object can-bus,id=canbus -machine canbus0=canbus -machine canbus1=canbus
```

To connect CANFD0 and CANFD1 to separate buses:

```
-object can-bus,id=canbus0 -object can-bus,id=canbus1 \
-machine canbus0=canbus0 -machine canbus1=canbus1
```

The SocketCAN interface can connect to a Physical or a Virtual CAN interfaces on the host machine. Please check this document to learn about CAN interface on Linux: docs/system/devices/can.rst

To connect CANFD0 and CANFD1 to host machine's CAN interface can0:

```
-object can-bus,id=canbus -machine canbus0=canbus -machine canbus1=canbus
-object can-host-socketcan,id=canhost0,if=can0,canbus=canbus
```

Xen Device Emulation Backend (`xenpvh`)

This machine is a little unusual compared to others as QEMU just acts as an IOREQ server to register/connect with Xen Hypervisor. Control of the VMs themselves is left to the Xen tooling.

When TPM is enabled, this machine also creates a tpm-tis-device at a user input tpm base address, adds a TPM emulator and connects to a swtpm application running on host machine via chardev socket. This enables xenpvh to support TPM functionalities for a guest domain.

More information about TPM use and installing swtpm linux application can be found in the *QEMU TPM Device* section.

Example for starting swtpm on host machine:

```
mkdir /tmp/vtpm2
swtpm socket --tpmstate dir=/tmp/vtpm2 \
  --ctrl type=unixio,path=/tmp/vtpm2/swtpm-sock &
```

Sample QEMU xenpvh commands for running and connecting with Xen:

```
qemu-system-aarch64 -xen-domid 1 \
  -chardev socket,id=libxl-cmd,path=qmp-libxl-1,server=on,wait=off \
  -mon chardev=libxl-cmd,mode=control \
  -chardev socket,id=libxenstat-cmd,path=qmp-libxenstat-1,server=on,wait=off \
  -mon chardev=libxenstat-cmd,mode=control \
  -xen-attach -name guest0 -vnc none -display none -nographic \
  -machine xenpvh -m 1301 \
  -chardev socket,id=chrtpm,path=tmp/vtpm2/swtpm-sock \
  -tpmdev emulator,id=tpm0,chardev=chrtpm -machine tpm-base-addr=0x0C000000
```

In above QEMU command, last two lines are for connecting xenpvh QEMU to swtpm via chardev socket.

Emulated CPU architecture support

A-profile CPU architecture support

QEMU's TCG emulation includes support for the Armv5, Armv6, Armv7 and Armv8 versions of the A-profile architecture. It also has support for the following architecture extensions:

- FEAT_AA32BF16 (AArch32 BFloat16 instructions)
- FEAT_AA32EL0 (Support for AArch32 at EL0)
- FEAT_AA32EL1 (Support for AArch32 at EL1)
- FEAT_AA32EL2 (Support for AArch32 at EL2)
- FEAT_AA32EL3 (Support for AArch32 at EL3)
- FEAT_AA32HPD (AArch32 hierarchical permission disables)
- FEAT_AA32I8MM (AArch32 Int8 matrix multiplication instructions)
- FEAT_AA64EL0 (Support for AArch64 at EL0)
- FEAT_AA64EL1 (Support for AArch64 at EL1)
- FEAT_AA64EL2 (Support for AArch64 at EL2)
- FEAT_AA64EL3 (Support for AArch64 at EL3)
- FEAT_AdvSIMD (Advanced SIMD Extension)
- FEAT_AES (AESD and AESE instructions)
- FEAT_Armv9_Crypto (Armv9 Cryptographic Extension)
- FEAT_ASID16 (16 bit ASID)
- FEAT_BBM at level 2 (Translation table break-before-make levels)
- FEAT_BF16 (AArch64 BFloat16 instructions)
- FEAT_BTI (Branch Target Identification)

- FEAT_CCIDX (Extended cache index)
- FEAT_CRC32 (CRC32 instructions)
- FEAT_Crypto (Cryptographic Extension)
- FEAT_CSV2 (Cache speculation variant 2)
- FEAT_CSV2_1p1 (Cache speculation variant 2, version 1.1)
- FEAT_CSV2_1p2 (Cache speculation variant 2, version 1.2)
- FEAT_CSV2_2 (Cache speculation variant 2, version 2)
- FEAT_CSV2_3 (Cache speculation variant 2, version 3)
- FEAT_CSV3 (Cache speculation variant 3)
- FEAT_DGH (Data gathering hint)
- FEAT_DIT (Data Independent Timing instructions)
- FEAT_DPB (DC CVAP instruction)
- FEAT_DPB2 (DC CVADP instruction)
- FEAT_Debugv8p1 (Debug with VHE)
- FEAT_Debugv8p2 (Debug changes for v8.2)
- FEAT_Debugv8p4 (Debug changes for v8.4)
- FEAT_DotProd (Advanced SIMD dot product instructions)
- FEAT_DoubleFault (Double Fault Extension)
- FEAT_E0PD (Preventing EL0 access to halves of address maps)
- FEAT_ECV (Enhanced Counter Virtualization)
- FEAT_EL0 (Support for execution at EL0)
- FEAT_EL1 (Support for execution at EL1)
- FEAT_EL2 (Support for execution at EL2)
- FEAT_EL3 (Support for execution at EL3)
- FEAT_EPAC (Enhanced pointer authentication)
- FEAT_ETS2 (Enhanced Translation Synchronization)
- FEAT_EVT (Enhanced Virtualization Traps)
- FEAT_F32MM (Single-precision Matrix Multiplication)
- FEAT_F64MM (Double-precision Matrix Multiplication)
- FEAT_FCMA (Floating-point complex number instructions)
- FEAT_FGT (Fine-Grained Traps)
- FEAT_FHM (Floating-point half-precision multiplication instructions)
- FEAT_FP (Floating Point extensions)
- FEAT_FP16 (Half-precision floating-point data processing)
- FEAT_FPAC (Faulting on AUT* instructions)
- FEAT_FPACCOMBINE (Faulting on combined pointer authentication instructions)

- FEAT_FPACC_SPEC (Speculative behavior of combined pointer authentication instructions)
- FEAT_FRINTTS (Floating-point to integer instructions)
- FEAT_FlagM (Flag manipulation instructions v2)
- FEAT_FlagM2 (Enhancements to flag manipulation instructions)
- FEAT_GTG (Guest translation granule size)
- FEAT_HAFDBS (Hardware management of the access flag and dirty bit state)
- FEAT_HBC (Hinted conditional branches)
- FEAT_HCX (Support for the HCRX_EL2 register)
- FEAT_HPDS (Hierarchical permission disables)
- FEAT_HPDS2 (Translation table page-based hardware attributes)
- FEAT_HPMN0 (Setting of MDCR_EL2.HPMN to zero)
- FEAT_I8MM (AArch64 Int8 matrix multiplication instructions)
- FEAT_IDST (ID space trap handling)
- FEAT_IESB (Implicit error synchronization event)
- FEAT_JSCVT (JavaScript conversion instructions)
- FEAT_LOR (Limited ordering regions)
- FEAT_LPA (Large Physical Address space)
- FEAT_LPA2 (Large Physical and virtual Address space v2)
- FEAT_LRCPC (Load-acquire RCpc instructions)
- FEAT_LRCPC2 (Load-acquire RCpc instructions v2)
- FEAT_LSE (Large System Extensions)
- FEAT_LSE2 (Large System Extensions v2)
- FEAT_LVA (Large Virtual Address space)
- FEAT_MixedEnd (Mixed-endian support)
- FEAT_MixdEndEL0 (Mixed-endian support at EL0)
- FEAT_MOPS (Standardization of memory operations)
- FEAT_MTE (Memory Tagging Extension)
- FEAT_MTE2 (Memory Tagging Extension)
- FEAT_MTE3 (MTE Asymmetric Fault Handling)
- FEAT_MTE_ASYM_FAULT (Memory tagging asymmetric faults)
- FEAT_NMI (Non-maskable Interrupt)
- FEAT_NV (Nested Virtualization)
- FEAT_NV2 (Enhanced nested virtualization support)
- FEAT_PACIMP (Pointer authentication - IMPLEMENTATION DEFINED algorithm)
- FEAT_PACQARMA3 (Pointer authentication - QARMA3 algorithm)
- FEAT_PACQARMA5 (Pointer authentication - QARMA5 algorithm)

- FEAT_PAN (Privileged access never)
- FEAT_PAN2 (AT S1E1R and AT S1E1W instruction variants affected by PSTATE.PAN)
- FEAT_PAN3 (Support for SCTLR_ELx.EPAN)
- FEAT_PAuth (Pointer authentication)
- FEAT_PAuth2 (Enhancements to pointer authentication)
- FEAT_PMULL (PMULL, PMULL2 instructions)
- FEAT_PMUv3 (PMU extension version 3)
- FEAT_PMUv3p1 (PMU Extensions v3.1)
- FEAT_PMUv3p4 (PMU Extensions v3.4)
- FEAT_PMUv3p5 (PMU Extensions v3.5)
- FEAT_RAS (Reliability, availability, and serviceability)
- FEAT_RASv1p1 (RAS Extension v1.1)
- FEAT_RDM (Advanced SIMD rounding double multiply accumulate instructions)
- FEAT_RME (Realm Management Extension) (NB: support status in QEMU is experimental)
- FEAT_RNG (Random number generator)
- FEAT_S2FWB (Stage 2 forced Write-Back)
- FEAT_SB (Speculation Barrier)
- FEAT_SEL2 (Secure EL2)
- FEAT_SHA1 (SHA1 instructions)
- FEAT_SHA256 (SHA256 instructions)
- FEAT_SHA3 (Advanced SIMD SHA3 instructions)
- FEAT_SHA512 (Advanced SIMD SHA512 instructions)
- FEAT_SM3 (Advanced SIMD SM3 instructions)
- FEAT_SM4 (Advanced SIMD SM4 instructions)
- FEAT_SME (Scalable Matrix Extension)
- FEAT_SME_FA64 (Full A64 instruction set in Streaming SVE mode)
- FEAT_SME_F64F64 (Double-precision floating-point outer product instructions)
- FEAT_SME_I16I64 (16-bit to 64-bit integer widening outer product instructions)
- FEAT_SVE (Scalable Vector Extension)
- FEAT_SVE_AES (Scalable Vector AES instructions)
- FEAT_SVE_BitPerm (Scalable Vector Bit Permutes instructions)
- FEAT_SVE_PMULL128 (Scalable Vector PMULL instructions)
- FEAT_SVE_SHA3 (Scalable Vector SHA3 instructions)
- FEAT_SVE_SM4 (Scalable Vector SM4 instructions)
- FEAT_SVE2 (Scalable Vector Extension version 2)
- FEAT_SPECRES (Speculation restriction instructions)

- FEAT_SSBS (Speculative Store Bypass Safe)
- FEAT_TGran16K (Support for 16KB memory translation granule size at stage 1)
- FEAT_TGran4K (Support for 4KB memory translation granule size at stage 1)
- FEAT_TGran64K (Support for 64KB memory translation granule size at stage 1)
- FEAT_TIDCP1 (EL0 use of IMPLEMENTATION DEFINED functionality)
- FEAT_TLBIOS (TLB invalidate instructions in Outer Shareable domain)
- FEAT_TLBIRANGE (TLB invalidate range instructions)
- FEAT_TTCNP (Translation table Common not private translations)
- FEAT_TTL (Translation Table Level)
- FEAT_TTST (Small translation tables)
- FEAT_UAO (Unprivileged Access Override control)
- FEAT_VHE (Virtualization Host Extensions)
- FEAT_VMID16 (16-bit VMID)
- FEAT_XNX (Translation table stage 2 Unprivileged Execute-never)

For information on the specifics of these extensions, please refer to the Armv8-A Arm Architecture Reference Manual.

When a specific named CPU is being emulated, only those features which are present in hardware for that CPU are emulated. (If a feature is not in the list above then it is not supported, even if the real hardware should have it.) The `max` CPU enables all features.

R-profile CPU architecture support

QEMU's TCG emulation support for R-profile CPUs is currently limited. We emulate only the Cortex-R5 and Cortex-R5F CPUs.

M-profile CPU architecture support

QEMU's TCG emulation includes support for Armv6-M, Armv7-M, Armv8-M, and Armv8.1-M versions of the M-profile architucture. It also has support for the following architecture extensions:

- FP (Floating-point Extension)
- FPCXT (FPCXT access instructions)
- HP (Half-precision floating-point instructions)
- LOB (Low Overhead loops and Branch future)
- M (Main Extension)
- MPU (Memory Protection Unit Extension)
- PXN (Privileged Execute Never)
- RAS (Reliability, Serviceability and Availability): "minimum RAS Extension" only
- S (Security Extension)
- ST (System Timer Extension)

For information on the specifics of these extensions, please refer to the Armv8-M Arm Architecture Reference Manual.

When a specific named CPU is being emulated, only those features which are present in hardware for that CPU are emulated. (If a feature is not in the list above then it is not supported, even if the real hardware should have it.) There is no equivalent of the `max` CPU for M-profile.

Arm CPU features

Arm CPU Features

CPU features are optional features that a CPU of supporting type may choose to implement or not. In QEMU, optional CPU features have corresponding boolean CPU proprieties that, when enabled, indicate that the feature is implemented, and, conversely, when disabled, indicate that it is not implemented. An example of an Arm CPU feature is the Performance Monitoring Unit (PMU). CPU types such as the Cortex-A15 and the Cortex-A57, which respectively implement Arm architecture reference manuals ARMv7-A and ARMv8-A, may both optionally implement PMUs. For example, if a user wants to use a Cortex-A15 without a PMU, then the `-cpu` parameter should contain `pmu=off` on the QEMU command line, i.e. `-cpu cortex-a15,pmu=off`.

As not all CPU types support all optional CPU features, then whether or not a CPU property exists depends on the CPU type. For example, CPUs that implement the ARMv8-A architecture reference manual may optionally support the AArch32 CPU feature, which may be enabled by disabling the `aarch64` CPU property. A CPU type such as the Cortex-A15, which does not implement ARMv8-A, will not have the `aarch64` CPU property.

QEMU's support may be limited for some CPU features, only partially supporting the feature or only supporting the feature under certain configurations. For example, the `aarch64` CPU feature, which, when disabled, enables the optional AArch32 CPU feature, is only supported when using the KVM accelerator and when running on a host CPU type that supports the feature. While `aarch64` currently only works with KVM, it could work with TCG. CPU features that are specific to KVM are prefixed with "kvm-" and are described in "KVM VCPU Features".

CPU Feature Probing

Determining which CPU features are available and functional for a given CPU type is possible with the `query-cpu-model-expansion` QMP command. Below are some examples where `scripts/qmp/qmp-shell` (see the top comment block in the script for usage) is used to issue the QMP commands.

1. Determine which CPU features are available for the `max` CPU type (Note, we started QEMU with qemu-system-aarch64, so `max` is implementing the ARMv8-A reference manual in this case):

```
(QEMU) query-cpu-model-expansion type=full model={"name":"max"}
{ "return": {
  "model": { "name": "max", "props": {
  "sve1664": true, "pmu": true, "sve1792": true, "sve1920": true,
  "sve128": true, "aarch64": true, "sve1024": true, "sve": true,
  "sve640": true, "sve768": true, "sve1408": true, "sve256": true,
  "sve1152": true, "sve512": true, "sve384": true, "sve1536": true,
  "sve896": true, "sve1280": true, "sve2048": true
}}}}
```

We see that the `max` CPU type has the `pmu`, `aarch64`, `sve`, and many `sve<N>` CPU features. We also see that all the CPU features are enabled, as they are all `true`. (The `sve<N>` CPU features are all optional SVE vector lengths (see "SVE CPU Properties"). While with TCG all SVE vector lengths can be supported, when KVM is in use it's more likely that only a few lengths will be supported, if SVE is supported at all.)

(2) Let's try to disable the PMU:

```
(QEMU) query-cpu-model-expansion type=full model={"name":"max","props":{"pmu":false}
↪}
{ "return": {
  "model": { "name": "max", "props": {
  "sve1664": true, "pmu": false, "sve1792": true, "sve1920": true,
  "sve128": true, "aarch64": true, "sve1024": true, "sve": true,
  "sve640": true, "sve768": true, "sve1408": true, "sve256": true,
  "sve1152": true, "sve512": true, "sve384": true, "sve1536": true,
  "sve896": true, "sve1280": true, "sve2048": true
}}}}
```

We see it worked, as `pmu` is now `false`.

(3) Let's try to disable `aarch64`, which enables the AArch32 CPU feature:

```
(QEMU) query-cpu-model-expansion type=full model={"name":"max","props":{"aarch64
↪":false}}
{"error": {
 "class": "GenericError", "desc":
 "'aarch64' feature cannot be disabled unless KVM is enabled and 32-bit EL1 is␣
↪supported"
}}
```

It looks like this feature is limited to a configuration we do not currently have.

(4) Let's disable `sve` and see what happens to all the optional SVE vector lengths:

```
(QEMU) query-cpu-model-expansion type=full model={"name":"max","props":{"sve":false}
↪}
{ "return": {
  "model": { "name": "max", "props": {
  "sve1664": false, "pmu": true, "sve1792": false, "sve1920": false,
  "sve128": false, "aarch64": true, "sve1024": false, "sve": false,
  "sve640": false, "sve768": false, "sve1408": false, "sve256": false,
  "sve1152": false, "sve512": false, "sve384": false, "sve1536": false,
  "sve896": false, "sve1280": false, "sve2048": false
}}}}
```

As expected they are now all `false`.

(5) Let's try probing CPU features for the Cortex-A15 CPU type:

```
(QEMU) query-cpu-model-expansion type=full model={"name":"cortex-a15"}
{"return": {"model": {"name": "cortex-a15", "props": {"pmu": true}}}}
```

Only the `pmu` CPU feature is available.

A note about CPU feature dependencies

It's possible for features to have dependencies on other features. I.e. it may be possible to change one feature at a time without error, but when attempting to change all features at once an error could occur depending on the order they are processed. It's also possible changing all at once doesn't generate an error, because a feature's dependencies are satisfied with other features, but the same feature cannot be changed independently without error. For these reasons callers should always attempt to make their desired changes all at once in order to ensure the collection is valid.

A note about CPU models and KVM

Named CPU models generally do not work with KVM. There are a few cases that do work, e.g. using the named CPU model `cortex-a57` with KVM on a seattle host, but mostly if KVM is enabled the `host` CPU type must be used. This means the guest is provided all the same CPU features as the host CPU type has. And, for this reason, the `host` CPU type should enable all CPU features that the host has by default. Indeed it's even a bit strange to allow disabling CPU features that the host has when using the `host` CPU type, but in the absence of CPU models it's the best we can do if we want to launch guests without all the host's CPU features enabled.

Enabling KVM also affects the `query-cpu-model-expansion` QMP command. The affect is not only limited to specific features, as pointed out in example (3) of "CPU Feature Probing", but also to which CPU types may be expanded. When KVM is enabled, only the `max`, `host`, and current CPU type may be expanded. This restriction is necessary as it's not possible to know all CPU types that may work with KVM, but it does impose a small risk of users experiencing unexpected errors. For example on a seattle, as mentioned above, the `cortex-a57` CPU type is also valid when KVM is enabled. Therefore a user could use the `host` CPU type for the current type, but then attempt to query `cortex-a57`, however that query will fail with our restrictions. This shouldn't be an issue though as management layers and users have been preferring the `host` CPU type for use with KVM for quite some time. Additionally, if the KVM-enabled QEMU instance running on a seattle host is using the `cortex-a57` CPU type, then querying `cortex-a57` will work.

Using CPU Features

After determining which CPU features are available and supported for a given CPU type, then they may be selectively enabled or disabled on the QEMU command line with that CPU type:

```
$ qemu-system-aarch64 -M virt -cpu max,pmu=off,sve=on,sve128=on,sve256=on
```

The example above disables the PMU and enables the first two SVE vector lengths for the `max` CPU type. Note, the `sve=on` isn't actually necessary, because, as we observed above with our probe of the `max` CPU type, `sve` is already on by default. Also, based on our probe of defaults, it would seem we need to disable many SVE vector lengths, rather than only enabling the two we want. This isn't the case, because, as disabling many SVE vector lengths would be quite verbose, the `sve<N>` CPU properties have special semantics (see "SVE CPU Property Parsing Semantics").

KVM VCPU Features

KVM VCPU features are CPU features that are specific to KVM, such as paravirt features or features that enable CPU virtualization extensions. The features' CPU properties are only available when KVM is enabled and are named with the prefix "kvm-". KVM VCPU features may be probed, enabled, and disabled in the same way as other CPU features. Below is the list of KVM VCPU features and their descriptions.

kvm-no-adjvtime
: By default kvm-no-adjvtime is disabled. This means that by default the virtual time adjustment is enabled (vtime is not *not* adjusted).

When virtual time adjustment is enabled each time the VM transitions back to running state the VCPU's virtual counter is updated to ensure stopped time is not counted. This avoids time jumps surprising guest OSes and applications, as long as they use the virtual counter for timekeeping. However it has the side effect of the virtual and physical counters diverging. All timekeeping based on the virtual counter will appear to lag behind any timekeeping that does not subtract VM stopped time. The guest may resynchronize its virtual counter with other time sources as needed.

Enable kvm-no-adjvtime to disable virtual time adjustment, also restoring the legacy (pre-5.0) behavior.

kvm-steal-time
Since v5.2, kvm-steal-time is enabled by default when KVM is enabled, the feature is supported, and the guest is 64-bit.

When kvm-steal-time is enabled a 64-bit guest can account for time its CPUs were not running due to the host not scheduling the corresponding VCPU threads. The accounting statistics may influence the guest scheduler behavior and/or be exposed to the guest userspace.

TCG VCPU Features

TCG VCPU features are CPU features that are specific to TCG. Below is the list of TCG VCPU features and their descriptions.

pauth
Enable or disable `FEAT_Pauth` entirely.

pauth-impdef
When `pauth` is enabled, select the QEMU implementation defined algorithm.

pauth-qarma3
When `pauth` is enabled, select the architected QARMA3 algorithm.

Without either `pauth-impdef` or `pauth-qarma3` enabled, the architected QARMA5 algorithm is used. The architected QARMA5 and QARMA3 algorithms have good cryptographic properties, but can be quite slow to emulate. The impdef algorithm used by QEMU is non-cryptographic but significantly faster.

SVE CPU Properties

There are two types of SVE CPU properties: `sve` and `sve<N>`. The first is used to enable or disable the entire SVE feature, just as the `pmu` CPU property completely enables or disables the PMU. The second type is used to enable or disable specific vector lengths, where `N` is the number of bits of the length. The `sve<N>` CPU properties have special dependencies and constraints, see "SVE CPU Property Dependencies and Constraints" below. Additionally, as we want all supported vector lengths to be enabled by default, then, in order to avoid overly verbose command lines (command lines full of `sve<N>=off`, for all `N` not wanted), we provide the parsing semantics listed in "SVE CPU Property Parsing Semantics".

SVE CPU Property Dependencies and Constraints

1) At least one vector length must be enabled when `sve` is enabled.

2) If a vector length `N` is enabled, then, when KVM is enabled, all smaller, host supported vector lengths must also be enabled. If KVM is not enabled, then only all the smaller, power-of-two vector lengths must be enabled. E.g. with KVM if the host supports all vector lengths up to 512-bits (128, 256, 384, 512), then if `sve512` is enabled, the 128-bit vector length, 256-bit vector length, and 384-bit vector length must also be enabled. Without KVM, the 384-bit vector length would not be required.

3) If KVM is enabled then only vector lengths that the host CPU type support may be enabled. If SVE is not supported by the host, then no `sve*` properties may be enabled.

SVE CPU Property Parsing Semantics

1) If SVE is disabled (`sve=off`), then which SVE vector lengths are enabled or disabled is irrelevant to the guest, as the entire SVE feature is disabled and that disables all vector lengths for the guest. However QEMU will still track any `sve<N>` CPU properties provided by the user. If later an `sve=on` is provided, then the guest will get only the enabled lengths. If no `sve=on` is provided and there are explicitly enabled vector lengths, then an error is generated.

2) If SVE is enabled (`sve=on`), but no `sve<N>` CPU properties are provided, then all supported vector lengths are enabled, which when KVM is not in use means including the non-power-of-two lengths, and, when KVM is in use, it means all vector lengths supported by the host processor.

3) If SVE is enabled, then an error is generated when attempting to disable the last enabled vector length (see constraint (1) of "SVE CPU Property Dependencies and Constraints").

4) If one or more vector lengths have been explicitly enabled and at least one of the dependency lengths of the maximum enabled length has been explicitly disabled, then an error is generated (see constraint (2) of "SVE CPU Property Dependencies and Constraints").

5) When KVM is enabled, if the host does not support SVE, then an error is generated when attempting to enable any `sve*` properties (see constraint (3) of "SVE CPU Property Dependencies and Constraints").

6) When KVM is enabled, if the host does support SVE, then an error is generated when attempting to enable any vector lengths not supported by the host (see constraint (3) of "SVE CPU Property Dependencies and Constraints").

7) If one or more `sve<N>` CPU properties are set `off`, but no `sve<N>`, CPU properties are set `on`, then the specified vector lengths are disabled but the default for any unspecified lengths remains enabled. When KVM is not enabled, disabling a power-of-two vector length also disables all vector lengths larger than the power-of-two length. When KVM is enabled, then disabling any supported vector length also disables all larger vector lengths (see constraint (2) of "SVE CPU Property Dependencies and Constraints").

8) If one or more `sve<N>` CPU properties are set to `on`, then they are enabled and all unspecified lengths default to disabled, except for the required lengths per constraint (2) of "SVE CPU Property Dependencies and Constraints", which will even be auto-enabled if they were not explicitly enabled.

9) If SVE was disabled (`sve=off`), allowing all vector lengths to be explicitly disabled (i.e. avoiding the error specified in (3) of "SVE CPU Property Parsing Semantics"), then if later an `sve=on` is provided an error will be generated. To avoid this error, one must enable at least one vector length prior to enabling SVE.

SVE CPU Property Examples

1) Disable SVE:

```
$ qemu-system-aarch64 -M virt -cpu max,sve=off
```

2) Implicitly enable all vector lengths for the `max` CPU type:

```
$ qemu-system-aarch64 -M virt -cpu max
```

3) When KVM is enabled, implicitly enable all host CPU supported vector lengths with the `host` CPU type:

```
$ qemu-system-aarch64 -M virt,accel=kvm -cpu host
```

4) Only enable the 128-bit vector length:

```
$ qemu-system-aarch64 -M virt -cpu max,sve128=on
```

5) Disable the 512-bit vector length and all larger vector lengths, since 512 is a power-of-two. This results in all the smaller, uninitialized lengths (128, 256, and 384) defaulting to enabled:

```
$ qemu-system-aarch64 -M virt -cpu max,sve512=off
```

6) Enable the 128-bit, 256-bit, and 512-bit vector lengths:

```
$ qemu-system-aarch64 -M virt -cpu max,sve128=on,sve256=on,sve512=on
```

7) The same as (6), but since the 128-bit and 256-bit vector lengths are required for the 512-bit vector length to be enabled, then allow them to be auto-enabled:

```
$ qemu-system-aarch64 -M virt -cpu max,sve512=on
```

8) Do the same as (7), but by first disabling SVE and then re-enabling it:

```
$ qemu-system-aarch64 -M virt -cpu max,sve=off,sve512=on,sve=on
```

9) Force errors regarding the last vector length:

```
$ qemu-system-aarch64 -M virt -cpu max,sve128=off
$ qemu-system-aarch64 -M virt -cpu max,sve=off,sve128=off,sve=on
```

SVE CPU Property Recommendations

The examples in "SVE CPU Property Examples" exhibit many ways to select vector lengths which developers may find useful in order to avoid overly verbose command lines. However, the recommended way to select vector lengths is to explicitly enable each desired length. Therefore only example's (1), (4), and (6) exhibit recommended uses of the properties.

SME CPU Property Examples

1) Disable SME:

```
$ qemu-system-aarch64 -M virt -cpu max,sme=off
```

2) Implicitly enable all vector lengths for the `max` CPU type:

```
$ qemu-system-aarch64 -M virt -cpu max
```

3) Only enable the 256-bit vector length:

```
$ qemu-system-aarch64 -M virt -cpu max,sme256=on
```

3) Enable the 256-bit and 1024-bit vector lengths:

```
$ qemu-system-aarch64 -M virt -cpu max,sme256=on,sme1024=on
```

4) Disable the 512-bit vector length. This results in all the other lengths supported by `max` defaulting to enabled (128, 256, 1024 and 2048):

```
$ qemu-system-aarch64 -M virt -cpu max,sve512=off
```

SVE User-mode Default Vector Length Property

For qemu-aarch64, the cpu property `sve-default-vector-length=N` is defined to mirror the Linux kernel parameter file `/proc/sys/abi/sve_default_vector_length`. The default length, `N`, is in units of bytes and must be between 16 and 8192. If not specified, the default vector length is 64.

If the default length is larger than the maximum vector length enabled, the actual vector length will be reduced. Note that the maximum vector length supported by QEMU is 256.

If this property is set to `-1` then the default vector length is set to the maximum possible length.

SME CPU Properties

The SME CPU properties are much like the SVE properties: `sme` is used to enable or disable the entire SME feature, and `sme<N>` is used to enable or disable specific vector lengths. Finally, `sme_fa64` is used to enable or disable `FEAT_SME_FA64`, which allows execution of the "full a64" instruction set while Streaming SVE mode is enabled.

SME is not supported by KVM at this time.

At least one vector length must be enabled when `sme` is enabled, and all vector lengths must be powers of 2. The maximum vector length supported by qemu is 2048 bits. Otherwise, there are no additional constraints on the set of vector lengths supported by SME.

SME User-mode Default Vector Length Property

For qemu-aarch64, the cpu property `sme-default-vector-length=N` is defined to mirror the Linux kernel parameter file `/proc/sys/abi/sme_default_vector_length`. The default length, `N`, is in units of bytes and must be between 16 and 8192. If not specified, the default vector length is 32.

As with `sve-default-vector-length`, if the default length is larger than the maximum vector length enabled, the actual vector length will be reduced. If this property is set to `-1` then the default vector length is set to the maximum possible length.

RME CPU Properties

The status of RME support with QEMU is experimental. At this time we only support RME within the CPU proper, not within the SMMU or GIC. The feature is enabled by the CPU property `x-rme`, with the `x-` prefix present as a reminder of the experimental status, and defaults off.

The method for enabling RME will change in some future QEMU release without notice or backward compatibility.

RME Level 0 GPT Size Property

To aid firmware developers in testing different possible CPU configurations, `x-l0gptsz=S` may be used to specify the value to encode into `GPCCR_EL3.L0GPTSZ`, a read-only field that specifies the size of the Level 0 Granule Protection Table. Legal values for `S` are 30, 34, 36, and 39; the default is 30.

As with `x-rme`, the `x-l0gptsz` property may be renamed or removed in some future QEMU release.

2.23.2 AVR System emulator

Use the executable `qemu-system-avr` to emulate a AVR 8 bit based machine. These can have one of the following cores: avr1, avr2, avr25, avr3, avr31, avr35, avr4, avr5, avr51, avr6, avrtiny, xmega2, xmega3, xmega4, xmega5, xmega6 and xmega7.

As for now it supports few Arduino boards for educational and testing purposes. These boards use a ATmega controller, which model is limited to USART & 16-bit timer devices, enough to run FreeRTOS based applications (like https://github.com/seharris/qemu-avr-tests/blob/master/free-rtos/Demo/AVR_ATMega2560_GCC/demo.elf).

Following are examples of possible usages, assuming demo.elf is compiled for AVR cpu

- Continuous non interrupted execution:

```
qemu-system-avr -machine mega2560 -bios demo.elf
```

- Continuous non interrupted execution with serial output into telnet window:

```
qemu-system-avr -M mega2560 -bios demo.elf -nographic \
                -serial tcp::5678,server=on,wait=off
```

  and then in another shell:

```
telnet localhost 5678
```

- Debugging with GDB debugger:

```
qemu-system-avr -machine mega2560 -bios demo.elf -s -S
```

  and then in another shell:

```
avr-gdb demo.elf
```

  and then within GDB shell:

```
target remote :1234
```

- Print out executed instructions (that have not been translated by the JIT compiler yet):

```
qemu-system-avr -machine mega2560 -bios demo.elf -d in_asm
```

2.23.3 ColdFire System emulator

Use the executable `qemu-system-m68k` to simulate a ColdFire machine. The emulator is able to boot a uClinux kernel.

The M5208EVB emulation includes the following devices:

- MCF5208 ColdFire V2 Microprocessor (ISA A+ with EMAC).
- Three Two on-chip UARTs.
- Fast Ethernet Controller (FEC)

The AN5206 emulation includes the following devices:

- MCF5206 ColdFire V2 Microprocessor.
- Two on-chip UARTs.

2.23.4 MIPS System emulator

Four executables cover simulation of 32 and 64-bit MIPS systems in both endian options, `qemu-system-mips`, `qemu-system-mipsel` `qemu-system-mips64` and `qemu-system-mips64el`. Five different machine types are emulated:

- The MIPS Malta prototype board "malta"
- An ACER Pica "pica61". This machine needs the 64-bit emulator.
- MIPS emulator pseudo board "mipssim"
- A MIPS Magnum R4000 machine "magnum". This machine needs the 64-bit emulator.

The Malta emulation supports the following devices:

- Core board with MIPS 24Kf CPU and Galileo system controller
- PIIX4 PCI/USB/SMbus controller
- The Multi-I/O chip's serial device
- PCI network cards (PCnet32 and others)
- Malta FPGA serial device
- Cirrus (default) or any other PCI VGA graphics card

The Boston board emulation supports the following devices:

- Xilinx FPGA, which includes a PCIe root port and an UART
- Intel EG20T PCH connects the I/O peripherals, but only the SATA bus is emulated

The ACER Pica emulation supports:

- MIPS R4000 CPU
- PC-style IRQ and DMA controllers
- PC Keyboard
- IDE controller

The MIPS Magnum R4000 emulation supports:

- MIPS R4000 CPU
- PC-style IRQ controller

- PC Keyboard
- SCSI controller
- G364 framebuffer

The Fuloong 2E emulation supports:

- Loongson 2E CPU
- Bonito64 system controller as North Bridge
- VT82C686 chipset as South Bridge
- RTL8139D as a network card chipset

The Loongson-3 virtual platform emulation supports:

- Loongson 3A CPU
- LIOINTC as interrupt controller
- GPEX and virtio as peripheral devices
- Both KVM and TCG supported

The mipssim pseudo board emulation provides an environment similar to what the proprietary MIPS emulator uses for running Linux. It supports:

- A range of MIPS CPUs, default is the 24Kf
- PC style serial port
- MIPSnet network emulation

Supported CPU model configurations on MIPS hosts

QEMU supports variety of MIPS CPU models:

Supported CPU models for MIPS32 hosts

The following CPU models are supported for use on MIPS32 hosts. Administrators / applications are recommended to use the CPU model that matches the generation of the host CPUs in use. In a deployment with a mixture of host CPU models between machines, if live migration compatibility is required, use the newest CPU model that is compatible across all desired hosts.

mips32r6-generic
: MIPS32 Processor (Release 6, 2015)

P5600
: MIPS32 Processor (P5600, 2014)

M14K, M14Kc
: MIPS32 Processor (M14K, 2009)

74Kf
: MIPS32 Processor (74K, 2007)

34Kf
: MIPS32 Processor (34K, 2006)

24Kc, 24KEc, 24Kf
: MIPS32 Processor (24K, 2003)

4Kc, 4Km, 4KEcR1, 4KEmR1, 4KEc, 4KEm
MIPS32 Processor (4K, 1999)

Supported CPU models for MIPS64 hosts

The following CPU models are supported for use on MIPS64 hosts. Administrators / applications are recommended to use the CPU model that matches the generation of the host CPUs in use. In a deployment with a mixture of host CPU models between machines, if live migration compatibility is required, use the newest CPU model that is compatible across all desired hosts.

I6400
MIPS64 Processor (Release 6, 2014)

Loongson-2E
MIPS64 Processor (Loongson 2, 2006)

Loongson-2F
MIPS64 Processor (Loongson 2, 2008)

Loongson-3A1000
MIPS64 Processor (Loongson 3, 2010)

Loongson-3A4000
MIPS64 Processor (Loongson 3, 2018)

mips64dspr2
MIPS64 Processor (Release 2, 2006)

MIPS64R2-generic, 5KEc, 5KEf
MIPS64 Processor (Release 2, 2002)

20Kc
MIPS64 Processor (20K, 2000

5Kc, 5Kf
MIPS64 Processor (5K, 1999)

VR5432
MIPS64 Processor (VR, 1998)

R4000
MIPS64 Processor (MIPS III, 1991)

Supported CPU models for nanoMIPS hosts

The following CPU models are supported for use on nanoMIPS hosts. Administrators / applications are recommended to use the CPU model that matches the generation of the host CPUs in use. In a deployment with a mixture of host CPU models between machines, if live migration compatibility is required, use the newest CPU model that is compatible across all desired hosts.

I7200
MIPS I7200 (nanoMIPS, 2018)

Preferred CPU models for MIPS hosts

The following CPU models are preferred for use on different MIPS hosts:

MIPS III
R4000

MIPS32R2
34Kf

MIPS64R6
I6400

nanoMIPS
I7200

nanoMIPS System emulator

Executable `qemu-system-mipsel` also covers simulation of 32-bit nanoMIPS system in little endian mode:

- nanoMIPS I7200 CPU

Example of `qemu-system-mipsel` usage for nanoMIPS is shown below:

Download <disk_image_file> from https://mipsdistros.mips.com/LinuxDistro/nanomips/buildroot/index.html.

Download <kernel_image_file> from https://mipsdistros.mips.com/LinuxDistro/nanomips/kernels/v4.15.18-432-gb2eb9a8b07a1-20180627102142/index.html.

Start system emulation of Malta board with nanoMIPS I7200 CPU:

```
qemu-system-mipsel -cpu I7200 -kernel <kernel_image_file> \
    -M malta -serial stdio -m <memory_size> -hda <disk_image_file> \
    -append "mem=256m@0x0 rw console=ttyS0 vga=cirrus vesa=0x111 root=/dev/sda"
```

2.23.5 PowerPC System emulator

Board-specific documentation

You can get a complete list by running `qemu-system-ppc64 --machine help`.

AmigaNG boards (`amigaone`, `pegasos2`, `sam460ex`)

These PowerPC machines emulate boards that are primarily used for running Amiga like OSes (AmigaOS 4, MorphOS and AROS) but these can also run Linux which is what this section documents.

Eyetech AmigaOne/Mai Logic Teron (amigaone)

The amigaone machine emulates an AmigaOne XE mainboard by Eyetech which is a rebranded Mai Logic Teron board with modified U-Boot firmware to support AmigaOS 4.

Emulated devices

- PowerPC 7457 CPU (can also use -cpu g3, 750cxe, 750fx or 750gx)
- Articia S north bridge
- VIA VT82C686B south bridge
- PCI VGA compatible card (guests may need other card instead)
- PS/2 keyboard and mouse

Firmware

A firmware binary is necessary for the boot process. It is a modified U-Boot under GPL but its source is lost so it cannot be included in QEMU. A binary is available at https://www.hyperion-entertainment.com/index.php/downloads?view=files&parent=28. The ROM image is in the last 512kB which can be extracted with the following command:

```
$ tail -c 524288 updater.image > u-boot-amigaone.bin
```

The BIOS emulator in the firmware is unable to run QEMU's standard vgabios so VGABIOS-lgpl-latest.bin is needed instead which can be downloaded from http://www.nongnu.org/vgabios.

Running Linux

There are some Linux images under the following link that work on the amigaone machine: https://sourceforge.net/projects/amigaone-linux/files/debian-installer/. To boot the system run:

```
$ qemu-system-ppc -machine amigaone -bios u-boot-amigaone.bin \
                  -cdrom "A1 Linux Net Installer.iso" \
                  -device ati-vga,model=rv100,romfile=VGABIOS-lgpl-latest.bin
```

From the firmware menu that appears select Boot sequence → Amiga Multiboot Options and set Boot device 1 to Onboard VIA IDE CDROM. Then hit escape until the main screen appears again, hit escape once more and from the exit menu that appears select either Save settings and exit or Use settings for this session only. It may take a long time loading the kernel into memory but eventually it boots and the installer becomes visible. The ati-vga RV100 emulation is not complete yet so only frame buffer works, DRM and 3D is not available.

Genesi/bPlan Pegasos II (pegasos2)

The `pegasos2` machine emulates the Pegasos II sold by Genesi and designed by bPlan. Its schematics are available at https://www.powerdeveloper.org/platforms/pegasos/schematics.

Emulated devices

- PowerPC 7457 CPU (can also use `-cpu g3` or `750cxe`)
- Marvell MV64361 Discovery II north bridge
- VIA VT8231 south bridge
- PCI VGA compatible card (guests may need other card instead)
- PS/2 keyboard and mouse

Firmware

The Pegasos II board has an Open Firmware compliant ROM based on SmartFirmware with some changes that are not open-sourced therefore the ROM binary cannot be included in QEMU. An updater was available from bPlan, it can be found in the Internet Archive. The ROM image can be extracted from it with the following command:

```
$ tail -c +85581 up050404 | head -c 524288 > pegasos2.rom
```

Running Linux

The PowerPC version of Debian 8.11 supported Pegasos II. The BIOS emulator in the firmware binary is unable to run QEMU's standard vgabios so it needs to be disabled. To boot the system run:

```
$ qemu-system-ppc -machine pegasos2 -bios pegasos2.rom \
                  -cdrom debian-8.11.0-powerpc-netinst.iso \
                  -device VGA,romfile="" -serial stdio
```

At the firmware `ok` prompt enter `boot cd install/pegasos`.

Alternatively, it is possible to boot the kernel directly without firmware ROM using the QEMU built-in minimal Virtual Open Firmware (VOF) emulation which is also supported on `pegasos2`. For this, extract the kernel `install/powerpc/vmlinuz-chrp.initrd` from the CD image, then run:

```
$ qemu-system-ppc -machine pegasos2 -serial stdio \
                  -kernel vmlinuz-chrp.initrd -append "---" \
                  -cdrom debian-8.11.0-powerpc-netinst.iso
```

aCube Sam460ex (`sam460ex`)

The `sam460ex` machine emulates the Sam460ex board by aCube which is based on the AMCC PowerPC 460EX SoC (that despite its name has a PPC440 CPU core).

Firmware

The board has a firmware based on an older U-Boot version with modifications to support booting AmigaOS 4. The firmware ROM is included with QEMU.

Emulated devices

- PowerPC 460EX SoC
- M41T80 serial RTC chip
- Silicon Motion SM501 display parts (identical to SM502 on real board)
- Silicon Image SiI3112 2 port SATA controller
- USB keyboard and mouse

Running Linux

The only Linux distro that supported Sam460ex out of box was CruxPPC 2.x. It can be booted by running:

```
$ qemu-system-ppc -machine sam460ex -serial stdio \
                  -drive if=none,id=cd,format=raw,file=crux-ppc-2.7a.iso \
                  -device ide-cd,drive=cd,bus=ide.1
```

There are some other kernels and instructions for booting other distros on aCube's product page at https://www.acube-systems.biz/index.php?page=hardware&pid=5 but those are untested.

Embedded family boards

- `bamboo` bamboo
- `mpc8544ds` mpc8544ds
- `ppce500` generic paravirt e500 platform
- `ref405ep` ref405ep
- `sam460ex` aCube Sam460ex
- `virtex-ml507` Xilinx Virtex ML507 reference design

PowerMac family boards (`g3beige`, `mac99`)

Use the executable `qemu-system-ppc` to simulate a complete PowerMac PowerPC system.

- `g3beige` Heathrow based PowerMAC
- `mac99` Mac99 based PowerMAC

Supported devices

QEMU emulates the following PowerMac peripherals:

- UniNorth or Grackle PCI Bridge
- PCI VGA compatible card with VESA Bochs Extensions
- 2 PMAC IDE interfaces with hard disk and CD-ROM support
- NE2000 PCI adapters
- Non Volatile RAM
- VIA-CUDA with ADB keyboard and mouse.

Missing devices

- To be identified

Firmware

Since version 0.9.1, QEMU uses OpenBIOS https://www.openbios.org/ for the g3beige and mac99 PowerMac and the 40p machines. OpenBIOS is a free (GPL v2) portable firmware implementation. The goal is to implement a 100% IEEE 1275-1994 (referred to as Open Firmware) compliant firmware.

PowerNV family boards (`powernv8`, `powernv9`, `powernv10`)

PowerNV (as Non-Virtualized) is the "bare metal" platform using the OPAL firmware. It runs Linux on IBM and OpenPOWER systems and it can be used as an hypervisor OS, running KVM guests, or simply as a host OS.

The PowerNV QEMU machine tries to emulate a PowerNV system at the level of the skiboot firmware, which loads the OS and provides some runtime services. Power Systems have a lower firmware (HostBoot) that does low level system initialization, like DRAM training. This is beyond the scope of what QEMU addresses today.

Supported devices

- Multi processor support for POWER8, POWER8NVL and POWER9.
- XSCOM, serial communication sideband bus to configure chiplets.
- Simple LPC Controller.
- Processor Service Interface (PSI) Controller.
- Interrupt Controller, XICS (POWER8) and XIVE (POWER9) and XIVE2 (Power10).
- POWER8 PHB3 PCIe Host bridge and POWER9 PHB4 PCIe Host bridge.

- Simple OCC is an on-chip micro-controller used for power management tasks.
- iBT device to handle BMC communication, with the internal BMC simulator provided by QEMU or an external BMC such as an Aspeed QEMU machine.
- PNOR containing the different firmware partitions.

Missing devices

A lot is missing, among which :

- I2C controllers (yet to be merged).
- NPU/NPU2/NPU3 controllers.
- EEH support for PCIe Host bridge controllers.
- NX controller.
- VAS controller.
- chipTOD (Time Of Day).
- Self Boot Engine (SBE).
- FSI bus.

Firmware

The OPAL firmware (OpenPower Abstraction Layer) for OpenPower systems includes the runtime services `skiboot` and the bootloader kernel and initramfs `skiroot`. Source code can be found on the OpenPOWER account at GitHub.

Prebuilt images of `skiboot` and `skiroot` are made available on the OpenPOWER site.

QEMU includes a prebuilt image of `skiboot` which is updated when a more recent version is required by the models.

Current acceleration status

KVM acceleration in Linux Power hosts is provided by the kvm-hv and kvm-pr modules. kvm-hv is adherent to PAPR and it's not compliant with powernv. kvm-pr in theory could be used as a valid accel option but this isn't supported by kvm-pr at this moment.

To spare users from dealing with not so informative errors when attempting to use accel=kvm, the powernv machine will throw an error informing that KVM is not supported. This can be revisited in the future if kvm-pr (or any other KVM alternative) is usable as KVM accel for this machine.

Boot options

Here is a simple setup with one e1000e NIC :

```
$ qemu-system-ppc64 -m 2G -machine powernv9 -smp 2,cores=2,threads=1 \
-accel tcg,thread=single \
-device e1000e,netdev=net0,mac=C0:FF:EE:00:00:02,bus=pcie.0,addr=0x0 \
-netdev user,id=net0,hostfwd=::20022-:22,hostname=pnv \
-kernel ./zImage.epapr  \
```

(continues on next page)

(continued from previous page)

```
-initrd ./rootfs.cpio.xz \
-nographic
```

and a SATA disk :

```
-device ich9-ahci,id=sata0,bus=pcie.1,addr=0x0 \
-drive file=./ubuntu-ppc64le.qcow2,if=none,id=drive0,format=qcow2,cache=none \
-device ide-hd,bus=sata0.0,unit=0,drive=drive0,id=ide,bootindex=1 \
```

Complex PCIe configuration

Six PHBs are defined per chip (POWER9) but no default PCI layout is provided (to be compatible with libvirt). One PCI device can be added on any of the available PCIe slots using command line options such as:

```
-device e1000e,netdev=net0,mac=C0:FF:EE:00:00:02,bus=pcie.0,addr=0x0
-netdev bridge,id=net0,helper=/usr/libexec/qemu-bridge-helper,br=virbr0,id=hostnet0

-device megasas,id=scsi0,bus=pcie.0,addr=0x0
-drive file=./ubuntu-ppc64le.qcow2,if=none,id=drive-scsi0-0-0-0,format=qcow2,cache=none
-device scsi-hd,bus=scsi0.0,channel=0,scsi-id=0,lun=0,drive=drive-scsi0-0-0-0,id=scsi0-0-
↪0-0,bootindex=2
```

Here is a full example with two different storage controllers on different PHBs, each with a disk, the second PHB is empty :

```
$ qemu-system-ppc64 -m 2G -machine powernv9 -smp 2,cores=2,threads=1 -accel tcg,
↪thread=single \
-kernel ./zImage.epapr -initrd ./rootfs.cpio.xz -bios ./skiboot.lid \
\
-device megasas,id=scsi0,bus=pcie.0,addr=0x0 \
-drive file=./rhel7-ppc64le.qcow2,if=none,id=drive-scsi0-0-0-0,format=qcow2,cache=none \
-device scsi-hd,bus=scsi0.0,channel=0,scsi-id=0,lun=0,drive=drive-scsi0-0-0-0,id=scsi0-0-
↪0-0,bootindex=2 \
\
-device pcie-pci-bridge,id=bridge1,bus=pcie.1,addr=0x0 \
\
-device ich9-ahci,id=sata0,bus=bridge1,addr=0x1 \
-drive file=./ubuntu-ppc64le.qcow2,if=none,id=drive0,format=qcow2,cache=none \
-device ide-hd,bus=sata0.0,unit=0,drive=drive0,id=ide,bootindex=1 \
-device e1000e,netdev=net0,mac=C0:FF:EE:00:00:02,bus=bridge1,addr=0x2 \
-netdev bridge,helper=/usr/libexec/qemu-bridge-helper,br=virbr0,id=net0 \
-device nec-usb-xhci,bus=bridge1,addr=0x7 \
\
-serial mon:stdio -nographic
```

You can also use VIRTIO devices :

```
-drive file=./fedora-ppc64le.qcow2,if=none,snapshot=on,id=drive0 \
-device virtio-blk-pci,drive=drive0,id=blk0,bus=pcie.0 \
\
-netdev tap,helper=/usr/lib/qemu/qemu-bridge-helper,br=virbr0,id=netdev0 \
```

(continues on next page)

(continued from previous page)

```
-device virtio-net-pci,netdev=netdev0,id=net0,bus=pcie.1 \
\
-fsdev local,id=fsdev0,path=$HOME,security_model=passthrough \
-device virtio-9p-pci,fsdev=fsdev0,mount_tag=host,bus=pcie.2
```

Multi sockets

The number of sockets is deduced from the number of CPUs and the number of cores. `-smp 2,cores=1` will define a machine with 2 sockets of 1 core, whereas `-smp 2,cores=2` will define a machine with 1 socket of 2 cores. `-smp 8,cores=2`, 4 sockets of 2 cores.

BMC configuration

OpenPOWER systems negotiate the shutdown and reboot with their BMC. The QEMU PowerNV machine embeds an IPMI BMC simulator using the iBT interface and should offer the same power features.

If you want to define your own BMC, use `-nodefaults` and specify one on the command line :

```
-device ipmi-bmc-sim,id=bmc0 -device isa-ipmi-bt,bmc=bmc0,irq=10
```

The files palmetto-SDR.bin and palmetto-FRU.bin define a Sensor Data Record repository and a Field Replaceable Unit inventory for a Palmetto BMC. They can be used to extend the QEMU BMC simulator.

```
-device ipmi-bmc-sim,sdrfile=./palmetto-SDR.bin,fruareasize=256,frudatafile=./palmetto-
↪FRU.bin,id=bmc0 \
-device isa-ipmi-bt,bmc=bmc0,irq=10
```

The PowerNV machine can also be run with an external IPMI BMC device connected to a remote QEMU machine acting as BMC, using these options :

```
-chardev socket,id=ipmi0,host=localhost,port=9002,reconnect=10 \
-device ipmi-bmc-extern,id=bmc0,chardev=ipmi0 \
-device isa-ipmi-bt,bmc=bmc0,irq=10 \
-nodefaults
```

NVRAM

Use a MTD drive to add a PNOR to the machine, and get a NVRAM :

```
-drive file=./witherspoon.pnor,format=raw,if=mtd
```

Maintainer contact information

Cédric Le Goater <clg@kaod.org>

ppce500 generic platform (ppce500)

QEMU for PPC supports a special `ppce500` machine designed for emulation and virtualization purposes.

Supported devices

The `ppce500` machine supports the following devices:

- PowerPC e500 series core (e500v2/e500mc/e5500/e6500)
- Configuration, Control, and Status Register (CCSR)
- Multicore Programmable Interrupt Controller (MPIC) with MSI support
- 1 16550A UART device
- 1 Freescale MPC8xxx I2C controller
- 1 Pericom pt7c4338 RTC via I2C
- 1 Freescale MPC8xxx GPIO controller
- Power-off functionality via one GPIO pin
- 1 Freescale MPC8xxx PCI host controller
- VirtIO devices via PCI bus
- 1 Freescale Enhanced Secure Digital Host controller (eSDHC)
- 1 Freescale Enhanced Triple Speed Ethernet controller (eTSEC)

Hardware configuration information

The `ppce500` machine automatically generates a device tree blob ("dtb") which it passes to the guest, if there is no `-dtb` option. This provides information about the addresses, interrupt lines and other configuration of the various devices in the system.

If users want to provide their own DTB, they can use the `-dtb` option. These DTBs should have the following requirements:

- The number of subnodes under /cpus node should match QEMU's `-smp` option
- The /memory reg size should match QEMU's selected ram_size via `-m`

Both `qemu-system-ppc` and `qemu-system-ppc64` provide emulation for the following 32-bit PowerPC CPUs:

- e500v2
- e500mc

Additionally `qemu-system-ppc64` provides support for the following 64-bit PowerPC CPUs:

- e5500
- e6500

The CPU type can be specified via the `-cpu` command line. If not specified, it creates a machine with e500v2 core. The following example shows an e6500 based machine creation:

```
$ qemu-system-ppc64 -nographic -M ppce500 -cpu e6500
```

Boot options

The `ppce500` machine can start using the standard -kernel functionality for loading a payload like an OS kernel (e.g.: Linux), or U-Boot firmware.

When -bios is omitted, the default pc-bios/u-boot.e500 firmware image is used as the BIOS. QEMU follows below truth table to select which payload to execute:

-bios	-kernel	payload
N	N	u-boot
N	Y	kernel
Y	don't care	u-boot

When both -bios and -kernel are present, QEMU loads U-Boot and U-Boot in turns automatically loads the kernel image specified by the -kernel parameter via U-Boot's built-in "bootm" command, hence a legacy uImage format is required in such scenario.

Running Linux kernel

Linux mainline v5.11 release is tested at the time of writing. To build a Linux mainline kernel that can be booted by the `ppce500` machine in 64-bit mode, simply configure the kernel using the defconfig configuration:

```
$ export ARCH=powerpc
$ export CROSS_COMPILE=powerpc-linux-
$ make corenet64_smp_defconfig
$ make menuconfig
```

then manually select the following configuration:

> Platform support > Freescale Book-E Machine Type > QEMU generic e500 platform

To boot the newly built Linux kernel in QEMU with the `ppce500` machine:

```
$ qemu-system-ppc64 -M ppce500 -cpu e5500 -smp 4 -m 2G \
    -display none -serial stdio \
    -kernel vmlinux \
    -initrd /path/to/rootfs.cpio \
    -append "root=/dev/ram"
```

To build a Linux mainline kernel that can be booted by the `ppce500` machine in 32-bit mode, use the same 64-bit configuration steps except the defconfig file should use corenet32_smp_defconfig.

To boot the 32-bit Linux kernel:

```
$ qemu-system-ppc64 -M ppce500 -cpu e500mc -smp 4 -m 2G \
    -display none -serial stdio \
    -kernel vmlinux \
```

(continues on next page)

(continued from previous page)

```
    -initrd /path/to/rootfs.cpio \
    -append "root=/dev/ram"
```

Running U-Boot

U-Boot mainline v2021.07 release is tested at the time of writing. To build a U-Boot mainline bootloader that can be booted by the `ppce500` machine, use the qemu-ppce500_defconfig with similar commands as described above for Linux:

```
$ export CROSS_COMPILE=powerpc-linux-
$ make qemu-ppce500_defconfig
```

You will get u-boot file in the build tree.

When U-Boot boots, you will notice the following if using with `-cpu e6500`:

```
CPU:   Unknown, Version: 0.0, (0x00000000)
Core:  e6500, Version: 2.0, (0x80400020)
```

This is because we only specified a core name to QEMU and it does not have a meaningful SVR value which represents an actual SoC that integrates such core. You can specify a real world SoC device that QEMU has built-in support but all these SoCs are e500v2 based MPC85xx series, hence you cannot test anything built for P4080 (e500mc), P5020 (e5500) and T2080 (e6500).

Networking

By default a VirtIO standard PCI networking device is connected as an ethernet interface at PCI address 0.1.0, but we can switch that to an e1000 NIC by:

```
$ qemu-system-ppc64 -M ppce500 -smp 4 -m 2G \
                    -display none -serial stdio \
                    -bios u-boot \
                    -nic tap,ifname=tap0,script=no,downscript=no,model=e1000
```

The QEMU `ppce500` machine can also dynamically instantiate an eTSEC device if "-device eTSEC" is given to QEMU:

```
-netdev tap,ifname=tap0,script=no,downscript=no,id=net0 -device eTSEC,netdev=net0
```

Root file system on flash drive

Rather than using a root file system on ram disk, it is possible to have it on CFI flash. Given an ext2 image whose size must be a power of two, it can be used as follows:

```
$ qemu-system-ppc64 -M ppce500 -cpu e500mc -smp 4 -m 2G \
    -display none -serial stdio \
    -kernel vmlinux \
    -drive if=pflash,file=/path/to/rootfs.ext2,format=raw \
    -append "rootwait root=/dev/mtdblock0"
```

Alternatively, the root file system can also reside on an emulated SD card whose size must again be a power of two:

```
$ qemu-system-ppc64 -M ppce500 -cpu e500mc -smp 4 -m 2G \
    -display none -serial stdio \
    -kernel vmlinux \
    -device sd-card,drive=mydrive \
    -drive id=mydrive,if=none,file=/path/to/rootfs.ext2,format=raw \
    -append "rootwait root=/dev/mmcblk0"
```

Prep machine (40p)

Use the executable `qemu-system-ppc` to simulate a complete 40P (PREP)

Supported devices

QEMU emulates the following 40P (PREP) peripherals:

- PCI Bridge
- PCI VGA compatible card with VESA Bochs Extensions
- 2 IDE interfaces with hard disk and CD-ROM support
- Floppy disk
- PCnet network adapters
- Serial port
- PREP Non Volatile RAM
- PC compatible keyboard and mouse.

pSeries family boards (pseries)

The Power machine para-virtualized environment described by the Linux on Power Architecture Reference ([LoPAR]) document is called pSeries. This environment is also known as sPAPR, System p guests, or simply Power Linux guests (although it is capable of running other operating systems, such as AIX).

Even though pSeries is designed to behave as a guest environment, it is also capable of acting as a hypervisor OS, providing, on that role, nested virtualization capabilities.

Supported devices

- Multi processor support for many Power processors generations: POWER7, POWER7+, POWER8, POWER8NVL, POWER9, and Power10. Support for POWER5+ exists, but its state is unknown.
- Interrupt Controller, XICS (POWER8) and XIVE (POWER9 and Power10)
- vPHB PCIe Host bridge.
- vscsi and vnet devices, compatible with the same devices available on a PowerVM hypervisor with VIOS managing LPARs.
- Virtio based devices.
- PCIe device pass through.

Missing devices

- SPICE support.

Firmware

The pSeries platform in QEMU comes with 2 firmwares:

SLOF (Slimline Open Firmware) is an implementation of the IEEE 1275-1994, Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices.

SLOF performs bus scanning, PCI resource allocation, provides the client interface to boot from block devices and network.

QEMU includes a prebuilt image of SLOF which is updated when a more recent version is required.

VOF (Virtual Open Firmware) is a minimalistic firmware to work with `-machine pseries,x-vof=on`. When enabled, the firmware acts as a slim shim and QEMU implements parts of the IEEE 1275 Open Firmware interface.

VOF does not have device drivers, does not do PCI resource allocation and relies on `-kernel` used with Linux kernels recent enough (v5.4+) to PCI resource assignment. It is ideal to use with petitboot.

Booting via `-kernel` supports the following:

kernel	pseries,x-vof=off	pseries,x-vof=on
vmlinux BE	✓	✓
vmlinux LE	✓	✓
zImage.pseries BE	✓[1]	✓[1]
zImage.pseries LE	✓	✓

[1] must set kernel-addr=0

Build directions

```
./configure --target-list=ppc64-softmmu && make
```

Running instructions

Someone can select the pSeries machine type by running QEMU with the following options:

```
qemu-system-ppc64 -M pseries <other QEMU arguments>
```

sPAPR devices

The sPAPR specification defines a set of para-virtualized devices, which are also supported by the pSeries machine in QEMU and can be instantiated with the `-device` option:

- `spapr-vlan` : a virtual network interface.
- `spapr-vscsi` : a virtual SCSI disk interface.
- `spapr-rng` : a pseudo-device for passing random number generator data to the guest (see the H_RANDOM hypercall feature for details).
- `spapr-vty`: a virtual teletype.
- `spapr-pci-host-bridge`: a PCI host bridge.
- `tpm-spapr`: a Trusted Platform Module (TPM).
- `spapr-tpm-proxy`: a TPM proxy.

These are compatible with the devices historically available for use when running the IBM PowerVM hypervisor with LPARs.

However, since these devices have originally been specified with another hypervisor and non-Linux guests in mind, you should use the virtio counterparts (virtio-net, virtio-blk/scsi and virtio-rng for instance) if possible instead, since they will most probably give you better performance with Linux guests in a QEMU environment.

The pSeries machine in QEMU is always instantiated with the following devices:

- A NVRAM device (`spapr-nvram`).
- A virtual teletype (`spapr-vty`).
- A PCI host bridge (`spapr-pci-host-bridge`).

Hence, it is not needed to add them manually, unless you use the `-nodefaults` command line option in QEMU.

In the case of the default `spapr-nvram` device, if someone wants to make the contents of the NVRAM device persistent, they will need to specify a PFLASH device when starting QEMU, i.e. either use `-drive if=pflash,file=<filename>,format=raw` to set the default PFLASH device, or specify one with an ID (`-drive if=none,file=<filename>,format=raw,id=pfid`) and pass that ID to the NVRAM device with `-global spapr-nvram.drive=pfid`.

sPAPR specification

The main source of documentation on the sPAPR standard is the [LoPAR] document. However, documentation specific to QEMU's implementation of the specification can also be found in QEMU documentation:

sPAPR Dynamic Reconfiguration

sPAPR or pSeries guests make use of a facility called dynamic reconfiguration to handle hot plugging of dynamic "physical" resources like PCI cards, or "logical"/para-virtual resources like memory, CPUs, and "physical" host-bridges, which are generally managed by the host/hypervisor and provided to guests as virtualized resources. The specifics of dynamic reconfiguration are documented extensively in section 13 of the Linux on Power Architecture Reference document ([LoPAR]). This document provides a summary of that information as it applies to the implementation within QEMU.

Dynamic-reconfiguration Connectors

To manage hot plug/unplug of these resources, a firmware abstraction known as a Dynamic Resource Connector (DRC) is used to assign a particular dynamic resource to the guest, and provide an interface for the guest to manage configuration/removal of the resource associated with it.

Device tree description of DRCs

A set of four Open Firmware device tree array properties are used to describe the name/index/power-domain/type of each DRC allocated to a guest at boot time. There may be multiple sets of these arrays, rooted at different paths in the device tree depending on the type of resource the DRCs manage.

In some cases, the DRCs themselves may be provided by a dynamic resource, such as the DRCs managing PCI slots on a hot plugged PHB. In this case the arrays would be fetched as part of the device tree retrieval interfaces for hot plugged resources described under *Guest->Host interface to manage dynamic resources*.

The array properties are described below. Each entry/element in an array describes the DRC identified by the element in the corresponding position of `ibm,drc-indexes`:

`ibm,drc-names`

First 4-bytes: big-endian (BE) encoded integer denoting the number of entries.

Each entry: a NULL-terminated <name> string encoded as a byte array.

<name> values for logical/virtual resources are defined in the Linux on Power Architecture Reference ([LoPAR]) section 13.5.2.4, and basically consist of the type of the resource followed by a space and a numerical value that's unique across resources of that type.

<name> values for "physical" resources such as PCI or VIO devices are defined as being "location codes", which are the "location labels" of each encapsulating device, starting from the chassis down to the individual slot for the device, concatenated by a hyphen. This provides a mapping of resources to a physical location in a chassis for debugging purposes. For QEMU, this mapping is less important, so we assign a location code that conforms to naming specifications, but is simply a location label for the slot by itself to simplify the implementation. The naming convention for location labels is documented in detail in the [LoPAR] section 12.3.1.5, and in our case amounts to using C<n> for PCI/VIO device slots, where <n> is unique across all PCI/VIO device slots.

`ibm,drc-indexes`

First 4-bytes: BE-encoded integer denoting the number of entries.

Each 4-byte entry: BE-encoded <index> integer that is unique across all DRCs in the machine.

<index> is arbitrary, but in the case of QEMU we try to maintain the convention used to assign them to pSeries guests on pHyp (the hypervisor portion of PowerVM):

`bit[31:28]`: integer encoding of <type>, where <type> is:

1 for CPU resource.

2 for PHB resource.

3 for VIO resource.

4 for PCI resource.

8 for memory resource.

bit[27:0]: integer encoding of <id>, where <id> is unique across all resources of specified type.

ibm,drc-power-domains

First 4-bytes: BE-encoded integer denoting the number of entries.

Each 4-byte entry: 32-bit, BE-encoded <index> integer that specifies the power domain the resource will be assigned to. In the case of QEMU we associated all resources with a "live insertion" domain, where the power is assumed to be managed automatically. The integer value for this domain is a special value of -1.

ibm,drc-types

First 4-bytes: BE-encoded integer denoting the number of entries.

Each entry: a NULL-terminated <type> string encoded as a byte array. <type> is assigned as follows:

"CPU" for a CPU.

"PHB" for a physical host-bridge.

"SLOT" for a VIO slot.

"28" for a PCI slot.

"MEM" for memory resource.

Guest->Host interface to manage dynamic resources

Each DRC is given a globally unique DRC index, and resources associated with a particular DRC are configured/managed by the guest via a number of RTAS calls which reference individual DRCs based on the DRC index. This can be considered the guest->host interface.

rtas-set-power-level

Set the power level for a specified power domain.

arg[0]: integer identifying power domain.

arg[1]: new power level for the domain, 0-100.

output[0]: status, 0 on success.

output[1]: power level after command.

rtas-get-power-level

Get the power level for a specified power domain.

`arg[0]`: integer identifying power domain.

`output[0]`: status, `0` on success.

`output[1]`: current power level.

rtas-set-indicator

Set the state of an indicator or sensor.

`arg[0]`: integer identifying sensor/indicator type.

`arg[1]`: index of sensor, for DR-related sensors this is generally the DRC index.

`arg[2]`: desired sensor value.

`output[0]`: status, `0` on success.

For the purpose of this document we focus on the indicator/sensor types associated with a DRC. The types are:

- `9001`: `isolation-state`, controls/indicates whether a device has been made accessible to a guest. Supported sensor values:

  `0`: `isolate`, device is made inaccessible by guest OS.

  `1`: `unisolate`, device is made available to guest OS.

- `9002`: `dr-indicator`, controls "visual" indicator associated with device. Supported sensor values:

  `0`: `inactive`, resource may be safely removed.

  `1`: `active`, resource is in use and cannot be safely removed.

  `2`: `identify`, used to visually identify slot for interactive hot plug.

  `3`: `action`, in most cases, used in the same manner as identify.

- `9003`: `allocation-state`, generally only used for "logical" DR resources to request the allocation/deallocation of a resource prior to acquiring it via `isolation-state->unisolate`, or after releasing it via `isolation-state->isolate`, respectively. For "physical" DR (like PCI hot plug/unplug) the pre-allocation of the resource is implied and this sensor is unused. Supported sensor values:

  `0`: `unusable`, tell firmware/system the resource can be unallocated/reclaimed and added back to the system resource pool.

  `1`: `usable`, request the resource be allocated/reserved for use by guest OS.

  `2`: `exchange`, used to allocate a spare resource to use for fail-over in certain situations. Unused in QEMU.

  `3`: `recover`, used to reclaim a previously allocated resource that's not currently allocated to the guest OS. Unused in QEMU.

rtas-get-sensor-state:

Used to read an indicator or sensor value.

`arg[0]`: integer identifying sensor/indicator type.

`arg[1]`: index of sensor, for DR-related sensors this is generally the DRC index

`output[0]`: status, 0 on success

For DR-related operations, the only noteworthy sensor is `dr-entity-sense`, which has a type value of `9003`, as `allocation-state` does in the case of `rtas-set-indicator`. The semantics/encodings of the sensor values are distinct however.

Supported sensor values for `dr-entity-sense` (`9003`) sensor:

`0`: empty.

For physical resources: DRC/slot is empty.

For logical resources: unused.

`1`: present.

For physical resources: DRC/slot is populated with a device/resource.

For logical resources: resource has been allocated to the DRC.

`2`: unusable.

For physical resources: unused.

For logical resources: DRC has no resource allocated to it.

`3`: exchange.

For physical resources: unused.

For logical resources: resource available for exchange (see `allocation-state` sensor semantics above).

`4`: recovery.

For physical resources: unused.

For logical resources: resource available for recovery (see `allocation-state` sensor semantics above).

rtas-ibm-configure-connector

Used to fetch an OpenFirmware device tree description of the resource associated with a particular DRC.

`arg[0]`: guest physical address of 4096-byte work area buffer.

`arg[1]`: 0, or address of additional 4096-byte work area buffer; only non-zero if a prior RTAS response indicated a need for additional memory.

`output[0]`: status:

`0`: completed transmittal of device tree node.

`1`: instruct guest to prepare for next device tree sibling node.

`2`: instruct guest to prepare for next device tree child node.

`3`: instruct guest to prepare for next device tree property.

`4`: instruct guest to ascend to parent device tree node.

`5`: instruct guest to provide additional work-area buffer via `arg[1]`.

`990x`: instruct guest that operation took too long and to try again later.

The DRC index is encoded in the first 4-bytes of the first work area buffer. Work area (`wa`) layout, using 4-byte offsets:

`wa[0]`: DRC index of the DRC to fetch device tree nodes from.

`wa[1]`: `0` (hard-coded).

`wa[2]`:

For next-sibling/next-child response:

`wa` offset of null-terminated string denoting the new node's name.

For next-property response:

`wa` offset of null-terminated string denoting new property's name.

`wa[3]`: for next-property response (unused otherwise):

Byte-length of new property's value.

`wa[4]`: for next-property response (unused otherwise):

New property's value, encoded as an OFDT-compatible byte array.

Hot plug/unplug events

For most DR operations, the hypervisor will issue host->guest add/remove events using the EPOW/check-exception notification framework, where the host issues a check-exception interrupt, then provides an RTAS event log via an rtas-check-exception call issued by the guest in response. This framework is documented by PAPR+ v2.7, and already use in by QEMU for generating powerdown requests via EPOW events.

For DR, this framework has been extended to include hotplug events, which were previously unneeded due to direct manipulation of DR-related guest userspace tools by host-level management such as an HMC. This level of management is not applicable to KVM on Power, hence the reason for extending the notification framework to support hotplug events.

The format for these EPOW-signalled events is described below under *Hot plug/unplug event structure*. Note that these events are not formally part of the PAPR+ specification, and have been superseded by a newer format, also described below under *Hot plug/unplug event structure*, and so are now deemed a "legacy" format. The formats are similar, but the "modern" format contains additional fields/flags, which are denoted for the purposes of this documentation with `#ifdef GUEST_SUPPORTS_MODERN` guards.

QEMU should assume support only for "legacy" fields/flags unless the guest advertises support for the "modern" format via `ibm,client-architecture-support` hcall by setting byte 5, bit 6 of it's `ibm,architecture-vec-5` option vector structure (as described by [LoPAR], section B.5.2.3). As with "legacy" format events, "modern" format events are surfaced to the guest via check-exception RTAS calls, but use a dedicated event source to signal the guest. This event source is advertised to the guest by the addition of a `hot-plug-events` node under `/event-sources` node of the guest's device tree using the standard format described in [LoPAR], section B.5.12.2.

Hot plug/unplug event structure

The hot plug specific payload in QEMU is implemented as follows (with all values encoded in big-endian format):

```
struct rtas_event_log_v6_hp {
#define SECTION_ID_HOTPLUG              0x4850 /* HP */
    struct section_header {
        uint16_t section_id;            /* set to SECTION_ID_HOTPLUG */
        uint16_t section_length;        /* sizeof(rtas_event_log_v6_hp),
                                         * plus the length of the DRC name
                                         * if a DRC name identifier is
                                         * specified for hotplug_identifier
                                         */
        uint8_t section_version;        /* version 1 */
        uint8_t section_subtype;        /* unused */
        uint16_t creator_component_id;  /* unused */
    } hdr;
#define RTAS_LOG_V6_HP_TYPE_CPU         1
#define RTAS_LOG_V6_HP_TYPE_MEMORY      2
#define RTAS_LOG_V6_HP_TYPE_SLOT        3
#define RTAS_LOG_V6_HP_TYPE_PHB         4
#define RTAS_LOG_V6_HP_TYPE_PCI         5
    uint8_t hotplug_type;               /* type of resource/device */
#define RTAS_LOG_V6_HP_ACTION_ADD       1
#define RTAS_LOG_V6_HP_ACTION_REMOVE    2
    uint8_t hotplug_action;             /* action (add/remove) */
#define RTAS_LOG_V6_HP_ID_DRC_NAME          1
#define RTAS_LOG_V6_HP_ID_DRC_INDEX         2
#define RTAS_LOG_V6_HP_ID_DRC_COUNT         3
#ifdef GUEST_SUPPORTS_MODERN
#define RTAS_LOG_V6_HP_ID_DRC_COUNT_INDEXED 4
#endif
    uint8_t hotplug_identifier;         /* type of the resource identifier,
                                         * which serves as the discriminator
                                         * for the 'drc' union field below
                                         */
#ifdef GUEST_SUPPORTS_MODERN
    uint8_t capabilities;               /* capability flags, currently unused
                                         * by QEMU
                                         */
#else
    uint8_t reserved;
#endif
    union {
        uint32_t index;                 /* DRC index of resource to take action
                                         * on
                                         */
        uint32_t count;                 /* number of DR resources to take
                                         * action on (guest chooses which)
                                         */
#ifdef GUEST_SUPPORTS_MODERN
        struct {
            uint32_t count;             /* number of DR resources to take
```

(continues on next page)

(continued from previous page)

```
                                    * action on
                                    */
            uint32_t index;        /* DRC index of first resource to take
                                    * action on. guest will take action
                                    * on DRC index <index> through
                                    * DRC index <index + count - 1> in
                                    * sequential order
                                    */
        } count_indexed;
#endif
        char name[1];              /* string representing the name of the
                                    * DRC to take action on
                                    */
    } drc;
} QEMU_PACKED;
```

ibm,lrdr-capacity

`ibm,lrdr-capacity` is a property in the /rtas device tree node that identifies the dynamic reconfiguration capabilities of the guest. It consists of a triple consisting of <phys>, <size> and <maxcpus>.

<phys>, encoded in BE format represents the maximum address in bytes and hence the maximum memory that can be allocated to the guest.

<size>, encoded in BE format represents the size increments in which memory can be hot-plugged to the guest.

<maxcpus>, a BE-encoded integer, represents the maximum number of processors that the guest can have.

`pseries` guests use this property to note the maximum allowed CPUs for the guest.

ibm,dynamic-reconfiguration-memory

`ibm,dynamic-reconfiguration-memory` is a device tree node that represents dynamically reconfigurable logical memory blocks (LMB). This node is generated only when the guest advertises the support for it via `ibm,client-architecture-support` call. Memory that is not dynamically reconfigurable is represented by `/memory` nodes. The properties of this node that are of interest to the sPAPR memory hotplug implementation in QEMU are described here.

ibm,lmb-size

This 64-bit integer defines the size of each dynamically reconfigurable LMB.

ibm,associativity-lookup-arrays

This property defines a lookup array in which the NUMA associativity information for each LMB can be found. It is a property encoded array that begins with an integer M, the number of associativity lists followed by an integer N, the number of entries per associativity list and terminated by M associativity lists each of length N integers.

This property provides the same information as given by `ibm,associativity` property in a `/memory` node. Each assigned LMB has an index value between 0 and M-1 which is used as an index into this table to select which associativity list to use for the LMB. This index value for each LMB is defined in `ibm,dynamic-memory` property.

ibm,dynamic-memory

This property describes the dynamically reconfigurable memory. It is a property encoded array that has an integer N, the number of LMBs followed by N LMB list entries.

Each LMB list entry consists of the following elements:

- Logical address of the start of the LMB encoded as a 64-bit integer. This corresponds to `reg` property in `/memory` node.
- DRC index of the LMB that corresponds to `ibm,my-drc-index` property in a `/memory` node.
- Four bytes reserved for expansion.
- Associativity list index for the LMB that is used as an index into `ibm,associativity-lookup-arrays` property described earlier. This is used to retrieve the right associativity list to be used for this LMB.
- A 32-bit flags word. The bit at bit position `0x00000008` defines whether the LMB is assigned to the partition as of boot time.

ibm,dynamic-memory-v2

This property describes the dynamically reconfigurable memory. This is an alternate and newer way to describe dynamically reconfigurable memory. It is a property encoded array that has an integer N (the number of LMB set entries) followed by N LMB set entries. There is an LMB set entry for each sequential group of LMBs that share common attributes.

Each LMB set entry consists of the following elements:

- Number of sequential LMBs in the entry represented by a 32-bit integer.
- Logical address of the first LMB in the set encoded as a 64-bit integer.
- DRC index of the first LMB in the set.
- Associativity list index that is used as an index into `ibm,associativity-lookup-arrays` property described earlier. This is used to retrieve the right associativity list to be used for all the LMBs in this set.
- A 32-bit flags word that applies to all the LMBs in the set.

sPAPR hypervisor calls

When used with the `pseries` machine type, `qemu-system-ppc64` implements a set of hypervisor calls (a.k.a. hcalls) defined in the Linux on Power Architecture Reference ([LoPAR]) document. This document is a subset of the Power Architecture Platform Reference (PAPR+) specification (IBM internal only), which is what PowerVM, the IBM proprietary hypervisor, adheres to.

The subset in LoPAR is selected based on the requirements of Linux as a guest.

In addition to those calls, we have added our own private hypervisor calls which are mostly used as a private interface between the firmware running in the guest and QEMU.

All those hypercalls start at hcall number 0xf000 which correspond to an implementation specific range in PAPR.

H_RTAS (0xf000)

RTAS stands for Run-Time Abstraction Sercies and is a set of runtime services generally provided by the firmware inside the guest to the operating system. It predates the existence of hypervisors (it was originally an extension to Open Firmware) and is still used by PAPR and LoPAR to provide various services that are not performance sensitive.

We currently implement the RTAS services in QEMU itself. The actual RTAS "firmware" blob in the guest is a small stub of a few instructions which calls our private H_RTAS hypervisor call to pass the RTAS calls to QEMU.

Arguments:

`r3`: `H_RTAS (0xf000)`

`r4`: Guest physical address of RTAS parameter block.

Returns:

`H_SUCCESS`: Successfully called the RTAS function (RTAS result will have been stored in the parameter block).

`H_PARAMETER`: Unknown token.

H_LOGICAL_MEMOP (0xf001)

When the guest runs in "real mode" (in powerpc terminology this means with MMU disabled, i.e. guest effective address equals to guest physical address), it only has access to a subset of memory and no I/Os.

PAPR and LoPAR provides a set of hypervisor calls to perform cacheable or non-cacheable accesses to any guest physical addresses that the guest can use in order to access IO devices while in real mode.

This is typically used by the firmware running in the guest.

However, doing a hypercall for each access is extremely inefficient (even more so when running KVM) when accessing the frame buffer. In that case, things like scrolling become unusably slow.

This hypercall allows the guest to request a "memory op" to be applied to memory. The supported memory ops at this point are to copy a range of memory (supports overlap of source and destination) and XOR which is used by our SLOF firmware to invert the screen.

Arguments:

`r3 ``:  ``H_LOGICAL_MEMOP (0xf001)`

`r4`: Guest physical address of destination.

`r5`: Guest physical address of source.

r6: Individual element size, defined by the binary logarithm of the desired size. Supported values are:

0 = 1 byte

1 = 2 bytes

2 = 4 bytes

3 = 8 bytes

r7: Number of elements.

r8: Operation. Supported values are:

0: copy

1: xor

Returns:

H_SUCCESS: Success.

H_PARAMETER: Invalid argument.

NUMA mechanics for sPAPR (pseries machines)

NUMA in sPAPR works different than the System Locality Distance Information Table (SLIT) in ACPI. The logic is explained in the LOPAPR 1.1 chapter 15, "Non Uniform Memory Access (NUMA) Option". This document aims to complement this specification, providing details of the elements that impacts how QEMU views NUMA in pseries.

Associativity and ibm,associativity property

Associativity is defined as a group of platform resources that has similar mean performance (or in our context here, distance) relative to everyone else outside of the group.

The format of the ibm,associativity property varies with the value of bit 0 of byte 5 of the ibm,architecture-vec-5 property. The format with bit 0 equal to zero is deprecated. The current format, with the bit 0 with the value of one, makes ibm,associativity property represent the physical hierarchy of the platform, as one or more lists that starts with the highest level grouping up to the smallest. Considering the following topology:

```
Mem M1 ---- Proc P1    |
-----------------      | Socket S1  ---|
      chip C1          |               |
                                       | HW module 1 (MOD1)
Mem M2 ---- Proc P2    |               |
-----------------      | Socket S2  ---|
      chip C2          |
```

The ibm,associativity property for the processors would be:

- P1: {MOD1, S1, C1, P1}
- P2: {MOD1, S2, C2, P2}

Each allocable resource has an ibm,associativity property. The LOPAPR specification allows multiple lists to be present in this property, considering that the same resource can have multiple connections to the platform.

Relative Performance Distance and ibm,associativity-reference-points

The ibm,associativity-reference-points property is an array that is used to define the relevant performance/distance related boundaries, defining the NUMA levels for the platform.

The definition of its elements also varies with the value of bit 0 of byte 5 of the ibm,architecture-vec-5 property. The format with bit 0 equal to zero is also deprecated. With the current format, each integer of the ibm,associativity-reference-points represents an 1 based ordinal index (i.e. the first element is 1) of the ibm,associativity array. The first boundary is the most significant to application performance, followed by less significant boundaries. Allocated resources that belongs to the same performance boundaries are expected to have relative NUMA distance that matches the relevancy of the boundary itself. Resources that belongs to the same first boundary will have the shortest distance from each other. Subsequent boundaries represents greater distances and degraded performance.

Using the previous example, the following setting reference points defines three NUMA levels:

- ibm,associativity-reference-points = {0x3, 0x2, 0x1}

The first NUMA level (0x3) is interpreted as the third element of each ibm,associativity array, the second level is the second element and the third level is the first element. Let's also consider that elements belonging to the first NUMA level have distance equal to 10 from each other, and each NUMA level doubles the distance from the previous. This means that the second would be 20 and the third level 40. For the P1 and P2 processors, we would have the following NUMA levels:

```
* ibm,associativity-reference-points = {0x3, 0x2, 0x1}

* P1: associativity{MOD1, S1, C1, P1}

First NUMA level (0x3) => associativity[2] = C1
Second NUMA level (0x2) => associativity[1] = S1
Third NUMA level (0x1) => associativity[0] = MOD1

* P2: associativity{MOD1, S2, C2, P2}

First NUMA level (0x3) => associativity[2] = C2
Second NUMA level (0x2) => associativity[1] = S2
Third NUMA level (0x1) => associativity[0] = MOD1

P1 and P2 have the same third NUMA level, MOD1: Distance between them = 40
```

Changing the ibm,associativity-reference-points array changes the performance distance attributes for the same associativity arrays, as the following example illustrates:

```
* ibm,associativity-reference-points = {0x2}

* P1: associativity{MOD1, S1, C1, P1}

First NUMA level (0x2) => associativity[1] = S1

* P2: associativity{MOD1, S2, C2, P2}

First NUMA level (0x2) => associativity[1] = S2

P1 and P2 does not have a common performance boundary. Since this is a one level
NUMA configuration, distance between them is one boundary above the first
level, 20.
```

In a hypothetical platform where all resources inside the same hardware module is considered to be on the same performance boundary:

```
* ibm,associativity-reference-points = {0x1}

* P1: associativity{MOD1, S1, C1, P1}

First NUMA level (0x1) => associativity[0] = MOD0

* P2: associativity{MOD1, S2, C2, P2}

First NUMA level (0x1) => associativity[0] = MOD0

P1 and P2 belongs to the same first order boundary. The distance between then
is 10.
```

How the pseries Linux guest calculates NUMA distances

Another key difference between ACPI SLIT and the LOPAPR regarding NUMA is how the distances are expressed. The SLIT table provides the NUMA distance value between the relevant resources. LOPAPR does not provide a standard way to calculate it. We have the ibm,associativity for each resource, which provides a common-performance hierarchy, and the ibm,associativity-reference-points array that tells which level of associativity is considered to be relevant or not.

The result is that each OS is free to implement and to interpret the distance as it sees fit. For the pseries Linux guest, each level of NUMA duplicates the distance of the previous level, and the maximum amount of levels is limited to MAX_DISTANCE_REF_POINTS = 4 (from arch/powerpc/mm/numa.c in the kernel tree). This results in the following distances:

- both resources in the first NUMA level: 10
- resources one NUMA level apart: 20
- resources two NUMA levels apart: 40
- resources three NUMA levels apart: 80
- resources four NUMA levels apart: 160

pseries NUMA mechanics

Starting in QEMU 5.2, the pseries machine considers user input when setting NUMA topology of the guest. The overall design is:

- ibm,associativity-reference-points is set to {0x4, 0x3, 0x2, 0x1}, allowing for 4 distinct NUMA distance values based on the NUMA levels
- ibm,max-associativity-domains supports multiple associativity domains in all NUMA levels, granting user flexibility
- ibm,associativity for all resources varies with user input

These changes are only effective for pseries-5.2 and newer machines that are created with more than one NUMA node (disconsidering NUMA nodes created by the machine itself, e.g. NVLink 2 GPUs). The now legacy support has been around for such a long time, with users seeing NUMA distances 10 and 40 (and 80 if using NVLink2 GPUs), and there is no need to disrupt the existing experience of those guests.

To bring the user experience x86 users have when tuning up NUMA, we had to operate under the current pseries Linux kernel logic described in *How the pseries Linux guest calculates NUMA distances*. The result is that we needed to translate NUMA distance user input to pseries Linux kernel input.

Translating user distance to kernel distance

User input for NUMA distance can vary from 10 to 254. We need to translate that to the values that the Linux kernel operates on (10, 20, 40, 80, 160). This is how it is being done:

- user distance 11 to 30 will be interpreted as 20
- user distance 31 to 60 will be interpreted as 40
- user distance 61 to 120 will be interpreted as 80
- user distance 121 and beyond will be interpreted as 160
- user distance 10 stays 10

The reasoning behind this approximation is to avoid any round up to the local distance (10), keeping it exclusive to the 4th NUMA level (which is still exclusive to the node_id). All other ranges were chosen under the developer discretion of what would be (somewhat) sensible considering the user input. Any other strategy can be used here, but in the end the reality is that we'll have to accept that a large array of values will be translated to the same NUMA topology in the guest, e.g. this user input:

```
    0   1   2
0  10  31 120
1  31  10  30
2 120  30  10
```

And this other user input:

```
    0   1   2
0  10  60  61
1  60  10  11
2  61  11  10
```

Will both be translated to the same values internally:

```
    0   1   2
0  10  40  80
1  40  10  20
2  80  20  10
```

Users are encouraged to use only the kernel values in the NUMA definition to avoid being taken by surprise with that the guest is actually seeing in the topology. There are enough potential surprises that are inherent to the associativity domain assignment process, discussed below.

How associativity domains are assigned

LOPAPR allows more than one associativity array (or 'string') per allocated resource. This would be used to represent that the resource has multiple connections with the board, and then the operational system, when deciding NUMA distancing, should consider the associativity information that provides the shortest distance.

The spapr implementation does not support multiple associativity arrays per resource, neither does the pseries Linux kernel. We'll have to represent the NUMA topology using one associativity per resource, which means that choices and compromises are going to be made.

Consider the following NUMA topology entered by user input:

```
    0   1   2   3
0  10  40  20  40
1  40  10  80  40
2  20  80  10  20
3  40  40  20  10
```

All the associativity arrays are initialized with NUMA id in all associativity domains:

- node 0: 0 0 0 0
- node 1: 1 1 1 1
- node 2: 2 2 2 2
- node 3: 3 3 3 3

Honoring just the relative distances of node 0 to every other node, we find the NUMA level matches (considering the reference points {0x4, 0x3, 0x2, 0x1}) for each distance:

- distance from 0 to 1 is 40 (no match at 0x4 and 0x3, will match at 0x2)
- distance from 0 to 2 is 20 (no match at 0x4, will match at 0x3)
- distance from 0 to 3 is 40 (no match at 0x4 and 0x3, will match at 0x2)

We'll copy the associativity domains of node 0 to all other nodes, based on the NUMA level matches. Between 0 and 1, a match in 0x2, we'll also copy the domains 0x2 and 0x1 from 0 to 1 as well. This will give us:

- node 0: 0 0 0 0
- node 1: 0 0 1 1

Doing the same to node 2 and node 3, these are the associativity arrays after considering all matches with node 0:

- node 0: 0 0 0 0
- node 1: 0 0 1 1
- node 2: 0 0 0 2
- node 3: 0 0 3 3

The distances related to node 0 are accounted for. For node 1, and keeping in mind that we don't need to revisit node 0 again, the distance from node 1 to 2 is 80, matching at 0x1, and distance from 1 to 3 is 40, match in 0x2. Repeating the same logic of copying all domains up to the NUMA level match:

- node 0: 0 0 0 0
- node 1: 1 0 1 1
- node 2: 1 0 0 2
- node 3: 1 0 3 3

In the last step we will analyze just nodes 2 and 3. The desired distance between 2 and 3 is 20, i.e. a match in 0x3:

- node 0: 0 0 0 0
- node 1: 1 0 1 1
- node 2: 1 0 0 2
- node 3: 1 0 0 3

The kernel will read these arrays and will calculate the following NUMA topology for the guest:

```
    0   1   2   3
0  10  40  20  20
1  40  10  40  40
2  20  40  10  20
3  20  40  20  10
```

Note that this is not what the user wanted - the desired distance between 0 and 3 is 40, we calculated it as 20. This is what the current logic and implementation constraints of the kernel and QEMU will provide inside the LOPAPR specification.

Users are welcome to use this knowledge and experiment with the input to get the NUMA topology they want, or as closer as they want. The important thing is to keep expectations up to par with what we are capable of provide at this moment: an approximation.

Limitations of the implementation

As mentioned above, the pSeries NUMA distance logic is, in fact, a way to approximate user choice. The Linux kernel, and PAPR itself, does not provide QEMU with the ways to fully map user input to actual NUMA distance the guest will use. These limitations creates two notable limitations in our support:

- Asymmetrical topologies aren't supported. We only support NUMA topologies where the distance from node A to B is always the same as B to A. We do not support any A-B pair where the distance back and forth is asymmetric. For example, the following topology isn't supported and the pSeries guest will not boot with this user input:

```
    0   1
0  10  40
1  20  10
```

- 'non-transitive' topologies will be poorly translated to the guest. This is the kind of topology where the distance from a node A to B is X, B to C is X, but the distance A to C is not X. E.g.:

```
    0   1   2   3
0  10  20  20  40
1  20  10  80  40
2  20  80  10  20
3  40  40  20  10

In the example above, distance 0 to 2 is 20, 2 to 3 is 20, but 0 to 3 is 40.
The kernel will always match with the shortest associativity domain possible,
and we're attempting to retain the previous established relations between the
nodes. This means that a distance equal to 20 between nodes 0 and 2 and the
same distance 20 between nodes 2 and 3 will cause the distance between 0 and 3
to also be 20.
```

Legacy (5.1 and older) pseries NUMA mechanics

In short, we can summarize the NUMA distances seem in pseries Linux guests, using QEMU up to 5.1, as follows:

- local distance, i.e. the distance of the resource to its own NUMA node: 10
- if it's a NVLink GPU device, distance: 80
- every other resource, distance: 40

The way the pseries Linux guest calculates NUMA distances has a direct effect on what QEMU users can expect when doing NUMA tuning. As of QEMU 5.1, this is the default ibm,associativity-reference-points being used in the pseries machine:

ibm,associativity-reference-points = {0x4, 0x4, 0x2}

The first and second level are equal, 0x4, and a third one was added in commit a6030d7e0b35 exclusively for NVLink GPUs support. This means that regardless of how the ibm,associativity properties are being created in the device tree, the pseries Linux guest will only recognize three scenarios as far as NUMA distance goes:

- if the resources belongs to the same first NUMA level = 10
- second level is skipped since it's equal to the first
- all resources that aren't a NVLink GPU, it is guaranteed that they will belong to the same third NUMA level, having distance = 40
- for NVLink GPUs, distance = 80 from everything else

This also means that user input in QEMU command line does not change the NUMA distancing inside the guest for the pseries machine.

Hypervisor calls and the Ultravisor

On PPC64 systems supporting Protected Execution Facility (PEF), system memory can be placed in a secured region where only an ultravisor running in firmware can provide access to. pSeries guests on such systems can communicate with the ultravisor (via ultracalls) to switch to a secure virtual machine (SVM) mode where the guest's memory is relocated to this secured region, making its memory inaccessible to normal processes/guests running on the host.

The various ultracalls/hypercalls relating to SVM mode are currently only documented internally, but are planned for direct inclusion into the Linux on Power Architecture Reference document ([LoPAR]). An internal ACR has been filed to reserve a hypercall number range specific to this use case to avoid any future conflicts with the IBM internally maintained Power Architecture Platform Reference (PAPR+) documentation specification. This document summarizes some of these details as they relate to QEMU.

Hypercalls needed by the ultravisor

Switching to SVM mode involves a number of hcalls issued by the ultravisor to the hypervisor to orchestrate the movement of guest memory to secure memory and various other aspects of the SVM mode. Numbers are assigned for these hcalls within the reserved range `0xEF00-0xEF80`. The below documents the hcalls relevant to QEMU.

H_TPM_COMM (0xef10)

SVM file systems are encrypted using a symmetric key. This key is then wrapped/encrypted using the public key of a trusted system which has the private key stored in the system's TPM. An Ultravisor will use this hcall to unwrap/unseal the symmetric key using the system's TPM device or a TPM Resource Manager associated with the device.

The Ultravisor sets up a separate session key with the TPM in advance during host system boot. All sensitive in and out values will be encrypted using the session key. Though the hypervisor will see the in and out buffers in raw form, any sensitive contents will generally be encrypted using this session key.

Arguments:

r3: H_TPM_COMM (0xef10)

r4: TPM operation, one of:

TPM_COMM_OP_EXECUTE (0x1): send a request to a TPM and receive a response, opening a new TPM session if one has not already been opened.

TPM_COMM_OP_CLOSE_SESSION (0x2): close the existing TPM session, if any.

r5: in_buffer, guest physical address of buffer containing the request. Caller may use the same address for both request and response.

r6: in_size, size of the in buffer. Must be less than or equal to 4 KB.

r7: out_buffer, guest physical address of buffer to store the response. Caller may use the same address for both request and response.

r8: out_size, size of the out buffer. Must be at least 4 KB, as this is the maximum request/response size supported by most TPM implementations, including the TPM Resource Manager in the linux kernel.

Return values:

r3: one of the following values:

H_Success: request processed successfully.

H_PARAMETER: invalid TPM operation.

H_P2: in_buffer is invalid.

H_P3: in_size is invalid.

H_P4: out_buffer is invalid.

H_P5: out_size is invalid.

H_RESOURCE: problem communicating with TPM.

H_FUNCTION: TPM access is not currently allowed/configured.

r4: For TPM_COMM_OP_EXECUTE, the size of the response will be stored here upon success.

XIVE for sPAPR (pseries machines)

The POWER9 processor comes with a new interrupt controller architecture, called XIVE as "eXternal Interrupt Virtualization Engine". It supports a larger number of interrupt sources and offers virtualization features which enables the HW to deliver interrupts directly to virtual processors without hypervisor assistance.

A QEMU `pseries` machine (which is PAPR compliant) using POWER9 processors can run under two interrupt modes:

- *Legacy Compatibility Mode*

  the hypervisor provides identical interfaces and similar functionality to PAPR+ Version 2.7. This is the default mode

  It is also referred as *XICS* in QEMU.

- *XIVE native exploitation mode*

  the hypervisor provides new interfaces to manage the XIVE control structures, and provides direct control for interrupt management through MMIO pages.

Which interrupt modes can be used by the machine is negotiated with the guest O/S during the Client Architecture Support negotiation sequence. The two modes are mutually exclusive.

Both interrupt mode share the same IRQ number space. See below for the layout.

CAS Negotiation

QEMU advertises the supported interrupt modes in the device tree property `ibm,arch-vec-5-platform-support` in byte 23 and the OS Selection for XIVE is indicated in the `ibm,architecture-vec-5` property byte 23.

The interrupt modes supported by the machine depend on the CPU type (POWER9 is required for XIVE) but also on the machine property `ic-mode` which can be set on the command line. It can take the following values: `xics`, `xive`, and `dual` which is the default mode. `dual` means that both modes XICS **and** XIVE are supported and if the guest OS supports XIVE, this mode will be selected.

The chosen interrupt mode is activated after a reconfiguration done in a machine reset.

KVM negotiation

When the guest starts under KVM, the capabilities of the host kernel and QEMU are also negotiated. Depending on the version of the host kernel, KVM will advertise the XIVE capability to QEMU or not.

Nevertheless, the available interrupt modes in the machine should not depend on the XIVE KVM capability of the host. On older kernels without XIVE KVM support, QEMU will use the emulated XIVE device as a fallback and on newer kernels (>=5.2), the KVM XIVE device.

XIVE native exploitation mode is not supported for KVM nested guests, VMs running under a L1 hypervisor (KVM on pSeries). In that case, the hypervisor will not advertise the KVM capability and QEMU will use the emulated XIVE device, same as for older versions of KVM.

As a final refinement, the user can also switch the use of the KVM device with the machine option `kernel_irqchip`.

XIVE support in KVM

For guest OSes supporting XIVE, the resulting interrupt modes on host kernels with XIVE KVM support are the following:

ic-mode	kernel_irqchip		
/	allowed (default)	off	on
dual (default)	XIVE KVM	XIVE emul.	XIVE KVM
xive	XIVE KVM	XIVE emul.	XIVE KVM
xics	XICS KVM	XICS emul.	XICS KVM

For legacy guest OSes without XIVE support, the resulting interrupt modes are the following:

ic-mode	kernel_irqchip		
/	allowed (default)	off	on
dual (default)	XICS KVM	XICS emul.	XICS KVM
xive	QEMU error(3)	QEMU error(3)	QEMU error(3)
xics	XICS KVM	XICS emul.	XICS KVM

(3) QEMU fails at CAS with `Guest requested unavailable interrupt mode (XICS), either don't set the ic-mode machine property or try ic-mode=xics or ic-mode=dual`

No XIVE support in KVM

For guest OSes supporting XIVE, the resulting interrupt modes on host kernels without XIVE KVM support are the following:

ic-mode	kernel_irqchip		
/	allowed (default)	off	on
dual (default)	XIVE emul.(1)	XIVE emul.	QEMU error (2)
xive	XIVE emul.(1)	XIVE emul.	QEMU error (2)
xics	XICS KVM	XICS emul.	XICS KVM

(1) QEMU warns with `warning: kernel_irqchip requested but unavailable: IRQ_XIVE capability must be present for KVM` In some cases (old host kernels or KVM nested guests), one may hit a QEMU/KVM incompatibility due to device destruction in reset. QEMU fails with `KVM is incompatible with ic-mode=dual,kernel-irqchip=on`

(2) QEMU fails with `kernel_irqchip requested but unavailable: IRQ_XIVE capability must be present for KVM`

For legacy guest OSes without XIVE support, the resulting interrupt modes are the following:

ic-mode	kernel_irqchip		
/	allowed (default)	off	on
dual (default)	QEMU error(4)	XICS emul.	QEMU error(4)
xive	QEMU error(3)	QEMU error(3)	QEMU error(3)
xics	XICS KVM	XICS emul.	XICS KVM

(3) QEMU fails at CAS with `Guest requested unavailable interrupt mode (XICS), either don't set the ic-mode machine property or try ic-mode=xics or ic-mode=dual`

(4) QEMU/KVM incompatibility due to device destruction in reset. QEMU fails with `KVM is incompatible with ic-mode=dual,kernel-irqchip=on`

XIVE Device tree properties

The properties for the PAPR interrupt controller node when the *XIVE native exploitation mode* is selected should contain:

- `device_type`

  value should be "power-ivpe".

- `compatible`

  value should be "ibm,power-ivpe".

- `reg`

  contains the base address and size of the thread interrupt managnement areas (TIMA), for the User level and for the Guest OS level. Only the Guest OS level is taken into account today.

- `ibm,xive-eq-sizes`

  the size of the event queues. One cell per size supported, contains log2 of size, in ascending order.

- `ibm,xive-lisn-ranges`

  the IRQ interrupt number ranges assigned to the guest for the IPIs.

The root node also exports :

- `ibm,plat-res-int-priorities`

  contains a list of priorities that the hypervisor has reserved for its own use.

IRQ number space

IRQ Number space of the `pseries` machine is 8K wide and is the same for both interrupt mode. The different ranges are defined as follow :

- `0x0000 .. 0x0FFF` 4K CPU IPIs (only used under XIVE)
- `0x1000 .. 0x1000` 1 EPOW
- `0x1001 .. 0x1001` 1 HOTPLUG
- `0x1002 .. 0x10FF` unused
- `0x1100 .. 0x11FF` 256 VIO devices
- `0x1200 .. 0x127F` 32x4 LSIs for PHB devices
- `0x1280 .. 0x12FF` unused
- `0x1300 .. 0x1FFF` PHB MSIs (dynamically allocated)

Monitoring XIVE

The state of the XIVE interrupt controller can be queried through the monitor commands `info pic`. The output comes in two parts.

First, the state of the thread interrupt context registers is dumped for each CPU :

```
(qemu) info pic
CPU[0000]:   QW   NSR CPPR IPB LSMFB ACK# INC AGE PIPR  W2
CPU[0000]: USER    00   00  00    00   00  00  00   00  00000000
CPU[0000]:   OS    00   ff  00    00   ff  00  ff   ff  80000400
CPU[0000]: POOL    00   00  00    00   00  00  00   00  00000000
CPU[0000]: PHYS    00   00  00    00   00  00  00   ff  00000000
...
```

In the case of a `pseries` machine, QEMU acts as the hypervisor and only the O/S and USER register rings make sense. `W2` contains the vCPU CAM line which is set to the VP identifier.

Then comes the routing information which aggregates the EAS and the END configuration:

```
...
LISN         PQ    EISN     CPU/PRIO EQ
00000000 MSI --    00000010   0/6    380/16384 @1fe3e0000 ^1 [ 80000010 ... ]
00000001 MSI --    00000010   1/6    305/16384 @1fc230000 ^1 [ 80000010 ... ]
00000002 MSI --    00000010   2/6    220/16384 @1fc2f0000 ^1 [ 80000010 ... ]
00000003 MSI --    00000010   3/6    201/16384 @1fc390000 ^1 [ 80000010 ... ]
00000004 MSI -Q  M 00000000
00000005 MSI -Q  M 00000000
00000006 MSI -Q  M 00000000
00000007 MSI -Q  M 00000000
00001000 MSI --    00000012   0/6    380/16384 @1fe3e0000 ^1 [ 80000010 ... ]
00001001 MSI --    00000013   0/6    380/16384 @1fe3e0000 ^1 [ 80000010 ... ]
00001100 MSI --    00000100   1/6    305/16384 @1fc230000 ^1 [ 80000010 ... ]
00001101 MSI -Q  M 00000000
00001200 LSI -Q  M 00000000
00001201 LSI -Q  M 00000000
00001202 LSI -Q  M 00000000
00001203 LSI -Q  M 00000000
00001300 MSI --    00000102   1/6    305/16384 @1fc230000 ^1 [ 80000010 ... ]
00001301 MSI --    00000103   2/6    220/16384 @1fc2f0000 ^1 [ 80000010 ... ]
00001302 MSI --    00000104   3/6    201/16384 @1fc390000 ^1 [ 80000010 ... ]
```

The source information and configuration:

- The `LISN` column outputs the interrupt number of the source in range `[ 0x0 ... 0x1FFF ]` and its type : `MSI` or `LSI`
- The `PQ` column reflects the state of the PQ bits of the source :
  - `--` source is ready to take events
  - `P-` an event was sent and an EOI is PENDING
  - `PQ` an event was QUEUED
  - `-Q` source is OFF

  a `M` indicates that source is *MASKED* at the EAS level,

The targeting configuration :

- The `EISN` column is the event data that will be queued in the event queue of the O/S.
- The `CPU/PRIO` column is the tuple defining the CPU number and priority queue serving the source.
- The `EQ` column outputs :
  - the current index of the event queue/ the max number of entries
  - the O/S event queue address
  - the toggle bit
  - the last entries that were pushed in the event queue.

Switching between the KVM-PR and KVM-HV kernel module

Currently, there are two implementations of KVM on Power, `kvm_hv.ko` and `kvm_pr.ko`.

If a host supports both KVM modes, and both KVM kernel modules are loaded, it is possible to switch between the two modes with the `kvm-type` parameter:

- Use `qemu-system-ppc64 -M pseries,accel=kvm,kvm-type=PR` to use the `kvm_pr.ko` kernel module.
- Use `qemu-system-ppc64 -M pseries,accel=kvm,kvm-type=HV` to use `kvm_hv.ko` instead.

KVM-PR

KVM-PR uses the so-called **PR**oblem state of the PPC CPUs to run the guests, i.e. the virtual machine is run in user mode and all privileged instructions trap and have to be emulated by the host. That means you can run KVM-PR inside a pSeries guest (or a PowerVM LPAR for that matter), and that is where it has originated, as historically (prior to POWER7) it was not possible to run Linux on hypervisor mode on a Power processor (this function was restricted to PowerVM, the IBM proprietary hypervisor).

Because all privileged instructions are trapped, guests that use a lot of privileged instructions run quite slow with KVM-PR. On the other hand, because of that, this kernel module can run on pretty much every PPC hardware, and is able to emulate a lot of guests CPUs. This module can even be used to run other PowerPC guests like an emulated PowerMac.

As KVM-PR can be run inside a pSeries guest, it can also provide nested virtualization capabilities (i.e. running a guest from within a guest).

It is important to notice that, as KVM-HV provides a much better execution performance, maintenance work has been much more focused on it in the past years. Maintenance for KVM-PR has been minimal.

In order to run KVM-PR guests with POWER9 processors, someone will need to start QEMU with `kernel_irqchip=off` command line option.

KVM-HV

KVM-HV uses the hypervisor mode of more recent Power processors, that allow access to the bare metal hardware directly. Although POWER7 had this capability, it was only starting with POWER8 that this was officially supported by IBM.

Originally, KVM-HV was only available when running on a PowerNV platform (a.k.a. Power bare metal). Although it runs on a PowerNV platform, it can only be used to start pSeries guests. As the pSeries guest doesn't have access to the hypervisor mode of the Power CPU, it wasn't possible to run KVM-HV on a guest. This limitation has been lifted, and now it is possible to run KVM-HV inside pSeries guests as well, making nested virtualization possible with KVM-HV.

As KVM-HV has access to privileged instructions, guests that use a lot of these can run much faster than with KVM-PR. On the other hand, the guest CPU has to be of the same type as the host CPU this way, e.g. it is not possible to specify an embedded PPC CPU for the guest with KVM-HV. However, there is at least the possibility to run the guest in a backward-compatibility mode of the previous CPUs generations, e.g. you can run a POWER7 guest on a POWER8 host by using `-cpu POWER8,compat=power7` as parameter to QEMU.

Modules support

As noticed in the sections above, each module can run in a different environment. The following table shows with which environment each module can run. As long as you are in a supported environment, you can run KVM-PR or KVM-HV nested. Combinations not shown in the table are not available.

Platform	Host type	Bits	Page table format	KVM-HV	KVM-PR
PowerNV	bare metal	32	hash	no	yes
			radix	N/A	N/A
		64	hash	yes	yes
			radix	yes	no
pSeries[1]	PowerNV	32	hash	no	yes
			radix	N/A	N/A
		64	hash	no	yes
			radix	yes[2]	no
	PowerVM	32	hash	no	yes
			radix	N/A	N/A
		64	hash	no	yes
			radix[3]	no	yes

POWER (PAPR) Protected Execution Facility (PEF)

Protected Execution Facility (PEF), also known as Secure Guest support is a feature found on IBM POWER9 and POWER10 processors.

If a suitable firmware including an Ultravisor is installed, it adds an extra memory protection mode to the CPU. The ultravisor manages a pool of secure memory which cannot be accessed by the hypervisor.

When this feature is enabled in QEMU, a guest can use ultracalls to enter "secure mode". This transfers most of its memory to secure memory, where it cannot be eavesdropped by a compromised hypervisor.

Launching

To launch a guest which will be permitted to enter PEF secure mode:

```
$ qemu-system-ppc64 \
    -object pef-guest,id=pef0 \
    -machine confidential-guest-support=pef0 \
    ...
```

[1] On POWER9 DD2.1 processors, the page table format on the host and guest must be the same.
[2] KVM-HV cannot run nested on POWER8 machines.
[3] Introduced on Power10 machines.

Live Migration

Live migration is not yet implemented for PEF guests. For consistency, QEMU currently prevents migration if the PEF feature is enabled, whether or not the guest has actually entered secure mode.

Maintainer contact information

Cédric Le Goater <clg@kaod.org>

Daniel Henrique Barboza <danielhb413@gmail.com>

2.23.6 OpenRISC System emulator

QEMU can emulate 32-bit OpenRISC CPUs using the `qemu-system-or1k` executable.

OpenRISC CPUs are generally built into "system-on-chip" (SoC) designs that run on FPGAs. These SoCs are based on the same core architecture as the or1ksim (the original OpenRISC instruction level simulator) which QEMU supports. For this reason QEMU does not need to support many different boards to support the OpenRISC hardware ecosystem.

The OpenRISC CPU supported by QEMU is the `or1200`, it supports an MMU and can run linux.

Choosing a board model

For QEMU's OpenRISC system emulation, you must specify which board model you want to use with the `-M` or `--machine` option; the default machine is `or1k-sim`.

If you intend to boot Linux, it is possible to have a single kernel image that will boot on any of the QEMU machines. To do this one would compile all required drivers into the kernel. This is possible because QEMU will create a device tree structure that describes the QEMU machine and pass a pointer to the structure to the kernel. The kernel can then use this to configure itself for the machine.

However, typically users will have specific firmware images for a specific machine.

If you already have a system image or a kernel that works on hardware and you want to boot with QEMU, check whether QEMU lists that machine in its `-machine help` output. If it is listed, then you can probably use that board model. If it is not listed, then unfortunately your image will almost certainly not boot on QEMU. (You might be able to extract the filesystem and use that with a different kernel which boots on a system that QEMU does emulate.)

If you don't care about reproducing the idiosyncrasies of a particular bit of hardware, such as small amount of RAM, no PCI or other hard disk, etc., and just want to run Linux, the best option is to use the `virt` board. This is a platform which doesn't correspond to any real hardware and is designed for use in virtual machines. You'll need to compile Linux with a suitable configuration for running on the `virt` board. `virt` supports PCI, virtio and large amounts of RAM.

Board-specific documentation

Or1ksim board

The QEMU Or1ksim machine emulates the standard OpenRISC board simulator which is also the standard SoC configuration.

Supported devices

- 16550A UART
- ETHOC Ethernet controller
- SMP (OpenRISC multicore using ompic)

Boot options

The Or1ksim machine can be started using the `-kernel` and `-initrd` options to load a Linux kernel and optional disk image.

```
$ qemu-system-or1k -cpu or1200 -M or1k-sim -nographic \
      -kernel vmlinux \
      -initrd initramfs.cpio.gz \
      -m 128
```

Linux guest kernel configuration

The 'or1ksim_defconfig' for Linux openrisc kernels includes the right drivers for the or1ksim machine. If you would like to run an SMP system choose the 'simple_smp_defconfig' config.

Hardware configuration information

The `or1k-sim` board automatically generates a device tree blob ("dtb") which it passes to the guest. This provides information about the addresses, interrupt lines and other configuration of the various devices in the system.

The location of the DTB will be passed in register `r3` to the guest operating system.

'virt' generic virtual platform

The `virt` board is a platform which does not correspond to any real hardware; it is designed for use in virtual machines. It is the recommended board type if you simply want to run a guest such as Linux and do not care about reproducing the idiosyncrasies and limitations of a particular bit of real-world hardware.

Supported devices

- PCI/PCIe devices
- 8 virtio-mmio transport devices
- 16550A UART
- Goldfish RTC
- SiFive Test device for poweroff and reboot
- SMP (OpenRISC multicore using ompic)

Boot options

The virt machine can be started using the `-kernel` and `-initrd` options to load a Linux kernel and optional disk image. For example:

```
$ qemu-system-or1k -cpu or1220 -M or1k-sim -nographic \
      -device virtio-net-device,netdev=user -netdev user,id=user,net=10.9.0.1/24,host=10.
↪9.0.100 \
      -device virtio-blk-device,drive=d0 -drive file=virt.qcow2,id=d0,if=none,
↪format=qcow2 \
      -kernel vmlinux \
      -initrd initramfs.cpio.gz \
      -m 128
```

Linux guest kernel configuration

The 'virt_defconfig' for Linux openrisc kernels includes the right drivers for the `virt` machine.

Hardware configuration information

The `virt` board automatically generates a device tree blob ("dtb") which it passes to the guest. This provides information about the addresses, interrupt lines and other configuration of the various devices in the system.

The location of the DTB will be passed in register `r3` to the guest operating system.

Emulated CPU architecture support

OpenRISC 1000 CPU architecture support

QEMU's TCG emulation includes support for the OpenRISC or1200 implementation of the OpenRISC 1000 cpu architecture.

The or1200 cpu also has support for the following instruction subsets:

- ORBIS32 (OpenRISC Basic Instruction Set)
- ORFPX32 (OpenRISC Floating-Point eXtension)

In addition to the instruction subsets the QEMU TCG emulation also has support for most Class II (optional) instructions.

For information on all OpenRISC instructions please refer to the latest architecture manual available on the OpenRISC website in the OpenRISC Architecture section.

OpenRISC CPU features

CPU Features

The QEMU emulation of the OpenRISC architecture provides following built in features.

- Shadow GPRs
- MMU TLB with 128 entries, 1 way
- Power Management (PM)
- Programmable Interrupt Controller (PIC)
- Tick Timer

These features are on by default and the presence can be confirmed by checking the contents of the Unit Presence Register (`UPR`) and CPU Configuration Register (`CPUCFGR`).

2.23.7 RISC-V System emulator

QEMU can emulate both 32-bit and 64-bit RISC-V CPUs. Use the `qemu-system-riscv64` executable to simulate a 64-bit RISC-V machine, `qemu-system-riscv32` executable to simulate a 32-bit RISC-V machine.

QEMU has generally good support for RISC-V guests. It has support for several different machines. The reason we support so many is that RISC-V hardware is much more widely varying than x86 hardware. RISC-V CPUs are generally built into "system-on-chip" (SoC) designs created by many different companies with different devices, and these SoCs are then built into machines which can vary still further even if they use the same SoC.

For most boards the CPU type is fixed (matching what the hardware has), so typically you don't need to specify the CPU type by hand, except for special cases like the `virt` board.

Choosing a board model

For QEMU's RISC-V system emulation, you must specify which board model you want to use with the `-M` or `--machine` option; there is no default.

Because RISC-V systems differ so much and in fundamental ways, typically operating system or firmware images intended to run on one machine will not run at all on any other. This is often surprising for new users who are used to the x86 world where every system looks like a standard PC. (Once the kernel has booted, most user space software cares much less about the detail of the hardware.)

If you already have a system image or a kernel that works on hardware and you want to boot with QEMU, check whether QEMU lists that machine in its `-machine help` output. If it is listed, then you can probably use that board model. If it is not listed, then unfortunately your image will almost certainly not boot on QEMU. (You might be able to extract the file system and use that with a different kernel which boots on a system that QEMU does emulate.)

If you don't care about reproducing the idiosyncrasies of a particular bit of hardware, such as small amount of RAM, no PCI or other hard disk, etc., and just want to run Linux, the best option is to use the `virt` board. This is a platform which doesn't correspond to any real hardware and is designed for use in virtual machines. You'll need to compile Linux with a suitable configuration for running on the `virt` board. `virt` supports PCI, virtio, recent CPUs and large amounts of RAM. It also supports 64-bit CPUs.

Board-specific documentation

Unfortunately many of the RISC-V boards QEMU supports are currently undocumented; you can get a complete list by running `qemu-system-riscv64 --machine help`, or `qemu-system-riscv32 --machine help`.

Microchip PolarFire SoC Icicle Kit (`microchip-icicle-kit`)

Microchip PolarFire SoC Icicle Kit integrates a PolarFire SoC, with one SiFive's E51 plus four U54 cores and many on-chip peripherals and an FPGA.

For more details about Microchip PolarFire SoC, please see: https://www.microsemi.com/product-directory/soc-fpgas/5498-polarfire-soc-fpga

The Icicle Kit board information can be found here: https://www.microsemi.com/existing-parts/parts/152514

Supported devices

The `microchip-icicle-kit` machine supports the following devices:

- 1 E51 core
- 4 U54 cores
- Core Level Interruptor (CLINT)
- Platform-Level Interrupt Controller (PLIC)
- L2 Loosely Integrated Memory (L2-LIM)
- DDR memory controller
- 5 MMUARTs
- 1 DMA controller
- 2 GEM Ethernet controllers
- 1 SDHC storage controller

Boot options

The `microchip-icicle-kit` machine can start using the standard -bios functionality for loading its BIOS image, aka Hart Software Services (HSS). HSS loads the second stage bootloader U-Boot from an SD card. Then a kernel can be loaded from U-Boot. It also supports direct kernel booting via the -kernel option along with the device tree blob via -dtb. When direct kernel boot is used, the OpenSBI fw_dynamic BIOS image is used to boot a payload like U-Boot or OS kernel directly.

The user provided DTB should have the following requirements:

- The /cpus node should contain at least one subnode for E51 and the number of subnodes should match QEMU's `-smp` option
- The /memory reg size should match QEMU's selected ram_size via `-m`
- Should contain a node for the CLINT device with a compatible string "riscv,clint0"

QEMU follows below truth table to select which payload to execute:

-bios	-kernel	-dtb	payload
N	N	don't care	HSS
Y	don't care	don't care	HSS
N	Y	Y	kernel

The memory is set to 1537 MiB by default which is the minimum required high memory size by HSS. A sanity check on ram size is performed in the machine init routine to prompt user to increase the RAM size to > 1537 MiB when less than 1537 MiB ram is detected.

Running HSS

HSS 2020.12 release is tested at the time of writing. To build an HSS image that can be booted by the `microchip-icicle-kit` machine, type the following in the HSS source tree:

```
$ export CROSS_COMPILE=riscv64-linux-
$ cp boards/mpfs-icicle-kit-es/def_config .config
$ make BOARD=mpfs-icicle-kit-es
```

Download the official SD card image released by Microchip and prepare it for QEMU usage:

```
$ wget ftp://ftpsoc.microsemi.com/outgoing/core-image-minimal-dev-icicle-kit-es-sd-
↪20201009141623.rootfs.wic.gz
$ gunzip core-image-minimal-dev-icicle-kit-es-sd-20201009141623.rootfs.wic.gz
$ qemu-img resize core-image-minimal-dev-icicle-kit-es-sd-20201009141623.rootfs.wic 4G
```

Then we can boot the machine by:

```
$ qemu-system-riscv64 -M microchip-icicle-kit -smp 5 \
    -bios path/to/hss.bin -sd path/to/sdcard.img \
    -nic user,model=cadence_gem \
    -nic tap,ifname=tap,model=cadence_gem,script=no \
    -display none -serial stdio \
    -chardev socket,id=serial1,path=serial1.sock,server=on,wait=on \
    -serial chardev:serial1
```

With above command line, current terminal session will be used for the first serial port. Open another terminal window, and use `minicom` to connect the second serial port.

```
$ minicom -D unix\#serial1.sock
```

HSS output is on the first serial port (stdio) and U-Boot outputs on the second serial port. U-Boot will automatically load the Linux kernel from the SD card image.

Direct Kernel Boot

Sometimes we just want to test booting a new kernel, and transforming the kernel image to the format required by the HSS bootflow is tedious. We can use '-kernel' for direct kernel booting just like other RISC-V machines do.

In this mode, the OpenSBI fw_dynamic BIOS image for 'generic' platform is used to boot an S-mode payload like U-Boot or OS kernel directly.

For example, the following commands show building a U-Boot image from U-Boot mainline v2021.07 for the Microchip Icicle Kit board:

```
$ export CROSS_COMPILE=riscv64-linux-
$ make microchip_mpfs_icicle_defconfig
```

Then we can boot the machine by:

```
$ qemu-system-riscv64 -M microchip-icicle-kit -smp 5 -m 2G \
    -sd path/to/sdcard.img \
    -nic user,model=cadence_gem \
    -nic tap,ifname=tap,model=cadence_gem,script=no \
    -display none -serial stdio \
    -kernel path/to/u-boot/build/dir/u-boot.bin \
    -dtb path/to/u-boot/build/dir/u-boot.dtb
```

CAVEATS:

- Check the "stdout-path" property in the /chosen node in the DTB to determine which serial port is used for the serial console, e.g.: if the console is set to the second serial port, change to use "-serial null -serial stdio".
- The default U-Boot configuration uses CONFIG_OF_SEPARATE hence the ELF image `u-boot` cannot be passed to "-kernel" as it does not contain the DTB hence `u-boot.bin` has to be used which does contain one. To use the ELF image, we need to change to CONFIG_OF_EMBED or CONFIG_OF_PRIOR_STAGE.

Shakti C Reference Platform (`shakti_c`)

Shakti C Reference Platform is a reference platform based on arty a7 100t for the Shakti SoC.

Shakti SoC is a SoC based on the Shakti C-class processor core. Shakti C is a 64bit RV64GCSUN processor core.

For more details on Shakti SoC, please see: https://gitlab.com/shaktiproject/cores/shakti-soc/-/blob/master/fpga/boards/artya7-100t/c-class/README.rst

For more info on the Shakti C-class core, please see: https://c-class.readthedocs.io/en/latest/

Supported devices

The `shakti_c` machine supports the following devices:

- 1 C-class core
- Core Level Interruptor (CLINT)
- Platform-Level Interrupt Controller (PLIC)
- 1 UART

Boot options

The `shakti_c` machine can start using the standard -bios functionality for loading the baremetal application or opensbi.

Boot the machine

Shakti SDK

Shakti SDK can be used to generate the baremetal example UART applications.

```
$ git clone https://gitlab.com/behindbytes/shakti-sdk.git
$ cd shakti-sdk
$ make software PROGRAM=loopback TARGET=artix7_100t
```

Binary would be generated in:
software/examples/uart_applns/loopback/output/loopback.shakti

You could also download the precompiled example applications using below commands.

```
$ wget -c https://gitlab.com/behindbytes/shakti-binaries/-/raw/master/sdk/shakti_sdk_
↪qemu.zip
$ unzip shakti_sdk_qemu.zip
```

Then we can run the UART example using:

```
$ qemu-system-riscv64 -M shakti_c -nographic \
   -bios path/to/shakti_sdk_qemu/loopback.shakti
```

OpenSBI

We can also run OpenSBI with Test Payload.

```
$ git clone https://github.com/riscv/opensbi.git -b v0.9
$ cd opensbi
$ wget -c https://gitlab.com/behindbytes/shakti-binaries/-/raw/master/dts/shakti.dtb
$ export CROSS_COMPILE=riscv64-unknown-elf-
$ export FW_FDT_PATH=./shakti.dtb
$ make PLATFORM=generic
```

fw_payload.elf would be generated in build/platform/generic/firmware/fw_payload.elf. Boot it using the below qemu command.

```
$ qemu-system-riscv64 -M shakti_c -nographic \
   -bios path/to/fw_payload.elf
```

SiFive HiFive Unleashed (sifive_u)

SiFive HiFive Unleashed Development Board is the ultimate RISC-V development board featuring the Freedom U540 multi-core RISC-V processor.

Supported devices

The sifive_u machine supports the following devices:

- 1 E51 / E31 core
- Up to 4 U54 / U34 cores
- Core Local Interruptor (CLINT)
- Platform-Level Interrupt Controller (PLIC)
- Power, Reset, Clock, Interrupt (PRCI)
- L2 Loosely Integrated Memory (L2-LIM)
- DDR memory controller
- 2 UARTs
- 1 GEM Ethernet controller
- 1 GPIO controller
- 1 One-Time Programmable (OTP) memory with stored serial number
- 1 DMA controller
- 2 QSPI controllers
- 1 ISSI 25WP256 flash
- 1 SD card in SPI mode
- PWM0 and PWM1

Please note the real world HiFive Unleashed board has a fixed configuration of 1 E51 core and 4 U54 core combination and the RISC-V core boots in 64-bit mode. With QEMU, one can create a machine with 1 E51 core and up to 4 U54 cores. It is also possible to create a 32-bit variant with the same peripherals except that the RISC-V cores are replaced by the 32-bit ones (E31 and U34), to help testing of 32-bit guest software.

Hardware configuration information

The sifive_u machine automatically generates a device tree blob ("dtb") which it passes to the guest, if there is no -dtb option. This provides information about the addresses, interrupt lines and other configuration of the various devices in the system. Guest software should discover the devices that are present in the generated DTB instead of using a DTB for the real hardware, as some of the devices are not modeled by QEMU and trying to access these devices may cause unexpected behavior.

If users want to provide their own DTB, they can use the -dtb option. These DTBs should have the following requirements:

- The /cpus node should contain at least one subnode for E51 and the number of subnodes should match QEMU's -smp option
- The /memory reg size should match QEMU's selected ram_size via -m

- Should contain a node for the CLINT device with a compatible string "riscv,clint0" if using with OpenSBI BIOS images

Boot options

The `sifive_u` machine can start using the standard -kernel functionality for loading a Linux kernel, a VxWorks kernel, a modified U-Boot bootloader (S-mode) or ELF executable with the default OpenSBI firmware image as the -bios. It also supports booting the unmodified U-Boot bootloader using the standard -bios functionality.

Machine-specific options

The following machine-specific options are supported:

- serial=nnn

  The board serial number. When not given, the default serial number 1 is used.

  SiFive reserves the first 1 KiB of the 16 KiB OTP memory for internal use. The current usage is only used to store the serial number of the board at offset 0xfc. U-Boot reads the serial number from the OTP memory, and uses it to generate a unique MAC address to be programmed to the on-chip GEM Ethernet controller. When multiple QEMU `sifive_u` machines are created and connected to the same subnet, they all have the same MAC address hence it creates an unusable network. In such scenario, user should give different values to serial= when creating different `sifive_u` machines.

- start-in-flash

  When given, QEMU's ROM codes jump to QSPI memory-mapped flash directly. Otherwise QEMU will jump to DRAM or L2LIM depending on the msel= value. When not given, it defaults to direct DRAM booting.

- msel=[6|11]

  Mode Select (MSEL[3:0]) pins value, used to control where to boot from.

  The FU540 SoC supports booting from several sources, which are controlled using the Mode Select pins on the chip. Typically, the boot process runs through several stages before it begins execution of user-provided programs. These stages typically include the following:

  1. Zeroth Stage Boot Loader (ZSBL), which is contained in an on-chip mask ROM and provided by QEMU. Note QEMU implemented ROM codes are not the same as what is programmed in the hardware. The QEMU one is a simplified version, but it provides the same functionality as the hardware.
  2. First Stage Boot Loader (FSBL), which brings up PLLs and DDR memory. This is U-Boot SPL.
  3. Second Stage Boot Loader (SSBL), which further initializes additional peripherals as needed. This is U-Boot proper combined with an OpenSBI fw_dynamic firmware image.

  msel=6 means FSBL and SSBL are both on the QSPI flash. msel=11 means FSBL and SSBL are both on the SD card.

Running Linux kernel

Linux mainline v5.10 release is tested at the time of writing. To build a Linux mainline kernel that can be booted by the `sifive_u` machine in 64-bit mode, simply configure the kernel using the defconfig configuration:

```
$ export ARCH=riscv
$ export CROSS_COMPILE=riscv64-linux-
$ make defconfig
$ make
```

To boot the newly built Linux kernel in QEMU with the `sifive_u` machine:

```
$ qemu-system-riscv64 -M sifive_u -smp 5 -m 2G \
    -display none -serial stdio \
    -kernel arch/riscv/boot/Image \
    -initrd /path/to/rootfs.ext4 \
    -append "root=/dev/ram"
```

Alternatively, we can use a custom DTB to boot the machine by inserting a CLINT node in fu540-c000.dtsi in the Linux kernel,

```
clint: clint@2000000 {
    compatible = "riscv,clint0";
    interrupts-extended = <&cpu0_intc 3 &cpu0_intc 7
                           &cpu1_intc 3 &cpu1_intc 7
                           &cpu2_intc 3 &cpu2_intc 7
                           &cpu3_intc 3 &cpu3_intc 7
                           &cpu4_intc 3 &cpu4_intc 7>;
    reg = <0x00 0x2000000 0x00 0x10000>;
};
```

with the following command line options:

```
$ qemu-system-riscv64 -M sifive_u -smp 5 -m 8G \
    -display none -serial stdio \
    -kernel arch/riscv/boot/Image \
    -dtb arch/riscv/boot/dts/sifive/hifive-unleashed-a00.dtb \
    -initrd /path/to/rootfs.ext4 \
    -append "root=/dev/ram"
```

To build a Linux mainline kernel that can be booted by the `sifive_u` machine in 32-bit mode, use the rv32_defconfig configuration. A patch is required to fix the 32-bit boot issue for Linux kernel v5.10.

```
$ export ARCH=riscv
$ export CROSS_COMPILE=riscv64-linux-
$ curl https://patchwork.kernel.org/project/linux-riscv/patch/20201219001356.2887782-1-
↪atish.patra@wdc.com/mbox/ > riscv.patch
$ git am riscv.patch
$ make rv32_defconfig
$ make
```

Replace `qemu-system-riscv64` with `qemu-system-riscv32` in the command line above to boot the 32-bit Linux kernel. A rootfs image containing 32-bit applications shall be used in order for kernel to boot to user space.

Running VxWorks kernel

VxWorks 7 SR0650 release is tested at the time of writing. To build a 64-bit VxWorks mainline kernel that can be booted by the `sifive_u` machine, simply create a VxWorks source build project based on the sifive_generic BSP, and a VxWorks image project to generate the bootable VxWorks image, by following the BSP documentation instructions.

A pre-built 64-bit VxWorks 7 image for HiFive Unleashed board is available as part of the VxWorks SDK for testing as well. Instructions to download the SDK:

```
$ wget https://labs.windriver.com/downloads/wrsdk-vxworks7-sifive-hifive-1.01.tar.bz2
$ tar xvf wrsdk-vxworks7-sifive-hifive-1.01.tar.bz2
$ ls bsps/sifive_generic_1_0_0_0/uboot/uVxWorks
```

To boot the VxWorks kernel in QEMU with the `sifive_u` machine, use:

```
$ qemu-system-riscv64 -M sifive_u -smp 5 -m 2G \
    -display none -serial stdio \
    -nic tap,ifname=tap0,script=no,downscript=no \
    -kernel /path/to/vxWorks \
    -append "gem(0,0)host:vxWorks h=192.168.200.1 e=192.168.200.2:ffffff00 u=target␣
↪pw=vxTarget f=0x01"
```

It is also possible to test 32-bit VxWorks on the `sifive_u` machine. Create a 32-bit project to build the 32-bit VxWorks image, and use exact the same command line options with `qemu-system-riscv32`.

Running U-Boot

U-Boot mainline v2024.01 release is tested at the time of writing. To build a U-Boot mainline bootloader that can be booted by the `sifive_u` machine, use the sifive_unleashed_defconfig with similar commands as described above for Linux:

```
$ export CROSS_COMPILE=riscv64-linux-
$ export OPENSBI=/path/to/opensbi-riscv64-generic-fw_dynamic.bin
$ make sifive_unleashed_defconfig
```

You will get spl/u-boot-spl.bin and u-boot.itb file in the build tree.

To start U-Boot using the `sifive_u` machine, prepare an SPI flash image, or SD card image that is properly partitioned and populated with correct contents. genimage can be used to generate these images.

A sample configuration file for a 128 MiB SD card image is:

```
$ cat genimage_sdcard.cfg
image sdcard.img {
        size = 128M

        hdimage {
                gpt = true
        }

        partition u-boot-spl {
                image = "u-boot-spl.bin"
                offset = 17K
                partition-type-uuid = 5B193300-FC78-40CD-8002-E86C45580B47
```

(continues on next page)

(continued from previous page)

```
        }

        partition u-boot {
                image = "u-boot.itb"
                offset = 1041K
                partition-type-uuid = 2E54B353-1271-4842-806F-E436D6AF6985
        }
}
```

SPI flash image has slightly different partition offsets, and the size has to be 32 MiB to match the ISSI 25WP256 flash on the real board:

```
$ cat genimage_spi-nor.cfg
image spi-nor.img {
        size = 32M

        hdimage {
                gpt = true
        }

        partition u-boot-spl {
                image = "u-boot-spl.bin"
                offset = 20K
                partition-type-uuid = 5B193300-FC78-40CD-8002-E86C45580B47
        }

        partition u-boot {
                image = "u-boot.itb"
                offset = 1044K
                partition-type-uuid = 2E54B353-1271-4842-806F-E436D6AF6985
        }
}
```

Assume U-Boot binaries are put in the same directory as the config file, we can generate the image by:

```
$ genimage --config genimage_<boot_src>.cfg --inputpath .
```

Boot U-Boot from SD card, by specifying msel=11 and pass the SD card image to QEMU `sifive_u` machine:

```
$ qemu-system-riscv64 -M sifive_u,msel=11 -smp 5 -m 8G \
    -display none -serial stdio \
    -bios /path/to/u-boot-spl.bin \
    -drive file=/path/to/sdcard.img,if=sd
```

Changing msel= value to 6, allows booting U-Boot from the SPI flash:

```
$ qemu-system-riscv64 -M sifive_u,msel=6 -smp 5 -m 8G \
    -display none -serial stdio \
    -bios /path/to/u-boot-spl.bin \
    -drive file=/path/to/spi-nor.img,if=mtd
```

Note when testing U-Boot, QEMU automatically generated device tree blob is not used because U-Boot itself embeds device tree blobs for U-Boot SPL and U-Boot proper. Hence the number of cores and size of memory have to match the real hardware, ie: 5 cores (-smp 5) and 8 GiB memory (-m 8G).

Above use case is to run upstream U-Boot for the SiFive HiFive Unleashed board on QEMU `sifive_u` machine out of the box. This allows users to develop and test the recommended RISC-V boot flow with a real world use case: ZSBL (in QEMU) loads U-Boot SPL from SD card or SPI flash to L2LIM, then U-Boot SPL loads the combined payload image of OpenSBI fw_dynamic firmware and U-Boot proper.

However sometimes we want to have a quick test of booting U-Boot on QEMU without the needs of preparing the SPI flash or SD card images, an alternate way can be used, which is to create a U-Boot S-mode image by modifying the configuration of U-Boot:

```
$ export CROSS_COMPILE=riscv64-linux-
$ make sifive_unleashed_defconfig
$ ./scripts/config --enable OF_BOARD
$ ./scripts/config --disable BINMAN_FDT
$ ./scripts/config --disable SPL
$ make olddefconfig
```

This changes U-Boot to use the QEMU generated device tree blob, and bypass running the U-Boot SPL stage.

Boot the 64-bit U-Boot S-mode image directly:

```
$ qemu-system-riscv64 -M sifive_u -smp 5 -m 2G \
    -display none -serial stdio \
    -kernel /path/to/u-boot.bin
```

It's possible to create a 32-bit U-Boot S-mode image as well.

```
$ export CROSS_COMPILE=riscv64-linux-
$ make sifive_unleashed_defconfig
$ ./scripts/config --disable ARCH_RV64I
$ ./scripts/config --enable ARCH_RV32I
$ ./scripts/config --set-val TEXT_BASE 0x80400000
$ ./scripts/config --enable OF_BOARD
$ ./scripts/config --disable BINMAN_FDT
$ ./scripts/config --disable SPL
$ make olddefconfig
```

Use the same command line options to boot the 32-bit U-Boot S-mode image:

```
$ qemu-system-riscv32 -M sifive_u -smp 5 -m 2G \
    -display none -serial stdio \
    -kernel /path/to/u-boot.bin
```

'virt' Generic Virtual Platform (`virt`)

The `virt` board is a platform which does not correspond to any real hardware; it is designed for use in virtual machines. It is the recommended board type if you simply want to run a guest such as Linux and do not care about reproducing the idiosyncrasies and limitations of a particular bit of real-world hardware.

Supported devices

The `virt` machine supports the following devices:

- Up to 512 generic RV32GC/RV64GC cores, with optional extensions
- Core Local Interruptor (CLINT)
- Platform-Level Interrupt Controller (PLIC)
- CFI parallel NOR flash memory
- 1 NS16550 compatible UART
- 1 Google Goldfish RTC
- 1 SiFive Test device
- 8 virtio-mmio transport devices
- 1 generic PCIe host bridge
- The fw_cfg device that allows a guest to obtain data from QEMU

The hypervisor extension has been enabled for the default CPU, so virtual machines with hypervisor extension can simply be used without explicitly declaring.

Hardware configuration information

The `virt` machine automatically generates a device tree blob ("dtb") which it passes to the guest, if there is no `-dtb` option. This provides information about the addresses, interrupt lines and other configuration of the various devices in the system. Guest software should discover the devices that are present in the generated DTB.

If users want to provide their own DTB, they can use the `-dtb` option. These DTBs should have the following requirements:

- The number of subnodes of the /cpus node should match QEMU's `-smp` option
- The /memory reg size should match QEMU's selected ram_size via `-m`
- Should contain a node for the CLINT device with a compatible string "riscv,clint0" if using with OpenSBI BIOS images

Boot options

The `virt` machine can start using the standard -kernel functionality for loading a Linux kernel, a VxWorks kernel, an S-mode U-Boot bootloader with the default OpenSBI firmware image as the -bios. It also supports the recommended RISC-V bootflow: U-Boot SPL (M-mode) loads OpenSBI fw_dynamic firmware and U-Boot proper (S-mode), using the standard -bios functionality.

Using flash devices

By default, the first flash device (pflash0) is expected to contain S-mode firmware code. It can be configured as read-only, with the second flash device (pflash1) available to store configuration data.

For example, booting edk2 looks like

```
$ qemu-system-riscv64 \
   -blockdev node-name=pflash0,driver=file,read-only=on,filename=<edk2_code> \
   -blockdev node-name=pflash1,driver=file,filename=<edk2_vars> \
   -M virt,pflash0=pflash0,pflash1=pflash1 \
   ... other args ....
```

For TCG guests only, it is also possible to boot M-mode firmware from the first flash device (pflash0) by additionally passing `-bios none`, as in

```
$ qemu-system-riscv64 \
   -bios none \
   -blockdev node-name=pflash0,driver=file,read-only=on,filename=<m_mode_code> \
   -M virt,pflash0=pflash0 \
   ... other args ....
```

Firmware images used for pflash must be exactly 32 MiB in size.

Machine-specific options

The following machine-specific options are supported:

- aclint=[on|off]

  When this option is "on", ACLINT devices will be emulated instead of SiFive CLINT. When not specified, this option is assumed to be "off". This option is restricted to the TCG accelerator.

- acpi=[on|off|auto]

  When this option is "on" (which is the default), ACPI tables are generated and exposed as firmware tables etc/acpi/rsdp and etc/acpi/tables.

- aia=[none|aplic|aplic-imsic]

  This option allows selecting interrupt controller defined by the AIA (advanced interrupt architecture) specification. The "aia=aplic" selects APLIC (advanced platform level interrupt controller) to handle wired interrupts whereas the "aia=aplic-imsic" selects APLIC and IMSIC (incoming message signaled interrupt controller) to handle both wired interrupts and MSIs. When not specified, this option is assumed to be "none" which selects SiFive PLIC to handle wired interrupts.

- aia-guests=nnn

  The number of per-HART VS-level AIA IMSIC pages to be emulated for a guest having AIA IMSIC (i.e. "aia=aplic-imsic" selected). When not specified, the default number of per-HART VS-level AIA IMSIC pages is 0.

Running Linux kernel

Linux mainline v5.12 release is tested at the time of writing. To build a Linux mainline kernel that can be booted by the `virt` machine in 64-bit mode, simply configure the kernel using the defconfig configuration:

```
$ export ARCH=riscv
$ export CROSS_COMPILE=riscv64-linux-
$ make defconfig
$ make
```

To boot the newly built Linux kernel in QEMU with the `virt` machine:

```
$ qemu-system-riscv64 -M virt -smp 4 -m 2G \
    -display none -serial stdio \
    -kernel arch/riscv/boot/Image \
    -initrd /path/to/rootfs.cpio \
    -append "root=/dev/ram"
```

To build a Linux mainline kernel that can be booted by the `virt` machine in 32-bit mode, use the rv32_defconfig configuration. A patch is required to fix the 32-bit boot issue for Linux kernel v5.12.

```
$ export ARCH=riscv
$ export CROSS_COMPILE=riscv64-linux-
$ curl https://patchwork.kernel.org/project/linux-riscv/patch/20210627135117.28641-1-
↪bmeng.cn@gmail.com/mbox/ > riscv.patch
$ git am riscv.patch
$ make rv32_defconfig
$ make
```

Replace `qemu-system-riscv64` with `qemu-system-riscv32` in the command line above to boot the 32-bit Linux kernel. A rootfs image containing 32-bit applications shall be used in order for kernel to boot to user space.

Running U-Boot

U-Boot mainline v2021.04 release is tested at the time of writing. To build an S-mode U-Boot bootloader that can be booted by the `virt` machine, use the qemu-riscv64_smode_defconfig with similar commands as described above for Linux:

```
$ export CROSS_COMPILE=riscv64-linux-
$ make qemu-riscv64_smode_defconfig
```

Boot the 64-bit U-Boot S-mode image directly:

```
$ qemu-system-riscv64 -M virt -smp 4 -m 2G \
    -display none -serial stdio \
    -kernel /path/to/u-boot.bin
```

To test booting U-Boot SPL which in M-mode, which in turn loads a FIT image that bundles OpenSBI fw_dynamic firmware and U-Boot proper (S-mode) together, build the U-Boot images using riscv64_spl_defconfig:

```
$ export CROSS_COMPILE=riscv64-linux-
$ export OPENSBI=/path/to/opensbi-riscv64-generic-fw_dynamic.bin
$ make qemu-riscv64_spl_defconfig
```

The minimal QEMU commands to run U-Boot SPL are:

```
$ qemu-system-riscv64 -M virt -smp 4 -m 2G \
    -display none -serial stdio \
    -bios /path/to/u-boot-spl \
    -device loader,file=/path/to/u-boot.itb,addr=0x80200000
```

To test 32-bit U-Boot images, switch to use qemu-riscv32_smode_defconfig and riscv32_spl_defconfig builds, and replace `qemu-system-riscv64` with `qemu-system-riscv32` in the command lines above to boot the 32-bit U-Boot.

Enabling TPM

A TPM device can be connected to the virt board by following the steps below.

First launch the TPM emulator:

```
$ swtpm socket --tpm2 -t -d --tpmstate dir=/tmp/tpm \
      --ctrl type=unixio,path=swtpm-sock
```

Then launch QEMU with some additional arguments to link a TPM device to the backend:

```
$ qemu-system-riscv64 \
  ... other args .... \
  -chardev socket,id=chrtpm,path=swtpm-sock \
  -tpmdev emulator,id=tpm0,chardev=chrtpm \
  -device tpm-tis-device,tpmdev=tpm0
```

The TPM device can be seen in the memory tree and the generated device tree and should be accessible from the guest software.

RISC-V CPU firmware

When using the `sifive_u` or `virt` machine there are three different firmware boot options:

- `-bios default`

This is the default behaviour if no `-bios` option is included. This option will load the default OpenSBI firmware automatically. The firmware is included with the QEMU release and no user interaction is required. All a user needs to do is specify the kernel they want to boot with the `-kernel` option

- `-bios none`

QEMU will not automatically load any firmware. It is up to the user to load all the images they need.

- `-bios <file>`

Tells QEMU to load the specified file as the firmware.

2.23.8 RX System emulator

Use the executable `qemu-system-rx` to simulate RX target (GDB simulator). This target emulated following devices.

- R5F562N8 MCU
  - On-chip memory (ROM 512KB, RAM 96KB)
  - Interrupt Control Unit (ICUa)
  - 8Bit Timer x 1CH (TMR0,1)
  - Compare Match Timer x 2CH (CMT0,1)
  - Serial Communication Interface x 1CH (SCI0)
- External memory 16MByte

Example of `qemu-system-rx` usage for RX is shown below:

Download <`u-boot_image_file`> from https://osdn.net/users/ysato/pf/qemu/dl/u-boot.bin.gz

Start emulation of rx-virt::
qemu-system-rx -M gdbsim-r5f562n8 -bios <u-boot_image_file>

Download `kernel_image_file` from https://osdn.net/users/ysato/pf/qemu/dl/zImage

Download `device_tree_blob` from https://osdn.net/users/ysato/pf/qemu/dl/rx-virt.dtb

Start emulation of rx-virt::

qemu-system-rx -M gdbsim-r5f562n8
-kernel <kernel_image_file> -dtb <device_tree_blob> -append "earlycon"

2.23.9 s390x System emulator

QEMU can emulate z/Architecture (in particular, 64 bit) s390x systems via the `qemu-system-s390x` binary. Only one machine type, `s390-ccw-virtio`, is supported (with versioning for compatibility handling).

When using KVM as accelerator, QEMU can emulate CPUs up to the generation of the host. When using the default cpu model with TCG as accelerator, QEMU will emulate a subset of z13 cpu features that should be enough to run distributions built for the z13.

Device support

QEMU will not emulate most of the traditional devices found under LPAR or z/VM; virtio devices (especially using virtio-ccw) make up the bulk of the available devices. Passthrough of host devices via vfio-pci, vfio-ccw, or vfio-ap is also available.

Adjunct Processor (AP) Device

Contents

Introduction

The IBM Adjunct Processor (AP) Cryptographic Facility is comprised of three AP instructions and from 1 to 256 PCIe cryptographic adapter cards. These AP devices provide cryptographic functions to all CPUs assigned to a linux system running in an IBM Z system LPAR.

On s390x, AP adapter cards are exposed via the AP bus. This document describes how those cards may be made available to KVM guests using the VFIO mediated device framework.

AP Architectural Overview

In order understand the terminology used in the rest of this document, let's start with some definitions:

- AP adapter

  An AP adapter is an IBM Z adapter card that can perform cryptographic functions. There can be from 0 to 256 adapters assigned to an LPAR depending on the machine model. Adapters assigned to the LPAR in which a linux host is running will be available to the linux host. Each adapter is identified by a number from 0 to 255; however, the maximum adapter number allowed is determined by machine model. When installed, an AP adapter is accessed by AP instructions executed by any CPU.

- AP domain

  An adapter is partitioned into domains. Each domain can be thought of as a set of hardware registers for processing AP instructions. An adapter can hold up to 256 domains; however, the maximum domain number allowed is determined by machine model. Each domain is identified by a number from 0 to 255. Domains can be further classified into two types:

  - Usage domains are domains that can be accessed directly to process AP commands
  - Control domains are domains that are accessed indirectly by AP commands sent to a usage domain to control or change the domain; for example, to set a secure private key for the domain.

- AP Queue

  An AP queue is the means by which an AP command-request message is sent to an AP usage domain inside a specific AP. An AP queue is identified by a tuple comprised of an AP adapter ID (APID) and an AP queue index

(APQI). The APQI corresponds to a given usage domain number within the adapter. This tuple forms an AP Queue Number (APQN) uniquely identifying an AP queue. AP instructions include a field containing the APQN to identify the AP queue to which the AP command-request message is to be sent for processing.

- AP Instructions:

  There are three AP instructions:

  - NQAP: to enqueue an AP command-request message to a queue
  - DQAP: to dequeue an AP command-reply message from a queue
  - PQAP: to administer the queues

  AP instructions identify the domain that is targeted to process the AP command; this must be one of the usage domains. An AP command may modify a domain that is not one of the usage domains, but the modified domain must be one of the control domains.

Start Interpretive Execution (SIE) Instruction

A KVM guest is started by executing the Start Interpretive Execution (SIE) instruction. The SIE state description is a control block that contains the state information for a KVM guest and is supplied as input to the SIE instruction. The SIE state description contains a satellite control block called the Crypto Control Block (CRYCB). The CRYCB contains three fields to identify the adapters, usage domains and control domains assigned to the KVM guest:

- The AP Mask (APM) field is a bit mask that identifies the AP adapters assigned to the KVM guest. Each bit in the mask, from left to right, corresponds to an APID from 0-255. If a bit is set, the corresponding adapter is valid for use by the KVM guest.
- The AP Queue Mask (AQM) field is a bit mask identifying the AP usage domains assigned to the KVM guest. Each bit in the mask, from left to right, corresponds to an AP queue index (APQI) from 0-255. If a bit is set, the corresponding queue is valid for use by the KVM guest.
- The AP Domain Mask field is a bit mask that identifies the AP control domains assigned to the KVM guest. The ADM bit mask controls which domains can be changed by an AP command-request message sent to a usage domain from the guest. Each bit in the mask, from left to right, corresponds to a domain from 0-255. If a bit is set, the corresponding domain can be modified by an AP command-request message sent to a usage domain.

If you recall from the description of an AP Queue, AP instructions include an APQN to identify the AP adapter and AP queue to which an AP command-request message is to be sent (NQAP and PQAP instructions), or from which a command-reply message is to be received (DQAP instruction). The validity of an APQN is defined by the matrix calculated from the APM and AQM; it is the cross product of all assigned adapter numbers (APM) with all assigned queue indexes (AQM). For example, if adapters 1 and 2 and usage domains 5 and 6 are assigned to a guest, the APQNs (1,5), (1,6), (2,5) and (2,6) will be valid for the guest.

The APQNs can provide secure key functionality - i.e., a private key is stored on the adapter card for each of its domains - so each APQN must be assigned to at most one guest or the linux host.

Example 1: Valid configuration

	Guest1	Guest2
adapters	1, 2	1, 2
domains	5, 6	7

This is valid because both guests have a unique set of APQNs:

- Guest1 has APQNs (1,5), (1,6), (2,5) and (2,6);
- Guest2 has APQNs (1,7) and (2,7).

Example 2: Valid configuration

	Guest1	Guest2
adapters	1, 2	3, 4
domains	5, 6	5, 6

This is also valid because both guests have a unique set of APQNs:

- Guest1 has APQNs (1,5), (1,6), (2,5), (2,6);
- Guest2 has APQNs (3,5), (3,6), (4,5), (4,6)

Example 3: Invalid configuration

	Guest1	Guest2
adapters	1, 2	1
domains	5, 6	6, 7

This is an invalid configuration because both guests have access to APQN (1,6).

AP Matrix Configuration on Linux Host

A linux system is a guest of the LPAR in which it is running and has access to the AP resources configured for the LPAR. The LPAR's AP matrix is configured via its Activation Profile which can be edited on the HMC. When the linux system is started, the AP bus will detect the AP devices assigned to the LPAR and create the following in sysfs:

```
/sys/bus/ap
... [devices]
...... xx.yyyy
...... ...
...... cardxx
...... ...
```

Where:

cardxx
: is AP adapter number xx (in hex)

xx.yyyy
is an APQN with xx specifying the APID and yyyy specifying the APQI

For example, if AP adapters 5 and 6 and domains 4, 71 (0x47), 171 (0xab) and 255 (0xff) are configured for the LPAR, the sysfs representation on the linux host system would look like this:

```
/sys/bus/ap
... [devices]
...... 05.0004
...... 05.0047
...... 05.00ab
...... 05.00ff
...... 06.0004
...... 06.0047
...... 06.00ab
...... 06.00ff
...... card05
...... card06
```

A set of default device drivers are also created to control each type of AP device that can be assigned to the LPAR on which a linux host is running:

```
/sys/bus/ap
... [drivers]
...... [cex2acard]        for Crypto Express 2/3 accelerator cards
...... [cex2aqueue]       for AP queues served by Crypto Express 2/3
                          accelerator cards
...... [cex4card]         for Crypto Express 4/5/6 accelerator and coprocessor
                          cards
...... [cex4queue]        for AP queues served by Crypto Express 4/5/6
                          accelerator and coprocessor cards
...... [pcixcccard]       for Crypto Express 2/3 coprocessor cards
...... [pcixccqueue]      for AP queues served by Crypto Express 2/3
                          coprocessor cards
```

Binding AP devices to device drivers

There are two sysfs files that specify bitmasks marking a subset of the APQN range as 'usable by the default AP queue device drivers' or 'not usable by the default device drivers' and thus available for use by the alternate device driver(s). The sysfs locations of the masks are:

```
/sys/bus/ap/apmask
/sys/bus/ap/aqmask
```

The `apmask` is a 256-bit mask that identifies a set of AP adapter IDs (APID). Each bit in the mask, from left to right (i.e., from most significant to least significant bit in big endian order), corresponds to an APID from 0-255. If a bit is set, the APID is marked as usable only by the default AP queue device drivers; otherwise, the APID is usable by the vfio_ap device driver.

The `aqmask` is a 256-bit mask that identifies a set of AP queue indexes (APQI). Each bit in the mask, from left to right (i.e., from most significant to least significant bit in big endian order), corresponds to an APQI from 0-255. If a bit is set, the APQI is marked as usable only by the default AP queue device drivers; otherwise, the APQI is usable by the vfio_ap device driver.

Take, for example, the following mask:

```
0x7dffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

It indicates:

> 1, 2, 3, 4, 5, and 7-255 belong to the default drivers' pool, and 0 and 6 belong to the vfio_ap device driver's pool.

The APQN of each AP queue device assigned to the linux host is checked by the AP bus against the set of APQNs derived from the cross product of APIDs and APQIs marked as usable only by the default AP queue device drivers. If a match is detected, only the default AP queue device drivers will be probed; otherwise, the vfio_ap device driver will be probed.

By default, the two masks are set to reserve all APQNs for use by the default AP queue device drivers. There are two ways the default masks can be changed:

1. The sysfs mask files can be edited by echoing a string into the respective sysfs mask file in one of two formats:

   - An absolute hex string starting with 0x - like "0x12345678" - sets the mask. If the given string is shorter than the mask, it is padded with 0s on the right; for example, specifying a mask value of 0x41 is the same as specifying:

     ```
     0x4100000000000000000000000000000000000000000000000000000000000000
     ```

     Keep in mind that the mask reads from left to right (i.e., most significant to least significant bit in big endian order), so the mask above identifies device numbers 1 and 7 (`01000001`).

     If the string is longer than the mask, the operation is terminated with an error (EINVAL).

   - Individual bits in the mask can be switched on and off by specifying each bit number to be switched in a comma separated list. Each bit number string must be prepended with a (+) or minus (-) to indicate the corresponding bit is to be switched on (+) or off (-). Some valid values are:

     ```
     "+0"    switches bit 0 on
     "-13"   switches bit 13 off
     "+0x41" switches bit 65 on
     "-0xff" switches bit 255 off
     ```

     The following example:

     ```
     +0,-6,+0x47,-0xf0
     ```

     Switches bits 0 and 71 (0x47) on Switches bits 6 and 240 (0xf0) off

     Note that the bits not specified in the list remain as they were before the operation.

2. The masks can also be changed at boot time via parameters on the kernel command line like this:

   ```
   ap.apmask=0xffff ap.aqmask=0x40
   ```

   This would create the following masks:

   apmask:

   ```
   0xffff0000000000000000000000000000000000000000000000000000000000000000
   ```

   aqmask:

   ```
   0x4000000000000000000000000000000000000000000000000000000000000000
   ```

   Resulting in these two pools:

```
default drivers pool:    adapter 0-15, domain 1
alternate drivers pool:  adapter 16-255, domains 0, 2-255
```

Configuring an AP matrix for a linux guest

The sysfs interfaces for configuring an AP matrix for a guest are built on the VFIO mediated device framework. To configure an AP matrix for a guest, a mediated matrix device must first be created for the `/sys/devices/vfio_ap/matrix` device. When the vfio_ap device driver is loaded, it registers with the VFIO mediated device framework. When the driver registers, the sysfs interfaces for creating mediated matrix devices is created:

```
/sys/devices
... [vfio_ap]
......[matrix]
......... [mdev_supported_types]
............ [vfio_ap-passthrough]
............... create
............... [devices]
```

A mediated AP matrix device is created by writing a UUID to the attribute file named `create`, for example:

```
uuidgen > create
```

or

```
echo $uuid > create
```

When a mediated AP matrix device is created, a sysfs directory named after the UUID is created in the `devices` subdirectory:

```
/sys/devices
... [vfio_ap]
......[matrix]
......... [mdev_supported_types]
............ [vfio_ap-passthrough]
............... create
............... [devices]
................... [$uuid]
```

There will also be three sets of attribute files created in the mediated matrix device's sysfs directory to configure an AP matrix for the KVM guest:

```
/sys/devices
... [vfio_ap]
......[matrix]
......... [mdev_supported_types]
............ [vfio_ap-passthrough]
............... create
............... [devices]
................... [$uuid]
...................... assign_adapter
...................... assign_control_domain
...................... assign_domain
```

(continues on next page)

(continued from previous page)

```
..................... matrix
..................... unassign_adapter
..................... unassign_control_domain
..................... unassign_domain
```

assign_adapter

To assign an AP adapter to the mediated matrix device, its APID is written to the `assign_adapter` file. This may be done multiple times to assign more than one adapter. The APID may be specified using conventional semantics as a decimal, hexadecimal, or octal number. For example, to assign adapters 4, 5 and 16 to a mediated matrix device in decimal, hexadecimal and octal respectively:

```
echo 4 > assign_adapter
echo 0x5 > assign_adapter
echo 020 > assign_adapter
```

In order to successfully assign an adapter:

- The adapter number specified must represent a value from 0 up to the maximum adapter number allowed by the machine model. If an adapter number higher than the maximum is specified, the operation will terminate with an error (ENODEV).
- All APQNs that can be derived from the adapter ID being assigned and the IDs of the previously assigned domains must be bound to the vfio_ap device driver. If no domains have yet been assigned, then there must be at least one APQN with the specified APID bound to the vfio_ap driver. If no such APQNs are bound to the driver, the operation will terminate with an error (EADDRNOTAVAIL).
- No APQN that can be derived from the adapter ID and the IDs of the previously assigned domains can be assigned to another mediated matrix device. If an APQN is assigned to another mediated matrix device, the operation will terminate with an error (EADDRINUSE).

unassign_adapter

To unassign an AP adapter, its APID is written to the `unassign_adapter` file. This may also be done multiple times to unassign more than one adapter.

assign_domain

To assign a usage domain, the domain number is written into the `assign_domain` file. This may be done multiple times to assign more than one usage domain. The domain number is specified using conventional semantics as a decimal, hexadecimal, or octal number. For example, to assign usage domains 4, 8, and 71 to a mediated matrix device in decimal, hexadecimal and octal respectively:

```
echo 4 > assign_domain
echo 0x8 > assign_domain
echo 0107 > assign_domain
```

In order to successfully assign a domain:

- The domain number specified must represent a value from 0 up to the maximum domain number allowed by the machine model. If a domain number higher than the maximum is specified, the operation will terminate with an error (ENODEV).
- All APQNs that can be derived from the domain ID being assigned and the IDs of the previously assigned adapters must be bound to the vfio_ap device driver. If no domains have yet been assigned, then there must be at least one APQN with the specified APQI bound to the vfio_ap driver. If no such APQNs are bound to the driver, the operation will terminate with an error (EADDRNOTAVAIL).
- No APQN that can be derived from the domain ID being assigned and the IDs of the previously assigned adapters can be assigned to another mediated matrix device. If an APQN is assigned to another mediated

matrix device, the operation will terminate with an error (EADDRINUSE).

unassign_domain
To unassign a usage domain, the domain number is written into the `unassign_domain` file. This may be done multiple times to unassign more than one usage domain.

assign_control_domain
To assign a control domain, the domain number is written into the `assign_control_domain` file. This may be done multiple times to assign more than one control domain. The domain number may be specified using conventional semantics as a decimal, hexadecimal, or octal number. For example, to assign control domains 4, 8, and 71 to a mediated matrix device in decimal, hexadecimal and octal respectively:

```
echo 4 > assign_domain
echo 0x8 > assign_domain
echo 0107 > assign_domain
```

In order to successfully assign a control domain, the domain number specified must represent a value from 0 up to the maximum domain number allowed by the machine model. If a control domain number higher than the maximum is specified, the operation will terminate with an error (ENODEV).

unassign_control_domain
To unassign a control domain, the domain number is written into the `unassign_domain` file. This may be done multiple times to unassign more than one control domain.

Notes: No changes to the AP matrix will be allowed while a guest using the mediated matrix device is running. Attempts to assign an adapter, domain or control domain will be rejected and an error (EBUSY) returned.

Starting a Linux Guest Configured with an AP Matrix

To provide a mediated matrix device for use by a guest, the following option must be specified on the QEMU command line:

```
-device vfio_ap,sysfsdev=$path-to-mdev
```

The sysfsdev parameter specifies the path to the mediated matrix device. There are a number of ways to specify this path:

```
/sys/devices/vfio_ap/matrix/$uuid
/sys/bus/mdev/devices/$uuid
/sys/bus/mdev/drivers/vfio_mdev/$uuid
/sys/devices/vfio_ap/matrix/mdev_supported_types/vfio_ap-passthrough/devices/$uuid
```

When the linux guest is started, the guest will open the mediated matrix device's file descriptor to get information about the mediated matrix device. The `vfio_ap` device driver will update the APM, AQM, and ADM fields in the guest's CRYCB with the adapter, usage domain and control domains assigned via the mediated matrix device's sysfs attribute files. Programs running on the linux guest will then:

1. Have direct access to the APQNs derived from the cross product of the AP adapter numbers (APID) and queue indexes (APQI) specified in the APM and AQM fields of the guests's CRYCB respectively. These APQNs identify the AP queues that are valid for use by the guest; meaning, AP commands can be sent by the guest to any of these queues for processing.
2. Have authorization to process AP commands to change a control domain identified in the ADM field of the guest's CRYCB. The AP command must be sent to a valid APQN (see 1 above).

CPU model features:

Three CPU model features are available for controlling guest access to AP facilities:

1. AP facilities feature

   The AP facilities feature indicates that AP facilities are installed on the guest. This feature will be exposed for use only if the AP facilities are installed on the host system. The feature is s390-specific and is represented as a parameter of the -cpu option on the QEMU command line:

   ```
   qemu-system-s390x -cpu $model,ap=on|off
   ```

   Where:

   $model
   is the CPU model defined for the guest (defaults to the model of the host system if not specified).

   ap=on|off
   indicates whether AP facilities are installed (on) or not (off). The default for CPU models zEC12 or newer is `ap=on`. AP facilities must be installed on the guest if a vfio-ap device (`-device vfio-ap,sysfsdev=$path`) is configured for the guest, or the guest will fail to start.

2. Query Configuration Information (QCI) facility

   The QCI facility is used by the AP bus running on the guest to query the configuration of the AP facilities. This facility will be available only if the QCI facility is installed on the host system. The feature is s390-specific and is represented as a parameter of the -cpu option on the QEMU command line:

   ```
   qemu-system-s390x -cpu $model,apqci=on|off
   ```

   Where:

   $model
   is the CPU model defined for the guest

   apqci=on|off
   indicates whether the QCI facility is installed (on) or not (off). The default for CPU models zEC12 or newer is `apqci=on`; for older models, QCI will not be installed.

   If QCI is installed (`apqci=on`) but AP facilities are not (`ap=off`), an error message will be logged, but the guest will be allowed to start. It makes no sense to have QCI installed if the AP facilities are not; this is considered an invalid configuration.

   If the QCI facility is not installed, APQNs with an APQI greater than 15 will not be detected by the AP bus running on the guest.

3. Adjunct Process Facility Test (APFT) facility

   The APFT facility is used by the AP bus running on the guest to test the AP facilities available for a given AP queue. This facility will be available only if the APFT facility is installed on the host system. The feature is s390-specific and is represented as a parameter of the -cpu option on the QEMU command line:

   ```
   qemu-system-s390x -cpu $model,apft=on|off
   ```

   Where:

   $model
   is the CPU model defined for the guest (defaults to the model of the host system if not specified).

   apft=on|off
   indicates whether the APFT facility is installed (on) or not (off). The default for CPU models zEC12 and newer is `apft=on` for older models, APFT will not be installed.

   If APFT is installed (`apft=on`) but AP facilities are not (`ap=off`), an error message will be logged, but the guest will be allowed to start. It makes no sense to have APFT installed if the AP facilities are not; this is considered an invalid configuration.

It also makes no sense to turn APFT off because the AP bus running on the guest will not detect CEX4 and newer devices without it. Since only CEX4 and newer devices are supported for guest usage, no AP devices can be made accessible to a guest started without APFT installed.

Hot plug a vfio-ap device into a running guest

Only one vfio-ap device can be attached to the virtual machine's ap-bus, so a vfio-ap device can be hot plugged if and only if no vfio-ap device is attached to the bus already, whether via the QEMU command line or a prior hot plug action.

To hot plug a vfio-ap device, use the QEMU `device_add` command:

```
(qemu) device_add vfio-ap,sysfsdev="$path-to-mdev",id="$id"
```

Where the `$path-to-mdev` value specifies the absolute path to a mediated device to which AP resources to be used by the guest have been assigned. `$id` is the name value for the optional id parameter.

Note that on Linux guests, the AP devices will be created in the `/sys/bus/ap/devices` directory when the AP bus subsequently performs its periodic scan, so there may be a short delay before the AP devices are accessible on the guest.

The command will fail if:

- A vfio-ap device has already been attached to the virtual machine's ap-bus.
- The CPU model features for controlling guest access to AP facilities are not enabled (see 'CPU model features' subsection in the previous section).

Hot unplug a vfio-ap device from a running guest

A vfio-ap device can be unplugged from a running KVM guest if a vfio-ap device has been attached to the virtual machine's ap-bus via the QEMU command line or a prior hot plug action.

To hot unplug a vfio-ap device, use the QEMU `device_del` command:

```
(qemu) device_del "$id"
```

Where `$id` is the same id that was specified at device creation.

On a Linux guest, the AP devices will be removed from the `/sys/bus/ap/devices` directory on the guest when the AP bus subsequently performs its periodic scan, so there may be a short delay before the AP devices are no longer accessible by the guest.

The command will fail if the `$path-to-mdev` specified on the `device_del` command does not match the value specified when the vfio-ap device was attached to the virtual machine's ap-bus.

Example: Configure AP Matrices for Three Linux Guests

Let's now provide an example to illustrate how KVM guests may be given access to AP facilities. For this example, we will show how to configure three guests such that executing the lszcrypt command on the guests would look like this:

Guest1:

```
CARD.DOMAIN TYPE  MODE
------------------------------
05          CEX5C CCA-Coproc
05.0004     CEX5C CCA-Coproc
```

(continues on next page)

(continued from previous page)

```
05.00ab     CEX5C CCA-Coproc
06          CEX5A Accelerator
06.0004     CEX5A Accelerator
06.00ab     CEX5C CCA-Coproc
```

Guest2:

```
CARD.DOMAIN TYPE  MODE
------------------------------
05          CEX5A Accelerator
05.0047     CEX5A Accelerator
05.00ff     CEX5A Accelerator
```

Guest3:

```
CARD.DOMAIN TYPE  MODE
------------------------------
06          CEX5A Accelerator
06.0047     CEX5A Accelerator
06.00ff     CEX5A Accelerator
```

These are the steps:

1. Install the vfio_ap module on the linux host. The dependency chain for the vfio_ap module is:
   - iommu
   - s390
   - zcrypt
   - vfio
   - vfio_mdev
   - vfio_mdev_device
   - KVM

   To build the vfio_ap module, the kernel build must be configured with the following Kconfig elements selected:
   - IOMMU_SUPPORT
   - S390
   - ZCRYPT
   - S390_AP_IOMMU
   - VFIO
   - VFIO_MDEV
   - VFIO_MDEV_DEVICE
   - KVM

   If using make menuconfig select the following to build the vfio_ap module::

   -> Device Drivers

   -> IOMMU Hardware Support
   select S390 AP IOMMU Support

-> VFIO Non-Privileged userspace driver framework

-> Mediated device driver framework
-> VFIO driver for Mediated devices

-> I/O subsystem
-> VFIO support for AP devices

2. Secure the AP queues to be used by the three guests so that the host can not access them. To secure the AP queues 05.0004, 05.0047, 05.00ab, 05.00ff, 06.0004, 06.0047, 06.00ab, and 06.00ff for use by the vfio_ap device driver, the corresponding APQNs must be removed from the default queue drivers pool as follows:

```
echo -5,-6 > /sys/bus/ap/apmask

echo -4,-0x47,-0xab,-0xff > /sys/bus/ap/aqmask
```

This will result in AP queues 05.0004, 05.0047, 05.00ab, 05.00ff, 06.0004, 06.0047, 06.00ab, and 06.00ff getting bound to the vfio_ap device driver. The sysfs directory for the vfio_ap device driver will now contain symbolic links to the AP queue devices bound to it:

```
/sys/bus/ap
... [drivers]
...... [vfio_ap]
......... [05.0004]
......... [05.0047]
......... [05.00ab]
......... [05.00ff]
......... [06.0004]
......... [06.0047]
......... [06.00ab]
......... [06.00ff]
```

Keep in mind that only type 10 and newer adapters (i.e., CEX4 and later) can be bound to the vfio_ap device driver. The reason for this is to simplify the implementation by not needlessly complicating the design by supporting older devices that will go out of service in the relatively near future, and for which there are few older systems on which to test.

The administrator, therefore, must take care to secure only AP queues that can be bound to the vfio_ap device driver. The device type for a given AP queue device can be read from the parent card's sysfs directory. For example, to see the hardware type of the queue 05.0004:

```
cat /sys/bus/ap/devices/card05/hwtype
```

The hwtype must be 10 or higher (CEX4 or newer) in order to be bound to the vfio_ap device driver.

3. Create the mediated devices needed to configure the AP matrixes for the three guests and to provide an interface to the vfio_ap driver for use by the guests:

```
/sys/devices/vfio_ap/matrix/
... [mdev_supported_types]
...... [vfio_ap-passthrough] (passthrough mediated matrix device type)
......... create
......... [devices]
```

To create the mediated devices for the three guests:

```
uuidgen > create
uuidgen > create
uuidgen > create
```

or

```
echo $uuid1 > create
echo $uuid2 > create
echo $uuid3 > create
```

This will create three mediated devices in the [devices] subdirectory named after the UUID used to create the mediated device. We'll call them $uuid1, $uuid2 and $uuid3 and this is the sysfs directory structure after creation:

```
/sys/devices/vfio_ap/matrix/
... [mdev_supported_types]
...... [vfio_ap-passthrough]
......... [devices]
............ [$uuid1]
.............. assign_adapter
.............. assign_control_domain
.............. assign_domain
.............. matrix
.............. unassign_adapter
.............. unassign_control_domain
.............. unassign_domain

............ [$uuid2]
.............. assign_adapter
.............. assign_control_domain
.............. assign_domain
.............. matrix
.............. unassign_adapter
.............. unassign_control_domain
.............. unassign_domain

............ [$uuid3]
.............. assign_adapter
.............. assign_control_domain
.............. assign_domain
.............. matrix
.............. unassign_adapter
.............. unassign_control_domain
.............. unassign_domain
```

4. The administrator now needs to configure the matrixes for the mediated devices $uuid1 (for Guest1), $uuid2 (for Guest2) and $uuid3 (for Guest3).

   This is how the matrix is configured for Guest1:

```
echo 5 > assign_adapter
echo 6 > assign_adapter
echo 4 > assign_domain
echo 0xab > assign_domain
```

   Control domains can similarly be assigned using the assign_control_domain sysfs file.

If a mistake is made configuring an adapter, domain or control domain, you can use the `unassign_xxx` interfaces to unassign the adapter, domain or control domain.

To display the matrix configuration for Guest1:

```
cat matrix
```

The output will display the APQNs in the format `xx.yyyy`, where xx is the adapter number and yyyy is the domain number. The output for Guest1 will look like this:

```
05.0004
05.00ab
06.0004
06.00ab
```

This is how the matrix is configured for Guest2:

```
echo 5 > assign_adapter
echo 0x47 > assign_domain
echo 0xff > assign_domain
```

This is how the matrix is configured for Guest3:

```
echo 6 > assign_adapter
echo 0x47 > assign_domain
echo 0xff > assign_domain
```

5. Start Guest1:

```
/usr/bin/qemu-system-s390x ... -cpu host,ap=on,apqci=on,apft=on -device vfio-ap,
↪sysfsdev=/sys/devices/vfio_ap/matrix/$uuid1 ...
```

7. Start Guest2:

```
/usr/bin/qemu-system-s390x ... -cpu host,ap=on,apqci=on,apft=on -device vfio-ap,
↪sysfsdev=/sys/devices/vfio_ap/matrix/$uuid2 ...
```

7. Start Guest3:

```
/usr/bin/qemu-system-s390x ... -cpu host,ap=on,apqci=on,apft=on -device vfio-ap,
↪sysfsdev=/sys/devices/vfio_ap/matrix/$uuid3 ...
```

When the guest is shut down, the mediated matrix devices may be removed.

Using our example again, to remove the mediated matrix device $uuid1:

```
/sys/devices/vfio_ap/matrix/
... [mdev_supported_types]
...... [vfio_ap-passthrough]
......... [devices]
............ [$uuid1]
............... remove


echo 1 > remove
```

This will remove all of the mdev matrix device's sysfs structures including the mdev device itself. To recreate and reconfigure the mdev matrix device, all of the steps starting with step 3 will have to be performed again. Note that the remove will fail if a guest using the mdev is still running.

It is not necessary to remove an mdev matrix device, but one may want to remove it if no guest will use it during the remaining lifetime of the linux host. If the mdev matrix device is removed, one may want to also reconfigure the pool of adapters and queues reserved for use by the default drivers.

Limitations

- The KVM/kernel interfaces do not provide a way to prevent restoring an APQN to the default drivers pool of a queue that is still assigned to a mediated device in use by a guest. It is incumbent upon the administrator to ensure there is no mediated device in use by a guest to which the APQN is assigned lest the host be given access to the private data of the AP queue device, such as a private key configured specifically for the guest.
- Dynamically assigning AP resources to or unassigning AP resources from a mediated matrix device - see *Configuring an AP matrix for a linux guest* section above - while a running guest is using it is currently not supported.
- Live guest migration is not supported for guests using AP devices. If a guest is using AP devices, the vfio-ap device configured for the guest must be unplugged before migrating the guest (see *Hot unplug a vfio-ap device from a running guest* section above.)

The virtual channel subsystem

QEMU implements a virtual channel subsystem with subchannels, (mostly functionless) channel paths, and channel devices (virtio-ccw, 3270, and devices passed via vfio-ccw). It supports multiple subchannel sets (MSS) and multiple channel subsystems extended (MCSS-E).

All channel devices support the `devno` property, which takes a parameter in the form `<cssid>.<ssid>.<device number>`.

The default channel subsystem image id (`<cssid>`) is `0xfe`. Devices in there will show up in channel subsystem image `0` to guests that do not enable MCSS-E. Note that devices with a different cssid will not be visible if the guest OS does not enable MCSS-E (which is true for all supported guest operating systems today).

Supported values for the subchannel set id (`<ssid>`) range from `0-3`. Devices with a ssid that is not `0` will not be visible if the guest OS does not enable MSS (any Linux version that supports virtio also enables MSS). Any device may be put into any subchannel set, there is no restriction by device type.

The device number can range from `0-0xffff`.

If the `devno` property is not specified for a device, QEMU will choose the next free device number in subchannel set 0, skipping to the next subchannel set if no more device numbers are free.

QEMU places a device at the first free subchannel in the specified subchannel set. If a device is hotunplugged and later replugged, it may appear at a different subchannel. (This is similar to how z/VM works.)

Examples

- a virtio-net device, cssid/ssid/devno automatically assigned:

```
-device virtio-net-ccw
```

  In a Linux guest (without default devices and no other devices specified prior to this one), this will show up as `0.0.0000` under subchannel `0.0.0000`.

  The auto-assigned-properties in QEMU (as seen via e.g. `info qtree`) would be `dev_id = "fe.0.0000"` and `subch_id = "fe.0.0000"`.

- a virtio-rng device in subchannel set `0`:

```
-device virtio-rng-ccw,devno=fe.0.0042
```

  If added to the same Linux guest as above, it would show up as `0.0.0042` under subchannel `0.0.0001`.

  The properties for the device would be `dev_id = "fe.0.0042"` and `subch_id = "fe.0.0001"`.

- a virtio-gpu device in subchannel set `2`:

```
-device virtio-gpu-ccw,devno=fe.2.1111
```

  If added to the same Linux guest as above, it would show up as `0.2.1111` under subchannel `0.2.0000`.

  The properties for the device would be `dev_id = "fe.2.1111"` and `subch_id = "fe.2.0000"`.

- a virtio-mouse device in a non-standard channel subsystem image:

```
-device virtio-mouse-ccw,devno=2.0.2222
```

  This would not show up in a standard Linux guest.

  The properties for the device would be `dev_id = "2.0.2222"` and `subch_id = "2.0.0000"`.

- a virtio-keyboard device in another non-standard channel subsystem image:

```
-device virtio-keyboard-ccw,devno=0.0.1234
```

  This would not show up in a standard Linux guest, either, as `0` is not the standard channel subsystem image id.

  The properties for the device would be `dev_id = "0.0.1234"` and `subch_id = "0.0.0000"`.

3270 devices

The 3270 is the classic 'green-screen' console of the mainframes (see the IBM 3270 Wikipedia article).

The 3270 data stream is not implemented within QEMU; the device only provides TN3270 (a telnet extension; see RFC 854 and RFC 1576) and leaves the heavy lifting to an external 3270 terminal emulator (such as `x3270`) to make a single 3270 device available to a guest. Note that this supports basic features only.

To provide a 3270 device to a guest, create a `x-terminal3270` linked to a `tn3270` chardev. The guest will see a 3270 channel device. In order to actually be able to use it, attach the `x3270` emulator to the chardev.

Example configuration

- Make sure that 3270 support is enabled in the guest's Linux kernel. You need `CONFIG_TN3270` and at least one of `CONFIG_TN3270_TTY` (for additional ttys) or `CONFIG_TN3270_CONSOLE` (for a 3270 console).
- Add a `tn3270` chardev and a `x-terminal3270` to the QEMU command line:

```
-chardev socket,id=ch0,host=0.0.0.0,port=2300,wait=off,server=on,tn3270=on
-device x-terminal3270,chardev=ch0,devno=fe.0.000a,id=terminal0
```

- Start the guest. In the guest, use `chccwdev -e 0.0.000a` to enable the device.
- On the host, start the `x3270` emulator:

```
x3270 <host>:2300
```

- In the guest, locate the 3270 device node under `/dev/3270/` (say, `tty1`) and start a getty on it:

```
systemctl start serial-getty@3270-tty1.service
```

  This should get you an additional tty for logging into the guest.

- If you want to use the 3270 device as the Linux kernel console instead of an additional tty, you can also append `conmode=3270 condev=000a` to the guest's kernel command line. The kernel then should use the 3270 as console after the next boot.

Restrictions

3270 support is very basic. In particular:

- Only one 3270 device is supported.
- It has only been tested with Linux guests and the x3270 emulator.
- TLS/SSL is not supported.
- Resizing on reattach is not supported.
- Multiple commands in one inbound buffer (for example, when the reset key is pressed while the network is slow) are not supported.

Subchannel passthrough via vfio-ccw

vfio-ccw (based upon the mediated vfio device infrastructure) allows to make certain I/O subchannels and their devices available to a guest. The host will not interact with those subchannels/devices any more.

Note that while vfio-ccw should work with most non-QDIO devices, only ECKD DASDs have really been tested.

Example configuration

Step 1: configure the host device

As every mdev is identified by a uuid, the first step is to obtain one:

```
[root@host ~]# uuidgen
7e270a25-e163-4922-af60-757fc8ed48c6
```

Note: it is recommended to use the `mdevctl` tool for actually configuring the host device.

To define the same device as configured below to be started automatically, use

```
[root@host ~]# driverctl -b css set-override 0.0.0313 vfio_ccw
[root@host ~]# mdevctl define -u 7e270a25-e163-4922-af60-757fc8ed48c6 \
               -p 0.0.0313 -t vfio_ccw-io -a
```

If using `mdevctl` is not possible or wanted, follow the manual procedure below.

- Locate the subchannel for the device (in this example, `0.0.2b09`):

  ```
  [root@host ~]# lscss | grep 0.0.2b09 | awk '{print $2}'
  0.0.0313
  ```

- Unbind the subchannel (in this example, `0.0.0313`) from the standard I/O subchannel driver and bind it to the vfio-ccw driver:

  ```
  [root@host ~]# echo 0.0.0313 > /sys/bus/css/devices/0.0.0313/driver/unbind
  [root@host ~]# echo 0.0.0313 > /sys/bus/css/drivers/vfio_ccw/bind
  ```

- Create the mediated device (identified by the uuid):

  ```
  [root@host ~]# echo "7e270a25-e163-4922-af60-757fc8ed48c6" > \
  /sys/bus/css/devices/0.0.0313/mdev_supported_types/vfio_ccw-io/create
  ```

Step 2: configure QEMU

- Reference the created mediated device and (optionally) pick a device id to be presented in the guest (here, `fe.0.1234`, which will end up visible in the guest as `0.0.1234`:

  ```
  -device vfio-ccw,devno=fe.0.1234,sysfsdev=\
  /sys/bus/mdev/devices/7e270a25-e163-4922-af60-757fc8ed48c6
  ```

- Start the guest. The device (here, `0.0.1234`) should now be usable:

  ```
  [root@guest ~]# lscss -d 0.0.1234
  Device   Subchan.  DevType CU Type Use  PIM PAM POM  CHPID
  ------------------------------------------------------------------------------
  0.0.1234 0.0.0007  3390/0e 3990/e9      f0  f0  ff   1a2a3a0a 00000000
  [root@guest ~]# chccwdev -e 0.0.1234
  Setting device 0.0.1234 online
  Done
  [root@guest ~]# dmesg -t
  ```

(continues on next page)

(continued from previous page)

```
(...)
dasd-eckd 0.0.1234: A channel path to the device has become operational
dasd-eckd 0.0.1234: New DASD 3390/0E (CU 3990/01) with 10017 cylinders, 15 heads,␣
↪224 sectors
dasd-eckd 0.0.1234: DASD with 4 KB/block, 7212240 KB total size, 48 KB/track,␣
↪compatible disk layout
dasda:VOL1/  0X2B09: dasda1
```

PCI devices on s390x

PCI devices on s390x work differently than on other architectures and need to be configured in a slightly different way.

Every PCI device is linked with an additional `zpci` device. While the `zpci` device will be autogenerated if not specified, it is recommended to specify it explicitly so that you can pass s390-specific PCI configuration.

For example, in order to pass a PCI device `0000:00:00.0` through to the guest, you would specify:

```
qemu-system-s390x ... \
                  -device zpci,uid=1,fid=0,target=hostdev0,id=zpci1 \
                  -device vfio-pci,host=0000:00:00.0,id=hostdev0
```

Here, the zpci device is joined with the PCI device via the `target` property.

Note that we don't set bus, slot or function here for the guest as is common in other PCI implementations. Topology information is not available on s390x, and the guest will not see any of the bus, slot or function information specified on the command line.

Instead, `uid` and `fid` determine how the device is presented to the guest operating system.

In case of Linux, `uid` will be used in the `domain` part of the PCI identifier, and `fid` identifies the physical slot, i.e.:

```
qemu-system-s390x ... \
                  -device zpci,uid=7,fid=8,target=hostdev0,id=zpci1 \
                  ...
```

will be presented in the guest as:

```
# lspci -v
0007:00:00.0 ...
Physical Slot: 00000008
...
```

Architectural features

Boot devices on s390x

Booting with bootindex parameter

For classical mainframe guests (i.e. LPAR or z/VM installations), you always have to explicitly specify the disk where you want to boot from (or "IPL" from, in s390x-speak – IPL means "Initial Program Load"). In particular, there can also be only one boot device according to the architecture specification, thus specifying multiple boot devices is not possible (yet).

So for booting an s390x guest in QEMU, you should always mark the device where you want to boot from with the `bootindex` property, for example:

```
qemu-system-s390x -drive if=none,id=dr1,file=guest.qcow2 \
                  -device virtio-blk,drive=dr1,bootindex=1
```

For booting from a CD-ROM ISO image (which needs to include El-Torito boot information in order to be bootable), it is recommended to specify a `scsi-cd` device, for example like this:

```
qemu-system-s390x -blockdev file,node-name=c1,filename=... \
                  -device virtio-scsi \
                  -device scsi-cd,drive=c1,bootindex=1
```

Note that you really have to use the `bootindex` property to select the boot device. The old-fashioned `-boot order=...` command of QEMU (and also `-boot once=...`) is not supported on s390x.

Booting without bootindex parameter

The QEMU guest firmware (the so-called s390-ccw bios) has also some rudimentary support for scanning through the available block devices. So in case you did not specify a boot device with the `bootindex` property, there is still a chance that it finds a bootable device on its own and starts a guest operating system from it. However, this scanning algorithm is still very rough and may be incomplete, so that it might fail to detect a bootable device in many cases. It is really recommended to always specify the boot device with the `bootindex` property instead.

This also means that you should avoid the classical short-cut commands like `-hda`, `-cdrom` or `-drive if=virtio`, since it is not possible to specify the `bootindex` with these commands. Note that the convenience `-cdrom` option even does not give you a real (virtio-scsi) CD-ROM device on s390x. Due to technical limitations in the QEMU code base, you will get a virtio-blk device with this parameter instead, which might not be the right device type for installing a Linux distribution via ISO image. It is recommended to specify a CD-ROM device via `-device scsi-cd` (as mentioned above) instead.

Selecting kernels with the `loadparm` property

The `s390-ccw-virtio` machine supports the so-called `loadparm` parameter which can be used to select the kernel on the disk of the guest that the s390-ccw bios should boot. When starting QEMU, it can be specified like this:

```
qemu-system-s390x -machine s390-ccw-virtio,loadparm=<string>
```

The first way to use this parameter is to use the word `PROMPT` as the `<string>` here. In that case the s390-ccw bios will show a list of installed kernels on the disk of the guest and ask the user to enter a number to chose which kernel should be booted – similar to what can be achieved by specifying the `-boot menu=on` option when starting QEMU. Note that the menu list will only show the names of the installed kernels when using a DASD-like disk image with 4k byte sectors. On normal SCSI-style disks with 512-byte sectors, there is not enough space for the zipl loader on the disk to store the kernel names, so you only get a list without names here.

The second way to use this parameter is to use a number in the range from 0 to 31. The numbers that can be used here correspond to the numbers that are shown when using the `PROMPT` option, and the s390-ccw bios will then try to automatically boot the kernel that is associated with the given number. Note that `0` can be used to boot the default entry.

Booting from a network device

Beside the normal guest firmware (which is loaded from the file `s390-ccw.img` in the data directory of QEMU, or via the `-bios` option), QEMU ships with a small TFTP network bootloader firmware for virtio-net-ccw devices, too. This firmware is loaded from a file called `s390-netboot.img` in the QEMU data directory. In case you want to load it from a different filename instead, you can specify it via the `-global s390-ipl.netboot_fw=filename` command line option.

The `bootindex` property is especially important for booting via the network. If you don't specify the `bootindex` property here, the network bootloader firmware code won't get loaded into the guest memory so that the network boot will fail. For a successful network boot, try something like this:

```
qemu-system-s390x -netdev user,id=n1,tftp=...,bootfile=... \
                  -device virtio-net-ccw,netdev=n1,bootindex=1
```

The network bootloader firmware also has basic support for pxelinux.cfg-style configuration files. See the PXELINUX Configuration page for details how to set up the configuration file on your TFTP server. The supported configuration file entries are `DEFAULT`, `LABEL`, `KERNEL`, `INITRD` and `APPEND` (see the Syslinux Config file syntax for more information).

Protected Virtualization on s390x

The memory and most of the registers of Protected Virtual Machines (PVMs) are encrypted or inaccessible to the hypervisor, effectively prohibiting VM introspection when the VM is running. At rest, PVMs are encrypted and can only be decrypted by the firmware, represented by an entity called Ultravisor, of specific IBM Z machines.

Prerequisites

To run PVMs, a machine with the Protected Virtualization feature, as indicated by the Ultravisor Call facility (stfle bit 158), is required. The Ultravisor needs to be initialized at boot by setting `prot_virt=1` on the host's kernel command line.

Running PVMs requires using the KVM hypervisor.

If those requirements are met, the capability `KVM_CAP_S390_PROTECTED` will indicate that KVM can support PVMs on that LPAR.

Running a Protected Virtual Machine

To run a PVM you will need to select a CPU model which includes the `Unpack facility` (stfle bit 161 represented by the feature `unpack/S390_FEAT_UNPACK`), and add these options to the command line:

```
-object s390-pv-guest,id=pv0 \
-machine confidential-guest-support=pv0
```

Adding these options will:

- Ensure the `unpack` facility is available
- Enable the IOMMU by default for all I/O devices
- Initialize the PV mechanism

Passthrough (vfio) devices are currently not supported.

Host huge page backings are not supported. However guests can use huge pages as indicated by its facilities.

Boot Process

A secure guest image can either be loaded from disk or supplied on the QEMU command line. Booting from disk is done by the unmodified s390-ccw BIOS. I.e., the bootmap is interpreted, multiple components are read into memory and control is transferred to one of the components (zipl stage3). Stage3 does some fixups and then transfers control to some program residing in guest memory, which is normally the OS kernel. The secure image has another component prepended (stage3a) that uses the new diag308 subcodes 8 and 10 to trigger the transition into secure mode.

Booting from the image supplied on the QEMU command line requires that the file passed via -kernel has the same memory layout as would result from the disk boot. This memory layout includes the encrypted components (kernel, initrd, cmdline), the stage3a loader and metadata. In case this boot method is used, the command line options -initrd and -cmdline are ineffective. The preparation of a PVM image is done via the `genprotimg` tool from the s390-tools collection.

CPU topology on s390x

Since QEMU 8.2, CPU topology on s390x provides up to 3 levels of topology containers: drawers, books and sockets. They define a tree-shaped hierarchy.

The socket container has one or more CPU entries. Each of these CPU entries consists of a bitmap and three CPU attributes:

- CPU type
- entitlement
- dedication

Each bit set in the bitmap correspond to a core-id of a vCPU with matching attributes.

This documentation provides general information on S390 CPU topology, how to enable it and explains the new CPU attributes. For information on how to modify the S390 CPU topology and how to monitor polarization changes, see `docs/devel/s390-cpu-topology.rst`.

Prerequisites

To use the CPU topology, you currently need to choose the KVM accelerator. See *Virtualisation Accelerators* for more details about accelerators and how to select them.

The s390x host needs to use a Linux kernel v6.0 or newer (which provides the so-called `KVM_CAP_S390_CPU_TOPOLOGY` capability that allows QEMU to signal the CPU topology facility via the so-called STFLE bit 11 to the VM).

Enabling CPU topology

Currently, CPU topology is enabled by default only in the "host" CPU model.

Enabling CPU topology in another CPU model is done by setting the CPU flag `ctop` to `on` as in:

```
-cpu gen16b,ctop=on
```

Having the topology disabled by default allows migration between old and new QEMU without adding new flags.

Default topology usage

The CPU topology can be specified on the QEMU command line with the `-smp` or the `-device` QEMU command arguments.

Note also that since 7.2 threads are no longer supported in the topology and the `-smp` command line argument accepts only `threads=1`.

If none of the containers attributes (drawers, books, sockets) are specified for the `-smp` flag, the number of these containers is 1.

Thus the following two options will result in the same topology:

```
-smp cpus=5,drawer=1,books=1,sockets=8,cores=4,maxcpus=32
```

and

```
-smp cpus=5,sockets=8,cores=4,maxcpus=32
```

When a CPU is defined by the `-smp` command argument, its position inside the topology is calculated by adding the CPUs to the topology based on the core-id starting with core-0 at position 0 of socket-0, book-0, drawer-0 and filling all CPUs of socket-0 before filling socket-1 of book-0 and so on up to the last socket of the last book of the last drawer.

When a CPU is defined by the `-device` command argument, the tree topology attributes must all be defined or all not defined.

```
-device gen16b-s390x-cpu,drawer-id=1,book-id=1,socket-id=2,core-id=1
```

or

```
-device gen16b-s390x-cpu,core-id=1,dedicated=true
```

If none of the tree attributes (drawer, book, sockets), are specified for the `-device` argument, like for all CPUs defined with the `-smp` command argument the topology tree attributes will be set by simply adding the CPUs to the topology based on the core-id.

QEMU will not try to resolve collisions and will report an error if the CPU topology defined explicitly or implicitly on a `-device` argument collides with the definition of a CPU implicitly defined on the `-smp` argument.

When the topology modifier attributes are not defined for the `-device` command argument they takes following default values:

- dedicated: `false`
- entitlement: `medium`

Hot plug

New CPUs can be plugged using the device_add hmp command as in:

```
(qemu) device_add gen16b-s390x-cpu,core-id=9
```

The placement of the CPU is derived from the core-id as described above.

The topology can of course also be fully defined:

```
(qemu) device_add gen16b-s390x-cpu,drawer-id=1,book-id=1,socket-id=2,core-id=1
```

Examples

In the following machine we define 8 sockets with 4 cores each.

```
$ qemu-system-s390x -accel kvm -m 2G \
  -cpu gen16b,ctop=on \
  -smp cpus=5,sockets=8,cores=4,maxcpus=32 \
  -device host-s390x-cpu,core-id=14 \
```

A new CPUs can be plugged using the device_add hmp command as before:

```
(qemu) device_add gen16b-s390x-cpu,core-id=9
```

The core-id defines the placement of the core in the topology by starting with core 0 in socket 0 up to maxcpus.

In the example above:

- There are 5 CPUs provided to the guest with the `-smp` command line They will take the core-ids 0,1,2,3,4 As we have 4 cores in a socket, we have 4 CPUs provided to the guest in socket 0, with core-ids 0,1,2,3. The last CPU, with core-id 4, will be on socket 1.
- the core with ID 14 provided by the `-device` command line will be placed in socket 3, with core-id 14
- the core with ID 9 provided by the `device_add` qmp command will be placed in socket 2, with core-id 9

Polarization, entitlement and dedication

Polarization

The polarization affects how the CPUs of a shared host are utilized/distributed among guests. The guest determines the polarization by using the PTF instruction.

Polarization defines two models of CPU provisioning: horizontal and vertical.

The horizontal polarization is the default model on boot and after subsystem reset. When horizontal polarization is in effect all vCPUs should have about equal resource provisioning.

In the vertical polarization model vCPUs are unequal, but overall more resources might be available. The guest can make use of the vCPU entitlement information provided by the host to optimize kernel thread scheduling.

A subsystem reset puts all vCPU of the configuration into the horizontal polarization.

Entitlement

The vertical polarization specifies that the guest's vCPU can get different real CPU provisioning:

- a vCPU with vertical high entitlement specifies that this vCPU gets 100% of the real CPU provisioning.
- a vCPU with vertical medium entitlement specifies that this vCPU shares the real CPU with other vCPUs.
- a vCPU with vertical low entitlement specifies that this vCPU only gets real CPU provisioning when no other vCPUs needs it.

In the case a vCPU with vertical high entitlement does not use the real CPU, the unused "slack" can be dispatched to other vCPU with medium or low entitlement.

A vCPU can be "dedicated" in which case the vCPU is fully dedicated to a single real CPU.

The dedicated bit is an indication of affinity of a vCPU for a real CPU while the entitlement indicates the sharing or exclusivity of use.

Defining the topology on the command line

The topology can entirely be defined using -device cpu statements, with the exception of CPU 0 which must be defined with the -smp argument.

For example, here we set the position of the cores 1,2,3 to drawer 1, book 1, socket 2 and cores 0,9 and 14 to drawer 0, book 0, socket 0 without defining entitlement or dedication. Core 4 will be set on its default position on socket 1 (since we have 4 core per socket) and we define it as dedicated and with vertical high entitlement.

```
$ qemu-system-s390x -accel kvm -m 2G \
  -cpu gen16b,ctop=on \
  -smp cpus=1,sockets=8,cores=4,maxcpus=32 \
  \
  -device gen16b-s390x-cpu,drawer-id=1,book-id=1,socket-id=2,core-id=1 \
  -device gen16b-s390x-cpu,drawer-id=1,book-id=1,socket-id=2,core-id=2 \
  -device gen16b-s390x-cpu,drawer-id=1,book-id=1,socket-id=2,core-id=3 \
  \
  -device gen16b-s390x-cpu,drawer-id=0,book-id=0,socket-id=0,core-id=9 \
  -device gen16b-s390x-cpu,drawer-id=0,book-id=0,socket-id=0,core-id=14 \
  \
  -device gen16b-s390x-cpu,core-id=4,dedicated=on,entitlement=high
```

The entitlement defined for the CPU 4 will only be used after the guest successfully enables vertical polarization by using the PTF instruction.

2.23.10 Sparc32 System emulator

Use the executable `qemu-system-sparc` to simulate the following Sun4m architecture machines:

- SPARCstation 4
- SPARCstation 5
- SPARCstation 10
- SPARCstation 20
- SPARCserver 600MP
- SPARCstation LX
- SPARCstation Voyager
- SPARCclassic
- SPARCbook

The emulation is somewhat complete. SMP up to 16 CPUs is supported, but Linux limits the number of usable CPUs to 4.

The list of available CPUs can be viewed by starting QEMU with `-cpu help`. Optional boolean features can be added with a "+" in front of the feature name, or disabled with a "-" in front of the name, for example `-cpu TI-SuperSparc-II,+float128`.

QEMU emulates the following sun4m peripherals:

- IOMMU
- TCX or cgthree Frame buffer
- Lance (Am7990) Ethernet
- Non Volatile RAM M48T02/M48T08
- Slave I/O: timers, interrupt controllers, Zilog serial ports, *Sparc32 keyboard* and power/reset logic
- ESP SCSI controller with hard disk and CD-ROM support
- Floppy drive (not on SS-600MP)
- CS4231 sound device (only on SS-5, not working yet)

The number of peripherals is fixed in the architecture. Maximum memory size depends on the machine type, for SS-5 it is 256MB and for others 2047MB.

Since version 0.8.2, QEMU uses OpenBIOS https://www.openbios.org/. OpenBIOS is a free (GPL v2) portable firmware implementation. The goal is to implement a 100% IEEE 1275-1994 (referred to as Open Firmware) compliant firmware.

Please note that currently older Solaris kernels don't work; this is probably due to interface issues between OpenBIOS and Solaris.

2.23.11 Sparc64 System emulator

Use the executable `qemu-system-sparc64` to simulate a Sun4u (UltraSPARC PC-like machine), Sun4v (T1 PC-like machine), or generic Niagara (T1) machine. The Sun4u emulator is mostly complete, being able to run Linux, NetBSD and OpenBSD in headless (-nographic) mode. The Sun4v emulator is still a work in progress.

The Niagara T1 emulator makes use of firmware and OS binaries supplied in the S10image/ directory of the OpenSPARC T1 project http://download.oracle.com/technetwork/systems/opensparc/OpenSPARCT1_Arch.1.5.tar.bz2 and is able to boot the disk.s10hw2 Solaris image.

```
qemu-system-sparc64 -M niagara -L /path-to/S10image/ \
                    -nographic -m 256 \
                    -drive if=pflash,readonly=on,file=/S10image/disk.s10hw2
```

QEMU emulates the following peripherals:

- UltraSparc IIi APB PCI Bridge
- PCI VGA compatible card with VESA Bochs Extensions
- PS/2 mouse and keyboard
- Non Volatile RAM M48T59
- PC-compatible serial ports
- 2 PCI IDE interfaces with hard disk and CD-ROM support
- Floppy disk

2.23.12 x86 System emulator

Board-specific documentation

'microvm' virtual platform (`microvm`)

`microvm` is a machine type inspired by `Firecracker` and constructed after its machine model.

It's a minimalist machine type without `PCI` nor `ACPI` support, designed for short-lived guests. microvm also establishes a baseline for benchmarking and optimizing both QEMU and guest operating systems, since it is optimized for both boot time and footprint.

Supported devices

The microvm machine type supports the following devices:

- ISA bus
- i8259 PIC (optional)
- i8254 PIT (optional)
- MC146818 RTC (optional)
- One ISA serial port (optional)
- LAPIC
- IOAPIC (with kernel-irqchip=split by default)
- kvmclock (if using KVM)
- fw_cfg
- Up to eight virtio-mmio devices (configured by the user)

Limitations

Currently, microvm does *not* support the following features:

- PCI-only devices.
- Hotplug of any kind.
- Live migration across QEMU versions.

Using the microvm machine type

Machine-specific options

It supports the following machine-specific options:

- microvm.x-option-roms=bool (Set off to disable loading option ROMs)
- microvm.pit=OnOffAuto (Enable i8254 PIT)
- microvm.isa-serial=bool (Set off to disable the instantiation an ISA serial port)
- microvm.pic=OnOffAuto (Enable i8259 PIC)

- microvm.rtc=OnOffAuto (Enable MC146818 RTC)
- microvm.auto-kernel-cmdline=bool (Set off to disable adding virtio-mmio devices to the kernel cmdline)

Boot options

By default, microvm uses `qboot` as its BIOS, to obtain better boot times, but it's also compatible with `SeaBIOS`.

As no current FW is able to boot from a block device using `virtio-mmio` as its transport, a microvm-based VM needs to be run using a host-side kernel and, optionally, an initrd image.

Running a microvm-based VM

By default, microvm aims for maximum compatibility, enabling both legacy and non-legacy devices. In this example, a VM is created without passing any additional machine-specific option, using the legacy `ISA serial` device as console:

```
$ qemu-system-x86_64 -M microvm \
   -enable-kvm -cpu host -m 512m -smp 2 \
   -kernel vmlinux -append "earlyprintk=ttyS0 console=ttyS0 root=/dev/vda" \
   -nodefaults -no-user-config -nographic \
   -serial stdio \
   -drive id=test,file=test.img,format=raw,if=none \
   -device virtio-blk-device,drive=test \
   -netdev tap,id=tap0,script=no,downscript=no \
   -device virtio-net-device,netdev=tap0
```

While the example above works, you might be interested in reducing the footprint further by disabling some legacy devices. If you're using `KVM`, you can disable the `RTC`, making the Guest rely on `kvmclock` exclusively. Additionally, if your host's CPUs have the `TSC_DEADLINE` feature, you can also disable both the i8259 PIC and the i8254 PIT (make sure you're also emulating a CPU with such feature in the guest).

This is an example of a VM with all optional legacy features disabled:

```
$ qemu-system-x86_64 \
   -M microvm,x-option-roms=off,pit=off,pic=off,isa-serial=off,rtc=off \
   -enable-kvm -cpu host -m 512m -smp 2 \
   -kernel vmlinux -append "console=hvc0 root=/dev/vda" \
   -nodefaults -no-user-config -nographic \
   -chardev stdio,id=virtiocon0 \
   -device virtio-serial-device \
   -device virtconsole,chardev=virtiocon0 \
   -drive id=test,file=test.img,format=raw,if=none \
   -device virtio-blk-device,drive=test \
   -netdev tap,id=tap0,script=no,downscript=no \
   -device virtio-net-device,netdev=tap0
```

Triggering a guest-initiated shut down

As the microvm machine type includes just a small set of system devices, some x86 mechanisms for rebooting or shutting down the system, like sending a key sequence to the keyboard or writing to an ACPI register, doesn't have any effect in the VM.

The recommended way to trigger a guest-initiated shut down is by generating a `triple-fault`, which will cause the VM to initiate a reboot. Additionally, if the `-no-reboot` argument is present in the command line, QEMU will detect this event and terminate its own execution gracefully.

Linux does support this mechanism, but by default will only be used after other options have been tried and failed, causing the reboot to be delayed by a small number of seconds. It's possible to instruct it to try the triple-fault mechanism first, by adding `reboot=t` to the kernel's command line.

i440fx PC (`pc-i440fx`, `pc`)

Peripherals

The QEMU PC System emulator simulates the following peripherals:

- i440FX host PCI bridge and PIIX3 PCI to ISA bridge
- Cirrus CLGD 5446 PCI VGA card or dummy VGA card with Bochs VESA extensions (hardware level, including all non standard modes).
- PS/2 mouse and keyboard
- 2 PCI IDE interfaces with hard disk and CD-ROM support
- Floppy disk
- PCI and ISA network adapters
- Serial ports
- IPMI BMC, either and internal or external one
- Creative SoundBlaster 16 sound card
- ENSONIQ AudioPCI ES1370 sound card
- Intel 82801AA AC97 Audio compatible sound card
- Intel HD Audio Controller and HDA codec
- Adlib (OPL2) - Yamaha YM3812 compatible chip
- Gravis Ultrasound GF1 sound card
- CS4231A compatible sound card
- PC speaker
- PCI UHCI, OHCI, EHCI or XHCI USB controller and a virtual USB-1.1 hub.

SMP is supported with up to 255 CPUs (and 4096 CPUs for PC Q35 machine).

QEMU uses the PC BIOS from the Seabios project and the Plex86/Bochs LGPL VGA BIOS.

QEMU uses YM3812 emulation by Tatsuyuki Satoh.

QEMU uses GUS emulation (GUSEMU32 http://www.deinmeister.de/gusemu/) by Tibor "TS" Schütz.

Note that, by default, GUS shares IRQ(7) with parallel ports and so QEMU must be told to not have parallel ports to have working GUS.

```
qemu-system-x86_64 dos.img -device gus -parallel none
```

Alternatively:

```
qemu-system-x86_64 dos.img -device gus,irq=5
```

Or some other unclaimed IRQ.

CS4231A is the chip used in Windows Sound System and GUSMAX products

The PC speaker audio device can be configured using the pcspk-audiodev machine property, i.e.

```
qemu-system-x86_64 some.img -audiodev <backend>,id=<name> -machine pcspk-audiodev=<name>
```

Machine-specific options

It supports the following machine-specific options:

- `x-south-bridge=PIIX3|piix4-isa` (Experimental option to select a particular south bridge. Default: `PIIX3`)

Architectural features

Recommendations for KVM CPU model configuration on x86 hosts

The information that follows provides recommendations for configuring CPU models on x86 hosts. The goals are to maximise performance, while protecting guest OS against various CPU hardware flaws, and optionally enabling live migration between hosts with heterogeneous CPU models.

Two ways to configure CPU models with QEMU / KVM

(1) **Host passthrough**

This passes the host CPU model features, model, stepping, exactly to the guest. Note that KVM may filter out some host CPU model features if they cannot be supported with virtualization. Live migration is unsafe when this mode is used as libvirt / QEMU cannot guarantee a stable CPU is exposed to the guest across hosts. This is the recommended CPU to use, provided live migration is not required.

(2) **Named model**

QEMU comes with a number of predefined named CPU models, that typically refer to specific generations of hardware released by Intel and AMD. These allow the guest VMs to have a degree of isolation from the host CPU, allowing greater flexibility in live migrating between hosts with differing hardware. @end table

In both cases, it is possible to optionally add or remove individual CPU features, to alter what is presented to the guest by default.

Libvirt supports a third way to configure CPU models known as “Host model”. This uses the QEMU “Named model” feature, automatically picking a CPU model that is similar the host CPU, and then adding extra features to approximate the host model as closely as possible. This does not guarantee the CPU family, stepping, etc will precisely match the host CPU, as they would with “Host passthrough”, but gives much of the benefit of passthrough, while making live migration safe.

ABI compatibility levels for CPU models

The x86_64 architecture has a number of ABI compatibility levels defined. Traditionally most operating systems and toolchains would only target the original baseline ABI. It is expected that in future OS and toolchains are likely to target newer ABIs. The table that follows illustrates which ABI compatibility levels can be satisfied by the QEMU CPU models. Note that the table only lists the long term stable CPU model versions (eg Haswell-v4). In addition to what is listed, there are also many CPU model aliases which resolve to a different CPU model version, depending on the machine type is in use.

Table 5: x86-64 ABI compatibility levels

Model	baseline	v2	v3	v4
486-v1				
Broadwell-v1				
Broadwell-v2				
Broadwell-v3				
Broadwell-v4				
Cascadelake-Server-v1				
Cascadelake-Server-v2				
Cascadelake-Server-v3				
Cascadelake-Server-v4				
Cascadelake-Server-v5				
Conroe-v1				
Cooperlake-v1				
Cooperlake-v2				
Denverton-v1				
Denverton-v2				
Denverton-v3				
Dhyana-v1				
Dhyana-v2				
EPYC-Genoa-v1				
EPYC-Milan-v1				
EPYC-Milan-v2				
EPYC-Rome-v1				
EPYC-Rome-v2				
EPYC-Rome-v3				
EPYC-Rome-v4				
EPYC-v1				
EPYC-v2				
EPYC-v3				
EPYC-v4				
GraniteRapids-v1				
Haswell-v1				
Haswell-v2				
Haswell-v3				
Haswell-v4				
Icelake-Server-v1				
Icelake-Server-v2				
Icelake-Server-v3				
Icelake-Server-v4				
Icelake-Server-v5				
Icelake-Server-v6				

continues on next page

Table 5 – continued from previous page

Model	baseline	v2	v3	v4
IvyBridge-v1				
IvyBridge-v2				
KnightsMill-v1				
Nehalem-v1				
Nehalem-v2				
Opteron_G1-v1				
Opteron_G2-v1				
Opteron_G3-v1				
Opteron_G4-v1				
Opteron_G5-v1				
Penryn-v1				
SandyBridge-v1				
SandyBridge-v2				
SapphireRapids-v1				
SapphireRapids-v2				
Skylake-Client-v1				
Skylake-Client-v2				
Skylake-Client-v3				
Skylake-Client-v4				
Skylake-Server-v1				
Skylake-Server-v2				
Skylake-Server-v3				
Skylake-Server-v4				
Skylake-Server-v5				
Snowridge-v1				
Snowridge-v2				
Snowridge-v3				
Snowridge-v4				
Westmere-v1				
Westmere-v2				
athlon-v1				
core2duo-v1				
coreduo-v1				
kvm32-v1				
kvm64-v1				
n270-v1				
pentium-v1				
pentium2-v1				
pentium3-v1				
phenom-v1				
qemu32-v1				
qemu64-v1				

Preferred CPU models for Intel x86 hosts

The following CPU models are preferred for use on Intel hosts. Administrators / applications are recommended to use the CPU model that matches the generation of the host CPUs in use. In a deployment with a mixture of host CPU models between machines, if live migration compatibility is required, use the newest CPU model that is compatible across all desired hosts.

Cascadelake-Server, Cascadelake-Server-noTSX
Intel Xeon Processor (Cascade Lake, 2019), with "stepping" levels 6 or 7 only. (The Cascade Lake Xeon processor with *stepping 5 is vulnerable to MDS variants*.)

Skylake-Server, Skylake-Server-IBRS, Skylake-Server-IBRS-noTSX
Intel Xeon Processor (Skylake, 2016)

Skylake-Client, Skylake-Client-IBRS, Skylake-Client-noTSX-IBRS}
Intel Core Processor (Skylake, 2015)

Broadwell, Broadwell-IBRS, Broadwell-noTSX, Broadwell-noTSX-IBRS
Intel Core Processor (Broadwell, 2014)

Haswell, Haswell-IBRS, Haswell-noTSX, Haswell-noTSX-IBRS
Intel Core Processor (Haswell, 2013)

IvyBridge, IvyBridge-IBR
Intel Xeon E3-12xx v2 (Ivy Bridge, 2012)

SandyBridge, SandyBridge-IBRS
Intel Xeon E312xx (Sandy Bridge, 2011)

Westmere, Westmere-IBRS
Westmere E56xx/L56xx/X56xx (Nehalem-C, 2010)

Nehalem, Nehalem-IBRS
Intel Core i7 9xx (Nehalem Class Core i7, 2008)

Penryn
Intel Core 2 Duo P9xxx (Penryn Class Core 2, 2007)

Conroe
Intel Celeron_4x0 (Conroe/Merom Class Core 2, 2006)

Important CPU features for Intel x86 hosts

The following are important CPU features that should be used on Intel x86 hosts, when available in the host CPU. Some of them require explicit configuration to enable, as they are not included by default in some, or all, of the named CPU models listed above. In general all of these features are included if using "Host passthrough" or "Host model".

pcid
Recommended to mitigate the cost of the Meltdown (CVE-2017-5754) fix.

Included by default in Haswell, Broadwell & Skylake Intel CPU models.

Should be explicitly turned on for Westmere, SandyBridge, and IvyBridge Intel CPU models. Note that some desktop/mobile Westmere CPUs cannot support this feature.

spec-ctrl
Required to enable the Spectre v2 (CVE-2017-5715) fix.

Included by default in Intel CPU models with -IBRS suffix.

Must be explicitly turned on for Intel CPU models without -IBRS suffix.

Requires the host CPU microcode to support this feature before it can be used for guest CPUs.

stibp
Required to enable stronger Spectre v2 (CVE-2017-5715) fixes in some operating systems.

Must be explicitly turned on for all Intel CPU models.

Requires the host CPU microcode to support this feature before it can be used for guest CPUs.

ssbd
Required to enable the CVE-2018-3639 fix.

Not included by default in any Intel CPU model.

Must be explicitly turned on for all Intel CPU models.

Requires the host CPU microcode to support this feature before it can be used for guest CPUs.

pdpe1gb
Recommended to allow guest OS to use 1GB size pages.

Not included by default in any Intel CPU model.

Should be explicitly turned on for all Intel CPU models.

Note that not all CPU hardware will support this feature.

md-clear
Required to confirm the MDS (CVE-2018-12126, CVE-2018-12127, CVE-2018-12130, CVE-2019-11091) fixes.

Not included by default in any Intel CPU model.

Must be explicitly turned on for all Intel CPU models.

Requires the host CPU microcode to support this feature before it can be used for guest CPUs.

mds-no
Recommended to inform the guest OS that the host is *not* vulnerable to any of the MDS variants ([MFBDS] CVE-2018-12130, [MLPDS] CVE-2018-12127, [MSBDS] CVE-2018-12126).

This is an MSR (Model-Specific Register) feature rather than a CPUID feature, so it will not appear in the Linux `/proc/cpuinfo` in the host or guest. Instead, the host kernel uses it to populate the MDS vulnerability file in `sysfs`.

So it should only be enabled for VMs if the host reports @code{Not affected} in the `/sys/devices/system/cpu/vulnerabilities/mds` file.

taa-no
Recommended to inform that the guest that the host is `not` vulnerable to CVE-2019-11135, TSX Asynchronous Abort (TAA).

This too is an MSR feature, so it does not show up in the Linux `/proc/cpuinfo` in the host or guest.

It should only be enabled for VMs if the host reports `Not affected` in the `/sys/devices/system/cpu/vulnerabilities/tsx_async_abort` file.

tsx-ctrl
Recommended to inform the guest that it can disable the Intel TSX (Transactional Synchronization Extensions) feature; or, if the processor is vulnerable, use the Intel VERW instruction (a processor-level instruction that performs checks on memory access) as a mitigation for the TAA vulnerability. (For details, refer to Intel's deep dive into MDS.)

Expose this to the guest OS if and only if: (a) the host has TSX enabled; *and* (b) the guest has `rtm` CPU flag enabled.

By disabling TSX, KVM-based guests can avoid paying the price of mitigating TSX-based attacks.

Note that `tsx-ctrl` too is an MSR feature, so it does not show up in the Linux `/proc/cpuinfo` in the host or guest.

To validate that Intel TSX is indeed disabled for the guest, there are two ways: (a) check for the *absence* of `rtm` in the guest's `/proc/cpuinfo`; or (b) the `/sys/devices/system/cpu/vulnerabilities/tsx_async_abort` file in the guest should report `Mitigation: TSX disabled`.

Preferred CPU models for AMD x86 hosts

The following CPU models are preferred for use on AMD hosts. Administrators / applications are recommended to use the CPU model that matches the generation of the host CPUs in use. In a deployment with a mixture of host CPU models between machines, if live migration compatibility is required, use the newest CPU model that is compatible across all desired hosts.

EPYC, EPYC-IBPB
AMD EPYC Processor (2017)

Opteron_G5
AMD Opteron 63xx class CPU (2012)

Opteron_G4
AMD Opteron 62xx class CPU (2011)

Opteron_G3
AMD Opteron 23xx (Gen 3 Class Opteron, 2009)

Opteron_G2
AMD Opteron 22xx (Gen 2 Class Opteron, 2006)

Opteron_G1
AMD Opteron 240 (Gen 1 Class Opteron, 2004)

Important CPU features for AMD x86 hosts

The following are important CPU features that should be used on AMD x86 hosts, when available in the host CPU. Some of them require explicit configuration to enable, as they are not included by default in some, or all, of the named CPU models listed above. In general all of these features are included if using "Host passthrough" or "Host model".

ibpb
Required to enable the Spectre v2 (CVE-2017-5715) fix.

Included by default in AMD CPU models with -IBPB suffix.

Must be explicitly turned on for AMD CPU models without -IBPB suffix.

Requires the host CPU microcode to support this feature before it can be used for guest CPUs.

stibp
Required to enable stronger Spectre v2 (CVE-2017-5715) fixes in some operating systems.

Must be explicitly turned on for all AMD CPU models.

Requires the host CPU microcode to support this feature before it can be used for guest CPUs.

virt-ssbd
Required to enable the CVE-2018-3639 fix

Not included by default in any AMD CPU model.

Must be explicitly turned on for all AMD CPU models.

This should be provided to guests, even if amd-ssbd is also provided, for maximum guest compatibility.

Note for some QEMU / libvirt versions, this must be force enabled when when using "Host model", because this is a virtual feature that doesn't exist in the physical host CPUs.

amd-ssbd
Required to enable the CVE-2018-3639 fix

Not included by default in any AMD CPU model.

Must be explicitly turned on for all AMD CPU models.

This provides higher performance than `virt-ssbd` so should be exposed to guests whenever available in the host. `virt-ssbd` should none the less also be exposed for maximum guest compatibility as some kernels only know about `virt-ssbd`.

amd-no-ssb
Recommended to indicate the host is not vulnerable CVE-2018-3639

Not included by default in any AMD CPU model.

Future hardware generations of CPU will not be vulnerable to CVE-2018-3639, and thus the guest should be told not to enable its mitigations, by exposing amd-no-ssb. This is mutually exclusive with virt-ssbd and amd-ssbd.

pdpe1gb
Recommended to allow guest OS to use 1GB size pages

Not included by default in any AMD CPU model.

Should be explicitly turned on for all AMD CPU models.

Note that not all CPU hardware will support this feature.

Default x86 CPU models

The default QEMU CPU models are designed such that they can run on all hosts. If an application does not wish to do perform any host compatibility checks before launching guests, the default is guaranteed to work.

The default CPU models will, however, leave the guest OS vulnerable to various CPU hardware flaws, so their use is strongly discouraged. Applications should follow the earlier guidance to setup a better CPU configuration, with host passthrough recommended if live migration is not needed.

qemu32, qemu64
QEMU Virtual CPU version 2.5+ (32 & 64 bit variants)

`qemu64` is used for x86_64 guests and `qemu32` is used for i686 guests, when no `-cpu` argument is given to QEMU, or no <cpu> is provided in libvirt XML.

Other non-recommended x86 CPUs

The following CPUs models are compatible with most AMD and Intel x86 hosts, but their usage is discouraged, as they expose a very limited featureset, which prevents guests having optimal performance.

kvm32, kvm64
Common KVM processor (32 & 64 bit variants).

Legacy models just for historical compatibility with ancient QEMU versions.

486, athlon, phenom, coreduo, core2duo, n270, pentium, pentium2, pentium3
Various very old x86 CPU models, mostly predating the introduction of hardware assisted virtualization, that should thus not be required for running virtual machines.

Syntax for configuring CPU models

The examples below illustrate the approach to configuring the various CPU models / features in QEMU and libvirt.

QEMU command line

Host passthrough:

```
qemu-system-x86_64 -cpu host
```

Host passthrough with feature customization:

```
qemu-system-x86_64 -cpu host,vmx=off,...
```

Named CPU models:

```
qemu-system-x86_64 -cpu Westmere
```

Named CPU models with feature customization:

```
qemu-system-x86_64 -cpu Westmere,pcid=on,...
```

Libvirt guest XML

Host passthrough:

```
<cpu mode='host-passthrough'/>
```

Host passthrough with feature customization:

```
<cpu mode='host-passthrough'>
    <feature name="vmx" policy="disable"/>
    ...
</cpu>
```

Host model:

```
<cpu mode='host-model'/>
```

Host model with feature customization:

```
<cpu mode='host-model'>
    <feature name="vmx" policy="disable"/>
    ...
</cpu>
```

Named model:

```
<cpu mode='custom'>
    <model name="Westmere"/>
</cpu>
```

Named model with feature customization:

```
<cpu mode='custom'>
    <model name="Westmere"/>
    <feature name="pcid" policy="require"/>
    ...
</cpu>
```

Hyper-V Enlightenments

Description

In some cases when implementing a hardware interface in software is slow, KVM implements its own paravirtualized interfaces. This works well for Linux as guest support for such features is added simultaneously with the feature itself. It may, however, be hard-to-impossible to add support for these interfaces to proprietary OSes, namely, Microsoft Windows.

KVM on x86 implements Hyper-V Enlightenments for Windows guests. These features make Windows and Hyper-V guests think they're running on top of a Hyper-V compatible hypervisor and use Hyper-V specific features.

Setup

No Hyper-V enlightenments are enabled by default by either KVM or QEMU. In QEMU, individual enlightenments can be enabled through CPU flags, e.g:

`qemu-system-x86_64 --enable-kvm --cpu host,hv_relaxed,hv_vpindex,hv_time, ...`

Sometimes there are dependencies between enlightenments, QEMU is supposed to check that the supplied configuration is sane.

When any set of the Hyper-V enlightenments is enabled, QEMU changes hypervisor identification (CPUID 0x40000000..0x4000000A) to Hyper-V. KVM identification and features are kept in leaves 0x40000100..0x40000101.

Existing enlightenments

hv-relaxed
: This feature tells guest OS to disable watchdog timeouts as it is running on a hypervisor. It is known that some Windows versions will do this even when they see 'hypervisor' CPU flag.

hv-vapic
: Provides so-called VP Assist page MSR to guest allowing it to work with APIC more efficiently. In particular, this enlightenment allows paravirtualized (exit-less) EOI processing.

hv-spinlocks = xxx
: Enables paravirtualized spinlocks. The parameter indicates how many times spinlock acquisition should be attempted before indicating the situation to the hypervisor. A special value 0xffffffff indicates "never notify".

hv-vpindex
: Provides HV_X64_MSR_VP_INDEX (0x40000002) MSR to the guest which has Virtual processor index information. This enlightenment makes sense in conjunction with hv-synic, hv-stimer and other enlightenments which require the guest to know its Virtual Processor indices (e.g. when VP index needs to be passed in a hypercall).

hv-runtime
: Provides HV_X64_MSR_VP_RUNTIME (0x40000010) MSR to the guest. The MSR keeps the virtual processor

run time in 100ns units. This gives guest operating system an idea of how much time was 'stolen' from it (when the virtual CPU was preempted to perform some other work).

hv-crash
Provides HV_X64_MSR_CRASH_P0..HV_X64_MSR_CRASH_P5 (0x40000100..0x40000105) and HV_X64_MSR_CRASH_CTL (0x40000105) MSRs to the guest. These MSRs are written to by the guest when it crashes, HV_X64_MSR_CRASH_P0..HV_X64_MSR_CRASH_P5 MSRs contain additional crash information. This information is outputted in QEMU log and through QAPI. Note: unlike under genuine Hyper-V, write to HV_X64_MSR_CRASH_CTL causes guest to shutdown. This effectively blocks crash dump generation by Windows.

hv-time
Enables two Hyper-V-specific clocksources available to the guest: MSR-based Hyper-V clocksource (HV_X64_MSR_TIME_REF_COUNT, 0x40000020) and Reference TSC page (enabled via MSR HV_X64_MSR_REFERENCE_TSC, 0x40000021). Both clocksources are per-guest, Reference TSC page clocksource allows for exit-less time stamp readings. Using this enlightenment leads to significant speedup of all timestamp related operations.

hv-synic
Enables Hyper-V Synthetic interrupt controller - an extension of a local APIC. When enabled, this enlightenment provides additional communication facilities to the guest: SynIC messages and Events. This is a pre-requisite for implementing VMBus devices (not yet in QEMU). Additionally, this enlightenment is needed to enable Hyper-V synthetic timers. SynIC is controlled through MSRs HV_X64_MSR_SCONTROL..HV_X64_MSR_EOM (0x40000080..0x40000084) and HV_X64_MSR_SINT0..HV_X64_MSR_SINT15 (0x40000090..0x4000009F)

Requires: hv-vpindex

hv-stimer
Enables Hyper-V synthetic timers. There are four synthetic timers per virtual CPU controlled through HV_X64_MSR_STIMER0_CONFIG..HV_X64_MSR_STIMER3_COUNT (0x400000B0..0x400000B7) MSRs. These timers can work either in single-shot or periodic mode. It is known that certain Windows versions revert to using HPET (or even RTC when HPET is unavailable) extensively when this enlightenment is not provided; this can lead to significant CPU consumption, even when virtual CPU is idle.

Requires: hv-vpindex, hv-synic, hv-time

hv-tlbflush
Enables paravirtualized TLB shoot-down mechanism. On x86 architecture, remote TLB flush procedure requires sending IPIs and waiting for other CPUs to perform local TLB flush. In virtualized environment some virtual CPUs may not even be scheduled at the time of the call and may not require flushing (or, flushing may be postponed until the virtual CPU is scheduled). hv-tlbflush enlightenment implements TLB shoot-down through hypervisor enabling the optimization.

Requires: hv-vpindex

hv-ipi
Enables paravirtualized IPI send mechanism. HvCallSendSyntheticClusterIpi hypercall may target more than 64 virtual CPUs simultaneously, doing the same through APIC requires more than one access (and thus exit to the hypervisor).

Requires: hv-vpindex

hv-vendor-id = xxx
This changes Hyper-V identification in CPUID 0x40000000.EBX-EDX from the default "Microsoft Hv". The parameter should be no longer than 12 characters. According to the specification, guests shouldn't use this information and it is unknown if there is a Windows version which acts differently. Note: hv-vendor-id is not an enlightenment and thus doesn't enable Hyper-V identification when specified without some other enlightenment.

hv-reset
Provides HV_X64_MSR_RESET (0x40000003) MSR to the guest allowing it to reset itself by writing to it. Even

when this MSR is enabled, it is not a recommended way for Windows to perform system reboot and thus it may not be used.

hv-frequencies
Provides HV_X64_MSR_TSC_FREQUENCY (0x40000022) and HV_X64_MSR_APIC_FREQUENCY (0x40000023) allowing the guest to get its TSC/APIC frequencies without doing measurements.

hv-reenlightenment
The enlightenment is nested specific, it targets Hyper-V on KVM guests. When enabled, it provides HV_X64_MSR_REENLIGHTENMENT_CONTROL (0x40000106), HV_X64_MSR_TSC_EMULATION_CONTROL (0x40000107)and HV_X64_MSR_TSC_EMULATION_STATUS (0x40000108) MSRs allowing the guest to get notified when TSC frequency changes (only happens on migration) and keep using old frequency (through emulation in the hypervisor) until it is ready to switch to the new one. This, in conjunction with `hv-frequencies`, allows Hyper-V on KVM to pass stable clocksource (Reference TSC page) to its own guests.

Note, KVM doesn't fully support re-enlightenment notifications and doesn't emulate TSC accesses after migration so 'tsc-frequency=' CPU option also has to be specified to make migration succeed. The destination host has to either have the same TSC frequency or support TSC scaling CPU feature.

Recommended: `hv-frequencies`

hv-evmcs
The enlightenment is nested specific, it targets Hyper-V on KVM guests. When enabled, it provides Enlightened VMCS version 1 feature to the guest. The feature implements paravirtualized protocol between L0 (KVM) and L1 (Hyper-V) hypervisors making L2 exits to the hypervisor faster. The feature is Intel-only.

Note: some virtualization features (e.g. Posted Interrupts) are disabled when hv-evmcs is enabled. It may make sense to measure your nested workload with and without the feature to find out if enabling it is beneficial.

Requires: `hv-vapic`

hv-stimer-direct
Hyper-V specification allows synthetic timer operation in two modes: "classic", when expiration event is delivered as SynIC message and "direct", when the event is delivered via normal interrupt. It is known that nested Hyper-V can only use synthetic timers in direct mode and thus `hv-stimer-direct` needs to be enabled.

Requires: `hv-vpindex`, `hv-synic`, `hv-time`, `hv-stimer`

hv-avic (hv-apicv)
The enlightenment allows to use Hyper-V SynIC with hardware APICv/AVIC enabled. Normally, Hyper-V SynIC disables these hardware feature and suggests the guest to use paravirtualized AutoEOI feature. Note: enabling this feature on old hardware (without APICv/AVIC support) may have negative effect on guest's performance.

hv-no-nonarch-coresharing = on/off/auto
This enlightenment tells guest OS that virtual processors will never share a physical core unless they are reported as sibling SMT threads. This information is required by Windows and Hyper-V guests to properly mitigate SMT related CPU vulnerabilities.

When the option is set to 'auto' QEMU will enable the feature only when KVM reports that non-architectural coresharing is impossible, this means that hyper-threading is not supported or completely disabled on the host. This setting also prevents migration as SMT settings on the destination may differ. When the option is set to 'on' QEMU will always enable the feature, regardless of host setup. To keep guests secure, this can only be used in conjunction with exposing correct vCPU topology and vCPU pinning.

hv-version-id-build, hv-version-id-major, hv-version-id-minor, hv-version-id-spack, hv-version-id-sbranch, hv-version-id-snumber
This changes Hyper-V version identification in CPUID 0x40000002.EAX-EDX from the default (WS2016).

- `hv-version-id-build` sets 'Build Number' (32 bits)

- hv-version-id-major sets 'Major Version' (16 bits)
- hv-version-id-minor sets 'Minor Version' (16 bits)
- hv-version-id-spack sets 'Service Pack' (32 bits)
- hv-version-id-sbranch sets 'Service Branch' (8 bits)
- hv-version-id-snumber sets 'Service Number' (24 bits)

Note: hv-version-id-* are not enlightenments and thus don't enable Hyper-V identification when specified without any other enlightenments.

hv-syndbg
Enables Hyper-V synthetic debugger interface, this is a special interface used by Windows Kernel debugger to send the packets through, rather than sending them via serial/network . When enabled, this enlightenment provides additional communication facilities to the guest: SynDbg messages. This new communication is used by Windows Kernel debugger rather than sending packets via serial/network, adding significant performance boost over the other comm channels. This enlightenment requires a VMBus device (-device vmbus-bridge,irq=15).

Requires: hv-relaxed, hv_time, hv-vapic, hv-vpindex, hv-synic, hv-runtime, hv-stimer

hv-emsr-bitmap
The enlightenment is nested specific, it targets Hyper-V on KVM guests. When enabled, it allows L0 (KVM) and L1 (Hyper-V) hypervisors to collaborate to avoid unnecessary updates to L2 MSR-Bitmap upon vmexits. While the protocol is supported for both VMX (Intel) and SVM (AMD), the VMX implementation requires Enlightened VMCS (hv-evmcs) feature to also be enabled.

Recommended: hv-evmcs (Intel)

hv-xmm-input
Hyper-V specification allows to pass parameters for certain hypercalls using XMM registers ("XMM Fast Hypercall Input"). When the feature is in use, it allows for faster hypercalls processing as KVM can avoid reading guest's memory.

hv-tlbflush-ext
Allow for extended GVA ranges to be passed to Hyper-V TLB flush hypercalls (HvFlushVirtualAddressList/HvFlushVirtualAddressListEx).

Requires: hv-tlbflush

hv-tlbflush-direct
The enlightenment is nested specific, it targets Hyper-V on KVM guests. When enabled, it allows L0 (KVM) to directly handle TLB flush hypercalls from L2 guest without the need to exit to L1 (Hyper-V) hypervisor. While the feature is supported for both VMX (Intel) and SVM (AMD), the VMX implementation requires Enlightened VMCS (hv-evmcs) feature to also be enabled.

Requires: hv-vapic

Recommended: hv-evmcs (Intel)

Supplementary features

hv-passthrough
: In some cases (e.g. during development) it may make sense to use QEMU in 'pass-through' mode and give Windows guests all enlightenments currently supported by KVM. This pass-through mode is enabled by "hv-passthrough" CPU flag.

  Note: `hv-passthrough` flag only enables enlightenments which are known to QEMU (have corresponding 'hv-' flag) and copies `hv-spinlocks` and `hv-vendor-id` values from KVM to QEMU. `hv-passthrough` overrides all other 'hv-' settings on the command line. Also, enabling this flag effectively prevents migration as the list of enabled enlightenments may differ between target and destination hosts.

hv-enforce-cpuid
: By default, KVM allows the guest to use all currently supported Hyper-V enlightenments when Hyper-V CPUID interface was exposed, regardless of if some features were not announced in guest visible CPUIDs. `hv-enforce-cpuid` feature alters this behavior and only allows the guest to use exposed Hyper-V enlightenments.

Useful links

Hyper-V Top Level Functional specification and other information:

- https://github.com/MicrosoftDocs/Virtualization-Documentation
- https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/tlfs/tlfs

Xen HVM guest support

Description

KVM has support for hosting Xen guests, intercepting Xen hypercalls and event channel (Xen PV interrupt) delivery. This allows guests which expect to be run under Xen to be hosted in QEMU under Linux/KVM instead.

Using the split irqchip is mandatory for Xen support.

Setup

Xen mode is enabled by setting the `xen-version` property of the KVM accelerator, for example for Xen 4.17:

```
qemu-system-x86_64 --accel kvm,xen-version=0x40011,kernel-irqchip=split
```

Additionally, virtual APIC support can be advertised to the guest through the `xen-vapic` CPU flag:

```
qemu-system-x86_64 --accel kvm,xen-version=0x40011,kernel-irqchip=split --cpu host,
↪+xen-vapic
```

When Xen support is enabled, QEMU changes hypervisor identification (CPUID 0x40000000..0x4000000A) to Xen. The KVM identification and features are not advertised to a Xen guest. If Hyper-V is also enabled, the Xen identification moves to leaves 0x40000100..0x4000010A.

Properties

The following properties exist on the KVM accelerator object:

xen-version
This property contains the Xen version in `XENVER_version` form, with the major version in the top 16 bits and the minor version in the low 16 bits. Setting this property enables the Xen guest support. If Xen version 4.5 or greater is specified, the HVM leaf in Xen CPUID is populated. Xen version 4.6 enables the vCPU ID in CPUID, and version 4.17 advertises vCPU upcall vector support to the guest.

xen-evtchn-max-pirq
Xen PIRQs represent an emulated physical interrupt, either GSI or MSI, which can be routed to an event channel instead of to the emulated I/O or local APIC. By default, QEMU permits only 256 PIRQs because this allows maximum compatibility with 32-bit MSI where the higher bits of the PIRQ# would need to be in the upper 64 bits of the MSI message. For guests with large numbers of PCI devices (and none which are limited to 32-bit addressing) it may be desirable to increase this value.

xen-gnttab-max-frames
Xen grant tables are the means by which a Xen guest grants access to its memory for PV back ends (disk, network, etc.). Since QEMU only supports v1 grant tables which are 8 bytes in size, each page (each frame) of the grant table can reference 512 pages of guest memory. The default number of frames is 64, allowing for 32768 pages of guest memory to be accessed by PV backends through simultaneous grants. For guests with large numbers of PV devices and high throughput, it may be desirable to increase this value.

Xen paravirtual devices

The Xen PCI platform device is enabled automatically for a Xen guest. This allows a guest to unplug all emulated devices, in order to use paravirtual block and network drivers instead.

Those paravirtual Xen block, network (and console) devices can be created through the command line, and/or hot-plugged.

To provide a Xen console device, define a character device and then a device of type `xen-console` to connect to it. For the Xen console equivalent of the handy `-serial mon:stdio` option, for example:

```
-chardev stdio,mux=on,id=char0,signal=off -mon char0 \
-device xen-console,chardev=char0
```

The Xen network device is `xen-net-device`, which becomes the default NIC model for emulated Xen guests, meaning that just the default NIC provided by QEMU should automatically work and present a Xen network device to the guest.

Disks can be configured with '`-drive file=${GUEST_IMAGE},if=xen`' and will appear to the guest as `xvda` onwards.

Under Xen, the boot disk is typically available both via IDE emulation, and as a PV block device. Guest bootloaders typically use IDE to load the guest kernel, which then unplugs the IDE and continues with the Xen PV block device.

This configuration can be achieved as follows:

```
qemu-system-x86_64 --accel kvm,xen-version=0x40011,kernel-irqchip=split \
     -drive file=${GUEST_IMAGE},if=xen \
     -drive file=${GUEST_IMAGE},file.locking=off,if=ide
```

VirtIO devices can also be used; Linux guests may need to be dissuaded from unplugging them by adding '`xen_emul_unplug=never`' on their command line.

Booting Xen PV guests

Booting PV guest kernels is possible by using the Xen PV shim (a version of Xen itself, designed to run inside a Xen HVM guest and provide memory management services for one guest alone).

The Xen binary is provided as the `-kernel` and the guest kernel itself (or PV Grub image) as the `-initrd` image, which actually just means the first multiboot "module". For example:

```
qemu-system-x86_64 --accel kvm,xen-version=0x40011,kernel-irqchip=split \
     -chardev stdio,id=char0 -device xen-console,chardev=char0 \
     -display none  -m 1G  -kernel xen -initrd bzImage \
     -append "pv-shim console=xen,pv -- console=hvc0 root=/dev/xvda1" \
     -drive file=${GUEST_IMAGE},if=xen
```

The Xen image must be built with the `CONFIG_XEN_GUEST` and `CONFIG_PV_SHIM` options, and as of Xen 4.17, Xen's PV shim mode does not support using a serial port; it must have a Xen console or it will panic.

The example above provides the guest kernel command line after a separator (" `--` ") on the Xen command line, and does not provide the guest kernel with an actual initramfs, which would need to listed as a second multiboot module. For more complicated alternatives, see the command line *documentation* for the `-initrd` option.

Host OS requirements

The minimal Xen support in the KVM accelerator requires the host to be running Linux v5.12 or newer. Later versions add optimisations: Linux v5.17 added acceleration of interrupt delivery via the Xen PIRQ mechanism, and Linux v5.19 accelerated Xen PV timers and inter-processor interrupts (IPIs).

Paravirtualized KVM features

Description

In some cases when implementing hardware interfaces in software is slow, `KVM` implements its own paravirtualized interfaces.

Setup

Paravirtualized `KVM` features are represented as CPU flags. The following features are enabled by default for any CPU model when `KVM` acceleration is enabled:

- `kvmclock`
- `kvm-nopiodelay`
- `kvm-asyncpf`
- `kvm-steal-time`
- `kvm-pv-eoi`
- `kvmclock-stable-bit`

`kvm-msi-ext-dest-id` feature is enabled by default in x2apic mode with split irqchip (e.g. "-machine ...,kernel-irqchip=split -cpu ...,x2apic").

Note: when CPU model `host` is used, QEMU passes through all supported paravirtualized `KVM` features to the guest.

Existing features

kvmclock
Expose a KVM specific paravirtualized clocksource to the guest. Supported since Linux v2.6.26.

kvm-nopiodelay
The guest doesn't need to perform delays on PIO operations. Supported since Linux v2.6.26.

kvm-mmu
This feature is deprecated.

kvm-asyncpf
Enable asynchronous page fault mechanism. Supported since Linux v2.6.38. Note: since Linux v5.10 the feature is deprecated and not enabled by KVM. Use kvm-asyncpf-int instead.

kvm-steal-time
Enable stolen (when guest vCPU is not running) time accounting. Supported since Linux v3.1.

kvm-pv-eoi
Enable paravirtualized end-of-interrupt signaling. Supported since Linux v3.10.

kvm-pv-unhalt
Enable paravirtualized spinlocks support. Supported since Linux v3.12.

kvm-pv-tlb-flush
Enable paravirtualized TLB flush mechanism. Supported since Linux v4.16.

kvm-pv-ipi
Enable paravirtualized IPI mechanism. Supported since Linux v4.19.

kvm-poll-control
Enable host-side polling on HLT control from the guest. Supported since Linux v5.10.

kvm-pv-sched-yield
Enable paravirtualized sched yield feature. Supported since Linux v5.10.

kvm-asyncpf-int
Enable interrupt based asynchronous page fault mechanism. Supported since Linux v5.10.

kvm-msi-ext-dest-id
Support 'Extended Destination ID' for external interrupts. The feature allows to use up to 32768 CPUs without IRQ remapping (but other limits may apply making the number of supported vCPUs for a given configuration lower). Supported since Linux v5.10.

kvmclock-stable-bit
Tell the guest that guest visible TSC value can be fully trusted for kvmclock computations and no warps are expected. Supported since Linux v2.6.35.

Supplementary features

kvm-pv-enforce-cpuid
Limit the supported paravirtualized feature set to the exposed features only. Note, by default, KVM allows the guest to use all currently supported paravirtualized features even when they were not announced in guest visible CPUIDs. Supported since Linux v5.10.

Useful links

Please refer to Documentation/virt/kvm in Linux for additional details.

Software Guard eXtensions (SGX)

Overview

Intel Software Guard eXtensions (SGX) is a set of instructions and mechanisms for memory accesses in order to provide security accesses for sensitive applications and data. SGX allows an application to use its particular address space as an *enclave*, which is a protected area provides confidentiality and integrity even in the presence of privileged malware. Accesses to the enclave memory area from any software not resident in the enclave are prevented, including those from privileged software.

Virtual SGX

SGX feature is exposed to guest via SGX CPUID. Looking at SGX CPUID, we can report the same CPUID info to guest as on host for most of SGX CPUID. With reporting the same CPUID guest is able to use full capacity of SGX, and KVM doesn't need to emulate those info.

The guest's EPC base and size are determined by QEMU, and KVM needs QEMU to notify such info to it before it can initialize SGX for guest.

Virtual EPC

By default, QEMU does not assign EPC to a VM, i.e. fully enabling SGX in a VM requires explicit allocation of EPC to the VM. Similar to other specialized memory types, e.g. hugetlbfs, EPC is exposed as a memory backend.

SGX EPC is enumerated through CPUID, i.e. EPC "devices" need to be realized prior to realizing the vCPUs themselves, which occurs long before generic devices are parsed and realized. This limitation means that EPC does not require -maxmem as EPC is not treated as {cold,hot}plugged memory.

QEMU does not artificially restrict the number of EPC sections exposed to a guest, e.g. QEMU will happily allow you to create 64 1M EPC sections. Be aware that some kernels may not recognize all EPC sections, e.g. the Linux SGX driver is hardwired to support only 8 EPC sections.

The following QEMU snippet creates two EPC sections, with 64M pre-allocated to the VM and an additional 28M mapped but not allocated:

```
-object memory-backend-epc,id=mem1,size=64M,prealloc=on \
-object memory-backend-epc,id=mem2,size=28M \
-M sgx-epc.0.memdev=mem1,sgx-epc.1.memdev=mem2
```

Note:

The size and location of the virtual EPC are far less restricted compared to physical EPC. Because physical EPC is protected via range registers, the size of the physical EPC must be a power of two (though software sees a subset of the full EPC, e.g. 92M or 128M) and the EPC must be naturally aligned. KVM SGX's virtual EPC is purely a software construct and only requires the size and location to be page aligned. QEMU enforces the EPC size is a multiple of 4k and will ensure the base of the EPC is 4k aligned. To simplify the implementation, EPC is always located above 4g in the guest physical address space.

Migration

QEMU/KVM doesn't prevent live migrating SGX VMs, although from hardware's perspective, SGX doesn't support live migration, since both EPC and the SGX key hierarchy are bound to the physical platform. However live migration can be supported in the sense if guest software stack can support recreating enclaves when it suffers sudden lose of EPC; and if guest enclaves can detect SGX keys being changed, and handle gracefully. For instance, when ERESUME fails with #PF.SGX, guest software can gracefully detect it and recreate enclaves; and when enclave fails to unseal sensitive information from outside, it can detect such error and sensitive information can be provisioned to it again.

CPUID

Due to its myriad dependencies, SGX is currently not listed as supported in any of QEMU's built-in CPU configuration. To expose SGX (and SGX Launch Control) to a guest, you must either use `-cpu host` to pass-through the host CPU model, or explicitly enable SGX when using a built-in CPU model, e.g. via `-cpu <model>,+sgx` or `-cpu <model>,+sgx,+sgxlc`.

All SGX sub-features enumerated through CPUID, e.g. SGX2, MISCSELECT, ATTRIBUTES, etc… can be restricted via CPUID flags. Be aware that enforcing restriction of MISCSELECT, ATTRIBUTES and XFRM requires intercepting ECREATE, i.e. may marginally reduce SGX performance in the guest. All SGX sub-features controlled via -cpu are prefixed with "sgx", e.g.:

```
$ qemu-system-x86_64 -cpu help | xargs printf "%s\n" | grep sgx
sgx
sgx-debug
sgx-encls-c
sgx-enclv
sgx-exinfo
sgx-kss
sgx-mode64
sgx-provisionkey
sgx-tokenkey
sgx1
sgx2
sgxlc
```

The following QEMU snippet passes through the host CPU but restricts access to the provision and EINIT token keys:

```
-cpu host,-sgx-provisionkey,-sgx-tokenkey
```

SGX sub-features cannot be emulated, i.e. sub-features that are not present in hardware cannot be forced on via '-cpu'.

Virtualize SGX Launch Control

QEMU SGX support for Launch Control (LC) is passive, in the sense that it does not actively change the LC configuration. QEMU SGX provides the user the ability to set/clear the CPUID flag (and by extension the associated IA32_FEATURE_CONTROL MSR bit in fw_cfg) and saves/restores the LE Hash MSRs when getting/putting guest state, but QEMU does not add new controls to directly modify the LC configuration. Similar to hardware behavior, locking the LC configuration to a non-Intel value is left to guest firmware. Unlike host bios setting for SGX launch control(LC), there is no special bios setting for SGX guest by our design. If host is in locked mode, we can still allow creating VM with SGX.

Feature Control

QEMU SGX updates the `etc/msr_feature_control` fw_cfg entry to set the SGX (bit 18) and SGX LC (bit 17) flags based on their respective CPUID support, i.e. existing guest firmware will automatically set SGX and SGX LC accordingly, assuming said firmware supports fw_cfg.msr_feature_control.

Launching a guest

To launch a SGX guest:

```
qemu-system-x86_64 \
 -cpu host,+sgx-provisionkey \
 -object memory-backend-epc,id=mem1,size=64M,prealloc=on \
 -M sgx-epc.0.memdev=mem1,sgx-epc.0.node=0
```

Utilizing SGX in the guest requires a kernel/OS with SGX support. The support can be determined in guest by:

```
$ grep sgx /proc/cpuinfo
```

and SGX epc info by:

```
$ dmesg | grep sgx
[    0.182807] sgx: EPC section 0x140000000-0x143ffffff
[    0.183695] sgx: [Firmware Bug]: Unable to map EPC section to online node. Fallback␣
↪to the NUMA node 0.
```

To launch a SGX numa guest:

```
qemu-system-x86_64 \
 -cpu host,+sgx-provisionkey \
 -object memory-backend-ram,size=2G,host-nodes=0,policy=bind,id=node0 \
 -object memory-backend-epc,id=mem0,size=64M,prealloc=on,host-nodes=0,policy=bind \
 -numa node,nodeid=0,cpus=0-1,memdev=node0 \
 -object memory-backend-ram,size=2G,host-nodes=1,policy=bind,id=node1 \
 -object memory-backend-epc,id=mem1,size=28M,prealloc=on,host-nodes=1,policy=bind \
 -numa node,nodeid=1,cpus=2-3,memdev=node1 \
 -M sgx-epc.0.memdev=mem0,sgx-epc.0.node=0,sgx-epc.1.memdev=mem1,sgx-epc.1.node=1
```

and SGX epc numa info by:

```
$ dmesg | grep sgx
[    0.369937] sgx: EPC section 0x180000000-0x183ffffff
[    0.370259] sgx: EPC section 0x184000000-0x185bfffff

$ dmesg | grep SRAT
[    0.009981] ACPI: SRAT: Node 0 PXM 0 [mem 0x180000000-0x183ffffff]
[    0.009982] ACPI: SRAT: Node 1 PXM 1 [mem 0x184000000-0x185bfffff]
```

References

- SGX Homepage
- SGX SDK
- SGX specification: Intel SDM Volume 3

AMD Secure Encrypted Virtualization (SEV)

Secure Encrypted Virtualization (SEV) is a feature found on AMD processors.

SEV is an extension to the AMD-V architecture which supports running encrypted virtual machines (VMs) under the control of KVM. Encrypted VMs have their pages (code and data) secured such that only the guest itself has access to the unencrypted version. Each encrypted VM is associated with a unique encryption key; if its data is accessed by a different entity using a different key the encrypted guests data will be incorrectly decrypted, leading to unintelligible data.

Key management for this feature is handled by a separate processor known as the AMD secure processor (AMD-SP), which is present in AMD SOCs. Firmware running inside the AMD-SP provides commands to support a common VM lifecycle. This includes commands for launching, snapshotting, migrating and debugging the encrypted guest. These SEV commands can be issued via KVM_MEMORY_ENCRYPT_OP ioctls.

Secure Encrypted Virtualization - Encrypted State (SEV-ES) builds on the SEV support to additionally protect the guest register state. In order to allow a hypervisor to perform functions on behalf of a guest, there is architectural support for notifying a guest's operating system when certain types of VMEXITs are about to occur. This allows the guest to selectively share information with the hypervisor to satisfy the requested function.

Launching

Boot images (such as bios) must be encrypted before a guest can be booted. The `MEMORY_ENCRYPT_OP` ioctl provides commands to encrypt the images: `LAUNCH_START`, `LAUNCH_UPDATE_DATA`, `LAUNCH_MEASURE` and `LAUNCH_FINISH`. These four commands together generate a fresh memory encryption key for the VM, encrypt the boot images and provide a measurement than can be used as an attestation of a successful launch.

For a SEV-ES guest, the `LAUNCH_UPDATE_VMSA` command is also used to encrypt the guest register state, or VM save area (VMSA), for all of the guest vCPUs.

`LAUNCH_START` is called first to create a cryptographic launch context within the firmware. To create this context, guest owner must provide a guest policy, its public Diffie-Hellman key (PDH) and session parameters. These inputs should be treated as a binary blob and must be passed as-is to the SEV firmware.

The guest policy is passed as plaintext. A hypervisor may choose to read it, but should not modify it (any modification of the policy bits will result in bad measurement). The guest policy is a 4-byte data structure containing several flags that restricts what can be done on a running SEV guest. See SEV API Spec ([SEVAPI]) section 3 and 6.2 for more details.

The guest policy can be provided via the `policy` property:

```
# ${QEMU} \
   sev-guest,id=sev0,policy=0x1...\
```

Setting the "SEV-ES required" policy bit (bit 2) will launch the guest as a SEV-ES guest:

```
# ${QEMU} \
   sev-guest,id=sev0,policy=0x5...\
```

The guest owner provided DH certificate and session parameters will be used to establish a cryptographic session with the guest owner to negotiate keys used for the attestation.

The DH certificate and session blob can be provided via the `dh-cert-file` and `session-file` properties:

```
# ${QEMU} \
    sev-guest,id=sev0,dh-cert-file=<file1>,session-file=<file2>
```

`LAUNCH_UPDATE_DATA` encrypts the memory region using the cryptographic context created via the `LAUNCH_START` command. If required, this command can be called multiple times to encrypt different memory regions. The command also calculates the measurement of the memory contents as it encrypts.

`LAUNCH_UPDATE_VMSA` encrypts all the vCPU VMSAs for a SEV-ES guest using the cryptographic context created via the `LAUNCH_START` command. The command also calculates the measurement of the VMSAs as it encrypts them.

`LAUNCH_MEASURE` can be used to retrieve the measurement of encrypted memory and, for a SEV-ES guest, encrypted VMSAs. This measurement is a signature of the memory contents and, for a SEV-ES guest, the VMSA contents, that can be sent to the guest owner as an attestation that the memory and VMSAs were encrypted correctly by the firmware. The guest owner may wait to provide the guest confidential information until it can verify the attestation measurement. Since the guest owner knows the initial contents of the guest at boot, the attestation measurement can be verified by comparing it to what the guest owner expects.

`LAUNCH_FINISH` finalizes the guest launch and destroys the cryptographic context.

See SEV API Spec ([SEVAPI]) 'Launching a guest' usage flow (Appendix A) for the complete flow chart.

To launch a SEV guest:

```
# ${QEMU} \
    -machine ...,confidential-guest-support=sev0 \
    -object sev-guest,id=sev0,cbitpos=47,reduced-phys-bits=1
```

To launch a SEV-ES guest:

```
# ${QEMU} \
    -machine ...,confidential-guest-support=sev0 \
    -object sev-guest,id=sev0,cbitpos=47,reduced-phys-bits=1,policy=0x5
```

An SEV-ES guest has some restrictions as compared to a SEV guest. Because the guest register state is encrypted and cannot be updated by the VMM/hypervisor, a SEV-ES guest:

- Does not support SMM - SMM support requires updating the guest register state.
- Does not support reboot - a system reset requires updating the guest register state.
- Requires in-kernel irqchip - the burden is placed on the hypervisor to manage booting APs.

Calculating expected guest launch measurement

In order to verify the guest launch measurement, The Guest Owner must compute it in the exact same way as it is calculated by the AMD-SP. SEV API Spec ([SEVAPI]) section 6.5.1 describes the AMD-SP operations:

> GCTX.LD is finalized, producing the hash digest of all plaintext data imported into the guest.
>
> The launch measurement is calculated as:
>
> HMAC(0x04 || API_MAJOR || API_MINOR || BUILD || GCTX.POLICY || GCTX.LD || MNONCE; GCTX.TIK)
>
> where "||" represents concatenation.

The values of API_MAJOR, API_MINOR, BUILD, and GCTX.POLICY can be obtained from the `query-sev` qmp command.

The value of MNONCE is part of the response of `query-sev-launch-measure`: it is the last 16 bytes of the base64-decoded data field (see SEV API Spec ([SEVAPI]) section 6.5.2 Table 52: LAUNCH_MEASURE Measurement Buffer).

The value of GCTX.LD is `SHA256(firmware_blob || kernel_hashes_blob || vmsas_blob)`, where:

- `firmware_blob` is the content of the entire firmware flash file (for example, `OVMF.fd`). Note that you must build a stateless firmware file which doesn't use an NVRAM store, because the NVRAM area is not measured, and therefore it is not secure to use a firmware which uses state from an NVRAM store.
- if kernel is used, and `kernel-hashes=on`, then `kernel_hashes_blob` is the content of PaddedSevHashTable (including the zero padding), which itself includes the hashes of kernel, initrd, and cmdline that are passed to the guest. The PaddedSevHashTable struct is defined in `target/i386/sev.c`.
- if SEV-ES is enabled (`policy & 0x4 != 0`), `vmsas_blob` is the concatenation of all VMSAs of the guest vcpus. Each VMSA is 4096 bytes long; its content is defined inside Linux kernel code as `struct vmcb_save_area`, or in AMD APM Volume 2 ([APMVOL2]) Table B-2: VMCB Layout, State Save Area.

If kernel hashes are not used, or SEV-ES is disabled, use empty blobs for `kernel_hashes_blob` and `vmsas_blob` as needed.

Debugging

Since the memory contents of a SEV guest are encrypted, hypervisor access to the guest memory will return cipher text. If the guest policy allows debugging, then a hypervisor can use the DEBUG_DECRYPT and DEBUG_ENCRYPT commands to access the guest memory region for debug purposes. This is not supported in QEMU yet.

Snapshot/Restore

TODO

Live Migration

TODO

References

AMD Memory Encryption whitepaper

KVM Forum slides:

- AMD's Virtualization Memory Encryption (2016)
- Extending Secure Encrypted Virtualization With SEV-ES (2018)

AMD64 Architecture Programmer's Manual:

- SME is section 7.10
- SEV is section 15.34
- SEV-ES is section 15.35

OS requirements

On x86_64 hosts, the default set of CPU features enabled by the KVM accelerator require the host to be running Linux v4.5 or newer. Red Hat Enterprise Linux 7 is also supported, since the required functionality was backported.

2.23.13 Xtensa System emulator

Two executables cover simulation of both Xtensa endian options, `qemu-system-xtensa` and `qemu-system-xtensaeb`. Two different machine types are emulated:

- Xtensa emulator pseudo board "sim"
- Avnet LX60/LX110/LX200 board

The sim pseudo board emulation provides an environment similar to one provided by the proprietary Tensilica ISS. It supports:

- A range of Xtensa CPUs, default is the DC232B
- Console and filesystem access via semihosting calls

The Avnet LX60/LX110/LX200 emulation supports:

- A range of Xtensa CPUs, default is the DC232B
- 16550 UART
- OpenCores 10/100 Mbps Ethernet MAC

2.24 Security

2.24.1 Overview

This chapter explains the security requirements that QEMU is designed to meet and principles for securely deploying QEMU.

2.24.2 Security Requirements

QEMU supports many different use cases, some of which have stricter security requirements than others. The community has agreed on the overall security requirements that users may depend on. These requirements define what is considered supported from a security perspective.

Virtualization Use Case

The virtualization use case covers cloud and virtual private server (VPS) hosting, as well as traditional data center and desktop virtualization. These use cases rely on hardware virtualization extensions to execute guest code safely on the physical CPU at close-to-native speed.

The following entities are untrusted, meaning that they may be buggy or malicious:

- Guest
- User-facing interfaces (e.g. VNC, SPICE, WebSocket)
- Network protocols (e.g. NBD, live migration)

- User-supplied files (e.g. disk images, kernels, device trees)
- Passthrough devices (e.g. PCI, USB)

Bugs affecting these entities are evaluated on whether they can cause damage in real-world use cases and treated as security bugs if this is the case.

Non-virtualization Use Case

The non-virtualization use case covers emulation using the Tiny Code Generator (TCG). In principle the TCG and device emulation code used in conjunction with the non-virtualization use case should meet the same security requirements as the virtualization use case. However, for historical reasons much of the non-virtualization use case code was not written with these security requirements in mind.

Bugs affecting the non-virtualization use case are not considered security bugs at this time. Users with non-virtualization use cases must not rely on QEMU to provide guest isolation or any security guarantees.

2.24.3 Architecture

This section describes the design principles that ensure the security requirements are met.

Guest Isolation

Guest isolation is the confinement of guest code to the virtual machine. When guest code gains control of execution on the host this is called escaping the virtual machine. Isolation also includes resource limits such as throttling of CPU, memory, disk, or network. Guests must be unable to exceed their resource limits.

QEMU presents an attack surface to the guest in the form of emulated devices. The guest must not be able to gain control of QEMU. Bugs in emulated devices could allow malicious guests to gain code execution in QEMU. At this point the guest has escaped the virtual machine and is able to act in the context of the QEMU process on the host.

Guests often interact with other guests and share resources with them. A malicious guest must not gain control of other guests or access their data. Disk image files and network traffic must be protected from other guests unless explicitly shared between them by the user.

Principle of Least Privilege

The principle of least privilege states that each component only has access to the privileges necessary for its function. In the case of QEMU this means that each process only has access to resources belonging to the guest.

The QEMU process should not have access to any resources that are inaccessible to the guest. This way the guest does not gain anything by escaping into the QEMU process since it already has access to those same resources from within the guest.

Following the principle of least privilege immediately fulfills guest isolation requirements. For example, guest A only has access to its own disk image file `a.img` and not guest B's disk image file `b.img`.

In reality certain resources are inaccessible to the guest but must be available to QEMU to perform its function. For example, host system calls are necessary for QEMU but are not exposed to guests. A guest that escapes into the QEMU process can then begin invoking host system calls.

New features must be designed to follow the principle of least privilege. Should this not be possible for technical reasons, the security risk must be clearly documented so users are aware of the trade-off of enabling the feature.

Isolation mechanisms

Several isolation mechanisms are available to realize this architecture of guest isolation and the principle of least privilege. With the exception of Linux seccomp, these mechanisms are all deployed by management tools that launch QEMU, such as libvirt. They are also platform-specific so they are only described briefly for Linux here.

The fundamental isolation mechanism is that QEMU processes must run as unprivileged users. Sometimes it seems more convenient to launch QEMU as root to give it access to host devices (e.g. `/dev/net/tun`) but this poses a huge security risk. File descriptor passing can be used to give an otherwise unprivileged QEMU process access to host devices without running QEMU as root. It is also possible to launch QEMU as a non-root user and configure UNIX groups for access to `/dev/kvm`, `/dev/net/tun`, and other device nodes. Some Linux distros already ship with UNIX groups for these devices by default.

- SELinux and AppArmor make it possible to confine processes beyond the traditional UNIX process and file permissions model. They restrict the QEMU process from accessing processes and files on the host system that are not needed by QEMU.
- Resource limits and cgroup controllers provide throughput and utilization limits on key resources such as CPU time, memory, and I/O bandwidth.
- Linux namespaces can be used to make process, file system, and other system resources unavailable to QEMU. A namespaced QEMU process is restricted to only those resources that were granted to it.
- Linux seccomp is available via the QEMU `--sandbox` option. It disables system calls that are not needed by QEMU, thereby reducing the host kernel attack surface.

2.24.4 Sensitive configurations

There are aspects of QEMU that can have security implications which users & management applications must be aware of.

Monitor console (QMP and HMP)

The monitor console (whether used with QMP or HMP) provides an interface to dynamically control many aspects of QEMU's runtime operation. Many of the commands exposed will instruct QEMU to access content on the host file system and/or trigger spawning of external processes.

For example, the `migrate` command allows for the spawning of arbitrary processes for the purpose of tunnelling the migration data stream. The `blockdev-add` command instructs QEMU to open arbitrary files, exposing their content to the guest as a virtual disk.

Unless QEMU is otherwise confined using technologies such as SELinux, AppArmor, or Linux namespaces, the monitor console should be considered to have privileges equivalent to those of the user account QEMU is running under.

It is further important to consider the security of the character device backend over which the monitor console is exposed. It needs to have protection against malicious third parties which might try to make unauthorized connections, or perform man-in-the-middle attacks. Many of the character device backends do not satisfy this requirement and so must not be used for the monitor console.

The general recommendation is that the monitor console should be exposed over a UNIX domain socket backend to the local host only. Use of the TCP based character device backend is inappropriate unless configured to use both TLS encryption and authorization control policy on client connections.

In summary, the monitor console is considered a privileged control interface to QEMU and as such should only be made accessible to a trusted management application or user.

2.25 Multi-process QEMU

This document describes how to configure and use multi-process qemu. For the design document refer to docs/devel/multi-process.rst.

2.25.1 1) Configuration

multi-process is enabled by default for targets that enable KVM

2.25.2 2) Usage

Multi-process QEMU requires an orchestrator to launch.

Following is a description of command-line used to launch mpqemu.

- Orchestrator:
  - The Orchestrator creates a unix socketpair
  - It launches the remote process and passes one of the sockets to it via command-line.
  - It then launches QEMU and specifies the other socket as an option to the Proxy device object
- Remote Process:
  - QEMU can enter remote process mode by using the "remote" machine option.
  - The orchestrator creates a "remote-object" with details about the device and the file descriptor for the device
  - The remaining options are no different from how one launches QEMU with devices.
  - Example command-line for the remote process is as follows:

    /usr/bin/qemu-system-x86_64 -machine x-remote -device lsi53c895a,id=lsi0 -drive id=drive_image2,file=/build/ol7-nvme-test-1.qcow2 -device scsi-hd,id=drive2,drive=drive_image2,bus=lsi0.0,scsi-id=0 -object x-remote-object,id=robj1,devid=lsi0,fd=4,
- QEMU:
  - Since parts of the RAM are shared between QEMU & remote process, a memory-backend-memfd is required to facilitate this, as follows:

    -object memory-backend-memfd,id=mem,size=2G
  - A "x-pci-proxy-dev" device is created for each of the PCI devices emulated in the remote process. A "socket" sub-option specifies the other end of unix channel created by orchestrator. The "id" sub-option must be specified and should be the same as the "id" specified for the remote PCI device
  - Example commandline for QEMU is as follows:

    -device x-pci-proxy-dev,id=lsi0,socket=3

2.26 Confidential Guest Support

Traditionally, hypervisors such as QEMU have complete access to a guest's memory and other state, meaning that a compromised hypervisor can compromise any of its guests. A number of platforms have added mechanisms in hardware and/or firmware which give guests at least some protection from a compromised hypervisor. This is obviously especially desirable for public cloud environments.

These mechanisms have different names and different modes of operation, but are often referred to as Secure Guests or Confidential Guests. We use the term "Confidential Guest Support" to distinguish this from other aspects of guest security (such as security against attacks from other guests, or from network sources).

2.26.1 Running a Confidential Guest

To run a confidential guest you need to add two command line parameters:

1. Use `-object` to create a "confidential guest support" object. The type and parameters will vary with the specific mechanism to be used
2. Set the `confidential-guest-support` machine parameter to the ID of the object from (1).

Example (for AMD SEV):

```
qemu-system-x86_64 \
    <other parameters> \
    -machine ...,confidential-guest-support=sev0 \
    -object sev-guest,id=sev0,cbitpos=47,reduced-phys-bits=1
```

2.26.2 Supported mechanisms

Currently supported confidential guest mechanisms are:

- AMD Secure Encrypted Virtualization (SEV) (see *AMD Secure Encrypted Virtualization (SEV)*)
- POWER Protected Execution Facility (PEF) (see *POWER (PAPR) Protected Execution Facility (PEF)*)
- s390x Protected Virtualization (PV) (see *Protected Virtualization on s390x*)

Other mechanisms may be supported in future.

2.27 QEMU VM templating

This document explains how to use VM templating in QEMU.

For now, the focus is on VM memory aspects, and not about how to save and restore other VM state (i.e., migrate-to-file with `x-ignore-shared`).

2.27.1 Overview

With VM templating, a single template VM serves as the starting point for new VMs. This allows for fast and efficient replication of VMs, resulting in fast startup times and reduced memory consumption.

Conceptually, the VM state is frozen, to then be used as a basis for new VMs. The Copy-On-Write mechanism in the operating systems makes sure that new VMs are able to read template VM memory; however, any modifications stay private and don't modify the original template VM or any other created VM.

2.27.2 !!! Security Alert !!!

When effectively cloning VMs by VM templating, hardware identifiers (such as UUIDs and NIC MAC addresses), and similar data in the guest OS (such as machine IDs, SSH keys, certificates) that are supposed to be *unique* are no longer unique, which can be a security concern.

Please be aware of these implications and how to mitigate them for your use case, which might involve vmgenid, hot(un)plug of NIC, etc..

2.27.3 Memory configuration

In order to create the template VM, we have to make sure that VM memory ends up in a file, from where it can be reused for the new VMs:

Supply VM RAM via memory-backend-file, with `share=on` (modifications go to the file) and `readonly=off` (open the file writable). Note that `readonly=off` is implicit.

In the following command-line example, a 2GB VM is created, whereby VM RAM is to be stored in the `template` file.

```
qemu-system-x86_64 [...] -m 2g \
    -object memory-backend-file,id=pc.ram,mem-path=template,size=2g,share=on,... \
    -machine q35,memory-backend=pc.ram
```

If multiple memory backends are used (vNUMA, DIMMs), configure all memory backends accordingly.

Once the VM is in the desired state, stop the VM and save other VM state, leaving the current state of VM RAM reside in the file.

In order to have a new VM be based on a template VM, we have to configure VM RAM to be based on a template VM RAM file; however, the VM should not be able to modify file content.

Supply VM RAM via memory-backend-file, with `share=off` (modifications stay private), `readonly=on` (open the file readonly) and `rom=off` (don't make the memory readonly for the VM). Note that `share=off` is implicit and that other VM state has to be restored separately.

In the following command-line example, a 2GB VM is created based on the existing 2GB file `template`.

```
qemu-system-x86_64 [...] -m 2g \
    -object memory-backend-file,id=pc.ram,mem-path=template,size=2g,readonly=on,rom=off,.
↪.. \
    -machine q35,memory-backend=pc.ram
```

If multiple memory backends are used (vNUMA, DIMMs), configure all memory backends accordingly.

Note that `-mem-path` cannot be used for VM templating when creating the template VM or when starting new VMs based on a template VM.

2.27.4 Incompatible features

Some features are incompatible with VM templating, as the underlying file cannot be modified to discard VM RAM, or to actually share memory with another process.

vhost-user and multi-process QEMU

vhost-user and multi-process QEMU are incompatible with VM templating. These technologies rely on shared memory, however, the template VMs don't actually share memory (`share=off`), even though they are file-based.

virtio-balloon

virtio-balloon inflation and "free page reporting" cannot discard VM RAM and will repeatedly report errors. While virtio-balloon can be used for template VMs (e.g., report VM RAM stats), "free page reporting" should be disabled and the balloon should not be inflated.

virtio-mem

virtio-mem cannot discard VM RAM that is managed by the virtio-mem device. virtio-mem will fail early when realizing the device. To use VM templating with virtio-mem, either hotplug virtio-mem devices to the new VM, or don't supply any memory to the template VM using virtio-mem (requested-size=0), not using a template VM file as memory backend for the virtio-mem device.

VM migration

For VM migration, "x-release-ram" similarly relies on discarding of VM RAM on the migration source to free up migrated RAM, and will repeatedly report errors.

Postcopy live migration fails discarding VM RAM on the migration destination early and refuses to activate postcopy live migration. Note that postcopy live migration usually only works on selected filesystems (shmem/tmpfs, hugetlbfs) either way.

CHAPTER

THREE

USER MODE EMULATION

This section of the manual is the overall guide for users using QEMU for user-mode emulation. In this mode, QEMU can launch processes compiled for one CPU on another CPU.

3.1 QEMU User space emulator

3.1.1 Supported Operating Systems

The following OS are supported in user space emulation:

- Linux (referred as qemu-linux-user)
- BSD (referred as qemu-bsd-user)

3.1.2 Features

QEMU user space emulation has the following notable features:

System call translation:
QEMU includes a generic system call translator. This means that the parameters of the system calls can be converted to fix endianness and 32/64-bit mismatches between hosts and targets. IOCTLs can be converted too.

POSIX signal handling:
QEMU can redirect to the running program all signals coming from the host (such as `SIGALRM`), as well as synthesize signals from virtual CPU exceptions (for example `SIGFPE` when the program executes a division by zero).

QEMU relies on the host kernel to emulate most signal system calls, for example to emulate the signal mask. On Linux, QEMU supports both normal and real-time signals.

Threading:
On Linux, QEMU can emulate the `clone` syscall and create a real host thread (with a separate virtual CPU) for each emulated thread. Note that not all targets currently emulate atomic operations correctly. x86 and Arm use a global lock in order to preserve their semantics.

QEMU was conceived so that ultimately it can emulate itself. Although it is not very useful, it is an important test to show the power of the emulator.

3.1.3 Linux User space emulator

Command line options

```
qemu-i386 [-h] [-d] [-L path] [-s size] [-cpu model] [-g port] [-B offset] [-R size]␣
↪program [arguments...]
```

-h
Print the help

-L path
Set the x86 elf interpreter prefix (default=/usr/local/qemu-i386)

-s size
Set the x86 stack size in bytes (default=524288)

-cpu model
Select CPU model (-cpu help for list and additional feature selection)

-E var=value
Set environment var to value.

-U var
Remove var from the environment.

-B offset
Offset guest address by the specified number of bytes. This is useful when the address region required by guest applications is reserved on the host. This option is currently only supported on some hosts.

-R size
Pre-allocate a guest virtual address space of the given size (in bytes). "G", "M", and "k" suffixes may be used when specifying the size.

Debug options:

-d item1,...
Activate logging of the specified items (use '-d help' for a list of log items)

-g port
Wait gdb connection to port

-one-insn-per-tb
Run the emulation with one guest instruction per translation block. This slows down emulation a lot, but can be useful in some situations, such as when trying to analyse the logs produced by the `-d` option.

Environment variables:

QEMU_STRACE
Print system calls and arguments similar to the 'strace' program (NOTE: the actual 'strace' program will not work because the user space emulator hasn't implemented ptrace). At the moment this is incomplete. All system calls that don't have a specific argument format are printed with information for six arguments. Many flag-style arguments don't have decoders and will show up as numbers.

Other binaries

- user mode (Alpha)
  - `qemu-alpha` TODO.
- user mode (Arm)
  - `qemu-armeb` TODO.
  - `qemu-arm` is also capable of running Arm "Angel" semihosted ELF binaries (as implemented by the arm-elf and arm-eabi Newlib/GDB configurations), and arm-uclinux bFLT format binaries.
- user mode (ColdFire)
- user mode (M68K)
  - `qemu-m68k` is capable of running semihosted binaries using the BDM (m5xxx-ram-hosted.ld) or m68k-sim (sim.ld) syscall interfaces, and coldfire uClinux bFLT format binaries.

  The binary format is detected automatically.
- user mode (Cris)
  - `qemu-cris` TODO.
- user mode (i386)
  - `qemu-i386` TODO.
  - `qemu-x86_64` TODO.
- user mode (Microblaze)
  - `qemu-microblaze` TODO.
- user mode (MIPS)
  - `qemu-mips` executes 32-bit big endian MIPS binaries (MIPS O32 ABI).
  - `qemu-mipsel` executes 32-bit little endian MIPS binaries (MIPS O32 ABI).
  - `qemu-mips64` executes 64-bit big endian MIPS binaries (MIPS N64 ABI).
  - `qemu-mips64el` executes 64-bit little endian MIPS binaries (MIPS N64 ABI).
  - `qemu-mipsn32` executes 32-bit big endian MIPS binaries (MIPS N32 ABI).
  - `qemu-mipsn32el` executes 32-bit little endian MIPS binaries (MIPS N32 ABI).
- user mode (PowerPC)
  - `qemu-ppc64` TODO.
  - `qemu-ppc` TODO.
- user mode (SH4)
  - `qemu-sh4eb` TODO.
  - `qemu-sh4` TODO.
- user mode (SPARC)
  - `qemu-sparc` can execute Sparc32 binaries (Sparc32 CPU, 32 bit ABI).
  - `qemu-sparc32plus` can execute Sparc32 and SPARC32PLUS binaries (Sparc64 CPU, 32 bit ABI).
  - `qemu-sparc64` can execute some Sparc64 (Sparc64 CPU, 64 bit ABI) and SPARC32PLUS binaries (Sparc64 CPU, 32 bit ABI).

3.1.4 BSD User space emulator

BSD Status

- target Sparc64 on Sparc64: Some trivial programs work.

Quick Start

In order to launch a BSD process, QEMU needs the process executable itself and all the target dynamic libraries used by it.

- On Sparc64, you can just try to launch any process by using the native libraries:

```
qemu-sparc64 /bin/ls
```

Command line options

```
qemu-sparc64 [-h] [-d] [-L path] [-s size] [-bsd type] program [arguments...]
```

-h
Print the help

-L path
Set the library root path (default=/)

-s size
Set the stack size in bytes (default=524288)

-ignore-environment
Start with an empty environment. Without this option, the initial environment is a copy of the caller's environment.

-E var=value
Set environment var to value.

-U var
Remove var from the environment.

-bsd type
Set the type of the emulated BSD Operating system. Valid values are FreeBSD, NetBSD and OpenBSD (default).

Debug options:

-d item1,...
Activate logging of the specified items (use '-d help' for a list of log items)

-p pagesize
Act as if the host page size was 'pagesize' bytes

-one-insn-per-tb
Run the emulation with one guest instruction per translation block. This slows down emulation a lot, but can be useful in some situations, such as when trying to analyse the logs produced by the `-d` option.

CHAPTER

FOUR

TOOLS

This section of the manual documents QEMU's "tools": its command line utilities and other standalone programs.

4.1 QEMU disk image utility

4.1.1 Synopsis

qemu-img [*standard options*] *command* [*command options*]

4.1.2 Description

qemu-img allows you to create, convert and modify images offline. It can handle all image formats supported by QEMU.

Warning: Never use qemu-img to modify images in use by a running virtual machine or any other process; this may destroy the image. Also, be aware that querying an image that is being modified by another process may encounter inconsistent state.

4.1.3 Options

Standard options:

-h, --help

Display this help and exit

-V, --version

Display version information and exit

-T, --trace [[enable=]PATTERN][,events=FILE][,file=FILE]

Specify tracing options.

[enable=]PATTERN

Immediately enable events matching *PATTERN* (either event name or a globbing pattern). This option is only available if QEMU has been compiled with the `simple`, `log` or `ftrace` tracing backend. To specify multiple events or patterns, specify the `-trace` option multiple times.

Use `-trace help` to print a list of names of trace points.

events=FILE

Immediately enable events listed in *FILE*. The file must contain one event name (as listed in the `trace-events-all` file) per line; globbing patterns are accepted too. This option is only available if QEMU has been compiled with the `simple`, `log` or `ftrace` tracing backend.

`file=FILE`

Log output traces to *FILE*. This option is only available if QEMU has been compiled with the `simple` tracing backend.

The following commands are supported:

amend [--object OBJECTDEF] [--image-opts] [-p] [-q] [-f FMT] [-t CACHE] [--force] -o OPTIONS FILENAME

bench [-c COUNT] [-d DEPTH] [-f FMT] [--flush-interval=FLUSH_INTERVAL] [-i AIO] [-n] [--no-drain] [-o O

bitmap (--merge SOURCE | --add | --remove | --clear | --enable | --disable)...
[-b SOURCE_FILE [-F SOURCE_FMT]] [-g GRANULARITY] [--object OBJECTDEF] [--image-opts | -f FMT] FILENAME

check [--object OBJECTDEF] [--image-opts] [-q] [-f FMT] [--output=OFMT] [-r [leaks | all]] [-T SRC_CACH

commit [--object OBJECTDEF] [--image-opts] [-q] [-f FMT] [-t CACHE] [-b BASE] [-r RATE_LIMIT] [-d] [-p]

compare [--object OBJECTDEF] [--image-opts] [-f FMT] [-F FMT] [-T SRC_CACHE] [-p] [-q] [-s] [-U] FILENA

convert [--object OBJECTDEF] [--image-opts] [--target-image-opts] [--target-is-zero] [--bitmaps] [-U] [
..]] OUTPUT_FILENAME

create [--object OBJECTDEF] [-q] [-f FMT] [-b BACKING_FILE [-F BACKING_FMT]] [-u] [-o OPTIONS] FILENAME

dd [--image-opts] [-U] [-f FMT] [-O OUTPUT_FMT] [bs=BLOCK_SIZE] [count=BLOCKS] [skip=BLOCKS] if=INPUT o

info [--object OBJECTDEF] [--image-opts] [-f FMT] [--output=OFMT] [--backing-chain] [-U] FILENAME

map [--object OBJECTDEF] [--image-opts] [-f FMT] [--start-offset=OFFSET] [--max-length=LEN] [--output=O

measure [--output=OFMT] [-O OUTPUT_FMT] [-o OPTIONS] [--size N | [--object OBJECTDEF] [--image-opts] [-

snapshot [--object OBJECTDEF] [--image-opts] [-U] [-q] [-l | -a SNAPSHOT | -c SNAPSHOT | -d SNAPSHOT] F

rebase [--object OBJECTDEF] [--image-opts] [-U] [-q] [-f FMT] [-t CACHE] [-T SRC_CACHE] [-p] [-u] [-c]

resize [--object OBJECTDEF] [--image-opts] [-f FMT] [--preallocation=PREALLOC] [-q] [--shrink] FILENAME

Command parameters:

FILENAME is a disk image filename.

FMT is the disk image format. It is guessed automatically in most cases. See below for a description of the supported disk formats.

SIZE is the disk image size in bytes. Optional suffixes `k` or `K` (kilobyte, 1024) `M` (megabyte, 1024k) and `G` (gigabyte, 1024M) and T (terabyte, 1024G) are supported. `b` is ignored.

OUTPUT_FILENAME is the destination disk image filename.

OUTPUT_FMT is the destination format.

OPTIONS is a comma separated list of format specific options in a name=value format. Use `-o help` for an overview of the options supported by the used format or see the format descriptions below for details.

SNAPSHOT_PARAM is param used for internal snapshot, format is 'snapshot.id=[ID],snapshot.name=[NAME]' or '[ID_OR_NAME]'.

--object OBJECTDEF

is a QEMU user creatable object definition. See the *qemu(1)* manual page for a description of the object properties. The most common object type is a `secret`, which is used to supply passwords and/or encryption keys.

--image-opts

Indicates that the source *FILENAME* parameter is to be interpreted as a full option string, not a plain filename. This parameter is mutually exclusive with the *-f* parameter.

--target-image-opts

Indicates that the OUTPUT_FILENAME parameter(s) are to be interpreted as a full option string, not a plain filename. This parameter is mutually exclusive with the *-O* parameters. It is currently required to also use the *-n* parameter to skip image creation. This restriction may be relaxed in a future release.

--force-share (-U)

If specified, `qemu-img` will open the image in shared mode, allowing other QEMU processes to open it in write mode. For example, this can be used to get the image information (with 'info' subcommand) when the image is used by a running guest. Note that this could produce inconsistent results because of concurrent metadata changes, etc. This option is only allowed when opening images in read-only mode.

--backing-chain

Will enumerate information about backing files in a disk image chain. Refer below for further description.

-c

Indicates that target image must be compressed (qcow/qcow2 and vmdk with streamOptimized subformat only).

For qcow2, the compression algorithm can be specified with the `-o compression_type=...` option (see below).

-h

With or without a command, shows help and lists the supported formats.

-p

Display progress bar (compare, convert and rebase commands only). If the *-p* option is not used for a command that supports it, the progress is reported when the process receives a `SIGUSR1` or `SIGINFO` signal.

-q

Quiet mode - do not print any output (except errors). There's no progress bar in case both *-q* and *-p* options are used.

-S SIZE

Indicates the consecutive number of bytes that must contain only zeros for `qemu-img` to create a sparse image during conversion. This value is rounded down to the nearest 512 bytes. You may use the common size suffixes like `k` for kilobytes.

-t CACHE

Specifies the cache mode that should be used with the (destination) file. See the documentation of the emulator's `-drive cache=...` option for allowed values.

-T SRC_CACHE

Specifies the cache mode that should be used with the source file(s). See the documentation of the emulator's `-drive cache=...` option for allowed values.

Parameters to compare subcommand:

-f

First image format

-F

Second image format

-s

Strict mode - fail on different image size or sector allocation

Parameters to convert subcommand:

--bitmaps

Additionally copy all persistent bitmaps from the top layer of the source

-n

Skip the creation of the target volume

-m

Number of parallel coroutines for the convert process

-W

Allow out-of-order writes to the destination. This option improves performance, but is only recommended for preallocated devices like host devices or other raw block devices.

-C

Try to use copy offloading to move data from source image to target. This may improve performance if the data is remote, such as with NFS or iSCSI backends, but will not automatically sparsify zero sectors, and may result in a fully allocated target image depending on the host support for getting allocation information.

-r

Rate limit for the convert process

--salvage

Try to ignore I/O errors when reading. Unless in quiet mode (`-q`), errors will still be printed. Areas that cannot be read from the source will be treated as containing only zeroes.

--target-is-zero

Assume that reading the destination image will always return zeros. This parameter is mutually exclusive with a destination image that has a backing file. It is required to also use the `-n` parameter to skip image creation.

Parameters to dd subcommand:

bs=BLOCK_SIZE

Defines the block size

count=BLOCKS

Sets the number of input blocks to copy

if=INPUT

Sets the input file

of=OUTPUT

Sets the output file

skip=BLOCKS

Sets the number of input blocks to skip

Parameters to snapshot subcommand:

snapshot

Is the name of the snapshot to create, apply or delete

-a

Applies a snapshot (revert disk to saved state)

-c

Creates a snapshot

-d

Deletes a snapshot

-l

Lists all snapshots in the given image

Command description:

amend [--object OBJECTDEF] [--image-opts] [-p] [-q] [-f FMT] [-t CACHE] [--force] -o OPTIONS FILENAME

Amends the image format specific *OPTIONS* for the image file *FILENAME*. Not all file formats support this operation.

The set of options that can be amended are dependent on the image format, but note that amending the backing chain relationship should instead be performed with `qemu-img rebase`.

–force allows some unsafe operations. Currently for -f luks, it allows to erase the last encryption key, and to overwrite an active encryption key.

bench [-c COUNT] [-d DEPTH] [-f FMT] [--flush-interval=FLUSH_INTERVAL] [-i AIO] [-n] [--no-drain] [-o O

Run a simple sequential I/O benchmark on the specified image. If `-w` is specified, a write test is performed, otherwise a read test is performed.

A total number of *COUNT* I/O requests is performed, each *BUFFER_SIZE* bytes in size, and with *DEPTH* requests in parallel. The first request starts at the position given by *OFFSET*, each following request increases the current position by *STEP_SIZE*. If *STEP_SIZE* is not given, *BUFFER_SIZE* is used for its value.

If *FLUSH_INTERVAL* is specified for a write test, the request queue is drained and a flush is issued before new writes are made whenever the number of remaining requests is a multiple of *FLUSH_INTERVAL*. If additionally `--no-drain` is specified, a flush is issued without draining the request queue first.

if `-i` is specified, *AIO* option can be used to specify different AIO backends: `threads`, `native` or `io_uring`.

If `-n` is specified, the native AIO backend is used if possible. On Linux, this option only works if `-t none` or `-t directsync` is specified as well.

For write tests, by default a buffer filled with zeros is written. This can be overridden with a pattern byte specified by *PATTERN*.

bitmap (--merge SOURCE | --add | --remove | --clear | --enable | --disable)...
[-b SOURCE_FILE [-F SOURCE_FMT]] [-g GRANULARITY] [--object OBJECTDEF] [--image-opts | -f FMT] FILENAME

Perform one or more modifications of the persistent bitmap *BITMAP* in the disk image *FILENAME*. The various modifications are:

`--add` to create *BITMAP*, enabled to record future edits.

`--remove` to remove *BITMAP*.

`--clear` to clear *BITMAP*.

`--enable` to change *BITMAP* to start recording future edits.

`--disable` to change *BITMAP* to stop recording future edits.

`--merge` to merge the contents of the *SOURCE* bitmap into *BITMAP*.

Additional options include `-g` which sets a non-default *GRANULARITY* for `--add`, and `-b` and `-F` which select an alternative source file for all *SOURCE* bitmaps used by `--merge`.

To see what bitmaps are present in an image, use `qemu-img info`.

check [--object OBJECTDEF] [--image-opts] [-q] [-f FMT] [--output=OFMT] [-r [leaks | all]] [-T SRC_CACH

Perform a consistency check on the disk image *FILENAME*. The command can output in the format *OFMT* which is either `human` or `json`. The JSON output is an object of QAPI type `ImageCheck`.

If `-r` is specified, qemu-img tries to repair any inconsistencies found during the check. `-r leaks` repairs only cluster leaks, whereas `-r all` fixes all kinds of errors, with a higher risk of choosing the wrong fix or hiding corruption that has already occurred.

Only the formats `qcow2`, `qed`, `parallels`, `vhdx`, `vmdk` and `vdi` support consistency checks.

In case the image does not have any inconsistencies, check exits with `0`. Other exit codes indicate the kind of inconsistency found or if another error occurred. The following table summarizes all exit codes of the check subcommand:

0
Check completed, the image is (now) consistent

1
Check not completed because of internal errors

2
Check completed, image is corrupted

3
Check completed, image has leaked clusters, but is not corrupted

63
Checks are not supported by the image format

If `-r` is specified, exit codes representing the image state refer to the state after (the attempt at) repairing it. That is, a successful `-r all` will yield the exit code 0, independently of the image state before.

commit [--object OBJECTDEF] [--image-opts] [-q] [-f FMT] [-t CACHE] [-b BASE] [-r RATE_LIMIT] [-d] [-p]

Commit the changes recorded in *FILENAME* in its base image or backing file. If the backing file is smaller than the snapshot, then the backing file will be resized to be the same size as the snapshot. If the snapshot is smaller than the backing file, the backing file will not be truncated. If you want the backing file to match the size of the smaller snapshot, you can safely truncate it yourself once the commit operation successfully completes.

The image *FILENAME* is emptied after the operation has succeeded. If you do not need *FILENAME* afterwards and intend to drop it, you may skip emptying *FILENAME* by specifying the `-d` flag.

If the backing chain of the given image file *FILENAME* has more than one layer, the backing file into which the changes will be committed may be specified as *BASE* (which has to be part of *FILENAME*'s backing chain). If *BASE* is not specified, the immediate backing file of the top image (which is *FILENAME*) will be used. Note that after a commit operation all images between *BASE* and the top image will be invalid and may return garbage data when read. For this reason, `-b` implies `-d` (so that the top image stays valid).

The rate limit for the commit process is specified by `-r`.

compare [--object OBJECTDEF] [--image-opts] [-f FMT] [-F FMT] [-T SRC_CACHE] [-p] [-q] [-s] [-U] FILENA

Check if two images have the same content. You can compare images with different format or settings.

The format is probed unless you specify it by `-f` (used for *FILENAME1*) and/or `-F` (used for *FILENAME2*) option.

By default, images with different size are considered identical if the larger image contains only unallocated and/or zeroed sectors in the area after the end of the other image. In addition, if any sector is not allocated in one image and contains only zero bytes in the second one, it is evaluated as equal. You can use Strict mode by specifying the `-s` option. When compare runs in Strict mode, it fails in case image size differs or a sector is allocated in one image and is not allocated in the second one.

By default, compare prints out a result message. This message displays information that both images are same or the position of the first different byte. In addition, result message can report different image size in case Strict mode is used.

Compare exits with `0` in case the images are equal and with `1` in case the images differ. Other exit codes mean an error occurred during execution and standard error output should contain an error message. The following table summarizes all exit codes of the compare subcommand:

0
: Images are identical (or requested help was printed)

1
: Images differ

2
: Error on opening an image

3
: Error on checking a sector allocation

4
: Error on reading data

convert `[--object OBJECTDEF] [--image-opts] [--target-image-opts] [--target-is-zero] [--bitmaps [--skip-`
`..]] OUTPUT_FILENAME`

Convert the disk image *FILENAME* or a snapshot *SNAPSHOT_PARAM* to disk image *OUTPUT_FILENAME* using format *OUTPUT_FMT*. It can be optionally compressed (`-c` option) or use any format specific options like encryption (`-o` option).

Only the formats `qcow` and `qcow2` support compression. The compression is read-only. It means that if a compressed sector is rewritten, then it is rewritten as uncompressed data.

Image conversion is also useful to get smaller image when using a growable format such as `qcow`: the empty sectors are detected and suppressed from the destination image.

SPARSE_SIZE indicates the consecutive number of bytes (defaults to 4k) that must contain only zeros for `qemu-img` to create a sparse image during conversion. If *SPARSE_SIZE* is 0, the source will not be scanned for unallocated or zero sectors, and the destination image will always be fully allocated.

You can use the *BACKING_FILE* option to force the output image to be created as a copy on write image of the specified base image; the *BACKING_FILE* should have the same content as the input's base image, however the path, image format (as given by *BACKING_FMT*), etc may differ.

If a relative path name is given, the backing file is looked up relative to the directory containing *OUTPUT_FILENAME*.

If the `-n` option is specified, the target volume creation will be skipped. This is useful for formats such as `rbd` if the target volume has already been created with site specific options that cannot be supplied through `qemu-img`.

Out of order writes can be enabled with `-W` to improve performance. This is only recommended for preallocated devices like host devices or other raw block devices. Out of order write does not work in combination with creating compressed images.

NUM_COROUTINES specifies how many coroutines work in parallel during the convert process (defaults to 8).

Use of `--bitmaps` requests that any persistent bitmaps present in the original are also copied to the destination. If any bitmap is inconsistent in the source, the conversion will fail unless `--skip-broken-bitmaps` is also specified to copy only the consistent bitmaps.

create `[--object OBJECTDEF] [-q] [-f FMT] [-b BACKING_FILE [-F BACKING_FMT]] [-u] [-o OPTIONS] FILENAME`

Create the new disk image *FILENAME* of size *SIZE* and format *FMT*. Depending on the file format, you can add one or more *OPTIONS* that enable additional features of this format.

If the option *BACKING_FILE* is specified, then the image will record only the differences from *BACKING_FILE*. No size needs to be specified in this case. *BACKING_FILE* will never be modified unless you use the `commit` monitor command (or `qemu-img commit`).

If a relative path name is given, the backing file is looked up relative to the directory containing *FILENAME*.

Note that a given backing file will be opened to check that it is valid. Use the `-u` option to enable unsafe backing file mode, which means that the image will be created even if the associated backing file cannot be opened. A matching backing file must be created or additional options be used to make the backing file specification valid when you want to use an image created this way.

The size can also be specified using the *SIZE* option with `-o`, it doesn't need to be specified separately in this case.

dd `[--image-opts] [-U] [-f FMT] [-O OUTPUT_FMT] [bs=BLOCK_SIZE] [count=BLOCKS] [skip=BLOCKS] if=INPUT o`

dd copies from *INPUT* file to *OUTPUT* file converting it from *FMT* format to *OUTPUT_FMT* format.

The data is by default read and written using blocks of 512 bytes but can be modified by specifying *BLOCK_SIZE*. If count=*BLOCKS* is specified dd will stop reading input after reading *BLOCKS* input blocks.

The size syntax is similar to *dd(1)*'s size syntax.

info `[--object OBJECTDEF] [--image-opts] [-f FMT] [--output=OFMT] [--backing-chain] [-U] FILENAME`

Give information about the disk image *FILENAME*. Use it in particular to know the size reserved on disk which can be different from the displayed size. If VM snapshots are stored in the disk image, they are displayed too.

If a disk image has a backing file chain, information about each disk image in the chain can be recursively enumerated by using the option `--backing-chain`.

For instance, if you have an image chain like:

```
base.qcow2 <- snap1.qcow2 <- snap2.qcow2
```

To enumerate information about each disk image in the above chain, starting from top to base, do:

```
qemu-img info --backing-chain snap2.qcow2
```

The command can output in the format *OFMT* which is either `human` or `json`. The JSON output is an object of QAPI type `ImageInfo`; with `--backing-chain`, it is an array of `ImageInfo` objects.

`--output=human` reports the following information (for every image in the chain):

image
The image file name

file format
The image format

virtual size
The size of the guest disk

disk size
How much space the image file occupies on the host file system (may be shown as 0 if this information is unavailable, e.g. because there is no file system)

cluster_size
Cluster size of the image format, if applicable

encrypted
Whether the image is encrypted (only present if so)

cleanly shut down
This is shown as no if the image is dirty and will have to be auto-repaired the next time it is opened in qemu.

backing file
The backing file name, if present

backing file format
The format of the backing file, if the image enforces it

Snapshot list
A list of all internal snapshots

Format specific information
Further information whose structure depends on the image format. This section is a textual representation of the respective ImageInfoSpecific* QAPI object (e.g. ImageInfoSpecificQCow2 for qcow2 images).

map [--object OBJECTDEF] [--image-opts] [-f FMT] [--start-offset=OFFSET] [--max-length=LEN] [--output=O

Dump the metadata of image *FILENAME* and its backing file chain. In particular, this commands dumps the allocation state of every sector of *FILENAME*, together with the topmost file that allocates it in the backing file chain.

Two option formats are possible. The default format (human) only dumps known-nonzero areas of the file. Known-zero parts of the file are omitted altogether, and likewise for parts that are not allocated throughout the chain. qemu-img output will identify a file from where the data can be read, and the offset in the file. Each line will include four fields, the first three of which are hexadecimal numbers. For example the first line of:

Offset	Length	Mapped to	File
0	0x20000	0x50000	/tmp/overlay.qcow2
0x100000	0x10000	0x95380000	/tmp/backing.qcow2

means that 0x20000 (131072) bytes starting at offset 0 in the image are available in /tmp/overlay.qcow2 (opened in raw format) starting at offset 0x50000 (327680). Data that is compressed, encrypted, or otherwise not available in raw format will cause an error if human format is in use. Note that file names can include newlines, thus it is not safe to parse this output format in scripts.

The alternative format json will return an array of dictionaries in JSON format. It will include similar information in the start, length, offset fields; it will also include other more specific information:

- boolean field data: true if the sectors contain actual data, false if the sectors are either unallocated or stored as optimized all-zero clusters
- boolean field zero: true if the data is known to read as zero
- boolean field present: true if the data belongs to the backing chain, false if rebasing the backing chain onto a deeper file would pick up data from the deeper file;
- integer field depth: the depth within the backing chain at which the data was resolved; for example, a depth of 2 refers to the backing file of the backing file of *FILENAME*.

In JSON format, the offset field is optional; it is absent in cases where human format would omit the entry or exit with an error. If data is false and the offset field is present, the corresponding sectors in the file are not yet in use, but they are preallocated.

For more information, consult include/block/block.h in QEMU's source code.

measure [--output=OFMT] [-O OUTPUT_FMT] [-o OPTIONS] [--size N | [--object OBJECTDEF] [--image-opts] [-

Calculate the file size required for a new image. This information can be used to size logical volumes or SAN LUNs appropriately for the image that will be placed in them. The values reported are guaranteed to be large enough to fit the image. The command can output in the format *OFMT* which is either human or json. The JSON output is an object of QAPI type BlockMeasureInfo.

If the size *N* is given then act as if creating a new empty image file using `qemu-img create`. If *FILENAME* is given then act as if converting an existing image file using `qemu-img convert`. The format of the new file is given by *OUTPUT_FMT* while the format of an existing file is given by *FMT*.

A snapshot in an existing image can be specified using *SNAPSHOT_PARAM*.

The following fields are reported:

```
required size: 524288
fully allocated size: 1074069504
bitmaps size: 0
```

The `required size` is the file size of the new image. It may be smaller than the virtual disk size if the image format supports compact representation.

The `fully allocated size` is the file size of the new image once data has been written to all sectors. This is the maximum size that the image file can occupy with the exception of internal snapshots, dirty bitmaps, vmstate data, and other advanced image format features.

The `bitmaps size` is the additional size required in order to copy bitmaps from a source image in addition to the guest-visible data; the line is omitted if either source or destination lacks bitmap support, or 0 if bitmaps are supported but there is nothing to copy.

snapshot `[--object OBJECTDEF] [--image-opts] [-U] [-q] [-l | -a SNAPSHOT | -c SNAPSHOT | -d SNAPSHOT] FI`

List, apply, create or delete snapshots in image *FILENAME*.

rebase `[--object OBJECTDEF] [--image-opts] [-U] [-q] [-f FMT] [-t CACHE] [-T SRC_CACHE] [-p] [-u] [-c]`

Changes the backing file of an image. Only the formats `qcow2` and `qed` support changing the backing file.

The backing file is changed to *BACKING_FILE* and (if the image format of *FILENAME* supports this) the backing file format is changed to *BACKING_FMT*. If *BACKING_FILE* is specified as "" (the empty string), then the image is rebased onto no backing file (i.e. it will exist independently of any backing file).

If a relative path name is given, the backing file is looked up relative to the directory containing *FILENAME*.

CACHE specifies the cache mode to be used for *FILENAME*, whereas *SRC_CACHE* specifies the cache mode for reading backing files.

There are two different modes in which `rebase` can operate:

Safe mode
: This is the default mode and performs a real rebase operation. The new backing file may differ from the old one and `qemu-img rebase` will take care of keeping the guest-visible content of *FILENAME* unchanged.

  In order to achieve this, any clusters that differ between *BACKING_FILE* and the old backing file of *FILENAME* are merged into *FILENAME* before actually changing the backing file. With the `-c` option specified, the clusters which are being merged (but not the entire *FILENAME* image) are compressed when written.

  Note that the safe mode is an expensive operation, comparable to converting an image. It only works if the old backing file still exists.

Unsafe mode
: `qemu-img` uses the unsafe mode if `-u` is specified. In this mode, only the backing file name and format of *FILENAME* is changed without any checks on the file contents. The user must take care of specifying the correct new backing file, or the guest-visible content of the image will be corrupted.

  This mode is useful for renaming or moving the backing file to somewhere else. It can be used without an accessible old backing file, i.e. you can use it to fix an image whose backing file has already been moved/renamed.

You can use `rebase` to perform a "diff" operation on two disk images. This can be useful when you have copied or cloned a guest, and you want to get back to a thin image on top of a template or base image.

Say that base.img has been cloned as modified.img by copying it, and that the modified.img guest has run so there are now some changes compared to base.img. To construct a thin image called diff.qcow2 that contains just the differences, do:

```
qemu-img create -f qcow2 -b modified.img diff.qcow2
qemu-img rebase -b base.img diff.qcow2
```

At this point, modified.img can be discarded, since base.img + diff.qcow2 contains the same information.

resize [--object OBJECTDEF] [--image-opts] [-f FMT] [--preallocation=PREALLOC] [-q] [--shrink] FILENAME

Change the disk image as if it had been created with *SIZE*.

Before using this command to shrink a disk image, you MUST use file system and partitioning tools inside the VM to reduce allocated file systems and partition sizes accordingly. Failure to do so will result in data loss!

When shrinking images, the --shrink option must be given. This informs qemu-img that the user acknowledges all loss of data beyond the truncated image's end.

After using this command to grow a disk image, you must use file system and partitioning tools inside the VM to actually begin using the new space on the device.

When growing an image, the --preallocation option may be used to specify how the additional image area should be allocated on the host. See the format description in the *Notes* section which values are allowed. Using this option may result in slightly more data being allocated than necessary.

4.1.4 Notes

Supported image file formats:

raw

Raw disk image format (default). This format has the advantage of being simple and easily exportable to all other emulators. If your file system supports *holes* (for example in ext2 or ext3 on Linux or NTFS on Windows), then only the written sectors will reserve space. Use qemu-img info to know the real size used by the image or ls -ls on Unix/Linux.

Supported options:

preallocation
Preallocation mode (allowed values: off, falloc, full). falloc mode preallocates space for image by calling posix_fallocate(). full mode preallocates space for image by writing data to underlying storage. This data may or may not be zero, depending on the storage location.

qcow2

QEMU image format, the most versatile format. Use it to have smaller images (useful if your filesystem does not supports holes, for example on Windows), optional AES encryption, zlib or zstd based compression and support of multiple VM snapshots.

Supported options:

compat
Determines the qcow2 version to use. compat=0.10 uses the traditional image format that can be read by any QEMU since 0.10. compat=1.1 enables image format extensions that only QEMU 1.1 and newer understand (this is the default). Amongst others, this includes zero clusters, which allow efficient copy-on-read for sparse images.

backing_file
File name of a base image (see create subcommand)

backing_fmt
: Image format of the base image

compression_type
: This option configures which compression algorithm will be used for compressed clusters on the image. Note that setting this option doesn't yet cause the image to actually receive compressed writes. It is most commonly used with the `-c` option of `qemu-img convert`, but can also be used with the `compress` filter driver or backup block jobs with compression enabled.

  Valid values are `zlib` and `zstd`. For images that use `compat=0.10`, only `zlib` compression is available.

encryption
: If this option is set to `on`, the image is encrypted with 128-bit AES-CBC.

  The use of encryption in qcow and qcow2 images is considered to be flawed by modern cryptography standards, suffering from a number of design problems:

  - The AES-CBC cipher is used with predictable initialization vectors based on the sector number. This makes it vulnerable to chosen plaintext attacks which can reveal the existence of encrypted data.
  - The user passphrase is directly used as the encryption key. A poorly chosen or short passphrase will compromise the security of the encryption.
  - In the event of the passphrase being compromised there is no way to change the passphrase to protect data in any qcow images. The files must be cloned, using a different encryption passphrase in the new file. The original file must then be securely erased using a program like shred, though even this is ineffective with many modern storage technologies.
  - Initialization vectors used to encrypt sectors are based on the guest virtual sector number, instead of the host physical sector. When a disk image has multiple internal snapshots this means that data in multiple physical sectors is encrypted with the same initialization vector. With the CBC mode, this opens the possibility of watermarking attacks if the attack can collect multiple sectors encrypted with the same IV and some predictable data. Having multiple qcow2 images with the same passphrase also exposes this weakness since the passphrase is directly used as the key.

  Use of qcow / qcow2 encryption is thus strongly discouraged. Users are recommended to use an alternative encryption technology such as the Linux dm-crypt / LUKS system.

cluster_size
: Changes the qcow2 cluster size (must be between 512 and 2M). Smaller cluster sizes can improve the image file size whereas larger cluster sizes generally provide better performance.

preallocation
: Preallocation mode (allowed values: `off`, `metadata`, `falloc`, `full`). An image with preallocated metadata is initially larger but can improve performance when the image needs to grow. `falloc` and `full` preallocations are like the same options of `raw` format, but sets up metadata also.

lazy_refcounts
: If this option is set to `on`, reference count updates are postponed with the goal of avoiding metadata I/O and improving performance. This is particularly interesting with `cache=writethrough` which doesn't batch metadata updates. The tradeoff is that after a host crash, the reference count tables must be rebuilt, i.e. on the next open an (automatic) `qemu-img check -r all` is required, which may take some time.

  This option can only be enabled if `compat=1.1` is specified.

nocow
: If this option is set to `on`, it will turn off COW of the file. It's only valid on btrfs, no effect on other file systems.

Btrfs has low performance when hosting a VM image file, even more when the guest on the VM also using btrfs as file system. Turning off COW is a way to mitigate this bad performance. Generally there are two ways to turn off COW on btrfs:

- Disable it by mounting with nodatacow, then all newly created files will be NOCOW
- For an empty file, add the NOCOW file attribute. That's what this option does.

Note: this option is only valid to new or empty files. If there is an existing file which is COW and has data blocks already, it couldn't be changed to NOCOW by setting `nocow=on`. One can issue `lsattr filename` to check if the NOCOW flag is set or not (Capital 'C' is NOCOW flag).

data_file
Filename where all guest data will be stored. If this option is used, the qcow2 file will only contain the image's metadata.

Note: Data loss will occur if the given filename already exists when using this option with `qemu-img create` since `qemu-img` will create the data file anew, overwriting the file's original contents. To simply update the reference to point to the given pre-existing file, use `qemu-img amend`.

data_file_raw
If this option is set to `on`, QEMU will always keep the external data file consistent as a standalone read-only raw image.

It does this by forwarding all write accesses to the qcow2 file through to the raw data file, including their offsets. Therefore, data that is visible on the qcow2 node (i.e., to the guest) at some offset is visible at the same offset in the raw data file. This results in a read-only raw image. Writes that bypass the qcow2 metadata may corrupt the qcow2 metadata because the out-of-band writes may result in the metadata falling out of sync with the raw image.

If this option is `off`, QEMU will use the data file to store data in an arbitrary manner. The file's content will not make sense without the accompanying qcow2 metadata. Where data is written will have no relation to its offset as seen by the guest, and some writes (specifically zero writes) may not be forwarded to the data file at all, but will only be handled by modifying qcow2 metadata.

This option can only be enabled if `data_file` is set.

`Other`

QEMU also supports various other image file formats for compatibility with older QEMU versions or other hypervisors, including VMDK, VDI, VHD (vpc), VHDX, qcow1 and QED. For a full list of supported formats see `qemu-img --help`. For a more detailed description of these formats, see the QEMU block drivers reference documentation.

The main purpose of the block drivers for these formats is image conversion. For running VMs, it is recommended to convert the disk images to either raw or qcow2 in order to achieve good performance.

4.2 QEMU Storage Daemon

4.2.1 Synopsis

qemu-storage-daemon [options]

4.2.2 Description

qemu-storage-daemon provides disk image functionality from QEMU, qemu-img, and qemu-nbd in a long-running process controlled via QMP commands without running a virtual machine. It can export disk images, run block job operations, and perform other disk-related operations. The daemon is controlled via a QMP monitor and initial configuration from the command-line.

The daemon offers the following subset of QEMU features:

- Block nodes
- Block jobs
- Block exports
- Throttle groups
- Character devices
- Crypto and secrets
- QMP
- IOThreads

Commands can be sent over a QEMU Monitor Protocol (QMP) connection. See the *qemu-storage-daemon-qmp-ref(7)* manual page for a description of the commands.

The daemon runs until it is stopped using the quit QMP command or SIGINT/SIGHUP/SIGTERM.

Warning: Never modify images in use by a running virtual machine or any other process; this may destroy the image. Also, be aware that querying an image that is being modified by another process may encounter inconsistent state.

4.2.3 Options

Standard options:

-h, --help
Display help and exit

-V, --version
Display version information and exit

-T, --trace [[enable=]PATTERN][,events=FILE][,file=FILE]
Specify tracing options.

[enable=]PATTERN

Immediately enable events matching *PATTERN* (either event name or a globbing pattern). This option is only available if QEMU has been compiled with the simple, log or ftrace tracing backend. To specify multiple events or patterns, specify the -trace option multiple times.

Use -trace help to print a list of names of trace points.

events=FILE

Immediately enable events listed in *FILE*. The file must contain one event name (as listed in the trace-events-all file) per line; globbing patterns are accepted too. This option is only available if QEMU has been compiled with the simple, log or ftrace tracing backend.

file=FILE

Log output traces to *FILE*. This option is only available if QEMU has been compiled with the `simple` tracing backend.

--blockdev BLOCKDEVDEF

is a block node definition. See the *qemu(1)* manual page for a description of block node properties and the *qemu-block-drivers(7)* manual page for a description of driver-specific parameters.

--chardev CHARDEVDEF

is a character device definition. See the *qemu(1)* manual page for a description of character device properties. A common character device definition configures a UNIX domain socket:

```
--chardev socket,id=char1,path=/var/run/qsd-qmp.sock,server=on,wait=off
```

--export [type=]nbd,id=<id>,node-name=<node-name>[,name=<export-name>][,
writable=on|off][,bitmap=<name>]

--export [type=]vhost-user-blk,id=<id>,node-name=<node-name>,addr.type=unix,addr.
path=<socket-path>[,writable=on|off][,logical-block-size=<block-size>][,
num-queues=<num-queues>]

--export [type=]vhost-user-blk,id=<id>,node-name=<node-name>,addr.type=fd,addr.str=<fd>[,
writable=on|off][,logical-block-size=<block-size>][,num-queues=<num-queues>]

--export [type=]fuse,id=<id>,node-name=<node-name>,mountpoint=<file>[,growable=on|off][,
writable=on|off][,allow-other=on|off|auto]

--export [type=]vduse-blk,id=<id>,node-name=<node-name>,name=<vduse-name>[,
writable=on|off][,num-queues=<num-queues>][,queue-size=<queue-size>][,
logical-block-size=<block-size>][,serial=<serial-number>]

is a block export definition. `node-name` is the block node that should be exported. `writable` determines whether or not the export allows write requests for modifying data (the default is off).

The `nbd` export type requires `--nbd-server` (see below). `name` is the NBD export name (if not specified, it defaults to the given `node-name`). `bitmap` is the name of a dirty bitmap reachable from the block node, so the NBD client can use NBD_OPT_SET_META_CONTEXT with the metadata context name "qemu:dirty-bitmap:BITMAP" to inspect the bitmap.

The `vhost-user-blk` export type takes a vhost-user socket address on which it accept incoming connections. Both `addr.type=unix,addr.path=<socket-path>` for UNIX domain sockets and `addr.type=fd,addr.str=<fd>` for file descriptor passing are supported. `logical-block-size` sets the logical block size in bytes (the default is 512). `num-queues` sets the number of virtqueues (the default is 1).

The `fuse` export type takes a mount point, which must be a regular file, on which to export the given block node. That file will not be changed, it will just appear to have the block node's content while the export is active (very much like mounting a filesystem on a directory does not change what the directory contains, it only shows a different content while the filesystem is mounted). Consequently, applications that have opened the given file before the export became active will continue to see its original content. If `growable` is set, writes after the end of the exported file will grow the block node to fit. The `allow-other` option controls whether users other than the user running the process will be allowed to access the export. Note that enabling this option as a non-root user requires enabling the user_allow_other option in the global fuse.conf configuration file. Setting `allow-other` to auto (the default) will try enabling this option, and on error fall back to disabling it.

The `vduse-blk` export type takes a `name` (must be unique across the host) to create the VDUSE device. `num-queues` sets the number of virtqueues (the default is 1). `queue-size` sets the virtqueue descriptor table size (the default is 256).

The instantiated VDUSE device must then be added to the vDPA bus using the vdpa(8) command from the iproute2 project:

```
# vdpa dev add name <id> mgmtdev vduse
```

The device can be removed from the vDPA bus later as follows:

```
# vdpa dev del <id>
```

For more information about attaching vDPA devices to the host with virtio_vdpa.ko or attaching them to guests with vhost_vdpa.ko, see https://vdpa-dev.gitlab.io/.

For more information about VDUSE, see https://docs.kernel.org/userspace-api/vduse.html.

--monitor MONITORDEF

is a QMP monitor definition. See the *qemu(1)* manual page for a description of QMP monitor properties. A common QMP monitor definition configures a monitor on character device `char1`:

```
--monitor chardev=char1
```

--nbd-server addr.type=inet,addr.host=<host>,addr.port=<port>[,tls-creds=<id>][,tls-authz=<id>][,max-connections=<n>]

--nbd-server addr.type=unix,addr.path=<path>[,tls-creds=<id>][,tls-authz=<id>][,max-connections=<n>]

--nbd-server addr.type=fd,addr.str=<fd>[,tls-creds=<id>][,tls-authz=<id>][,max-connections=<n>]

is a server for NBD exports. Both TCP and UNIX domain sockets are supported. A listen socket can be provided via file descriptor passing (see Examples below). TLS encryption can be configured using `--object` tls-creds-* and authz-* secrets (see below).

To configure an NBD server on UNIX domain socket path `/var/run/qsd-nbd.sock`:

```
--nbd-server addr.type=unix,addr.path=/var/run/qsd-nbd.sock
```

--object help

--object <type>,help

--object <type>[,<property>=<value>...]

is a QEMU user creatable object definition. List object types with `help`. List object properties with `<type>,help`. See the *qemu(1)* manual page for a description of the object properties.

--pidfile PATH

is the path to a file where the daemon writes its pid. This allows scripts to stop the daemon by sending a signal:

```
$ kill -SIGTERM $(<path/to/qsd.pid)
```

A file lock is applied to the file so only one instance of the daemon can run with a given pid file path. The daemon unlinks its pid file when terminating.

The pid file is written after chardevs, exports, and NBD servers have been created but before accepting connections. The daemon has started successfully when the pid file is written and clients may begin connecting.

--daemonize

Daemonize the process. The parent process will exit once startup is complete (i.e., after the pid file has been or would have been written) or failure occurs. Its exit code reflects whether the child has started up successfully or failed to do so.

4.2.4 Examples

Launch the daemon with QMP monitor socket `qmp.sock` so clients can execute QMP commands:

```
$ qemu-storage-daemon \
    --chardev socket,path=qmp.sock,server=on,wait=off,id=char1 \
    --monitor chardev=char1
```

Launch the daemon from Python with a QMP monitor socket using file descriptor passing so there is no need to busy wait for the QMP monitor to become available:

```
#!/usr/bin/env python3
import subprocess
import socket

sock_path = '/var/run/qmp.sock'

with socket.socket(socket.AF_UNIX, socket.SOCK_STREAM) as listen_sock:
    listen_sock.bind(sock_path)
    listen_sock.listen()

    fd = listen_sock.fileno()

    subprocess.Popen(
        ['qemu-storage-daemon',
         '--chardev', f'socket,fd={fd},server=on,id=char1',
         '--monitor', 'chardev=char1'],
        pass_fds=[fd],
    )

# listen_sock was automatically closed when leaving the 'with' statement
# body. If the daemon process terminated early then the following connect()
# will fail with "Connection refused" because no process has the listen
# socket open anymore. Launch errors can be detected this way.

qmp_sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
qmp_sock.connect(sock_path)
...QMP interaction...
```

The same socket spawning approach also works with the `--nbd-server addr.type=fd,addr.str=<fd>` and `--export type=vhost-user-blk,addr.type=fd,addr.str=<fd>` options.

Export raw image file `disk.img` over NBD UNIX domain socket `nbd.sock`:

```
$ qemu-storage-daemon \
    --blockdev driver=file,node-name=disk,filename=disk.img \
    --nbd-server addr.type=unix,addr.path=nbd.sock \
    --export type=nbd,id=export,node-name=disk,writable=on
```

Export a qcow2 image file `disk.qcow2` as a vhost-user-blk device over UNIX domain socket `vhost-user-blk.sock`:

```
$ qemu-storage-daemon \
    --blockdev driver=file,node-name=file,filename=disk.qcow2 \
    --blockdev driver=qcow2,node-name=qcow2,file=file \
```

(continues on next page)

(continued from previous page)

```
    --export type=vhost-user-blk,id=export,addr.type=unix,addr.path=vhost-user-blk.sock,
↪node-name=qcow2
```

Export a qcow2 image file `disk.qcow2` via FUSE on itself, so the disk image file will then appear as a raw image:

```
$ qemu-storage-daemon \
    --blockdev driver=file,node-name=file,filename=disk.qcow2 \
    --blockdev driver=qcow2,node-name=qcow2,file=file \
    --export type=fuse,id=export,node-name=qcow2,mountpoint=disk.qcow2,writable=on
```

4.2.5 See also

qemu(1), *qemu-block-drivers(7)*, *qemu-storage-daemon-qmp-ref(7)*

4.3 QEMU Disk Network Block Device Server

4.3.1 Synopsis

qemu-nbd [*OPTION*]... *filename*

qemu-nbd -L [*OPTION*]...

qemu-nbd -d *dev*

4.3.2 Description

Export a QEMU disk image using the NBD protocol.

Other uses:

- Bind a /dev/nbdX block device to a QEMU server (on Linux).
- As a client to query exports of a remote NBD server.

4.3.3 Options

filename is a disk image filename, or a set of block driver options if *--image-opts* is specified.

dev is an NBD device.

--object `type,id=ID,...`

Define a new instance of the *type* object class identified by *ID*. See the *qemu(1)* manual page for full details of the properties supported. The common object types that it makes sense to define are the `secret` object, which is used to supply passwords and/or encryption keys, and the `tls-creds` object, which is used to supply TLS credentials for the `qemu-nbd` server or client.

-p, --port=PORT

TCP port to listen on as a server, or connect to as a client (default `10809`).

-o, --offset=OFFSET

The offset into the image.

-b, --bind=IFACE

The interface to bind to as a server, or connect to as a client (default `0.0.0.0`).

-k, --socket=PATH

Use a unix socket with path *PATH*.

--image-opts

Treat *filename* as a set of image options, instead of a plain filename. If this flag is specified, the `-f` flag should not be used, instead the `format=` option should be set.

-f, --format=FMT

Force the use of the block driver for format *FMT* instead of auto-detecting.

-r, --read-only

Export the disk as read-only.

-A, --allocation-depth

Expose allocation depth information via the `qemu:allocation-depth` metadata context accessible through NBD_OPT_SET_META_CONTEXT.

-B, --bitmap=NAME

If *filename* has a qcow2 persistent bitmap *NAME*, expose that bitmap via the `qemu:dirty-bitmap:NAME` metadata context accessible through NBD_OPT_SET_META_CONTEXT.

-s, --snapshot

Use *filename* as an external snapshot, create a temporary file with `backing_file`=*filename*, redirect the write to the temporary one.

-l, --load-snapshot=SNAPSHOT_PARAM

Load an internal snapshot inside *filename* and export it as an read-only device, SNAPSHOT_PARAM format is `snapshot.id=[ID],snapshot.name=[NAME]` or `[ID_OR_NAME]`

--cache=CACHE

The cache mode to be used with the file. Valid values are: `none`, `writeback` (the default), `writethrough`, `directsync` and `unsafe`. See the documentation of the emulator's `-drive cache=...` option for more info.

-n, --nocache

Equivalent to *--cache=none*.

--aio=AIO

Set the asynchronous I/O mode between `threads` (the default), `native` (Linux only), and `io_uring` (Linux 5.1+).

--discard=DISCARD

Control whether `discard` (also known as `trim` or `unmap`) requests are ignored or passed to the filesystem. *DISCARD* is one of `ignore` (or `off`), `unmap` (or `on`). The default is `ignore`.

--detect-zeroes=DETECT_ZEROES

Control the automatic conversion of plain zero writes by the OS to driver-specific optimized zero write commands. *DETECT_ZEROES* is one of `off`, `on`, or `unmap`. `unmap` converts a zero write to an unmap operation and can only be used if *DISCARD* is set to `unmap`. The default is `off`.

-c, --connect=DEV

Connect *filename* to NBD device *DEV* (Linux only).

-d, --disconnect

Disconnect the device *DEV* (Linux only).

-e, --shared=NUM

Allow up to *NUM* clients to share the device (default 1), 0 for unlimited.

-t, --persistent

Don't exit on the last connection.

-x, --export-name=NAME

Set the NBD volume export name (default of a zero-length string).

-D, --description=DESCRIPTION

Set the NBD volume export description, as a human-readable string.

-L, --list

Connect as a client and list all details about the exports exposed by a remote NBD server. This enables list mode, and is incompatible with options that change behavior related to a specific export (such as *--export-name*, *--offset*, ...).

--tls-creds=ID

Enable mandatory TLS encryption for the server by setting the ID of the TLS credentials object previously created with the *--object* option; or provide the credentials needed for connecting as a client in list mode.

--tls-hostname=hostname

When validating an x509 certificate received over a TLS connection, the hostname that the NBD client used to connect will be checked against information in the server provided certificate. Sometimes it might be required to override the hostname used to perform this check. For example, if the NBD client is using a tunnel from localhost to connect to the remote server, the *--tls-hostname* option should be used to set the officially expected hostname of the remote NBD server. This can also be used if accessing NBD over a UNIX socket where there is no inherent hostname available. This is only permitted when acting as a NBD client with the *--list* option.

--fork

Fork off the server process and exit the parent once the server is running.

--pid-file=PATH

Store the server's process ID in the given file.

--tls-authz=ID

Specify the ID of a qauthz object previously created with the *--object* option. This will be used to authorize connecting users against their x509 distinguished name.

-v, --verbose

Display extra debugging information. This option also keeps the original *STDERR* stream open if the `qemu-nbd` process is daemonized due to other options like *--fork* or *-c*.

-h, --help

Display this help and exit.

-V, --version

Display version information and exit.

-T, --trace [[enable=]PATTERN][,events=FILE][,file=FILE]

Specify tracing options.

`[enable=]PATTERN`

Immediately enable events matching *PATTERN* (either event name or a globbing pattern). This option is only available if QEMU has been compiled with the `simple`, `log` or `ftrace` tracing backend. To specify multiple events or patterns, specify the `-trace` option multiple times.

Use `-trace help` to print a list of names of trace points.

`events=FILE`

Immediately enable events listed in *FILE*. The file must contain one event name (as listed in the `trace-events-all` file) per line; globbing patterns are accepted too. This option is only available if QEMU has been compiled with the `simple`, `log` or `ftrace` tracing backend.

`file=FILE`

Log output traces to *FILE*. This option is only available if QEMU has been compiled with the `simple` tracing backend.

4.3.4 Examples

Start a server listening on port 10809 that exposes only the guest-visible contents of a qcow2 file, with no TLS encryption, and with the default export name (an empty string). The command is one-shot, and will block until the first successful client disconnects:

```
qemu-nbd -f qcow2 file.qcow2
```

Start a long-running server listening with encryption on port 10810, and allow clients with a specific X.509 certificate to connect to a 1 megabyte subset of a raw file, using the export name 'subset':

```
qemu-nbd \
  --object tls-creds-x509,id=tls0,endpoint=server,dir=/path/to/qemutls \
  --object 'authz-simple,id=auth0,identity=CN=laptop.example.com,,\
            O=Example Org,,L=London,,ST=London,,C=GB' \
  --tls-creds tls0 --tls-authz auth0 \
  -t -x subset -p 10810 \
  --image-opts driver=raw,offset=1M,size=1M,file.driver=file,file.filename=file.raw
```

Serve a read-only copy of a guest image over a Unix socket with as many as 5 simultaneous readers, with a persistent process forked as a daemon:

```
qemu-nbd --fork --persistent --shared=5 --socket=/path/to/sock \
  --read-only --format=qcow2 file.qcow2
```

Expose the guest-visible contents of a qcow2 file via a block device /dev/nbd0 (and possibly creating /dev/nbd0p1 and friends for partitions found within), then disconnect the device when done. Access to bind `qemu-nbd` to a /dev/nbd device generally requires root privileges, and may also require the execution of `modprobe nbd` to enable the kernel NBD client module. *CAUTION*: Do not use this method to mount filesystems from an untrusted guest image - a malicious guest may have prepared the image to attempt to trigger kernel bugs in partition probing or file system mounting.

```
qemu-nbd -c /dev/nbd0 -f qcow2 file.qcow2
qemu-nbd -d /dev/nbd0
```

Query a remote server to see details about what export(s) it is serving on port 10809, and authenticating via PSK:

```
qemu-nbd \
  --object tls-creds-psk,id=tls0,dir=/tmp/keys,username=eblake,endpoint=client \
  --tls-creds tls0 -L -b remote.example.com
```

4.3.5 See also

qemu(1), *qemu-img(1)*

4.4 QEMU persistent reservation helper

4.4.1 Synopsis

qemu-pr-helper [*OPTION*]

4.4.2 Description

Implements the persistent reservation helper for QEMU.

SCSI persistent reservations allow restricting access to block devices to specific initiators in a shared storage setup. When implementing clustering of virtual machines, it is a common requirement for virtual machines to send persistent reservation SCSI commands. However, the operating system restricts sending these commands to unprivileged programs because incorrect usage can disrupt regular operation of the storage fabric. QEMU's SCSI passthrough devices `scsi-block` and `scsi-generic` support passing guest persistent reservation requests to a privileged external helper program. **qemu-pr-helper** is that external helper; it creates a listener socket which will accept incoming connections for communication with QEMU.

If you want to run VMs in a setup like this, this helper should be started as a system service, and you should read the QEMU manual section on "persistent reservation managers" to find out how to configure QEMU to connect to the socket created by **qemu-pr-helper**.

After connecting to the socket, **qemu-pr-helper** can optionally drop root privileges, except for those capabilities that are needed for its operation.

qemu-pr-helper can also use the systemd socket activation protocol. In this case, the systemd socket unit should specify a Unix stream socket, like this:

```
[Socket]
ListenStream=/var/run/qemu-pr-helper.sock
```

4.4.3 Options

-d, --daemon
: run in the background (and create a PID file)

-q, --quiet
: decrease verbosity

-v, --verbose
: increase verbosity

-f, --pidfile=PATH
: PID file when running as a daemon. By default the PID file is created in the system runtime state directory, for example `/var/run/qemu-pr-helper.pid`.

-k, --socket=PATH

path to the socket. By default the socket is created in the system runtime state directory, for example `/var/run/qemu-pr-helper.sock`.

-T, --trace [[enable=]PATTERN][,events=FILE][,file=FILE]

Specify tracing options.

`[enable=]PATTERN`

Immediately enable events matching *PATTERN* (either event name or a globbing pattern). This option is only available if QEMU has been compiled with the `simple`, `log` or `ftrace` tracing backend. To specify multiple events or patterns, specify the `-trace` option multiple times.

Use `-trace help` to print a list of names of trace points.

`events=FILE`

Immediately enable events listed in *FILE*. The file must contain one event name (as listed in the `trace-events-all` file) per line; globbing patterns are accepted too. This option is only available if QEMU has been compiled with the `simple`, `log` or `ftrace` tracing backend.

`file=FILE`

Log output traces to *FILE*. This option is only available if QEMU has been compiled with the `simple` tracing backend.

-u, --user=USER

user to drop privileges to

-g, --group=GROUP

group to drop privileges to

-h, --help

Display a help message and exit.

-V, --version

Display version information and exit.

4.5 QEMU SystemTap trace tool

4.5.1 Synopsis

qemu-trace-stap [*GLOBAL-OPTIONS*] *COMMAND* [*COMMAND-OPTIONS*] *ARGS…*

4.5.2 Description

The `qemu-trace-stap` program facilitates tracing of the execution of QEMU emulators using SystemTap.

It is required to have the SystemTap runtime environment installed to use this program, since it is a wrapper around execution of the `stap` program.

4.5.3 Options

The following global options may be used regardless of which command is executed:

--verbose, -v

Display verbose information about command execution.

The following commands are valid:

list BINARY PATTERN...

List all the probe names provided by *BINARY* that match *PATTERN*.

If *BINARY* is not an absolute path, it will be located by searching the directories listed in the $PATH environment variable.

PATTERN is a plain string that is used to filter the results of this command. It may optionally contain a * wildcard to facilitate matching multiple probes without listing each one explicitly. Multiple *PATTERN* arguments may be given, causing listing of probes that match any of the listed names. If no *PATTERN* is given, the all possible probes will be listed.

For example, to list all probes available in the qemu-system-x86_64 binary:
```
$ qemu-trace-stap list qemu-system-x86_64
```
To filter the list to only cover probes related to QEMU's cryptographic subsystem, in a binary outside $PATH
```
$ qemu-trace-stap list /opt/qemu/9.0.50/bin/qemu-system-x86_64 'qcrypto*'
```

run OPTIONS BINARY PATTERN...

Run a trace session, printing formatted output any time a process that is executing *BINARY* triggers a probe matching *PATTERN*.

If *BINARY* is not an absolute path, it will be located by searching the directories listed in the $PATH environment variable.

PATTERN is a plain string that matches a probe name shown by the *LIST* command. It may optionally contain a * wildcard to facilitate matching multiple probes without listing each one explicitly. Multiple *PATTERN* arguments may be given, causing all matching probes to be monitored. At least one *PATTERN* is required, since stap is not capable of tracing all known QEMU probes concurrently without overflowing its trace buffer.

Invocation of this command does not need to be synchronized with invocation of the QEMU process(es). It will match probes on all existing running processes and all future launched processes, unless told to only monitor a specific process.

Valid command specific options are:

--pid=PID, **-p** PID

Restrict the tracing session so that it only triggers for the process identified by *PID*.

For example, to monitor all processes executing qemu-system-x86_64 as found on $PATH, displaying all I/O related probes:
```
$ qemu-trace-stap run qemu-system-x86_64 'qio*'
```
To monitor only the QEMU process with PID 1732
```
$ qemu-trace-stap run --pid=1732 qemu-system-x86_64 'qio*'
```
To monitor QEMU processes running an alternative binary outside of $PATH, displaying verbose information about setup of the tracing environment:
```
$ qemu-trace-stap -v run /opt/qemu/9.0.50/bin/qemu-system-x86_64 'qio*'
```

4.5.4 See also

qemu(1), *stap(1)*

4.6 QEMU 9p virtfs proxy filesystem helper

4.6.1 Synopsis

virtfs-proxy-helper [*OPTIONS*]

4.6.2 Description

NOTE: The 9p 'proxy' backend is deprecated (since QEMU 8.1) and will be removed, along with this daemon, in a future version of QEMU!

Pass-through security model in QEMU 9p server needs root privilege to do few file operations (like chown, chmod to any mode/uid:gid). There are two issues in pass-through security model:

- TOCTTOU vulnerability: Following symbolic links in the server could provide access to files beyond 9p export path.
- Running QEMU with root privilege could be a security issue.

To overcome above issues, following approach is used: A new filesystem type 'proxy' is introduced. Proxy FS uses chroot + socket combination for securing the vulnerability known with following symbolic links. Intention of adding a new filesystem type is to allow qemu to run in non-root mode, but doing privileged operations using socket IO.

Proxy helper (a stand alone binary part of qemu) is invoked with root privileges. Proxy helper chroots into 9p export path and creates a socket pair or a named socket based on the command line parameter. QEMU and proxy helper communicate using this socket. QEMU proxy fs driver sends filesystem request to proxy helper and receives the response from it.

The proxy helper is designed so that it can drop root privileges except for the capabilities needed for doing filesystem operations.

4.6.3 Options

The following options are supported:

-h

Display help and exit

-p, --path PATH

Path to export for proxy filesystem driver

-f, --fd SOCKET_ID

Use given file descriptor as socket descriptor for communicating with qemu proxy fs drier. Usually a helper like libvirt will create socketpair and pass one of the fds as parameter to this option.

-s, --socket SOCKET_FILE

Creates named socket file for communicating with qemu proxy fs driver

-u, --uid UID

uid to give access to named socket file; used in combination with -g.

-g, --gid GID

gid to give access to named socket file; used in combination with -u.

-n, --nodaemon

Run as a normal program. By default program will run in daemon mode

CHAPTER

FIVE

SYSTEM EMULATION MANAGEMENT AND INTEROPERABILITY

This section of the manual contains documents and specifications that are useful for making QEMU interoperate with other software.

5.1 Barrier client protocol

QEMU's `input-barrier` device implements the client end of the KVM (Keyboard-Video-Mouse) software Barrier.

This document briefly describes the protocol as we implement it.

5.1.1 Message format

Message format between the server and client is in two parts:

1. the payload length, a 32bit integer in network endianness
2. the payload

The payload starts with a 4byte string (without NUL) which is the command. The first command between the server and the client is the only command not encoded on 4 bytes ("Barrier"). The remaining part of the payload is decoded according to the command.

5.1.2 Protocol Description

This comes from `barrier/src/lib/barrier/protocol_types.h`.

barrierCmdHello "Barrier"

Direction:
server -> client

Parameters:
`{ int16_t minor, int16_t major }`

Description:
Say hello to client

`minor` = protocol major version number supported by server

`major` = protocol minor version number supported by server

barrierCmdHelloBack "Barrier"

Direction:
client ->server

Parameters:
`{ int16_t minor, int16_t major, char *name}`

Description:
Respond to hello from server

`minor` = protocol major version number supported by client

`major` = protocol minor version number supported by client

`name` = client name

barrierCmdDInfo "DINF"

Direction:
client ->server

Parameters:
`{ int16_t x_origin, int16_t y_origin, int16_t width, int16_t height, int16_t x, int16_t y}`

Description:
The client screen must send this message in response to the barrierCmdQInfo message. It must also send this message when the screen's resolution changes. In this case, the client screen should ignore any barrierCmdDMouseMove messages until it receives a barrierCmdCInfoAck in order to prevent attempts to move the mouse off the new screen area.

barrierCmdCNoop "CNOP"

Direction:
client -> server

Parameters:
None

Description:
No operation

barrierCmdCClose "CBYE"

Direction:
server -> client

Parameters:
None

Description:
Close connection

barrierCmdCEnter "CINN"

Direction:
server -> client

Parameters:
`{ int16_t x, int16_t y, int32_t seq, int16_t modifier }`

Description:
Enter screen.

`x`, `y` = entering screen absolute coordinates

`seq` = sequence number, which is used to order messages between screens. the secondary screen must return this number with some messages

`modifier` = modifier key mask. this will have bits set for each toggle modifier key that is activated on entry to the screen. the secondary screen should adjust its toggle modifiers to reflect that state.

barrierCmdCLeave "COUT"

Direction:
server -> client

Parameters:
None

Description:
Leaving screen. the secondary screen should send clipboard data in response to this message for those clipboards that it has grabbed (i.e. has sent a barrierCmdCClipboard for and has not received a barrierCmdCClipboard for with a greater sequence number) and that were grabbed or have changed since the last leave.

barrierCmdCClipboard "CCLP"

Direction:
server -> client

Parameters:
`{ int8_t id, int32_t seq }`

Description:
Grab clipboard. Sent by screen when some other app on that screen grabs a clipboard.

`id` = the clipboard identifier

`seq` = sequence number. Client must use the sequence number passed in the most recent barrierCmdCEnter. the server always sends 0.

barrierCmdCScreenSaver “CSEC”

Direction:
server -> client

Parameters:
`{ int8_t started }`

Description:
Screensaver change.

`started` = Screensaver on primary has started (1) or closed (0)

barrierCmdCResetOptions “CROP”

Direction:
server -> client

Parameters:
None

Description:
Reset options. Client should reset all of its options to their defaults.

barrierCmdCInfoAck “CIAK”

Direction:
server -> client

Parameters:
None

Description:
Resolution change acknowledgment. Sent by server in response to a client screen’s barrierCmdDInfo. This is sent for every barrierCmdDInfo, whether or not the server had sent a barrierCmdQInfo.

barrierCmdCKeepAlive “CALV”

Direction:
server -> client

Parameters:
None

Description:
Keep connection alive. Sent by the server periodically to verify that connections are still up and running. clients must reply in kind on receipt. if the server gets an error sending the message or does not receive a reply within a reasonable time then the server disconnects the client. if the client doesn’t receive these (or any message) periodically then it should disconnect from the server. the appropriate interval is defined by an option.

barrierCmdDKeyDown “DKDN”

Direction:
server -> client

Parameters:
`{ int16_t keyid, int16_t modifier [,int16_t button] }`

Description:
Key pressed.

`keyid` = X11 key id

`modified` = modified mask

`button` = X11 Xkb keycode (optional)

barrierCmdDKeyRepeat “DKRP”

Direction:
server -> client

Parameters:
`{ int16_t keyid, int16_t modifier, int16_t repeat [,int16_t button] }`

Description:
Key auto-repeat.

`keyid` = X11 key id

`modified` = modified mask

`repeat` = number of repeats

`button` = X11 Xkb keycode (optional)

barrierCmdDKeyUp “DKUP”

Direction:
server -> client

Parameters:
`{ int16_t keyid, int16_t modifier [,int16_t button] }`

Description:
Key released.

`keyid` = X11 key id

`modified` = modified mask

`button` = X11 Xkb keycode (optional)

barrierCmdDMouseDown “DMDN”

Direction:
server -> client

Parameters:
`{ int8_t button }`

Description:
Mouse button pressed.

`button` = button id

barrierCmdDMouseUp “DMUP”

Direction:
server -> client

Parameters:
`{ int8_t button }`

Description:
Mouse button release.

`button` = button id

barrierCmdDMouseMove “DMMV”

Direction:
server -> client

Parameters:
`{ int16_t x, int16_t y }`

Description:
Absolute mouse moved.

`x`, `y` = absolute screen coordinates

barrierCmdDMouseRelMove “DMRM”

Direction:
server -> client

Parameters:
`{ int16_t x, int16_t y }`

Description:
Relative mouse moved.

`x`, `y` = r relative screen coordinates

barrierCmdDMouseWheel "DMWM"

Direction:
server -> client

Parameters:
`{ int16_t x , int16_t y }` or `{ int16_t y }`

Description:
Mouse scroll. The delta should be +120 for one tick forward (away from the user) or right and -120 for one tick backward (toward the user) or left.

`x` = x delta

`y` = y delta

barrierCmdDClipboard "DCLP"

Direction:
server -> client

Parameters:
`{ int8_t id, int32_t seq, int8_t mark, char *data }`

Description:
Clipboard data.

`id` = clipboard id

`seq` = sequence number. The sequence number is 0 when sent by the server. Client screens should use the/ sequence number from the most recent barrierCmdCEnter.

barrierCmdDSetOptions "DSOP"

Direction:
server -> client

Parameters:
`{ int32 t nb, { int32_t id, int32_t val }[] }`

Description:
Set options. Client should set the given option/value pairs.

`nb` = numbers of `{ id, val }` entries

`id` = option id

`val` = option new value

barrierCmdDFileTransfer “DFTR”

Direction:
server -> client

Parameters:
`{ int8_t mark, char *content }`

Description:
Transfer file data.

- `mark` = 0 means the content followed is the file size
- 1 means the content followed is the chunk data
- 2 means the file transfer is finished

barrierCmdDDragInfo “DDRG”

Direction:
server -> client

Parameters:
`{ int16_t nb, char *content }`

Description:
Drag information.

`nb` = number of dragging objects

`content` = object’s directory

barrierCmdQInfo “QINF”

Direction:
server -> client

Parameters:
None

Description:
Query screen info

Client should reply with a barrierCmdDInfo

barrierCmdEIncompatible “EICV”

Direction:
server -> client

Parameters:
`{ int16_t nb, major *minor }`

Description:
Incompatible version.

`major` = major version

`minor` = minor version

barrierCmdEBusy "EBSY"

Direction:
server -> client

Parameters:
None

Description:
Name provided when connecting is already in use.

barrierCmdEUnknown "EUNK"

Direction:
server -> client

Parameters:
None

Description:
Unknown client. Name provided when connecting is not in primary's screen configuration map.

barrierCmdEBad "EBAD"

Direction:
server -> client

Parameters:
None

Description:
Protocol violation. Server should disconnect after sending this message.

5.2 Dirty Bitmaps and Incremental Backup

Dirty Bitmaps are in-memory objects that track writes to block devices. They can be used in conjunction with various block job operations to perform incremental or differential backup regimens.

This document explains the conceptual mechanisms, as well as up-to-date, complete and comprehensive documentation on the API to manipulate them. (Hopefully, the "why", "what", and "how".)

The intended audience for this document is developers who are adding QEMU backup features to management applications, or power users who run and administer QEMU directly via QMP.

Contents

5.2.1 Overview

Bitmaps are bit vectors where each '1' bit in the vector indicates a modified ("dirty") segment of the corresponding block device. The size of the segment that is tracked is the granularity of the bitmap. If the granularity of a bitmap is 64K, each '1' bit means that a 64K region as a whole may have changed in some way, possibly by as little as one byte.

Smaller granularities mean more accurate tracking of modified disk data, but requires more computational overhead and larger bitmap sizes. Larger granularities mean smaller bitmap sizes, but less targeted backups.

The size of a bitmap (in bytes) can be computed as such:

`size` = ceil(ceil(`image_size` / `granularity`) / 8)

e.g. the size of a 64KiB granularity bitmap on a 2TiB image is:

`size` = ((2147483648K / 64K) / 8)

= 4194304B = 4MiB.

QEMU uses these bitmaps when making incremental backups to know which sections of the file to copy out. They are not enabled by default and must be explicitly added in order to begin tracking writes.

Bitmaps can be created at any time and can be attached to any arbitrary block node in the storage graph, but are most useful conceptually when attached to the root node attached to the guest's storage device model.

That is to say: It's likely most useful to track the guest's writes to disk, but you could theoretically track things like qcow2 metadata changes by attaching the bitmap elsewhere in the storage graph. This is beyond the scope of this document.

QEMU supports persisting these bitmaps to disk via the qcow2 image format. Bitmaps which are stored or loaded in this way are called "persistent", whereas bitmaps that are not are called "transient".

QEMU also supports the migration of both transient bitmaps (tracking any arbitrary image format) or persistent bitmaps (qcow2) via live migration.

5.2.2 Supported Image Formats

QEMU supports all documented features below on the qcow2 image format.

However, qcow2 is only strictly necessary for the persistence feature, which writes bitmap data to disk upon close. If persistence is not required for a specific use case, all bitmap features excepting persistence are available for any arbitrary image format.

For example, Dirty Bitmaps can be combined with the 'raw' image format, but any changes to the bitmap will be discarded upon exit.

Warning: Transient bitmaps will not be saved on QEMU exit! Persistent bitmaps are available only on qcow2 images.

5.2.3 Dirty Bitmap Names

Bitmap objects need a method to reference them in the API. All API-created and managed bitmaps have a human-readable name chosen by the user at creation time.

- A bitmap's name is unique to the node, but bitmaps attached to different nodes can share the same name. Therefore, all bitmaps are addressed via their (node, name) pair.
- The name of a user-created bitmap cannot be empty ("").
- Transient bitmaps can have JSON unicode names that are effectively not length limited. (QMP protocol may restrict messages to less than 64MiB.)
- Persistent storage formats may impose their own requirements on bitmap names and namespaces. Presently, only qcow2 supports persistent bitmaps. See docs/interop/qcow2.txt for more details on restrictions. Notably:
  - qcow2 bitmap names are limited to between 1 and 1023 bytes long.
  - No two bitmaps saved to the same qcow2 file may share the same name.
- QEMU occasionally uses bitmaps for internal use which have no name. They are hidden from API query calls, cannot be manipulated by the external API, are never persistent, nor ever migrated.

5.2.4 Bitmap Status

Dirty Bitmap objects can be queried with the QMP command query-block, and are visible via the BlockDirtyInfo QAPI structure.

This struct shows the name, granularity, and dirty byte count for each bitmap. Additionally, it shows several boolean status indicators:

- `recording`: This bitmap is recording writes.
- `busy`: This bitmap is in-use by an operation.
- `persistent`: This bitmap is a persistent type.
- `inconsistent`: This bitmap is corrupted and cannot be used.

The +`busy` status prohibits you from deleting, clearing, or otherwise modifying a bitmap, and happens when the bitmap is being used for a backup operation or is in the process of being loaded from a migration. Many of the commands documented below will refuse to work on such bitmaps.

The `+inconsistent` status similarly prohibits almost all operations, notably allowing only the `block-dirty-bitmap-remove` operation.

There is also a deprecated `status` field of type DirtyBitmapStatus. A bitmap historically had five visible states:

1. `Frozen`: This bitmap is currently in-use by an operation and is immutable. It can't be deleted, renamed, reset, etc.

   (This is now +`busy`.)

2. `Disabled`: This bitmap is not recording new writes.

   (This is now `-recording -busy`.)

3. `Active`: This bitmap is recording new writes.

   (This is now `+recording -busy`.)

4. `Locked`: This bitmap is in-use by an operation, and is immutable. The difference from "Frozen" was primarily implementation details.

   (This is now +`busy`.)

5. `Inconsistent`: This persistent bitmap was not saved to disk correctly, and can no longer be used. It remains in memory to serve as an indicator of failure.

   (This is now `+inconsistent`.)

These states are directly replaced by the status indicators and should not be used. The difference between `Frozen` and `Locked` is an implementation detail and should not be relevant to external users.

5.2.5 Basic QMP Usage

The primary interface to manipulating bitmap objects is via the QMP interface. If you are not familiar, see the *QEMU Machine Protocol Specification* for the protocol, and *QEMU QMP Reference Manual* for a full reference of all QMP commands.

Supported Commands

There are six primary bitmap-management API commands:

- `block-dirty-bitmap-add`
- `block-dirty-bitmap-remove`
- `block-dirty-bitmap-clear`
- `block-dirty-bitmap-disable`
- `block-dirty-bitmap-enable`
- `block-dirty-bitmap-merge`

And one related query command:

- `query-block`

Creation: block-dirty-bitmap-add

block-dirty-bitmap-add:

Creates a new bitmap that tracks writes to the specified node. granularity, persistence, and recording state can be adjusted at creation time.

Example

to create a new, actively recording persistent bitmap:

```
-> { "execute": "block-dirty-bitmap-add",
     "arguments": {
       "node": "drive0",
       "name": "bitmap0",
       "persistent": true,
     }
   }

<- { "return": {} }
```

- This bitmap will have a default granularity that matches the cluster size of its associated drive, if available, clamped to between [4KiB, 64KiB]. The current default for qcow2 is 64KiB.

Example

To create a new, disabled (`-recording`), transient bitmap that tracks changes in 32KiB segments:

```
-> { "execute": "block-dirty-bitmap-add",
     "arguments": {
       "node": "drive0",
       "name": "bitmap1",
       "granularity": 32768,
       "disabled": true
     }
   }
```

(continues on next page)

(continued from previous page)

```
<- { "return": {} }
```

Deletion: block-dirty-bitmap-remove

block-dirty-bitmap-remove:

Deletes a bitmap. Bitmaps that are +busy cannot be removed.

- Deleting a bitmap does not impact any other bitmaps attached to the same node, nor does it affect any backups already created from this bitmap or node.
- Because bitmaps are only unique to the node to which they are attached, you must specify the node/drive name here, too.
- Deleting a persistent bitmap will remove it from the qcow2 file.

Example

Remove a bitmap named `bitmap0` from node `drive0`:

```
-> { "execute": "block-dirty-bitmap-remove",
     "arguments": {
       "node": "drive0",
       "name": "bitmap0"
     }
   }

<- { "return": {} }
```

Resetting: block-dirty-bitmap-clear

block-dirty-bitmap-clear:

Clears all dirty bits from a bitmap. +busy bitmaps cannot be cleared.

- An incremental backup created from an empty bitmap will copy no data, as if nothing has changed.

Example

Clear all dirty bits from bitmap `bitmap0` on node `drive0`:

```
-> { "execute": "block-dirty-bitmap-clear",
     "arguments": {
       "node": "drive0",
       "name": "bitmap0"
     }
   }

<- { "return": {} }
```

Enabling: block-dirty-bitmap-enable

block-dirty-bitmap-enable:

"Enables" a bitmap, setting the `recording` bit to true, causing writes to begin being recorded. +`busy` bitmaps cannot be enabled.

- Bitmaps default to being enabled when created, unless configured otherwise.
- Persistent enabled bitmaps will remember their +`recording` status on load.

Example

To set +`recording` on bitmap `bitmap0` on node `drive0`:

```
-> { "execute": "block-dirty-bitmap-enable",
     "arguments": {
       "node": "drive0",
       "name": "bitmap0"
     }
   }

<- { "return": {} }
```

Enabling: block-dirty-bitmap-disable

block-dirty-bitmap-disable:

"Disables" a bitmap, setting the `recording` bit to false, causing further writes to begin being ignored. +`busy` bitmaps cannot be disabled.

Warning: This is potentially dangerous: QEMU makes no effort to stop any writes if there are disabled bitmaps on a node, and will not mark any disabled bitmaps as +`inconsistent` if any such writes do happen. Backups made from such bitmaps will not be able to be used to reconstruct a coherent image.

- Disabling a bitmap may be useful for examining which sectors of a disk changed during a specific time period, or for explicit management of differential backup windows.
- Persistent disabled bitmaps will remember their -`recording` status on load.

Example

To set -`recording` on bitmap `bitmap0` on node `drive0`:

```
-> { "execute": "block-dirty-bitmap-disable",
     "arguments": {
       "node": "drive0",
       "name": "bitmap0"
     }
   }

<- { "return": {} }
```

Merging, Copying: block-dirty-bitmap-merge

block-dirty-bitmap-merge:

Merges one or more bitmaps into a target bitmap. For any segment that is dirty in any one source bitmap, the target bitmap will mark that segment dirty.

- Merge takes one or more bitmaps as a source and merges them together into a single destination, such that any segment marked as dirty in any source bitmap(s) will be marked dirty in the destination bitmap.
- Merge does not create the destination bitmap if it does not exist. A blank bitmap can be created beforehand to achieve the same effect.
- The destination is not cleared prior to merge, so subsequent merge operations will continue to cumulatively mark more segments as dirty.
- If the merge operation should fail, the destination bitmap is guaranteed to be unmodified. The operation may fail if the source or destination bitmaps are busy, or have different granularities.
- Bitmaps can only be merged on the same node. There is only one "node" argument, so all bitmaps must be attached to that same node.
- Copy can be achieved by merging from a single source to an empty destination.

Example

Merge the data from `bitmap0` into the bitmap `new_bitmap` on node `drive0`. If `new_bitmap` was empty prior to this command, this achieves a copy.

```
-> { "execute": "block-dirty-bitmap-merge",
     "arguments": {
       "node": "drive0",
       "target": "new_bitmap",
       "bitmaps": [ "bitmap0" ]
     }
   }

<- { "return": {} }
```

Querying: query-block

query-block:

Not strictly a bitmaps command, but will return information about any bitmaps attached to nodes serving as the root for guest devices.

- The "inconsistent" bit will not appear when it is false, appearing only when the value is true to indicate there is a problem.

Example

Query the block sub-system of QEMU. The following json has trimmed irrelevant keys from the response to highlight only the bitmap-relevant portions of the API. This result highlights a bitmap `bitmap0` attached to the root node of device `drive0`.

```
-> {
     "execute": "query-block",
     "arguments": {}
   }

<- {
     "return": [ {
       "dirty-bitmaps": [ {
         "status": "active",
         "count": 0,
         "busy": false,
         "name": "bitmap0",
         "persistent": false,
         "recording": true,
         "granularity": 65536
       } ],
       "device": "drive0",
     } ]
   }
```

5.2.6 Bitmap Persistence

As outlined in *Supported Image Formats*, QEMU can persist bitmaps to qcow2 files. Demonstrated in *Creation: block-dirty-bitmap-add*, passing `persistent: true` to `block-dirty-bitmap-add` will persist that bitmap to disk.

Persistent bitmaps will be automatically loaded into memory upon load, and will be written back to disk upon close. Their usage should be mostly transparent.

However, if QEMU does not get a chance to close the file cleanly, the bitmap will be marked as `+inconsistent` at next load and considered unsafe to use for any operation. At this point, the only valid operation on such bitmaps is `block-dirty-bitmap-remove`.

Losing a bitmap in this way does not invalidate any existing backups that have been made from this bitmap, but no further backups will be able to be issued for this chain.

5.2.7 Transactions

Transactions are a QMP feature that allows you to submit multiple QMP commands at once, being guaranteed that they will all succeed or fail atomically, together. The interaction of bitmaps and transactions are demonstrated below.

See transaction in the QMP reference for more details.

Justification

Bitmaps can generally be modified at any time, but certain operations often only make sense when paired directly with other commands. When a VM is paused, it's easy to ensure that no guest writes occur between individual QMP commands. When a VM is running, this is difficult to accomplish with individual QMP commands that may allow guest writes to occur between each command.

For example, using only individual QMP commands, we could:

1. Boot the VM in a paused state.
2. Create a full drive backup of drive0.
3. Create a new bitmap attached to drive0, confident that nothing has been written to drive0 in the meantime.
4. Resume execution of the VM.
5. At a later point, issue incremental backups from `bitmap0`.

At this point, the bitmap and drive backup would be correctly in sync, and incremental backups made from this point forward would be correctly aligned to the full drive backup.

This is not particularly useful if we decide we want to start incremental backups after the VM has been running for a while, for which we would want to perform actions such as the following:

1. Boot the VM and begin execution.
2. Using a single transaction, perform the following operations:
   - Create `bitmap0`.
   - Create a full drive backup of `drive0`.
3. At a later point, issue incremental backups from `bitmap0`.

Note: As a consideration, if `bitmap0` is created prior to the full drive backup, incremental backups can still be authored from this bitmap, but they will copy extra segments reflecting writes that occurred prior to the backup operation. Transactions allow us to narrow critical points in time to reduce waste, or, in the other direction, to ensure that no segments are omitted.

Supported Bitmap Transactions

- `block-dirty-bitmap-add`
- `block-dirty-bitmap-clear`
- `block-dirty-bitmap-enable`
- `block-dirty-bitmap-disable`
- `block-dirty-bitmap-merge`

The usages for these commands are identical to their respective QMP commands, but see the sections below for concrete examples.

5.2.8 Incremental Backups - Push Model

Incremental backups are simply partial disk images that can be combined with other partial disk images on top of a base image to reconstruct a full backup from the point in time at which the incremental backup was issued.

The "Push Model" here references the fact that QEMU is "pushing" the modified blocks out to a destination. We will be using the blockdev-backup QMP command to create both full and incremental backups.

The command is a background job, which has its own QMP API for querying and management documented in Background jobs.

Example: New Incremental Backup Anchor Point

As outlined in the Transactions - *Justification* section, perhaps we want to create a new incremental backup chain attached to a drive.

This example creates a new, full backup of "drive0" and accompanies it with a new, empty bitmap that records writes from this point in time forward.

The target can be created with the help of blockdev-add or blockdev-create command.

Note: Any new writes that happen after this command is issued, even while the backup job runs, will be written locally and not to the backup destination. These writes will be recorded in the bitmap accordingly.

```
-> {
     "execute": "transaction",
     "arguments": {
       "actions": [
         {
           "type": "block-dirty-bitmap-add",
           "data": {
             "node": "drive0",
             "name": "bitmap0"
           }
         },
         {
           "type": "blockdev-backup",
           "data": {
             "device": "drive0",
             "target": "target0",
             "sync": "full"
           }
         }
       ]
     }
   }

<- { "return": {} }

<- {
     "timestamp": {
       "seconds": 1555436945,
       "microseconds": 179620
```

(continues on next page)

(continued from previous page)

```
     },
     "data": {
       "status": "created",
       "id": "drive0"
     },
     "event": "JOB_STATUS_CHANGE"
   }

...

<- {
     "timestamp": {...},
     "data": {
       "device": "drive0",
       "type": "backup",
       "speed": 0,
       "len": 68719476736,
       "offset": 68719476736
     },
     "event": "BLOCK_JOB_COMPLETED"
   }

<- {
     "timestamp": {...},
     "data": {
       "status": "concluded",
       "id": "drive0"
     },
     "event": "JOB_STATUS_CHANGE"
   }

<- {
     "timestamp": {...},
     "data": {
       "status": "null",
       "id": "drive0"
     },
     "event": "JOB_STATUS_CHANGE"
   }
```

A full explanation of the job transition semantics and the JOB_STATUS_CHANGE event are beyond the scope of this document and will be omitted in all subsequent examples; above, several more events have been omitted for brevity.

Note: Subsequent examples will omit all events except BLOCK_JOB_COMPLETED except where necessary to illustrate workflow differences.

Omitted events and json objects will be represented by ellipses: ...

Example: Resetting an Incremental Backup Anchor Point

If we want to start a new backup chain with an existing bitmap, we can also use a transaction to reset the bitmap while making a new full backup:

```
-> {
     "execute": "transaction",
     "arguments": {
       "actions": [
       {
         "type": "block-dirty-bitmap-clear",
         "data": {
           "node": "drive0",
           "name": "bitmap0"
         }
       },
       {
         "type": "blockdev-backup",
         "data": {
           "device": "drive0",
           "target": "target0",
           "sync": "full"
         }
       }
     ]
   }
 }

<- { "return": {} }

...

<- {
     "timestamp": {...},
     "data": {
       "device": "drive0",
       "type": "backup",
       "speed": 0,
       "len": 68719476736,
       "offset": 68719476736
     },
     "event": "BLOCK_JOB_COMPLETED"
   }

...
```

The result of this example is identical to the first, but we clear an existing bitmap instead of adding a new one.

Tip: In both of these examples, “bitmap0” is tied conceptually to the creation of new, full backups. This relationship is not saved or remembered by QEMU; it is up to the operator or management layer to remember which bitmaps are associated with which backups.

Example: First Incremental Backup

1. Create a full backup and sync it to a dirty bitmap using any method:
   - Either of the two live backup method demonstrated above,
   - Using QMP commands with the VM paused as in the *Justification* section, or
   - With the VM offline, manually copy the image and start the VM in a paused state, careful to add a new bitmap before the VM begins execution.

   Whichever method is chosen, let's assume that at the end of this step:
   - The full backup is named `drive0.full.qcow2`.
   - The bitmap we created is named `bitmap0`, attached to `drive0`.
2. Create a destination image for the incremental backup that utilizes the full backup as a backing image.
   - Let's assume the new incremental image is named `drive0.inc0.qcow2`:

```
$ qemu-img create -f qcow2 drive0.inc0.qcow2 \
  -b drive0.full.qcow2 -F qcow2
```

3. Add target block node:

```
-> {
     "execute": "blockdev-add",
     "arguments": {
       "node-name": "target0",
       "driver": "qcow2",
       "file": {
         "driver": "file",
         "filename": "drive0.inc0.qcow2"
       }
     }
   }

<- { "return": {} }
```

4. Issue an incremental backup command:

```
-> {
     "execute": "blockdev-backup",
     "arguments": {
       "device": "drive0",
       "bitmap": "bitmap0",
       "target": "target0",
       "sync": "incremental"
     }
   }

<- { "return": {} }

...

<- {
     "timestamp": {...},
```

(continues on next page)

(continued from previous page)

```
     "data": {
       "device": "drive0",
       "type": "backup",
       "speed": 0,
       "len": 68719476736,
       "offset": 68719476736
     },
     "event": "BLOCK_JOB_COMPLETED"
   }

...
```

This copies any blocks modified since the full backup was created into the `drive0.inc0.qcow2` file. During the operation, `bitmap0` is marked +`busy`. If the operation is successful, `bitmap0` will be cleared to reflect the "incremental" backup regimen, which only copies out new changes from each incremental backup.

Note: Any new writes that occur after the backup operation starts do not get copied to the destination. The backup's "point in time" is when the backup starts, not when it ends. These writes are recorded in a special bitmap that gets re-added to bitmap0 when the backup ends so that the next incremental backup can copy them out.

Example: Second Incremental Backup

1. Create a new destination image for the incremental backup that points to the previous one, e.g.: `drive0.inc1.qcow2`

```
$ qemu-img create -f qcow2 drive0.inc1.qcow2 \
  -b drive0.inc0.qcow2 -F qcow2
```

2. Add target block node:

```
-> {
     "execute": "blockdev-add",
     "arguments": {
       "node-name": "target0",
       "driver": "qcow2",
       "file": {
         "driver": "file",
         "filename": "drive0.inc1.qcow2"
       }
     }
   }

<- { "return": {} }
```

3. Issue a new incremental backup command. The only difference here is that we have changed the target image below.

```
-> {
     "execute": "blockdev-backup",
     "arguments": {
```

(continues on next page)

(continued from previous page)

```
       "device": "drive0",
       "bitmap": "bitmap0",
       "target": "target0",
       "sync": "incremental"
     }
   }

<- { "return": {} }

...

<- {
     "timestamp": {...},
     "data": {
       "device": "drive0",
       "type": "backup",
       "speed": 0,
       "len": 68719476736,
       "offset": 68719476736
     },
     "event": "BLOCK_JOB_COMPLETED"
   }

...
```

Because the first incremental backup from the previous example completed successfully, `bitmap0` was synchronized with `drive0.inc0.qcow2`. Here, we use `bitmap0` again to create a new incremental backup that targets the previous one, creating a chain of three images:

Diagram

```
+-------------------+   +-------------------+   +-------------------+
| drive0.full.qcow2 |<--| drive0.inc0.qcow2 |<--| drive0.inc1.qcow2 |
+-------------------+   +-------------------+   +-------------------+
```

Each new incremental backup re-synchronizes the bitmap to the latest backup authored, allowing a user to continue to "consume" it to create new backups on top of an existing chain.

In the above diagram, neither drive0.inc1.qcow2 nor drive0.inc0.qcow2 are complete images by themselves, but rely on their backing chain to reconstruct a full image. The dependency terminates with each full backup.

Each backup in this chain remains independent, and is unchanged by new entries made later in the chain. For instance, drive0.inc0.qcow2 remains a perfectly valid backup of the disk as it was when that backup was issued.

Example: Incremental Push Backups without Backing Files

Backup images are best kept off-site, so we often will not have the preceding backups in a chain available to link against. This is not a problem at backup time; we simply do not set the backing image when creating the destination image:

1. Create a new destination image with no backing file set. We will need to specify the size of the base image, because the backing file isn't available for QEMU to use to determine it.

```
$ qemu-img create -f qcow2 drive0.inc2.qcow2 64G
```

Note: Alternatively, you can omit `mode: "existing"` from the push backup commands to have QEMU create an image without a backing file for you, but you lose control over format options like compatibility and preallocation presets.

---

2. Add target block node:

```
-> {
     "execute": "blockdev-add",
     "arguments": {
       "node-name": "target0",
       "driver": "qcow2",
       "file": {
         "driver": "file",
         "filename": "drive0.inc2.qcow2"
       }
     }
   }

<- { "return": {} }
```

3. Issue a new incremental backup command. Apart from the new destination image, there is no difference from the last two examples.

```
-> {
     "execute": "blockdev-backup",
     "arguments": {
       "device": "drive0",
       "bitmap": "bitmap0",
       "target": "target0",
       "sync": "incremental"
     }
   }

<- { "return": {} }

...

<- {
     "timestamp": {...},
     "data": {
       "device": "drive0",
       "type": "backup",
       "speed": 0,
```

(continues on next page)

(continued from previous page)

```
        "len": 68719476736,
        "offset": 68719476736
      },
      "event": "BLOCK_JOB_COMPLETED"
    }

...
```

The only difference from the perspective of the user is that you will need to set the backing image when attempting to restore the backup:

```
$ qemu-img rebase drive0.inc2.qcow2 \
  -u -b drive0.inc1.qcow2
```

This uses the "unsafe" rebase mode to simply set the backing file to a file that isn't present.

It is also possible to use `--image-opts` to specify the entire backing chain by hand as an ephemeral property at runtime, but that is beyond the scope of this document.

Example: Multi-drive Incremental Backup

Assume we have a VM with two drives, "drive0" and "drive1" and we wish to back both of them up such that the two backups represent the same crash-consistent point in time.

1. For each drive, create an empty image:

```
$ qemu-img create -f qcow2 drive0.full.qcow2 64G
$ qemu-img create -f qcow2 drive1.full.qcow2 64G
```

2. Add target block nodes:

```
-> {
     "execute": "blockdev-add",
     "arguments": {
       "node-name": "target0",
       "driver": "qcow2",
       "file": {
         "driver": "file",
         "filename": "drive0.full.qcow2"
       }
     }
   }

<- { "return": {} }

-> {
     "execute": "blockdev-add",
     "arguments": {
       "node-name": "target1",
       "driver": "qcow2",
       "file": {
         "driver": "file",
         "filename": "drive1.full.qcow2"
```

(continues on next page)

(continued from previous page)

```
      }
    }
  }

<- { "return": {} }
```

3. Create a full (anchor) backup for each drive, with accompanying bitmaps:

```
-> {
     "execute": "transaction",
     "arguments": {
       "actions": [
         {
           "type": "block-dirty-bitmap-add",
           "data": {
             "node": "drive0",
             "name": "bitmap0"
           }
         },
         {
           "type": "block-dirty-bitmap-add",
           "data": {
             "node": "drive1",
             "name": "bitmap0"
           }
         },
         {
           "type": "blockdev-backup",
           "data": {
             "device": "drive0",
             "target": "target0",
             "sync": "full"
           }
         },
         {
           "type": "blockdev-backup",
           "data": {
             "device": "drive1",
             "target": "target1",
             "sync": "full"
           }
         }
       ]
     }
   }

<- { "return": {} }

...

<- {
     "timestamp": {...},
```

(continues on next page)

(continued from previous page)

```
     "data": {
       "device": "drive0",
       "type": "backup",
       "speed": 0,
       "len": 68719476736,
       "offset": 68719476736
     },
     "event": "BLOCK_JOB_COMPLETED"
   }

...

<- {
     "timestamp": {...},
     "data": {
       "device": "drive1",
       "type": "backup",
       "speed": 0,
       "len": 68719476736,
       "offset": 68719476736
     },
     "event": "BLOCK_JOB_COMPLETED"
   }

...
```

4. Later, create new destination images for each of the incremental backups that point to their respective full backups:

```
$ qemu-img create -f qcow2 drive0.inc0.qcow2 \
  -b drive0.full.qcow2 -F qcow2
$ qemu-img create -f qcow2 drive1.inc0.qcow2 \
  -b drive1.full.qcow2 -F qcow2
```

5. Add target block nodes:

```
-> {
     "execute": "blockdev-add",
     "arguments": {
       "node-name": "target0",
       "driver": "qcow2",
       "file": {
         "driver": "file",
         "filename": "drive0.inc0.qcow2"
       }
     }
   }

<- { "return": {} }

-> {
     "execute": "blockdev-add",
```

(continues on next page)

(continued from previous page)

```
     "arguments": {
       "node-name": "target1",
       "driver": "qcow2",
       "file": {
         "driver": "file",
         "filename": "drive1.inc0.qcow2"
       }
     }
   }

<- { "return": {} }
```

6. Issue a multi-drive incremental push backup transaction:

```
-> {
     "execute": "transaction",
     "arguments": {
       "actions": [
         {
           "type": "blockev-backup",
           "data": {
             "device": "drive0",
             "bitmap": "bitmap0",
             "sync": "incremental",
             "target": "target0"
           }
         },
         {
           "type": "blockdev-backup",
           "data": {
             "device": "drive1",
             "bitmap": "bitmap0",
             "sync": "incremental",
             "target": "target1"
           }
         },
       ]
     }
   }

<- { "return": {} }

...

<- {
     "timestamp": {...},
     "data": {
       "device": "drive0",
       "type": "backup",
       "speed": 0,
       "len": 68719476736,
       "offset": 68719476736
```

(continues on next page)

(continued from previous page)

```
     },
     "event": "BLOCK_JOB_COMPLETED"
   }

...

<- {
     "timestamp": {...},
     "data": {
       "device": "drive1",
       "type": "backup",
       "speed": 0,
       "len": 68719476736,
       "offset": 68719476736
     },
     "event": "BLOCK_JOB_COMPLETED"
   }

...
```

5.2.9 Push Backup Errors & Recovery

In the event of an error that occurs after a push backup job is successfully launched, either by an individual QMP command or a QMP transaction, the user will receive a `BLOCK_JOB_COMPLETE` event with a failure message, accompanied by a `BLOCK_JOB_ERROR` event.

In the case of a job being cancelled, the user will receive a `BLOCK_JOB_CANCELLED` event instead of a pair of COMPLETE and ERROR events.

In either failure case, the bitmap used for the failed operation is not cleared. It will contain all of the dirty bits it did at the start of the operation, plus any new bits that got marked during the operation.

Effectively, the "point in time" that a bitmap is recording differences against is kept at the issuance of the last successful incremental backup, instead of being moved forward to the start of this now-failed backup.

Once the underlying problem is addressed (e.g. more storage space is allocated on the destination), the incremental backup command can be retried with the same bitmap.

Example: Individual Failures

Incremental Push Backup jobs that fail individually behave simply as described above. This example demonstrates the single-job failure case:

1. Create a target image:

```
$ qemu-img create -f qcow2 drive0.inc0.qcow2 \
  -b drive0.full.qcow2 -F qcow2
```

2. Add target block node:

```
-> {
     "execute": "blockdev-add",
     "arguments": {
```

(continues on next page)

(continued from previous page)

```
       "node-name": "target0",
       "driver": "qcow2",
       "file": {
         "driver": "file",
         "filename": "drive0.inc0.qcow2"
       }
     }
   }

<- { "return": {} }
```

3. Attempt to create an incremental backup via QMP:

```
-> {
     "execute": "blockdev-backup",
     "arguments": {
       "device": "drive0",
       "bitmap": "bitmap0",
       "target": "target0",
       "sync": "incremental"
     }
   }

<- { "return": {} }
```

4. Receive a pair of events indicating failure:

```
<- {
     "timestamp": {...},
     "data": {
       "device": "drive0",
       "action": "report",
       "operation": "write"
     },
     "event": "BLOCK_JOB_ERROR"
   }

<- {
     "timestamp": {...},
     "data": {
       "speed": 0,
       "offset": 0,
       "len": 67108864,
       "error": "No space left on device",
       "device": "drive0",
       "type": "backup"
     },
     "event": "BLOCK_JOB_COMPLETED"
   }
```

5. Remove target node:

```
-> {
     "execute": "blockdev-del",
     "arguments": {
       "node-name": "target0",
     }
   }

<- { "return": {} }
```

6. Delete the failed image, and re-create it.

```
$ rm drive0.inc0.qcow2
$ qemu-img create -f qcow2 drive0.inc0.qcow2 \
  -b drive0.full.qcow2 -F qcow2
```

7. Add target block node:

```
-> {
     "execute": "blockdev-add",
     "arguments": {
       "node-name": "target0",
       "driver": "qcow2",
       "file": {
         "driver": "file",
         "filename": "drive0.inc0.qcow2"
       }
     }
   }

<- { "return": {} }
```

8. Retry the command after fixing the underlying problem, such as freeing up space on the backup volume:

```
-> {
     "execute": "blockdev-backup",
     "arguments": {
       "device": "drive0",
       "bitmap": "bitmap0",
       "target": "target0",
       "sync": "incremental"
     }
   }

<- { "return": {} }
```

9. Receive confirmation that the job completed successfully:

```
<- {
     "timestamp": {...},
     "data": {
       "device": "drive0",
       "type": "backup",
       "speed": 0,
```

(continues on next page)

(continued from previous page)

```
      "len": 67108864,
      "offset": 67108864
    },
    "event": "BLOCK_JOB_COMPLETED"
  }
```

Example: Partial Transactional Failures

QMP commands like blockdev-backup conceptually only start a job, and so transactions containing these commands may succeed even if the job it created later fails. This might have surprising interactions with notions of how a "transaction" ought to behave.

This distinction means that on occasion, a transaction containing such job launching commands may appear to succeed and return success, but later individual jobs associated with the transaction may fail. It is possible that a management application may have to deal with a partial backup failure after a "successful" transaction.

If multiple backup jobs are specified in a single transaction, if one of those jobs fails, it will not interact with the other backup jobs in any way by default. The job(s) that succeeded will clear the dirty bitmap associated with the operation, but the job(s) that failed will not. It is therefore not safe to delete any incremental backups that were created successfully in this scenario, even though others failed.

This example illustrates a transaction with two backup jobs, where one fails and one succeeds:

1. Issue the transaction to start a backup of both drives.

```
-> {
     "execute": "transaction",
     "arguments": {
       "actions": [
       {
         "type": "blockdev-backup",
         "data": {
           "device": "drive0",
           "bitmap": "bitmap0",
           "sync": "incremental",
           "target": "target0"
         }
       },
       {
         "type": "blockdev-backup",
         "data": {
           "device": "drive1",
           "bitmap": "bitmap0",
           "sync": "incremental",
           "target": "target1"
         }
       }]
     }
   }
```

2. Receive notice that the Transaction was accepted, and jobs were launched:

```
<- { "return": {} }
```

3. Receive notice that the first job has completed:

```
<- {
     "timestamp": {...},
     "data": {
       "device": "drive0",
       "type": "backup",
       "speed": 0,
       "len": 67108864,
       "offset": 67108864
     },
     "event": "BLOCK_JOB_COMPLETED"
   }
```

4. Receive notice that the second job has failed:

```
<- {
     "timestamp": {...},
     "data": {
       "device": "drive1",
       "action": "report",
       "operation": "read"
     },
     "event": "BLOCK_JOB_ERROR"
   }

...

<- {
     "timestamp": {...},
     "data": {
       "speed": 0,
       "offset": 0,
       "len": 67108864,
       "error": "Input/output error",
       "device": "drive1",
       "type": "backup"
     },
     "event": "BLOCK_JOB_COMPLETED"
   }
```

At the conclusion of the above example, `drive0.inc0.qcow2` is valid and must be kept, but `drive1.inc0.qcow2` is incomplete and should be deleted. If a VM-wide incremental backup of all drives at a point-in-time is to be made, new backups for both drives will need to be made, taking into account that a new incremental backup for drive0 needs to be based on top of `drive0.inc0.qcow2`.

For this example, an incremental backup for `drive0` was created, but not for `drive1`. The last VM-wide crash-consistent backup that is available in this case is the full backup:

```
[drive0.full.qcow2] <-- [drive0.inc0.qcow2]
[drive1.full.qcow2]
```

To repair this, issue a new incremental backup across both drives. The result will be backup chains that resemble the following:

```
[drive0.full.qcow2] <-- [drive0.inc0.qcow2] <-- [drive0.inc1.qcow2]
[drive1.full.qcow2] <--------------------------- [drive1.inc1.qcow2]
```

Example: Grouped Completion Mode

While jobs launched by transactions normally complete or fail individually, it's possible to instruct them to complete or fail together as a group. QMP transactions take an optional properties structure that can affect the behavior of the transaction.

The `completion-mode` transaction property can be either `individual` which is the default legacy behavior described above, or `grouped`, detailed below.

In `grouped` completion mode, no jobs will report success until all jobs are ready to report success. If any job fails, all other jobs will be cancelled.

Regardless of if a participating incremental backup job failed or was cancelled, their associated bitmaps will all be held at their existing points-in-time, as in individual failure cases.

Here's the same multi-drive backup scenario from *Example: Partial Transactional Failures*, but with the `grouped` completion-mode property applied:

1. Issue the multi-drive incremental backup transaction:

```
-> {
     "execute": "transaction",
     "arguments": {
       "properties": {
         "completion-mode": "grouped"
       },
       "actions": [
       {
         "type": "blockdev-backup",
         "data": {
           "device": "drive0",
           "bitmap": "bitmap0",
           "sync": "incremental",
           "target": "target0"
         }
       },
       {
         "type": "blockdev-backup",
         "data": {
           "device": "drive1",
           "bitmap": "bitmap0",
           "sync": "incremental",
           "target": "target1"
         }
       }]
     }
   }
```

2. Receive notice that the Transaction was accepted, and jobs were launched:

```
<- { "return": {} }
```

3. Receive notification that the backup job for `drive1` has failed:

```
<- {
     "timestamp": {...},
     "data": {
       "device": "drive1",
       "action": "report",
       "operation": "read"
     },
     "event": "BLOCK_JOB_ERROR"
   }

<- {
     "timestamp": {...},
     "data": {
       "speed": 0,
       "offset": 0,
       "len": 67108864,
       "error": "Input/output error",
       "device": "drive1",
       "type": "backup"
     },
     "event": "BLOCK_JOB_COMPLETED"
   }
```

4. Receive notification that the job for `drive0` has been cancelled:

```
<- {
     "timestamp": {...},
     "data": {
       "device": "drive0",
       "type": "backup",
       "speed": 0,
       "len": 67108864,
       "offset": 16777216
     },
     "event": "BLOCK_JOB_CANCELLED"
   }
```

At the conclusion of *this* example, both jobs have been aborted due to a failure. Both destination images should be deleted and are no longer of use.

The transaction as a whole can simply be re-issued at a later time.

5.3 D-Bus

5.3.1 Introduction

QEMU may be running with various helper processes involved:

- vhost-user* processes (gpu, virtfs, input, etc...)
- TPM emulation (or other devices)

- user networking (slirp)
- network services (DHCP/DNS, samba/ftp etc)
- background tasks (compression, streaming etc)
- client UI
- admin & cli

Having several processes allows stricter security rules, as well as greater modularity.

While QEMU itself uses QMP as primary IPC (and Spice/VNC for remote display), D-Bus is the de facto IPC of choice on Unix systems. The wire format is machine friendly, good bindings exist for various languages, and there are various tools available.

Using a bus, helper processes can discover and communicate with each other easily, without going through QEMU. The bus topology is also easier to apprehend and debug than a mesh. However, it is wise to consider the security aspects of it.

5.3.2 Security

A QEMU D-Bus bus should be private to a single VM. Thus, only cooperative tasks are running on the same bus to serve the VM.

D-Bus, the protocol and standard, doesn't have mechanisms to enforce security between peers once the connection is established. Peers may have additional mechanisms to enforce security rules, based for example on UNIX credentials.

The daemon can control which peers can send/recv messages using various metadata attributes, however, this is alone is not generally sufficient to make the deployment secure. The semantics of the actual methods implemented using D-Bus are just as critical. Peers need to carefully validate any information they received from a peer with a different trust level.

dbus-daemon policy

dbus-daemon can enforce various policies based on the UID/GID of the processes that are connected to it. It is thus a good idea to run helpers as different UID from QEMU and set appropriate policies.

Depending on the use case, you may choose different scenarios:

- Everything the same UID
  - Convenient for developers
  - Improved reliability - crash of one part doesn't take out entire VM
  - No security benefit over traditional QEMU, unless additional unless additional controls such as SELinux or AppArmor are applied
- Two UIDs, one for QEMU, one for dbus & helpers
  - Moderately improved user based security isolation
- Many UIDs, one for QEMU one for dbus and one for each helpers
  - Best user based security isolation
  - Complex to manager distinct UIDs needed for each VM

For example, to allow only `qemu` user to talk to `qemu-helper org.qemu.Helper1` service, a dbus-daemon policy may contain:

```
<policy user="qemu">
   <allow send_destination="org.qemu.Helper1"/>
   <allow receive_sender="org.qemu.Helper1"/>
</policy>

<policy user="qemu-helper">
   <allow own="org.qemu.Helper1"/>
</policy>
```

dbus-daemon can also perform SELinux checks based on the security context of the source and the target. For example, `virtiofs_t` could be allowed to send a message to `svirt_t`, but `virtiofs_t` wouldn't be allowed to send a message to `virtiofs_t`.

See dbus-daemon man page for details.

5.3.3 Guidelines

When implementing new D-Bus interfaces, it is recommended to follow the "D-Bus API Design Guidelines": https://dbus.freedesktop.org/doc/dbus-api-design.html

The "org.qemu.*" prefix is reserved for services implemented & distributed by the QEMU project.

5.3.4 QEMU Interfaces

D-Bus VMState

D-Bus display

5.4 D-Bus VMState

The QEMU dbus-vmstate object's aim is to migrate helpers' data running on a QEMU D-Bus bus. (refer to the *D-Bus* document for some recommendations on D-Bus usage)

Upon migration, QEMU will go through the queue of `org.qemu.VMState1` D-Bus name owners and query their `Id`. It must be unique among the helpers.

It will then save arbitrary data of each Id to be transferred in the migration stream and restored/loaded at the corresponding destination helper.

For now, the data amount to be transferred is arbitrarily limited to 1Mb. The state must be saved quickly (a fraction of a second). (D-Bus imposes a time limit on reply anyway, and migration would fail if data isn't given quickly enough.)

dbus-vmstate object can be configured with the expected list of helpers by setting its `id-list` property, with a comma-separated `Id` list.

5.4.1 org.qemu.VMState1 interface

`interface org.qemu.VMState1`

This interface must be implemented at the object path `/org/qemu/VMState1` to support helper migration.

`method Load`(*ay data*) →

Arguments

- `data` (*ay*) – data to restore the state.

The method called on destination with the state to restore.

The helper may be initially started in a waiting state (with an `-incoming` argument for example), and it may resume on success.

An error may be returned to the caller.

`method Save`() → *ay data*

Returns

- `data` (*ay*) – state data to save for later resume.

The method called on the source to get the current state to be migrated. The helper should continue to run normally.

An error may be returned to the caller.

`property Id`:s

Access
read-only

Emits Changed
yes

A string that identifies the helper uniquely. (maximum 256 bytes including terminating NUL byte)

Note: The VMState helper ID namespace is its own namespace. In particular, it is not related to QEMU "id" used in -object/-device objects.

5.5 D-Bus display

QEMU can export the VM display through D-Bus (when started with `-display dbus`), to allow out-of-process UIs, remote protocol servers or other interactive display usages.

Various specialized D-Bus interfaces are available on different object paths under `/org/qemu/Display1/`, depending on the VM configuration.

QEMU also implements the standard interfaces, such as org.freedesktop.DBus.Introspectable.

5.5.1 org.qemu.Display1.VM interface

interface org.qemu.Display1.VM

This interface is implemented on /org/qemu/Display1/VM.

property Name:s

Access
read-only

Emits Changed
yes

The name of the VM.

property UUID:s

Access
read-only

Emits Changed
yes

The UUID of the VM.

property ConsoleIDs:au

Access
read-only

Emits Changed
yes

The list of consoles available on /org/qemu/Display1/Console_$id.

property Interfaces:as

Access
read-only

Emits Changed
yes

This property lists extra interfaces provided by the /org/qemu/Display1/VM object, and can be used to detect the capabilities with which they are communicating.

Unlike the standard D-Bus Introspectable interface, querying this property does not require parsing XML.

(earlier version of the display interface do not provide this property)

5.5.2 org.qemu.Display1.Console interface

interface org.qemu.Display1.Console

This interface is implemented on /org/qemu/Display1/Console_$id. You may discover available consoles through introspection or with the *org.qemu.Display1.VM.ConsoleIDs* property.

A console is attached to a video device head. It may be "Graphic" or "Text" (see *Type* and other properties).

Interactions with a console may be done with *org.qemu.Display1.Keyboard*, *org.qemu.Display1.Mouse* and *org.qemu.Display1.MultiTouch* interfaces when available.

method RegisterListener(*ay listener*, *h listener*) →

Arguments

- **listener** (*h*) – a Unix socket FD, for peer-to-peer D-Bus communication.
- **listener** – a Unix socket FD, for peer-to-peer D-Bus communication.

Register a console listener, which will receive display updates, until it is disconnected.

Multiple listeners may be registered simultaneously.

The listener is expected to implement the *org.qemu.Display1.Listener* interface.

method SetUIInfo(*q width_mm*, *q height_mm*, *i xoff*, *i yoff*, *u width*, *u height*) →

Arguments

- **width_mm** (*q*) – the physical display width in millimeters.
- **height_mm** (*q*) – the physical display height in millimeters.
- **xoff** (*i*) – horizontal offset, in pixels.
- **yoff** (*i*) – vertical offset, in pixels.
- **width** (*u*) – console width, in pixels.
- **height** (*u*) – console height, in pixels.

Modify the dimensions and display settings.

property Label:s

Access
read-only

Emits Changed
yes

A user-friendly name for the console (for ex: “VGA”).

property Head:u

Access
read-only

Emits Changed
yes

Graphical device head number.

property Type:s

Access
read-only

Emits Changed
yes

Console type (“Graphic” or “Text”).

property Width:u

Access
read-only

Emits Changed
yes

Console width, in pixels.

`property Height`:u

Access
read-only

Emits Changed
yes

Console height, in pixels.

`property DeviceAddress`:s

Access
read-only

Emits Changed
yes

The device address (ex: "pci/0000/02.0").

`property Interfaces`:as

Access
read-only

Emits Changed
yes

This property lists extra interfaces provided by the `/org/qemu/Display1/Console_$id` object, and can be used to detect the capabilities with which they are communicating.

Unlike the standard D-Bus Introspectable interface, querying this property does not require parsing XML.

(earlier version of the display interface do not provide this property)

5.5.3 org.qemu.Display1.Keyboard interface

`interface org.qemu.Display1.Keyboard`

This interface is optionally implemented on `/org/qemu/Display1/Console_$id` (see *Console*).

`method Press`(*u keycode*) →

Arguments

- **keycode** (*u*) – QEMU key number (xtkbd + special re-encoding of high bit)

Send a key press event.

`method Release`(*u keycode*) →

Arguments

- **keycode** (*u*) – QEMU key number (xtkbd + special re-encoding of high bit)

Send a key release event.

property Modifiers:u

Access
read-only

Emits Changed
yes

The active keyboard modifiers:

```
Scroll = 1 << 0
Num    = 1 << 1
Caps   = 1 << 2
```

5.5.4 org.qemu.Display1.Mouse interface

interface org.qemu.Display1.Mouse

This interface is optionally implemented on `/org/qemu/Display1/Console_$id` (see *Console* documentation).

Button values:

```
Left       = 0
Middle     = 1
Right      = 2
Wheel-up   = 3
Wheel-down = 4
Side       = 5
Extra      = 6
```

method Press(*u button*) →

Arguments

- **button** (*u*) – *button value*.

Send a mouse button press event.

method Release(*u button*) →

Arguments

- **button** (*u*) – *button value*.

Send a mouse button release event.

method SetAbsPosition(*u x*, *u y*) →

Arguments

- **x** (*u*) – X position, in pixels.
- **y** (*u*) – Y position, in pixels.

Set the mouse pointer position.

Returns an error if not *IsAbsolute*.

method RelMotion(*i dx*, *i dy*) →

Arguments

- **dx** (*i*) – X-delta, in pixels.
- **dy** (*i*) – Y-delta, in pixels.

Move the mouse pointer position, relative to the current position.

Returns an error if *IsAbsolute*.

property IsAbsolute:b

Access
read-only

Emits Changed
yes

Whether the mouse is using absolute movements.

5.5.5 org.qemu.Display1.MultiTouch interface

interface org.qemu.Display1.MultiTouch

This interface in implemented on `/org/qemu/Display1/Console_$id` (see *Console* documentation).

Kind values:

```
Begin       = 0
Update      = 1
End         = 2
Cancel      = 3
```

method SendEvent(*u kind*, *t num_slot*, *d x*, *d y*) →

Arguments

- **kind** (*u*) – The touch event kind
- **num_slot** (*t*) – The slot number.
- **x** (*d*) – The x coordinates.
- **y** (*d*) – The y coordinates.

Send a touch gesture event.

property MaxSlots:i

Access
read-only

Emits Changed
yes

The maximum number of slots.

5.5.6 org.qemu.Display1.Listener interface

interface org.qemu.Display1.Listener

This client-side interface must be available on `/org/qemu/Display1/Listener` when registering the peer-to-peer connection with *Register*.

method Scanout(*u width*, *u height*, *u stride*, *u pixman_format*, *ay data*) →

Arguments

- **width** (*u*) – display width, in pixels.
- **height** (*u*) – display height, in pixels.
- **stride** (*u*) – data stride, in bytes.
- **pixman_format** (*u*) – image format (ex: `PIXMAN_X8R8G8B8`).
- **data** (*ay*) – image data.

Resize and update the display content.

The data to transfer for the display update may be large. The preferred scanout method is *ScanoutDMABUF*, used whenever possible.

method Update(*i x*, *i y*, *i width*, *i height*, *u stride*, *u pixman_format*, *ay data*) →

Arguments

- **x** (*i*) – X update position, in pixels.
- **y** (*i*) – Y update position, in pixels.
- **width** (*i*) – update width, in pixels.
- **height** (*i*) – update height, in pixels.
- **stride** (*u*) – data stride, in bytes.
- **pixman_format** (*u*) – image format (ex: `PIXMAN_X8R8G8B8`).
- **data** (*ay*) – display image data.

Update the display content.

This method is only called after a *Scanout* call.

method ScanoutDMABUF(*h dmabuf*, *u width*, *u height*, *u stride*, *u fourcc*, *t modifier*, *b y0_top*) →

Arguments

- **dmabuf** (*h*) – the DMABUF file descriptor.
- **width** (*u*) – display width, in pixels.
- **height** (*u*) – display height, in pixels.
- **stride** (*u*) – stride, in bytes.
- **fourcc** (*u*) – DMABUF fourcc.
- **modifier** (*t*) – DMABUF modifier.
- **y0_top** (*b*) – whether Y position 0 is the top or not.

Resize and update the display content with a DMABUF.

`method UpdateDMABUF`(*i x*, *i y*, *i width*, *i height*) →

Arguments

- **x** (*i*) – the X update position, in pixels.
- **y** (*i*) – the Y update position, in pixels.
- **width** (*i*) – the update width, in pixels.
- **height** (*i*) – the update height, in pixels.

Update the display content with the current DMABUF and the given region.

`method Disable`() →

Disable the display (turn it off).

`method MouseSet`(*i x*, *i y*, *i on*) →

Arguments

- **x** (*i*) – X mouse position, in pixels.
- **y** (*i*) – Y mouse position, in pixels.
- **on** (*i*) – whether the mouse is visible or not.

Set the mouse position and visibility.

`method CursorDefine`(*i width*, *i height*, *i hot_x*, *i hot_y*, *ay data*) →

Arguments

- **width** (*i*) – cursor width, in pixels.
- **height** (*i*) – cursor height, in pixels.
- **hot_x** (*i*) – hot-spot X position, in pixels.
- **hot_y** (*i*) – hot-spot Y position, in pixels.
- **data** (*ay*) – the cursor data.

Set the mouse cursor shape and hot-spot. The "data" must be ARGB, 32-bit per pixel.

`property Interfaces`:as

Access
read-only

Emits Changed
yes

This property lists extra interfaces provided by the /org/qemu/Display1/Listener object, and can be used to detect the capabilities with which they are communicating.

Unlike the standard D-Bus Introspectable interface, querying this property does not require parsing XML.

(earlier version of the display interface do not provide this property)

5.5.7 org.qemu.Display1.Listener.Win32.Map interface

interface org.qemu.Display1.Listener.Win32.Map

This optional client-side interface can complement org.qemu.Display1.Listener on `/org/qemu/Display1/Listener` for Windows specific shared memory scanouts.

method ScanoutMap(*t handle*, *u offset*, *u width*, *u height*, *u stride*, *u pixman_format*) →

Arguments

- **handle** (*t*) – the shared map handle value.
- **offset** (*u*) – mapping offset.
- **width** (*u*) – display width, in pixels.
- **height** (*u*) – display height, in pixels.
- **stride** (*u*) – stride, in bytes.
- **pixman_format** (*u*) – image format (ex: `PIXMAN_X8R8G8B8`).

Resize and update the display content with a shared map.

method UpdateMap(*i x*, *i y*, *i width*, *i height*) →

Arguments

- **x** (*i*) – the X update position, in pixels.
- **y** (*i*) – the Y update position, in pixels.
- **width** (*i*) – the update width, in pixels.
- **height** (*i*) – the update height, in pixels.

Update the display content with the current shared map and the given region.

5.5.8 org.qemu.Display1.Listener.Win32.D3d11 interface

interface org.qemu.Display1.Listener.Win32.D3d11

This optional client-side interface can complement org.qemu.Display1.Listener on `/org/qemu/Display1/Listener` for Windows specific Direct3D texture sharing of the scanouts.

method ScanoutTexture2d(*t handle*, *u texture_width*, *u texture_height*, *b y0_top*, *u x*, *u y*, *u width*, *u height*) →

Arguments

- **handle** (*t*) – the NT handle for the shared texture (to be opened back with ID3D11Device1::OpenSharedResource1).
- **texture_width** (*u*) – texture width, in pixels.
- **texture_height** (*u*) – texture height, in pixels.
- **y0_top** (*b*) – whether Y position 0 is the top or not.
- **x** (*u*) – the X scanout position, in pixels.
- **y** (*u*) – the Y scanout position, in pixels.
- **width** (*u*) – the scanout width, in pixels.

- **height** (*u*) – the scanout height, in pixels.

Resize and update the display content with a Direct3D 11 2D texture. You must acquire and release the associated KeyedMutex 0 during rendering.

method UpdateTexture2d(*i x*, *i y*, *i width*, *i height*) →

Arguments

- **x** (*i*) – the X update position, in pixels.
- **y** (*i*) – the Y update position, in pixels.
- **width** (*i*) – the update width, in pixels.
- **height** (*i*) – the update height, in pixels.

Update the display content with the current Direct3D 2D texture and the given region. You must acquire and release the associated KeyedMutex 0 during rendering.

5.5.9 org.qemu.Display1.Clipboard interface

interface org.qemu.Display1.Clipboard

This interface must be implemented by both the client and the server on `/org/qemu/Display1/Clipboard` to support clipboard sharing between the client and the guest.

Once *Register*'ed, method calls may be sent and received in both directions. Unregistered callers will get error replies.

Selection values:

```
Clipboard   = 0
Primary     = 1
Secondary   = 2
```

Serial counter

To solve potential clipboard races, clipboard grabs have an associated serial counter. It is set to 0 on registration, and incremented by 1 for each grab. The peer with the highest serial is the clipboard grab owner.

When a grab with a lower serial is received, it should be discarded.

When a grab is attempted with the same serial number as the current grab, the one coming from the client should have higher priority, and the client should gain clipboard grab ownership.

method Register() →

Register a clipboard session and reinitialize the serial counter.

The client must register itself, and is granted an exclusive access for handling the clipboard.

The server can reinitialize the session as well (to reset the counter).

method Unregister() →

Unregister the clipboard session.

method Grab(*u selection*, *u serial*, *as mimes*) →

Arguments

- **selection** (*u*) – a *selection value*.
- **serial** (*u*) – the current grab *serial*.

- **mimes** (*as*) – the list of available content MIME types.

Grab the clipboard, claiming current clipboard content.

method Release(*u selection*) →

Arguments

- **selection** (*u*) – a *selection value*.

Release the clipboard (does nothing if not the current owner).

method Request(*u selection*, *as mimes*) → *s reply_mime*, *ay data*

Arguments

- **selection** (*u*) – a *selection value*
- **mimes** (*as*) – requested MIME types (by order of preference).

Returns

- **reply_mime** (*s*) – the returned data MIME type.
- **data** (*ay*) – the clipboard data.

Request the clipboard content.

Return an error if the clipboard is empty, or the requested MIME types are unavailable.

property Interfaces:as

Access
read-only

Emits Changed
yes

This property lists extra interfaces provided by the /org/qemu/Display1/Clipboard object, and can be used to detect the capabilities with which they are communicating.

Unlike the standard D-Bus Introspectable interface, querying this property does not require parsing XML.

(earlier version of the display interface do not provide this property)

5.5.10 org.qemu.Display1.Audio interface

interface org.qemu.Display1.Audio

Audio backend may be available on `/org/qemu/Display1/Audio`.

method RegisterOutListener(*ay listener*, *h listener*) →

Arguments

- **listener** (*h*) – a Unix socket FD, for peer-to-peer D-Bus communication.
- **listener** – a Unix socket FD, for peer-to-peer D-Bus communication.

Register an audio backend playback handler.

Multiple listeners may be registered simultaneously.

The listener is expected to implement the *org.qemu.Display1.AudioOutListener* interface.

method RegisterInListener(*ay listener*, *h listener*) →

Arguments

- **listener** (*h*) – a Unix socket FD, for peer-to-peer D-Bus communication.
- **listener** – a Unix socket FD, for peer-to-peer D-Bus communication.

Register an audio backend record handler.

Multiple listeners may be registered simultaneously.

The listener is expected to implement the *org.qemu.Display1.AudioInListener* interface.

property Interfaces:as

Access
read-only

Emits Changed
yes

This property lists extra interfaces provided by the /org/qemu/Display1/Audio object, and can be used to detect the capabilities with which they are communicating.

Unlike the standard D-Bus Introspectable interface, querying this property does not require parsing XML.

(earlier version of the display interface do not provide this property)

5.5.11 org.qemu.Display1.AudioOutListener interface

interface org.qemu.Display1.AudioOutListener

This client-side interface must be available on `/org/qemu/Display1/AudioOutListener` when registering the peer-to-peer connection with *RegisterOutListener*.

method Init(*t id*, *y bits*, *b is_signed*, *b is_float*, *u freq*, *y nchannels*, *u bytes_per_frame*, *u bytes_per_second*, *b be*) →

Arguments

- **id** (*t*) – the stream ID.
- **bits** (*y*) – PCM bits per sample.
- **is_signed** (*b*) – whether the PCM data is signed.
- **is_float** (*b*) – PCM floating point format.
- **freq** (*u*) – the PCM frequency in Hz.
- **nchannels** (*y*) – the number of channels.
- **bytes_per_frame** (*u*) – the bytes per frame.
- **bytes_per_second** (*u*) – the bytes per second.
- **be** (*b*) – whether using big-endian format.

Initializes a PCM playback stream.

method Fini(*t id*) →

Arguments

- **id** (*t*) – the stream ID.

Finish & close a playback stream.

method SetEnabled(*t id*, *b enabled*) →

Arguments

- **id** (*t*) – the stream ID.
- **enabled** (*b*) –

Resume or suspend the playback stream.

method SetVolume(*t id*, *b mute*, *ay volume*) →

Arguments

- **id** (*t*) – the stream ID.
- **mute** (*b*) – whether the stream is muted.
- **volume** (*ay*) – the volume per-channel.

Set the stream volume and mute state (volume without unit, 0-255).

method Write(*t id*, *ay data*) →

Arguments

- **id** (*t*) – the stream ID.
- **data** (*ay*) – the PCM data.

PCM stream to play.

property Interfaces:as

Access
read-only

Emits Changed
yes

This property lists extra interfaces provided by the /org/qemu/Display1/AudioOutListener object, and can be used to detect the capabilities with which they are communicating.

Unlike the standard D-Bus Introspectable interface, querying this property does not require parsing XML.

(earlier version of the display interface do not provide this property)

5.5.12 org.qemu.Display1.AudioInListener interface

interface org.qemu.Display1.AudioInListener

This client-side interface must be available on `/org/qemu/Display1/AudioInListener` when registering the peer-to-peer connection with *RegisterInListener*.

method Init(*t id*, *y bits*, *b is_signed*, *b is_float*, *u freq*, *y nchannels*, *u bytes_per_frame*, *u bytes_per_second*, *b be*) →

Arguments

- **id** (*t*) – the stream ID.
- **bits** (*y*) – PCM bits per sample.
- **is_signed** (*b*) – whether the PCM data is signed.

- **is_float** (*b*) – PCM floating point format.
- **freq** (*u*) – the PCM frequency in Hz.
- **nchannels** (*y*) – the number of channels.
- **bytes_per_frame** (*u*) – the bytes per frame.
- **bytes_per_second** (*u*) – the bytes per second.
- **be** (*b*) – whether using big-endian format.

Initializes a PCM record stream.

method Fini(*t id*) →

Arguments

- **id** (*t*) – the stream ID.

Finish & close a record stream.

method SetEnabled(*t id*, *b enabled*) →

Arguments

- **id** (*t*) – the stream ID.
- **enabled** (*b*) –

Resume or suspend the record stream.

method SetVolume(*t id*, *b mute*, *ay volume*) →

Arguments

- **id** (*t*) – the stream ID.
- **mute** (*b*) – whether the stream is muted.
- **volume** (*ay*) – the volume per-channel.

Set the stream volume and mute state (volume without unit, 0-255).

method Read(*t id*, *t size*) → *ay data*

Arguments

- **id** (*t*) – the stream ID.
- **size** (*t*) – the amount to read, in bytes.

Returns

- **data** (*ay*) – the recorded data (which may be less than requested).

Read "size" bytes from the record stream.

property Interfaces:as

Access
read-only

Emits Changed
yes

This property lists extra interfaces provided by the /org/qemu/Display1/AudioInListener object, and can be used to detect the capabilities with which they are communicating.

Unlike the standard D-Bus Introspectable interface, querying this property does not require parsing XML.

(earlier version of the display interface do not provide this property)

5.5.13 org.qemu.Display1.Chardev interface

`interface org.qemu.Display1.Chardev`

Character devices may be available on `/org/qemu/Display1/Chardev_$id`.

They may be used for different kind of streams, which are identified via their FQDN *Name*.

Here are some known reserved kind names (the `org.qemu` prefix is reserved by QEMU):

org.qemu.console.serial.0
A serial console stream.

org.qemu.monitor.hmp.0
A QEMU HMP human monitor.

org.qemu.monitor.qmp.0
A QEMU QMP monitor.

org.qemu.usbredir
A usbredir stream.

`method Register`(*ay listener*, *h stream*) →

Arguments

- `listener` (*ay*) –
- `stream` (*h*) – a Unix FD to redirect the stream to.

Register a file-descriptor for the stream handling.

The current handler, if any, will be replaced.

`method SendBreak`() →

Send a break event to the character device.

`property Name`:s

Access
read-only

Emits Changed
yes

The FQDN name to identify the kind of stream. See *reserved names*.

`property FEOpened`:b

Access
read-only

Emits Changed
yes

Whether the front-end side is opened.

`property Echo`:b

Access
read-only

Emits Changed
yes

Whether the input should be echo'ed (for serial streams).

`property Owner`:s

Access
read-only

Emits Changed
yes

The D-Bus unique name of the registered handler.

`property Interfaces`:as

Access
read-only

Emits Changed
yes

This property lists extra interfaces provided by the `/org/qemu/Display1/Chardev_$i` object, and can be used to detect the capabilities with which they are communicating.

Unlike the standard D-Bus Introspectable interface, querying this property does not require parsing XML.

(earlier version of the display interface do not provide this property)

5.6 Live Block Device Operations

QEMU Block Layer currently (as of QEMU 2.9) supports four major kinds of live block device jobs – stream, commit, mirror, and backup. These can be used to manipulate disk image chains to accomplish certain tasks, namely: live copy data from backing files into overlays; shorten long disk image chains by merging data from overlays into backing files; live synchronize data from a disk image chain (including current active disk) to another target image; and point-in-time (and incremental) backups of a block device. Below is a description of the said block (QMP) primitives, and some (non-exhaustive list of) examples to illustrate their use.

Note: The file `qapi/block-core.json` in the QEMU source tree has the canonical QEMU API (QAPI) schema documentation for the QMP primitives discussed here.

Contents

5.6.1 Disk image backing chain notation

A simple disk image chain. (This can be created live using QMP `blockdev-snapshot-sync`, or offline via `qemu-img`):

```
                (Live QEMU)
                     |
                     .
                     V

        [A] <----- [B]

(backing file)    (overlay)
```

The arrow can be read as: Image [A] is the backing file of disk image [B]. And live QEMU is currently writing to image [B], consequently, it is also referred to as the "active layer".

There are two kinds of terminology that are common when referring to files in a disk image backing chain:

(1) Directional: 'base' and 'top'. Given the simple disk image chain above, image [A] can be referred to as 'base', and image [B] as 'top'. (This terminology can be seen in the QAPI schema file, block-core.json.)

(2) Relational: 'backing file' and 'overlay'. Again, taking the same simple disk image chain from the above, disk image [A] is referred to as the backing file, and image [B] as overlay.

Throughout this document, we will use the relational terminology.

Important: The overlay files can generally be any format that supports a backing file, although QCOW2 is the preferred

format and the one used in this document.

5.6.2 Brief overview of live block QMP primitives

The following are the four different kinds of live block operations that QEMU block layer supports.

(1) `block-stream`: Live copy of data from backing files into overlay files.

Note: Once the 'stream' operation has finished, three things to note:

(a) QEMU rewrites the backing chain to remove reference to the now-streamed and redundant backing file;

(b) the streamed file *itself* won't be removed by QEMU, and must be explicitly discarded by the user;

(c) the streamed file remains valid – i.e. further overlays can be created based on it. Refer the `block-stream` section further below for more details.

---

(2) `block-commit`: Live merge of data from overlay files into backing files (with the optional goal of removing the overlay file from the chain). Since QEMU 2.0, this includes "active `block-commit`" (i.e. merge the current active layer into the base image).

Note: Once the 'commit' operation has finished, there are three things to note here as well:

(a) QEMU rewrites the backing chain to remove reference to now-redundant overlay images that have been committed into a backing file;

(b) the committed file *itself* won't be removed by QEMU – it ought to be manually removed;

(c) however, unlike in the case of `block-stream`, the intermediate images will be rendered invalid – i.e. no more further overlays can be created based on them. Refer the `block-commit` section further below for more details.

---

(3) `drive-mirror` (and `blockdev-mirror`): Synchronize a running disk to another image.

(4) `blockdev-backup` (and the deprecated `drive-backup`): Point-in-time (live) copy of a block device to a destination.

5.6.3 Interacting with a QEMU instance

To show some example invocations of command-line, we will use the following invocation of QEMU, with a QMP server running over UNIX socket:

```
$ qemu-system-x86_64 -display none -no-user-config -nodefaults \
  -m 512 -blockdev \
  node-name=node-A,driver=qcow2,file.driver=file,file.node-name=file,file.filename=./a.
↪qcow2 \
  -device virtio-blk,drive=node-A,id=virtio0 \
  -monitor stdio -qmp unix:/tmp/qmp-sock,server=on,wait=off
```

The `-blockdev` command-line option, used above, is available from QEMU 2.9 onwards. In the above invocation, notice the `node-name` parameter that is used to refer to the disk image a.qcow2 ('node-A') – this is a cleaner way to refer to a disk image (as opposed to referring to it by spelling out file paths). So, we will continue to designate a `node-name` to each further disk image created (either via `blockdev-snapshot-sync`, or `blockdev-add`) as part of

the disk image chain, and continue to refer to the disks using their `node-name` (where possible, because `block-commit` does not yet, as of QEMU 2.9, accept `node-name` parameter) when performing various block operations.

To interact with the QEMU instance launched above, we will use the `qmp-shell` utility (located at: `qemu/scripts/qmp`, as part of the QEMU source directory), which takes key-value pairs for QMP commands. Invoke it as below (which will also print out the complete raw JSON syntax for reference – examples in the following sections):

```
$ ./qmp-shell -v -p /tmp/qmp-sock
(QEMU)
```

Note: In the event we have to repeat a certain QMP command, we will: for the first occurrence of it, show the `qmp-shell` invocation, *and* the corresponding raw JSON QMP syntax; but for subsequent invocations, present just the `qmp-shell` syntax, and omit the equivalent JSON output.

5.6.4 Example disk image chain

We will use the below disk image chain (and occasionally spelling it out where appropriate) when discussing various primitives:

```
[A] <-- [B] <-- [C] <-- [D]
```

Where [A] is the original base image; [B] and [C] are intermediate overlay images; image [D] is the active layer – i.e. live QEMU is writing to it. (The rule of thumb is: live QEMU will always be pointing to the rightmost image in a disk image chain.)

The above image chain can be created by invoking `blockdev-snapshot-sync` commands as following (which shows the creation of overlay image [B]) using the `qmp-shell` (our invocation also prints the raw JSON invocation of it):

```
(QEMU) blockdev-snapshot-sync node-name=node-A snapshot-file=b.qcow2 snapshot-node-
↪name=node-B format=qcow2
{
    "execute": "blockdev-snapshot-sync",
    "arguments": {
        "node-name": "node-A",
        "snapshot-file": "b.qcow2",
        "format": "qcow2",
        "snapshot-node-name": "node-B"
    }
}
```

Here, "node-A" is the name QEMU internally uses to refer to the base image [A] – it is the backing file, based on which the overlay image, [B], is created.

To create the rest of the overlay images, [C], and [D] (omitting the raw JSON output for brevity):

```
(QEMU) blockdev-snapshot-sync node-name=node-B snapshot-file=c.qcow2 snapshot-node-
↪name=node-C format=qcow2
(QEMU) blockdev-snapshot-sync node-name=node-C snapshot-file=d.qcow2 snapshot-node-
↪name=node-D format=qcow2
```

5.6.5 A note on points-in-time vs file names

In our disk image chain:

```
[A] <-- [B] <-- [C] <-- [D]
```

We have *three* points in time and an active layer:

- Point 1: Guest state when [B] was created is contained in file [A]
- Point 2: Guest state when [C] was created is contained in [A] + [B]
- Point 3: Guest state when [D] was created is contained in [A] + [B] + [C]
- Active layer: Current guest state is contained in [A] + [B] + [C] + [D]

Therefore, be aware with naming choices:

- Naming a file after the time it is created is misleading – the guest data for that point in time is *not* contained in that file (as explained earlier)
- Rather, think of files as a *delta* from the backing file

5.6.6 Live block streaming — `block-stream`

The `block-stream` command allows you to do live copy data from backing files into overlay images.

Given our original example disk image chain from earlier:

```
[A] <-- [B] <-- [C] <-- [D]
```

The disk image chain can be shortened in one of the following different ways (not an exhaustive list).

(1) Merge everything into the active layer: I.e. copy all contents from the base image, [A], and overlay images, [B] and [C], into [D], *while* the guest is running. The resulting chain will be a standalone image, [D] – with contents from [A], [B] and [C] merged into it (where live QEMU writes go to):

```
[D]
```

(2) Taking the same example disk image chain mentioned earlier, merge only images [B] and [C] into [D], the active layer. The result will be contents of images [B] and [C] will be copied into [D], and the backing file pointer of image [D] will be adjusted to point to image [A]. The resulting chain will be:

```
[A] <-- [D]
```

(3) Intermediate streaming (available since QEMU 2.8): Starting afresh with the original example disk image chain, with a total of four images, it is possible to copy contents from image [B] into image [C]. Once the copy is finished, image [B] can now be (optionally) discarded; and the backing file pointer of image [C] will be adjusted to point to [A]. I.e. after performing "intermediate streaming" of [B] into [C], the resulting image chain will be (where live QEMU is writing to [D]):

```
[A] <-- [C] <-- [D]
```

QMP invocation for `block-stream`

For *Case-1*, to merge contents of all the backing files into the active layer, where 'node-D' is the current active image (by default `block-stream` will flatten the entire chain); `qmp-shell` (and its corresponding JSON output):

```
(QEMU) block-stream device=node-D job-id=job0
{
    "execute": "block-stream",
    "arguments": {
        "device": "node-D",
        "job-id": "job0"
    }
}
```

For *Case-2*, merge contents of the images [B] and [C] into [D], where image [D] ends up referring to image [A] as its backing file:

```
(QEMU) block-stream device=node-D base-node=node-A job-id=job0
```

And for *Case-3*, of "intermediate" streaming", merge contents of images [B] into [C], where [C] ends up referring to [A] as its backing image:

```
(QEMU) block-stream device=node-C base-node=node-A job-id=job0
```

Progress of a `block-stream` operation can be monitored via the QMP command:

```
(QEMU) query-block-jobs
{
    "execute": "query-block-jobs",
    "arguments": {}
}
```

Once the `block-stream` operation has completed, QEMU will emit an event, `BLOCK_JOB_COMPLETED`. The intermediate overlays remain valid, and can now be (optionally) discarded, or retained to create further overlays based on them. Finally, the `block-stream` jobs can be restarted at anytime.

5.6.7 Live block commit — `block-commit`

The `block-commit` command lets you merge live data from overlay images into backing file(s). Since QEMU 2.0, this includes "live active commit" (i.e. it is possible to merge the "active layer", the right-most image in a disk image chain where live QEMU will be writing to, into the base image). This is analogous to `block-stream`, but in the opposite direction.

Again, starting afresh with our example disk image chain, where live QEMU is writing to the right-most image in the chain, [D]:

```
[A] <-- [B] <-- [C] <-- [D]
```

The disk image chain can be shortened in one of the following ways:

(1) Commit content from only image [B] into image [A]. The resulting chain is the following, where image [C] is adjusted to point at [A] as its new backing file:

```
[A] <-- [C] <-- [D]
```

(2) Commit content from images [B] and [C] into image [A]. The resulting chain, where image [D] is adjusted to point to image [A] as its new backing file:

```
[A] <-- [D]
```

(3) Commit content from images [B], [C], and the active layer [D] into image [A]. The resulting chain (in this case, a consolidated single image):

```
[A]
```

(4) Commit content from image only image [C] into image [B]. The resulting chain:

```
[A] <-- [B] <-- [D]
```

(5) Commit content from image [C] and the active layer [D] into image [B]. The resulting chain:

```
[A] <-- [B]
```

QMP invocation for `block-commit`

For *Case-1*, to merge contents only from image [B] into image [A], the invocation is as follows:

```
(QEMU) block-commit device=node-D base=a.qcow2 top=b.qcow2 job-id=job0
{
    "execute": "block-commit",
    "arguments": {
        "device": "node-D",
        "job-id": "job0",
        "top": "b.qcow2",
        "base": "a.qcow2"
    }
}
```

Once the above `block-commit` operation has completed, a `BLOCK_JOB_COMPLETED` event will be issued, and no further action is required. As the end result, the backing file of image [C] is adjusted to point to image [A], and the original 4-image chain will end up being transformed to:

```
[A] <-- [C] <-- [D]
```

Note: The intermediate image [B] is invalid (as in: no more further overlays based on it can be created).

Reasoning: An intermediate image after a 'stream' operation still represents that old point-in-time, and may be valid in that context. However, an intermediate image after a 'commit' operation no longer represents any point-in-time, and is invalid in any context.

However, *Case-3* (also called: "active `block-commit`") is a *two-phase* operation: In the first phase, the content from the active overlay, along with the intermediate overlays, is copied into the backing file (also called the base image). In the second phase, adjust the said backing file as the current active image – possible via issuing the command `block-job-complete`. Optionally, the `block-commit` operation can be cancelled by issuing the command `block-job-cancel`, but be careful when doing this.

Once the `block-commit` operation has completed, the event `BLOCK_JOB_READY` will be emitted, signalling that the synchronization has finished. Now the job can be gracefully completed by issuing the command `block-job-complete` – until such a command is issued, the 'commit' operation remains active.

The following is the flow for *Case-3* to convert a disk image chain such as this:

```
[A] <-- [B] <-- [C] <-- [D]
```

Into:

```
[A]
```

Where content from all the subsequent overlays, [B], and [C], including the active layer, [D], is committed back to [A] – which is where live QEMU is performing all its current writes).

Start the "active `block-commit`" operation:

```
(QEMU) block-commit device=node-D base=a.qcow2 top=d.qcow2 job-id=job0
{
    "execute": "block-commit",
    "arguments": {
        "device": "node-D",
        "job-id": "job0",
        "top": "d.qcow2",
        "base": "a.qcow2"
    }
}
```

Once the synchronization has completed, the event `BLOCK_JOB_READY` will be emitted.

Then, optionally query for the status of the active block operations. We can see the 'commit' job is now ready to be completed, as indicated by the line *"ready": true*:

```
(QEMU) query-block-jobs
{
    "execute": "query-block-jobs",
    "arguments": {}
}
{
    "return": [
        {
            "busy": false,
            "type": "commit",
            "len": 1376256,
            "paused": false,
            "ready": true,
            "io-status": "ok",
            "offset": 1376256,
            "device": "job0",
            "speed": 0
        }
    ]
}
```

Gracefully complete the 'commit' block device job:

```
(QEMU) block-job-complete device=job0
{
    "execute": "block-job-complete",
```

(continues on next page)

(continued from previous page)

```
    "arguments": {
        "device": "job0"
    }
}
{
    "return": {}
}
```

Finally, once the above job is completed, an event `BLOCK_JOB_COMPLETED` will be emitted.

Note: The invocation for rest of the cases (2, 4, and 5), discussed in the previous section, is omitted for brevity.

5.6.8 Live disk synchronization — `drive-mirror` and `blockdev-mirror`

Synchronize a running disk image chain (all or part of it) to a target image.

Again, given our familiar disk image chain:

```
[A] <-- [B] <-- [C] <-- [D]
```

The `drive-mirror` (and its newer equivalent `blockdev-mirror`) allows you to copy data from the entire chain into a single target image (which can be located on a different host), [E].

Note: When you cancel an in-progress 'mirror' job *before* the source and target are synchronized, `block-job-cancel` will emit the event `BLOCK_JOB_CANCELLED`. However, note that if you cancel a 'mirror' job *after* it has indicated (via the event `BLOCK_JOB_READY`) that the source and target have reached synchronization, then the event emitted by `block-job-cancel` changes to `BLOCK_JOB_COMPLETED`.

Besides the 'mirror' job, the "active `block-commit`" is the only other block device job that emits the event `BLOCK_JOB_READY`. The rest of the block device jobs ('stream', "non-active `block-commit`", and 'backup') end automatically.

So there are two possible actions to take, after a 'mirror' job has emitted the event `BLOCK_JOB_READY`, indicating that the source and target have reached synchronization:

(1) Issuing the command `block-job-cancel` (after it emits the event `BLOCK_JOB_COMPLETED`) will create a point-in-time (which is at the time of *triggering* the cancel command) copy of the entire disk image chain (or only the top-most image, depending on the `sync` mode), contained in the target image [E]. One use case for this is live VM migration with non-shared storage.

(2) Issuing the command `block-job-complete` (after it emits the event `BLOCK_JOB_COMPLETED`) will adjust the guest device (i.e. live QEMU) to point to the target image, [E], causing all the new writes from this point on to happen there.

About synchronization modes: The synchronization mode determines *which* part of the disk image chain will be copied to the target. Currently, there are four different kinds:

(1) `full` – Synchronize the content of entire disk image chain to the target

(2) `top` – Synchronize only the contents of the top-most disk image in the chain to the target

(3) `none` – Synchronize only the new writes from this point on.

Note: In the case of `blockdev-backup` (or deprecated `drive-backup`), the behavior of `none` synchronization mode is different. Normally, a `backup` job consists of two parts: Anything that is overwritten by the guest is first copied out to the backup, and in the background the whole image is copied from start to end. With `sync=none`, it's only the first part.

(4) `incremental` – Synchronize content that is described by the dirty bitmap

Note: Refer to the *Dirty Bitmaps and Incremental Backup* document in the QEMU source tree to learn about the detailed workings of the `incremental` synchronization mode.

QMP invocation for `drive-mirror`

To copy the contents of the entire disk image chain, from [A] all the way to [D], to a new target (`drive-mirror` will create the destination file, if it doesn't already exist), call it [E]:

```
(QEMU) drive-mirror device=node-D target=e.qcow2 sync=full job-id=job0
{
    "execute": "drive-mirror",
    "arguments": {
        "device": "node-D",
        "job-id": "job0",
        "target": "e.qcow2",
        "sync": "full"
    }
}
```

The `"sync": "full"`, from the above, means: copy the *entire* chain to the destination.

Following the above, querying for active block jobs will show that a 'mirror' job is "ready" to be completed (and QEMU will also emit an event, `BLOCK_JOB_READY`):

```
(QEMU) query-block-jobs
{
    "execute": "query-block-jobs",
    "arguments": {}
}
{
    "return": [
        {
            "busy": false,
            "type": "mirror",
            "len": 21757952,
            "paused": false,
            "ready": true,
            "io-status": "ok",
            "offset": 21757952,
            "device": "job0",
            "speed": 0
        }
    ]
}
```

And, as noted in the previous section, there are two possible actions at this point:

(a) Create a point-in-time snapshot by ending the synchronization. The point-in-time is at the time of *ending* the sync. (The result of the following being: the target image, [E], will be populated with content from the entire chain, [A] to [D]):

```
(QEMU) block-job-cancel device=job0
{
    "execute": "block-job-cancel",
    "arguments": {
        "device": "job0"
    }
}
```

(b) Or, complete the operation and pivot the live QEMU to the target copy:

```
(QEMU) block-job-complete device=job0
```

In either of the above cases, if you once again run the `query-block-jobs` command, there should not be any active block operation.

Comparing 'commit' and 'mirror': In both then cases, the overlay images can be discarded. However, with 'commit', the *existing* base image will be modified (by updating it with contents from overlays); while in the case of 'mirror', a *new* target image is populated with the data from the disk image chain.

QMP invocation for live storage migration with `drive-mirror` + NBD

Live storage migration (without shared storage setup) is one of the most common use-cases that takes advantage of the `drive-mirror` primitive and QEMU's built-in Network Block Device (NBD) server. Here's a quick walk-through of this setup.

Given the disk image chain:

```
[A] <-- [B] <-- [C] <-- [D]
```

Instead of copying content from the entire chain, synchronize *only* the contents of the *top*-most disk image (i.e. the active layer), [D], to a target, say, [TargetDisk].

Important: The destination host must already have the contents of the backing chain, involving images [A], [B], and [C], visible via other means – whether by `cp`, `rsync`, or by some storage array-specific command.)

Sometimes, this is also referred to as "shallow copy" – because only the "active layer", and not the rest of the image chain, is copied to the destination.

Note: In this example, for the sake of simplicity, we'll be using the same `localhost` as both source and destination.

As noted earlier, on the destination host the contents of the backing chain – from images [A] to [C] – are already expected to exist in some form (e.g. in a file called, `Contents-of-A-B-C.qcow2`). Now, on the destination host, let's create a target overlay image (with the image `Contents-of-A-B-C.qcow2` as its backing file), to which the contents of image [D] (from the source QEMU) will be mirrored to:

```
$ qemu-img create -f qcow2 -b ./Contents-of-A-B-C.qcow2 \
    -F qcow2 ./target-disk.qcow2
```

And start the destination QEMU (we already have the source QEMU running – discussed in the section: *Interacting with a QEMU instance*) instance, with the following invocation. (As noted earlier, for simplicity's sake, the destination QEMU is started on the same host, but it could be located elsewhere):

```
$ qemu-system-x86_64 -display none -no-user-config -nodefaults \
  -m 512 -blockdev \
  node-name=node-TargetDisk,driver=qcow2,file.driver=file,file.node-name=file,file.
↪filename=./target-disk.qcow2 \
  -device virtio-blk,drive=node-TargetDisk,id=virtio0 \
  -S -monitor stdio -qmp unix:./qmp-sock2,server=on,wait=off \
  -incoming tcp:localhost:6666
```

Given the disk image chain on source QEMU:

```
[A] <-- [B] <-- [C] <-- [D]
```

On the destination host, it is expected that the contents of the chain `[A] <-- [B] <-- [C]` are *already* present, and therefore copy *only* the content of image [D].

(1) [On *destination* QEMU] As part of the first step, start the built-in NBD server on a given host (local host, represented by `::`)and port:

```
(QEMU) nbd-server-start addr={"type":"inet","data":{"host":"::","port":"49153"}}
{
    "execute": "nbd-server-start",
    "arguments": {
        "addr": {
            "data": {
                "host": "::",
                "port": "49153"
            },
            "type": "inet"
        }
    }
}
```

(2) [On *destination* QEMU] And export the destination disk image using QEMU's built-in NBD server:

```
(QEMU) nbd-server-add device=node-TargetDisk writable=true
{
    "execute": "nbd-server-add",
    "arguments": {
        "device": "node-TargetDisk"
    }
}
```

(3) [On *source* QEMU] Then, invoke `drive-mirror` (NB: since we're running `drive-mirror` with `mode=existing` (meaning: synchronize to a pre-created file, therefore 'existing', file on the target host), with the synchronization mode as 'top' (`"sync: "top"`):

```
(QEMU) drive-mirror device=node-D target=nbd:localhost:49153:exportname=node-
↪TargetDisk sync=top mode=existing job-id=job0
{
    "execute": "drive-mirror",
    "arguments": {
```

(continues on next page)

(continued from previous page)

```
        "device": "node-D",
        "mode": "existing",
        "job-id": "job0",
        "target": "nbd:localhost:49153:exportname=node-TargetDisk",
        "sync": "top"
    }
}
```

(4) [On *source* QEMU] Once `drive-mirror` copies the entire data, and the event `BLOCK_JOB_READY` is emitted, issue `block-job-cancel` to gracefully end the synchronization, from source QEMU:

```
(QEMU) block-job-cancel device=job0
{
    "execute": "block-job-cancel",
    "arguments": {
        "device": "job0"
    }
}
```

(5) [On *destination* QEMU] Then, stop the NBD server:

```
(QEMU) nbd-server-stop
{
    "execute": "nbd-server-stop",
    "arguments": {}
}
```

(6) [On *destination* QEMU] Finally, resume the guest vCPUs by issuing the QMP command `cont`:

```
(QEMU) cont
{
    "execute": "cont",
    "arguments": {}
}
```

Note: Higher-level libraries (e.g. libvirt) automate the entire above process (although note that libvirt does not allow same-host migrations to localhost for other reasons).

Notes on `blockdev-mirror`

The `blockdev-mirror` command is equivalent in core functionality to `drive-mirror`, except that it operates at node-level in a BDS graph.

Also: for `blockdev-mirror`, the 'target' image needs to be explicitly created (using `qemu-img`) and attach it to live QEMU via `blockdev-add`, which assigns a name to the to-be created target node.

E.g. the sequence of actions to create a point-in-time backup of an entire disk image chain, to a target, using `blockdev-mirror` would be:

(0) Create the QCOW2 overlays, to arrive at a backing chain of desired depth

(1) Create the target image (using `qemu-img`), say, `e.qcow2`

(2) Attach the above created file (`e.qcow2`), run-time, using `blockdev-add` to QEMU

(3) Perform `blockdev-mirror` (use `"sync": "full"` to copy the entire chain to the target). And notice the event `BLOCK_JOB_READY`

(4) Optionally, query for active block jobs, there should be a 'mirror' job ready to be completed

(5) Gracefully complete the 'mirror' block device job, and notice the event `BLOCK_JOB_COMPLETED`

(6) Shutdown the guest by issuing the QMP `quit` command so that caches are flushed

(7) Then, finally, compare the contents of the disk image chain, and the target copy with `qemu-img compare`. You should notice: "Images are identical"

QMP invocation for `blockdev-mirror`

Given the disk image chain:

```
[A] <-- [B] <-- [C] <-- [D]
```

To copy the contents of the entire disk image chain, from [A] all the way to [D], to a new target, call it [E]. The following is the flow.

Create the overlay images, [B], [C], and [D]:

```
(QEMU) blockdev-snapshot-sync node-name=node-A snapshot-file=b.qcow2 snapshot-node-
↪name=node-B format=qcow2
(QEMU) blockdev-snapshot-sync node-name=node-B snapshot-file=c.qcow2 snapshot-node-
↪name=node-C format=qcow2
(QEMU) blockdev-snapshot-sync node-name=node-C snapshot-file=d.qcow2 snapshot-node-
↪name=node-D format=qcow2
```

Create the target image, [E]:

```
$ qemu-img create -f qcow2 e.qcow2 39M
```

Add the above created target image to QEMU, via `blockdev-add`:

```
(QEMU) blockdev-add driver=qcow2 node-name=node-E file={"driver":"file","filename":"e.
↪qcow2"}
{
    "execute": "blockdev-add",
    "arguments": {
        "node-name": "node-E",
        "driver": "qcow2",
        "file": {
            "driver": "file",
            "filename": "e.qcow2"
        }
    }
}
```

Perform `blockdev-mirror`, and notice the event `BLOCK_JOB_READY`:

```
(QEMU) blockdev-mirror device=node-B target=node-E sync=full job-id=job0
{
```

(continues on next page)

(continued from previous page)

```
    "execute": "blockdev-mirror",
    "arguments": {
        "device": "node-D",
        "job-id": "job0",
        "target": "node-E",
        "sync": "full"
    }
}
```

Query for active block jobs, there should be a 'mirror' job ready:

```
(QEMU) query-block-jobs
{
    "execute": "query-block-jobs",
    "arguments": {}
}
{
    "return": [
        {
            "busy": false,
            "type": "mirror",
            "len": 21561344,
            "paused": false,
            "ready": true,
            "io-status": "ok",
            "offset": 21561344,
            "device": "job0",
            "speed": 0
        }
    ]
}
```

Gracefully complete the block device job operation, and notice the event `BLOCK_JOB_COMPLETED`:

```
(QEMU) block-job-complete device=job0
{
    "execute": "block-job-complete",
    "arguments": {
        "device": "job0"
    }
}
{
    "return": {}
}
```

Shutdown the guest, by issuing the `quit` QMP command:

```
(QEMU) quit
{
    "execute": "quit",
    "arguments": {}
}
```

5.6.9 Live disk backup — `blockdev-backup` and the deprecated``drive-backup``

The `blockdev-backup` (and the deprecated `drive-backup`) allows you to create a point-in-time snapshot.

In this case, the point-in-time is when you *start* the `blockdev-backup` (or deprecated `drive-backup`) command.

QMP invocation for `drive-backup`

Note that `drive-backup` command is deprecated since QEMU 6.2 and will be removed in future.

Yet again, starting afresh with our example disk image chain:

```
[A] <-- [B] <-- [C] <-- [D]
```

To create a target image [E], with content populated from image [A] to [D], from the above chain, the following is the syntax. (If the target image does not exist, `drive-backup` will create it):

```
(QEMU) drive-backup device=node-D sync=full target=e.qcow2 job-id=job0
{
    "execute": "drive-backup",
    "arguments": {
        "device": "node-D",
        "job-id": "job0",
        "sync": "full",
        "target": "e.qcow2"
    }
}
```

Once the above `drive-backup` has completed, a `BLOCK_JOB_COMPLETED` event will be issued, indicating the live block device job operation has completed, and no further action is required.

Moving from the deprecated `drive-backup` to newer `blockdev-backup`

`blockdev-backup` differs from `drive-backup` in how you specify the backup target. With `blockdev-backup` you can't specify filename as a target. Instead you use `node-name` of existing block node, which you may add by `blockdev-add` or `blockdev-create` commands. Correspondingly, `blockdev-backup` doesn't have `mode` and `format` arguments which don't apply to an existing block node. See following sections for details and examples.

Notes on `blockdev-backup`

The `blockdev-backup` command operates at node-level in a Block Driver State (BDS) graph.

E.g. the sequence of actions to create a point-in-time backup of an entire disk image chain, to a target, using `blockdev-backup` would be:

(0) Create the QCOW2 overlays, to arrive at a backing chain of desired depth

(1) Create the target image (using `qemu-img`), say, `e.qcow2`

(2) Attach the above created file (`e.qcow2`), run-time, using `blockdev-add` to QEMU

(3) Perform `blockdev-backup` (use `"sync": "full"` to copy the entire chain to the target). And notice the event `BLOCK_JOB_COMPLETED`

(4) Shutdown the guest, by issuing the QMP `quit` command, so that caches are flushed

(5) Then, finally, compare the contents of the disk image chain, and the target copy with `qemu-img compare`. You should notice: "Images are identical"

The following section shows an example QMP invocation for `blockdev-backup`.

QMP invocation for `blockdev-backup`

Given a disk image chain of depth 1 where image [B] is the active overlay (live QEMU is writing to it):

```
[A] <-- [B]
```

The following is the procedure to copy the content from the entire chain to a target image (say, [E]), which has the full content from [A] and [B].

Create the overlay [B]:

```
(QEMU) blockdev-snapshot-sync node-name=node-A snapshot-file=b.qcow2 snapshot-node-
↪name=node-B format=qcow2
{
    "execute": "blockdev-snapshot-sync",
    "arguments": {
        "node-name": "node-A",
        "snapshot-file": "b.qcow2",
        "format": "qcow2",
        "snapshot-node-name": "node-B"
    }
}
```

Create a target image that will contain the copy:

```
$ qemu-img create -f qcow2 e.qcow2 39M
```

Then add it to QEMU via `blockdev-add`:

```
(QEMU) blockdev-add driver=qcow2 node-name=node-E file={"driver":"file","filename":"e.
↪qcow2"}
{
    "execute": "blockdev-add",
    "arguments": {
        "node-name": "node-E",
        "driver": "qcow2",
        "file": {
            "driver": "file",
            "filename": "e.qcow2"
        }
    }
}
```

Then invoke `blockdev-backup` to copy the contents from the entire image chain, consisting of images [A] and [B] to the target image 'e.qcow2':

```
(QEMU) blockdev-backup device=node-B target=node-E sync=full job-id=job0
{
    "execute": "blockdev-backup",
```

(continues on next page)

(continued from previous page)

```
    "arguments": {
        "device": "node-B",
        "job-id": "job0",
        "target": "node-E",
        "sync": "full"
    }
}
```

Once the above 'backup' operation has completed, the event, `BLOCK_JOB_COMPLETED` will be emitted, signalling successful completion.

Next, query for any active block device jobs (there should be none):

```
(QEMU) query-block-jobs
{
    "execute": "query-block-jobs",
    "arguments": {}
}
```

Shutdown the guest:

```
(QEMU) quit
{
        "execute": "quit",
            "arguments": {}
}
        "return": {}
}
```

Note: The above step is really important; if forgotten, an error, "Failed to get shared "write" lock on e.qcow2", will be thrown when you do `qemu-img compare` to verify the integrity of the disk image with the backup content.

The end result will be the image 'e.qcow2' containing a point-in-time backup of the disk image chain – i.e. contents from images [A] and [B] at the time the `blockdev-backup` command was initiated.

One way to confirm the backup disk image contains the identical content with the disk image chain is to compare the backup and the contents of the chain, you should see "Images are identical". (NB: this is assuming QEMU was launched with `-S` option, which will not start the CPUs at guest boot up):

```
$ qemu-img compare b.qcow2 e.qcow2
Warning: Image size mismatch!
Images are identical.
```

NOTE: The "Warning: Image size mismatch!" is expected, as we created the target image (e.qcow2) with 39M size.

5.7 Persistent reservation helper protocol

QEMU's SCSI passthrough devices, `scsi-block` and `scsi-generic`, can delegate implementation of persistent reservations to an external (and typically privileged) program. Persistent Reservations allow restricting access to block devices to specific initiators in a shared storage setup.

For a more detailed reference please refer to the SCSI Primary Commands standard, specifically the section on Reservations and the "PERSISTENT RESERVE IN" and "PERSISTENT RESERVE OUT" commands.

This document describes the socket protocol used between QEMU's `pr-manager-helper` object and the external program.

Contents

5.7.1 Connection and initialization

All data transmitted on the socket is big-endian.

After connecting to the helper program's socket, the helper starts a simple feature negotiation process by writing four bytes corresponding to the features it exposes (`supported_features`). QEMU reads it, then writes four bytes corresponding to the desired features of the helper program (`requested_features`).

If a bit is 1 in `requested_features` and 0 in `supported_features`, the corresponding feature is not supported by the helper and the connection is closed. On the other hand, it is acceptable for a bit to be 0 in `requested_features` and 1 in `supported_features`; in this case, the helper will not enable the feature.

Right now no feature is defined, so the two parties always write four zero bytes.

5.7.2 Command format

It is invalid to send multiple commands concurrently on the same socket. It is however possible to connect multiple sockets to the helper and send multiple commands to the helper for one or more file descriptors.

A command consists of a request and a response. A request consists of a 16-byte SCSI CDB. A file descriptor must be passed to the helper together with the SCSI CDB using ancillary data.

The CDB has the following limitations:

- the command (stored in the first byte) must be one of 0x5E (PERSISTENT RESERVE IN) or 0x5F (PERSISTENT RESERVE OUT).
- the allocation length (stored in bytes 7-8 of the CDB for PERSISTENT RESERVE IN) or parameter list length (stored in bytes 5-8 of the CDB for PERSISTENT RESERVE OUT) is limited to 8 KiB.

For PERSISTENT RESERVE OUT, the parameter list is sent right after the CDB. The length of the parameter list is taken from the CDB itself.

The helper's reply has the following structure:

- 4 bytes for the SCSI status

- 4 bytes for the payload size (nonzero only for PERSISTENT RESERVE IN and only if the SCSI status is 0x00, i.e. GOOD)
- 96 bytes for the SCSI sense data
- if the size is nonzero, the payload follows

The sense data is always sent to keep the protocol simple, even though it is only valid if the SCSI status is CHECK CONDITION (0x02).

The payload size is always less than or equal to the allocation length specified in the CDB for the PERSISTENT RESERVE IN command.

If the protocol is violated, the helper closes the socket.

5.8 QEMU Machine Protocol Specification

The QEMU Machine Protocol (QMP) is a JSON-based protocol which is available for applications to operate QEMU at the machine-level. It is also in use by the QEMU Guest Agent (QGA), which is available for host applications to interact with the guest operating system. This page specifies the general format of the protocol; details of the commands and data structures can be found in the *QEMU QMP Reference Manual* and the *QEMU Guest Agent Protocol Reference*.

Contents

5.8.1 Protocol Specification

This section details the protocol format. For the purpose of this document, "Server" is either QEMU or the QEMU Guest Agent, and "Client" is any application communicating with it via QMP.

JSON data structures, when mentioned in this document, are always in the following format:

    json-DATA-STRUCTURE-NAME

Where DATA-STRUCTURE-NAME is any valid JSON data structure, as defined by the JSON standard.

The server expects its input to be encoded in UTF-8, and sends its output encoded in ASCII.

For convenience, json-object members mentioned in this document will be in a certain order. However, in real protocol usage they can be in ANY order, thus no particular order should be assumed. On the other hand, use of json-array elements presumes that preserving order is important unless specifically documented otherwise. Repeating a key within a json-object gives unpredictable results.

Also for convenience, the server will accept an extension of `'single-quoted'` strings in place of the usual `"double-quoted"` json-string, and both input forms of strings understand an additional escape sequence of `\'` for a single quote. The server will only use double quoting on output.

General Definitions

All interactions transmitted by the Server are json-objects, always terminating with CRLF.

All json-objects members are mandatory when not specified otherwise.

Server Greeting

Right when connected the Server will issue a greeting message, which signals that the connection has been successfully established and that the Server is ready for capabilities negotiation (for more information refer to section *Capabilities Negotiation*).

The greeting message format is:

```
{ "QMP": { "version": json-object, "capabilities": json-array } }
```

Where:

- The `version` member contains the Server's version information (the format is the same as for the query-version command).
- The `capabilities` member specifies the availability of features beyond the baseline specification; the order of elements in this array has no particular significance.

Capabilities

Currently supported capabilities are:

oob

the QMP server supports "out-of-band" (OOB) command execution, as described in section *Out-of-band execution*.

Issuing Commands

The format for command execution is:

```
{ "execute": json-string, "arguments": json-object, "id": json-value }
```

or

```
{ "exec-oob": json-string, "arguments": json-object, "id": json-value }
```

Where:

- The `execute` or `exec-oob` member identifies the command to be executed by the server. The latter requests out-of-band execution.
- The `arguments` member is used to pass any arguments required for the execution of the command, it is optional when no arguments are required. Each command documents what contents will be considered valid when handling the json-argument.
- The `id` member is a transaction identification associated with the command execution, it is optional and will be part of the response if provided. The `id` member can be any json-value. A json-number incremented for each successive command works fine.

The actual commands are documented in the *QEMU QMP Reference Manual*.

Out-of-band execution

The server normally reads, executes and responds to one command after the other. The client therefore receives command responses in issue order.

With out-of-band execution enabled via *capabilities negotiation*, the server reads and queues commands as they arrive. It executes commands from the queue one after the other. Commands executed out-of-band jump the queue: the command get executed right away, possibly overtaking prior in-band commands. The client may therefore receive such a command's response before responses from prior in-band commands.

To be able to match responses back to their commands, the client needs to pass `id` with out-of-band commands. Passing it with all commands is recommended for clients that accept capability `oob`.

If the client sends in-band commands faster than the server can execute them, the server will stop reading requests until the request queue length is reduced to an acceptable range.

To ensure commands to be executed out-of-band get read and executed, the client should have at most eight in-band commands in flight.

Only a few commands support out-of-band execution. The ones that do have `"allow-oob": true` in the output of `query-qmp-schema`.

Commands Responses

There are two possible responses which the Server will issue as the result of a command execution: success or error.

As long as the commands were issued with a proper `id` field, then the same `id` field will be attached in the corresponding response message so that requests and responses can match. Clients should drop all the responses that have an unknown `id` field.

Success

The format of a success response is:

```
{ "return": json-value, "id": json-value }
```

Where:

- The `return` member contains the data returned by the command, which is defined on a per-command basis (usually a json-object or json-array of json-objects, but sometimes a json-number, json-string, or json-array of json-strings); it is an empty json-object if the command does not return data.
- The `id` member contains the transaction identification associated with the command execution if issued by the Client.

Error

The format of an error response is:

```
{ "error": { "class": json-string, "desc": json-string }, "id": json-value }
```

Where:

- The `class` member contains the error class name (eg. `"GenericError"`).
- The `desc` member is a human-readable error message. Clients should not attempt to parse this message.
- The `id` member contains the transaction identification associated with the command execution if issued by the Client.

NOTE: Some errors can occur before the Server is able to read the `id` member; in these cases the `id` member will not be part of the error response, even if provided by the client.

Asynchronous events

As a result of state changes, the Server may send messages unilaterally to the Client at any time, when not in the middle of any other response. They are called "asynchronous events".

The format of asynchronous events is:

```
{ "event": json-string, "data": json-object,
  "timestamp": { "seconds": json-number, "microseconds": json-number } }
```

Where:

- The `event` member contains the event's name.
- The `data` member contains event specific data, which is defined in a per-event basis. It is optional.
- The `timestamp` member contains the exact time of when the event occurred in the Server. It is a fixed json-object with time in seconds and microseconds relative to the Unix Epoch (1 Jan 1970); if there is a failure to retrieve host time, both members of the timestamp will be set to -1.

The actual asynchronous events are documented in the *QEMU QMP Reference Manual*.

Some events are rate-limited to at most one per second. If additional "similar" events arrive within one second, all but the last one are dropped, and the last one is delayed. "Similar" normally means same event type.

Forcing the JSON parser into known-good state

Incomplete or invalid input can leave the server's JSON parser in a state where it can't parse additional commands. To get it back into known-good state, the client should provoke a lexical error.

The cleanest way to do that is sending an ASCII control character other than `\t` (horizontal tab), `\r` (carriage return), or `\n` (new line).

Sadly, older versions of QEMU can fail to flag this as an error. If a client needs to deal with them, it should send a 0xFF byte.

QGA Synchronization

When a client connects to QGA over a transport lacking proper connection semantics such as virtio-serial, QGA may have read partial input from a previous client. The client needs to force QGA's parser into known-good state using the previous section's technique. Moreover, the client may receive output a previous client didn't read. To help with skipping that output, QGA provides the `guest-sync-delimited` command. Refer to its documentation for details.

5.8.2 QMP Examples

This section provides some examples of real QMP usage, in all of them -> marks text sent by the Client and <- marks replies by the Server.

Example

Server greeting

```
<- { "QMP": {"version": {"qemu": {"micro": 0, "minor": 0, "major": 3},
     "package": "v3.0.0"}, "capabilities": ["oob"] } }
```

Example

Capabilities negotiation

```
-> { "execute": "qmp_capabilities", "arguments": { "enable": ["oob"] } }
<- { "return": {}}
```

Example

Simple 'stop' execution

```
-> { "execute": "stop" }
<- { "return": {} }
```

Example

KVM information

```
-> { "execute": "query-kvm", "id": "example" }
<- { "return": { "enabled": true, "present": true }, "id": "example"}
```

Example

Parsing error

```
-> { "execute": }
<- { "error": { "class": "GenericError", "desc": "JSON parse error, expecting value" } }
```

Example

Powerdown event

```
<- { "timestamp": { "seconds": 1258551470, "microseconds": 802384 },
    "event": "POWERDOWN" }
```

Example

Out-of-band execution

```
-> { "exec-oob": "migrate-pause", "id": 42 }
<- { "id": 42,
     "error": { "class": "GenericError",
      "desc": "migrate-pause is currently only supported during postcopy-active state" }␣
↪}
```

5.8.3 Capabilities Negotiation

When a Client successfully establishes a connection, the Server is in Capabilities Negotiation mode.

In this mode only the `qmp_capabilities` command is allowed to run; all other commands will return the `CommandNotFound` error. Asynchronous messages are not delivered either.

Clients should use the `qmp_capabilities` command to enable capabilities advertised in the *Server Greeting* which they support.

When the `qmp_capabilities` command is issued, and if it does not return an error, the Server enters Command mode where capabilities changes take effect, all commands (except `qmp_capabilities`) are allowed and asynchronous messages are delivered.

5.8.4 Compatibility Considerations

All protocol changes or new features which modify the protocol format in an incompatible way are disabled by default and will be advertised by the capabilities array (in the *Server Greeting*). Thus, Clients can check that array and enable the capabilities they support.

The QMP Server performs a type check on the arguments to a command. It generates an error if a value does not have the expected type for its key, or if it does not understand a key that the Client included. The strictness of the Server catches wrong assumptions of Clients about the Server's schema. Clients can assume that, when such validation errors occur, they will be reported before the command generated any side effect.

However, Clients must not assume any particular:

- Length of json-arrays
- Size of json-objects; in particular, future versions of QEMU may add new keys and Clients should be able to ignore them
- Order of json-object members or json-array elements
- Amount of errors generated by a command, that is, new errors can be added to any existing command in newer versions of the Server

Any command or member name beginning with `x-` is deemed experimental, and may be withdrawn or changed in an incompatible manner in a future release.

Of course, the Server does guarantee to send valid JSON. But apart from this, a Client should be "conservative in what they send, and liberal in what they accept".

5.8.5 Downstream extension of QMP

We recommend that downstream consumers of QEMU do *not* modify QMP. Management tools should be able to support both upstream and downstream versions of QMP without special logic, and downstream extensions are inherently at odds with that.

However, we recognize that it is sometimes impossible for downstreams to avoid modifying QMP. Both upstream and downstream need to take care to preserve long-term compatibility and interoperability.

To help with that, QMP reserves JSON object member names beginning with `__` (double underscore) for downstream use ("downstream names"). This means upstream will never use any downstream names for its commands, arguments, errors, asynchronous events, and so forth.

Any new names downstream wishes to add must begin with `__`. To ensure compatibility with other downstreams, it is strongly recommended that you prefix your downstream names with `__RFQDN_` where RFQDN is a valid, reverse fully qualified domain name which you control. For example, a qemu-kvm specific monitor command would be:

```
(qemu) __org.linux-kvm_enable_irqchip
```

Downstream must not change the *server greeting* other than to offer additional capabilities. But see below for why even that is discouraged.

The section *Compatibility Considerations* applies to downstream as well as to upstream, obviously. It follows that downstream must behave exactly like upstream for any input not containing members with downstream names ("downstream members"), except it may add members with downstream names to its output.

Thus, a client should not be able to distinguish downstream from upstream as long as it doesn't send input with downstream members, and properly ignores any downstream members in the output it receives.

Advice on downstream modifications:

1. Introducing new commands is okay. If you want to extend an existing command, consider introducing a new one with the new behaviour instead.
2. Introducing new asynchronous messages is okay. If you want to extend an existing message, consider adding a new one instead.
3. Introducing new errors for use in new commands is okay. Adding new errors to existing commands counts as extension, so 1. applies.
4. New capabilities are strongly discouraged. Capabilities are for evolving the basic protocol, and multiple diverging basic protocol dialects are most undesirable.

5.9 QEMU Guest Agent

5.9.1 Synopsis

qemu-ga [*OPTIONS*]

5.9.2 Description

The QEMU Guest Agent is a daemon intended to be run within virtual machines. It allows the hypervisor host to perform various operations in the guest, such as:

- get information from the guest
- set the guest's system time
- read/write a file
- sync and freeze the filesystems
- suspend the guest
- reconfigure guest local processors
- set user's password
- ...

qemu-ga will read a system configuration file on startup (located at `/etc/qemu/qemu-ga.conf` by default), then parse remaining configuration options on the command line. For the same key, the last option wins, but the lists accumulate (see below for configuration file format).

5.9.3 Options

-m, --method=METHOD

Transport method: one of `unix-listen`, `virtio-serial`, or `isa-serial`, or `vsock-listen` (`virtio-serial` is the default).

-p, --path=PATH

Device/socket path (the default for virtio-serial is `/dev/virtio-ports/org.qemu.guest_agent.0`, the default for isa-serial is `/dev/ttyS0`). Socket addresses for vsock-listen are written as <cid>:<port>.

-l, --logfile=PATH

Set log file path (default is stderr).

-f, --pidfile=PATH

Specify pid file (default is /var/run/qemu-ga.pid).

-F, --fsfreeze-hook=PATH

Enable fsfreeze hook. Accepts an optional argument that specifies script to run on freeze/thaw. Script will be called with 'freeze'/'thaw' arguments accordingly (default is /etc/qemu/fsfreeze-hook). If using -F with an argument, do not follow -F with a space (for example: -F/var/run/fsfreezehook.sh).

-t, --statedir=PATH

Specify the directory to store state information (absolute paths only, default is /var/run).

-v, --verbose

Log extra debugging information.

-V, --version

Print version information and exit.

-d, --daemon

Daemonize after startup (detach from terminal).

-b, --block-rpcs=LIST

Comma-separated list of RPCs to disable (no spaces, use --block-rpcs=help to list available RPCs).

-a, --allow-rpcs=LIST

Comma-separated list of RPCs to enable (no spaces, use --allow-rpcs=help to list available RPCs).

-D, --dump-conf

Dump the configuration in a format compatible with qemu-ga.conf and exit.

-h, --help

Display this help and exit.

5.9.4 Files

The syntax of the qemu-ga.conf configuration file follows the Desktop Entry Specification, here is a quick summary: it consists of groups of key-value pairs, interspersed with comments.

```
# qemu-ga configuration sample
[general]
daemonize = 0
pidfile = /var/run/qemu-ga.pid
verbose = 0
method = virtio-serial
path = /dev/virtio-ports/org.qemu.guest_agent.0
statedir = /var/run
```

The list of keys follows the command line options:

Key	Key type
daemon	boolean
method	string
path	string
logfile	string
pidfile	string
fsfreeze-hook	string
statedir	string
verbose	boolean
block-rpcs	string list

5.9.5 See also

qemu(1)

5.10 QEMU Guest Agent Protocol Reference

Contents

5.10.1 General note concerning the use of guest agent interfaces

"unsupported" is a higher-level error than the errors that individual commands might document. The caller should always be prepared to receive QERR_UNSUPPORTED, even if the given command doesn't specify it, or doesn't document any failure mode at all.

5.10.2 QEMU guest agent protocol commands and structs

`guest-sync-delimited` (Command)

Echo back a unique integer value, and prepend to response a leading sentinel byte (0xFF) the client can check scan for.

This is used by clients talking to the guest agent over the wire to ensure the stream is in sync and doesn't contain stale data from previous client. It must be issued upon initial connection, and after any client-side timeouts (including timeouts on receiving a response to this command).

After issuing this request, all guest agent responses should be ignored until the response containing the unique integer value the client passed in is returned. Receival of the 0xFF sentinel byte must be handled as an indication that the client's lexer/tokenizer/parser state should be flushed/reset in preparation for reliably receiving the subsequent response. As an optimization, clients may opt to ignore all data until a sentinel value is receiving to avoid unnecessary processing of stale data.

Similarly, clients should also precede this *request* with a 0xFF byte to make sure the guest agent flushes any partially read JSON data from a previous client connection.

Arguments

`id: int`
: randomly generated 64-bit integer

Returns

The unique integer id passed in by the client

Since

1.1

guest-sync (Command)

Echo back a unique integer value

This is used by clients talking to the guest agent over the wire to ensure the stream is in sync and doesn't contain stale data from previous client. All guest agent responses should be ignored until the provided unique integer value is returned, and it is up to the client to handle stale whole or partially-delivered JSON text in such a way that this response can be obtained.

In cases where a partial stale response was previously received by the client, this cannot always be done reliably. One particular scenario being if qemu-ga responses are fed character-by-character into a JSON parser. In these situations, using guest-sync-delimited may be optimal.

For clients that fetch responses line by line and convert them to JSON objects, guest-sync should be sufficient, but note that in cases where the channel is dirty some attempts at parsing the response may result in a parser error.

Such clients should also precede this command with a 0xFF byte to make sure the guest agent flushes any partially read JSON data from a previous session.

Arguments

id: int
: randomly generated 64-bit integer

Returns

The unique integer id passed in by the client

Since

0.15.0

guest-ping (Command)

Ping the guest agent, a non-error return implies success

Since

0.15.0

guest-get-time (Command)

Get the information about guest's System Time relative to the Epoch of 1970-01-01 in UTC.

Returns

Time in nanoseconds.

Since

1.5

guest-set-time (Command)

Set guest time.

When a guest is paused or migrated to a file then loaded from that file, the guest OS has no idea that there was a big gap in the time. Depending on how long the gap was, NTP might not be able to resynchronize the guest.

This command tries to set guest's System Time to the given value, then sets the Hardware Clock (RTC) to the current System Time. This will make it easier for a guest to resynchronize without waiting for NTP. If no `time` is specified, then the time to set is read from RTC. However, this may not be supported on all platforms (i.e. Windows). If that's the case users are advised to always pass a value.

Arguments

`time`: `int` (optional)
: time of nanoseconds, relative to the Epoch of 1970-01-01 in UTC.

Since

1.5

GuestAgentCommandInfo (Object)

Information about guest agent commands.

Members

`name`: `string`
: name of the command

`enabled`: `boolean`
: whether command is currently enabled by guest admin

`success-response`: `boolean`
: whether command returns a response on success (since 1.7)

Since

1.1.0

GuestAgentInfo (Object)

Information about guest agent.

Members

version: string
guest agent version

supported_commands: array of GuestAgentCommandInfo
Information about guest agent commands

Since

0.15.0

guest-info (Command)

Get some information about the guest agent.

Returns

GuestAgentInfo

Since

0.15.0

guest-shutdown (Command)

Initiate guest-activated shutdown. Note: this is an asynchronous shutdown request, with no guarantee of successful shutdown.

Arguments

mode: string (optional)
"halt", "powerdown" (default), or "reboot"

This command does NOT return a response on success. Success condition is indicated by the VM exiting with a zero exit status or, when running with –no-shutdown, by issuing the query-status QMP command to confirm the VM status is "shutdown".

Since

0.15.0

guest-file-open (Command)

Open a file in the guest and retrieve a file handle for it

Arguments

path: string
: Full path to the file in the guest to open.

mode: string (optional)
: open mode, as per fopen(), "r" is the default.

Returns

Guest file handle

Since

0.15.0

guest-file-close (Command)

Close an open file in the guest

Arguments

handle: int
: filehandle returned by guest-file-open

Since

0.15.0

GuestFileRead (Object)

Result of guest agent file-read operation

Members

count: int
number of bytes read (note: count is *before* base64-encoding is applied)

buf-b64: string
base64-encoded bytes read

eof: boolean
whether EOF was encountered during read operation.

Since

0.15.0

guest-file-read (Command)

Read from an open file in the guest. Data will be base64-encoded. As this command is just for limited, ad-hoc debugging, such as log file access, the number of bytes to read is limited to 48 MB.

Arguments

handle: int
filehandle returned by guest-file-open

count: int (optional)
maximum number of bytes to read (default is 4KB, maximum is 48MB)

Returns

GuestFileRead

Since

0.15.0

GuestFileWrite (Object)

Result of guest agent file-write operation

Members

count: int
number of bytes written (note: count is actual bytes written, after base64-decoding of provided buffer)

eof: boolean
whether EOF was encountered during write operation.

Since

0.15.0

guest-file-write (Command)

Write to an open file in the guest.

Arguments

handle: int
filehandle returned by guest-file-open

buf-b64: string
base64-encoded string representing data to be written

count: int (optional)
bytes to write (actual bytes, after base64-decode), default is all content in buf-b64 buffer after base64 decoding

Returns

GuestFileWrite

Since

0.15.0

GuestFileSeek (Object)

Result of guest agent file-seek operation

Members

position: int
current file position

eof: boolean
whether EOF was encountered during file seek

Since

0.15.0

QGASeek (Enum)

Symbolic names for use in `guest-file-seek`

Values

set
: Set to the specified offset (same effect as 'whence':0)

cur
: Add offset to the current location (same effect as 'whence':1)

end
: Add offset to the end of the file (same effect as 'whence':2)

Since

2.6

GuestFileWhence (Alternate)

Controls the meaning of offset to `guest-file-seek`.

Members

value: int
: Integral value (0 for set, 1 for cur, 2 for end), available for historical reasons, and might differ from the host's or guest's SEEK_* values (since: 0.15)

name: QGASeek
: Symbolic name, and preferred interface

Since

2.6

guest-file-seek (Command)

Seek to a position in the file, as with fseek(), and return the current file position afterward. Also encapsulates ftell()'s functionality, with offset=0 and whence=1.

Arguments

handle: int
filehandle returned by guest-file-open

offset: int
bytes to skip over in the file stream

whence: GuestFileWhence
Symbolic or numeric code for interpreting offset

Returns

GuestFileSeek

Since

0.15.0

guest-file-flush (Command)

Write file changes buffered in userspace to disk/kernel buffers

Arguments

handle: int
filehandle returned by guest-file-open

Since

0.15.0

GuestFsfreezeStatus (Enum)

An enumeration of filesystem freeze states

Values

thawed
filesystems thawed/unfrozen

frozen
all non-network guest filesystems frozen

Since

0.15.0

guest-fsfreeze-status (Command)

Get guest fsfreeze state.

Returns

GuestFsfreezeStatus (“thawed”, “frozen”, etc., as defined below)

Note

This may fail to properly report the current state as a result of some other guest processes having issued an fs freeze/thaw.

Since

0.15.0

guest-fsfreeze-freeze (Command)

Sync and freeze all freezable, local guest filesystems. If this command succeeded, you may call `guest-fsfreeze-thaw` later to unfreeze.

On error, all filesystems will be thawed. If no filesystems are frozen as a result of this call, then `guest-fsfreeze-status` will remain “thawed” and calling `guest-fsfreeze-thaw` is not necessary.

Returns

Number of file systems currently frozen.

Note

On Windows, the command is implemented with the help of a Volume Shadow-copy Service DLL helper. The frozen state is limited for up to 10 seconds by VSS.

Since

0.15.0

guest-fsfreeze-freeze-list (Command)

Sync and freeze specified guest filesystems. See also guest-fsfreeze-freeze.

On error, all filesystems will be thawed.

Arguments

mountpoints: array of string (optional)
an array of mountpoints of filesystems to be frozen. If omitted, every mounted filesystem is frozen. Invalid mount points are ignored.

Returns

Number of file systems currently frozen.

Since

2.2

guest-fsfreeze-thaw (Command)

Unfreeze all frozen guest filesystems

Returns

Number of file systems thawed by this call

Note

if return value does not match the previous call to guest-fsfreeze-freeze, this likely means some freezable filesystems were unfrozen before this call, and that the filesystem state may have changed before issuing this command.

Since

0.15.0

GuestFilesystemTrimResult (Object)

Members

path: string
path that was trimmed

error: string (optional)
an error message when trim failed

trimmed: int (optional)
bytes trimmed for this path

minimum: int (optional)
reported effective minimum for this path

Since

2.4

GuestFilesystemTrimResponse (Object)

Members

paths: array of GuestFilesystemTrimResult
list of `GuestFilesystemTrimResult` per path that was trimmed

Since

2.4

guest-fstrim (Command)

Discard (or "trim") blocks which are not in use by the filesystem.

Arguments

minimum: int (optional)
Minimum contiguous free range to discard, in bytes. Free ranges smaller than this may be ignored (this is a hint and the guest may not respect it). By increasing this value, the fstrim operation will complete more quickly for filesystems with badly fragmented free space, although not all blocks will be discarded. The default value is zero, meaning "discard every free block".

Returns

A `GuestFilesystemTrimResponse` which contains the status of all trimmed paths. (since 2.4)

Since

1.2

`guest-suspend-disk` (Command)

Suspend guest to disk.

This command attempts to suspend the guest using three strategies, in this order:

- systemd hibernate
- pm-utils (via pm-hibernate)
- manual write into sysfs

This command does NOT return a response on success. There is a high chance the command succeeded if the VM exits with a zero exit status or, when running with –no-shutdown, by issuing the query-status QMP command to to confirm the VM status is "shutdown". However, the VM could also exit (or set its status to "shutdown") due to other reasons.

Errors

- If suspend to disk is not supported, Unsupported

Notes

It's strongly recommended to issue the guest-sync command before sending commands when the guest resumes

Since

1.1

`guest-suspend-ram` (Command)

Suspend guest to ram.

This command attempts to suspend the guest using three strategies, in this order:

- systemd hibernate
- pm-utils (via pm-hibernate)
- manual write into sysfs

IMPORTANT: guest-suspend-ram requires working wakeup support in QEMU. You should check QMP command query-current-machine returns wakeup-suspend-support: true before issuing this command. Failure in doing so can result in a suspended guest that QEMU will not be able to awaken, forcing the user to power cycle the guest to bring it back.

This command does NOT return a response on success. There are two options to check for success:

1. Wait for the SUSPEND QMP event from QEMU
2. Issue the query-status QMP command to confirm the VM status is “suspended”

Errors

- If suspend to ram is not supported, Unsupported

Notes

It’s strongly recommended to issue the guest-sync command before sending commands when the guest resumes

Since

1.1

`guest-suspend-hybrid` (Command)

Save guest state to disk and suspend to ram.

This command attempts to suspend the guest by executing, in this order:

- systemd hybrid-sleep
- pm-utils (via pm-suspend-hybrid)

IMPORTANT: guest-suspend-hybrid requires working wakeup support in QEMU. You should check QMP command query-current-machine returns wakeup-suspend-support: true before issuing this command. Failure in doing so can result in a suspended guest that QEMU will not be able to awaken, forcing the user to power cycle the guest to bring it back.

This command does NOT return a response on success. There are two options to check for success:

1. Wait for the SUSPEND QMP event from QEMU
2. Issue the query-status QMP command to confirm the VM status is “suspended”

Errors

- If hybrid suspend is not supported, Unsupported

Notes

It’s strongly recommended to issue the guest-sync command before sending commands when the guest resumes

Since

1.1

GuestIpAddressType (Enum)

An enumeration of supported IP address types

Values

ipv4
IP version 4

ipv6
IP version 6

Since

1.1

GuestIpAddress (Object)

Members

ip-address: string
IP address

ip-address-type: GuestIpAddressType
Type of `ip-address` (e.g. ipv4, ipv6)

prefix: int
Network prefix length of `ip-address`

Since

1.1

GuestNetworkInterfaceStat (Object)

Members

rx-bytes: int
total bytes received

rx-packets: int
total packets received

rx-errs: int
bad packets received

rx-dropped: int
receiver dropped packets

tx-bytes: int
total bytes transmitted

tx-packets: int
total packets transmitted

tx-errs: int
packet transmit problems

tx-dropped: int
dropped packets transmitted

Since

2.11

GuestNetworkInterface (Object)

Members

name: string
The name of interface for which info are being delivered

hardware-address: string (optional)
Hardware address of name

ip-addresses: array of GuestIpAddress (optional)
List of addresses assigned to name

statistics: GuestNetworkInterfaceStat (optional)
various statistic counters related to name (since 2.11)

Since

1.1

guest-network-get-interfaces (Command)

Get list of guest IP addresses, MAC addresses and netmasks.

Returns

List of GuestNetworkInterface

Since

1.1

GuestLogicalProcessor (Object)

Members

logical-id: int
Arbitrary guest-specific unique identifier of the VCPU.

online: boolean
Whether the VCPU is enabled.

can-offline: boolean (optional)
Whether offlining the VCPU is possible. This member is always filled in by the guest agent when the structure is returned, and always ignored on input (hence it can be omitted then).

Since

1.5

guest-get-vcpus (Command)

Retrieve the list of the guest's logical processors.

This is a read-only operation.

Returns

The list of all VCPUs the guest knows about. Each VCPU is put on the list exactly once, but their order is unspecified.

Since

1.5

guest-set-vcpus (Command)

Attempt to reconfigure (currently: enable/disable) logical processors inside the guest.

Arguments

vcpus: array of GuestLogicalProcessor
The logical processors to be reconfigured. This list is processed node by node in order. In each node `logical-id` is used to look up the guest VCPU, for which `online` specifies the requested state. The set of distinct `logical-id`'s is only required to be a subset of the guest-supported identifiers. There's no restriction on list length or on repeating the same `logical-id` (with possibly different `online` field). Preferably the input list should describe a modified subset of `guest-get-vcpus`' return value.

Returns

The length of the initial sublist that has been successfully processed. The guest agent maximizes this value. Possible cases:

- 0: if the `vcpus` list was empty on input. Guest state has not been changed. Otherwise,
- < length(`vcpus`): more than zero initial nodes have been processed, but not the entire `vcpus` list. Guest state has changed accordingly. To retrieve the error (assuming it persists), repeat the call with the successfully processed initial sublist removed. Otherwise,
- length(`vcpus`): call successful.

Errors

- If the reconfiguration of the first node in `vcpus` failed. Guest state has not been changed.

Since

1.5

`GuestDiskBusType` (Enum)

An enumeration of bus type of disks

Values

ide
IDE disks

fdc
floppy disks

scsi
SCSI disks

virtio
virtio disks

xen
Xen disks

usb
USB disks

uml
UML disks

sata
SATA disks

sd
SD cards

unknown
Unknown bus type

ieee1394
Win IEEE 1394 bus type

ssa
Win SSA bus type

fibre
Win fiber channel bus type

raid
Win RAID bus type

iscsi
Win iScsi bus type

sas
Win serial-attaches SCSI bus type

mmc
Win multimedia card (MMC) bus type

virtual
Win virtual bus type

file-backed-virtual
Win file-backed bus type

nvme
NVMe disks (since 7.1)

Since

2.2; 'Unknown' and all entries below since 2.4

GuestPCIAddress (Object)

Members

domain: int
domain id

bus: int
bus id

slot: int
slot id

function: int
function id

Since

2.2

GuestCCWAddress (Object)

Members

cssid: int
channel subsystem image id

ssid: int
subchannel set id

subchno: int
subchannel number

devno: int
device number

Since

6.0

GuestDiskAddress (Object)

Members

pci-controller: GuestPCIAddress
controller's PCI address (fields are set to -1 if invalid)

bus-type: GuestDiskBusType
bus type

bus: int
bus id

target: int
target id

unit: int
unit id

serial: string (optional)
serial number (since: 3.1)

dev: string (optional)
device node (POSIX) or device UNC (Windows) (since: 3.1)

ccw-address: GuestCCWAddress (optional)
CCW address on s390x (since: 6.0)

Since

2.2

GuestNVMeSmart (Object)

NVMe smart information, based on NVMe specification, section <SMART / Health Information (Log Identifier 02h)>

Members

critical-warning: int
Not documented

temperature: int
Not documented

available-spare: int
Not documented

available-spare-threshold: int
Not documented

percentage-used: int
Not documented

data-units-read-lo: int
Not documented

data-units-read-hi: int
Not documented

data-units-written-lo: int
Not documented

data-units-written-hi: int
Not documented

host-read-commands-lo: int
Not documented

host-read-commands-hi: int
Not documented

host-write-commands-lo: int
Not documented

host-write-commands-hi: int
Not documented

controller-busy-time-lo: int
Not documented

controller-busy-time-hi: int
Not documented

power-cycles-lo: int
Not documented

power-cycles-hi: int
Not documented

power-on-hours-lo: int
Not documented

power-on-hours-hi: int
Not documented

unsafe-shutdowns-lo: int
Not documented

unsafe-shutdowns-hi: int
Not documented

media-errors-lo: int
Not documented

media-errors-hi: int
Not documented

number-of-error-log-entries-lo: int
Not documented

number-of-error-log-entries-hi: int
Not documented

Since

7.1

GuestDiskSmart (Object)

Disk type related smart information.

Members

type: GuestDiskBusType
disk bus type

The members of GuestNVMeSmart when type is "nvme"

Since

7.1

GuestDiskInfo (Object)

Members

name: string
device node (Linux) or device UNC (Windows)

partition: boolean
whether this is a partition or disk

dependencies: array of string (optional)
list of device dependencies; e.g. for LVs of the LVM this will hold the list of PVs, for LUKS encrypted volume this will contain the disk where the volume is placed. (Linux)

address: GuestDiskAddress (optional)
disk address information (only for non-virtual devices)

alias: string (optional)
optional alias assigned to the disk, on Linux this is a name assigned by device mapper

smart: GuestDiskSmart (optional)
disk smart information (Since 7.1)

Since

5.2

guest-get-disks (Command)

Returns

The list of disks in the guest. For Windows these are only the physical disks. On Linux these are all root block devices of non-zero size including e.g. removable devices, loop devices, NBD, etc.

Since

5.2

GuestFilesystemInfo (Object)

Members

name: string
disk name

mountpoint: string
mount point path

type: string
file system type string

used-bytes: int (optional)
file system used bytes (since 3.0)

total-bytes: int (optional)
filesystem capacity in bytes for unprivileged users (since 3.0)

total-bytes-privileged: int (optional)
filesystem capacity in bytes for privileged users (since 9.1)

disk: array of GuestDiskAddress
an array of disk hardware information that the volume lies on, which may be empty if the disk type is not supported

Since

2.2

guest-get-fsinfo (Command)

Returns

The list of filesystems information mounted in the guest. The returned mountpoints may be specified to `guest-fsfreeze-freeze-list`. Network filesystems (such as CIFS and NFS) are not listed.

Since

2.2

guest-set-user-password (Command)

Arguments

username: string
: the user account whose password to change

password: string
: the new password entry string, base64 encoded

crypted: boolean
: true if password is already crypt()d, false if raw

If the `crypted` flag is true, it is the caller's responsibility to ensure the correct crypt() encryption scheme is used. This command does not attempt to interpret or report on the encryption scheme. Refer to the documentation of the guest operating system in question to determine what is supported.

Not all guest operating systems will support use of the `crypted` flag, as they may require the clear-text password

The `password` parameter must always be base64 encoded before transmission, even if already crypt()d, to ensure it is 8-bit safe when passed as JSON.

Since

2.3

GuestMemoryBlock (Object)

Members

phys-index: int
: Arbitrary guest-specific unique identifier of the MEMORY BLOCK.

online: boolean
: Whether the MEMORY BLOCK is enabled in guest.

can-offline: boolean (optional)
Whether offlining the MEMORY BLOCK is possible. This member is always filled in by the guest agent when the structure is returned, and always ignored on input (hence it can be omitted then).

Since

2.3

guest-get-memory-blocks (Command)

Retrieve the list of the guest's memory blocks.

This is a read-only operation.

Returns

The list of all memory blocks the guest knows about. Each memory block is put on the list exactly once, but their order is unspecified.

Since

2.3

GuestMemoryBlockResponseType (Enum)

An enumeration of memory block operation result.

Values

success
the operation of online/offline memory block is successful.

not-found
can't find the corresponding memoryXXX directory in sysfs.

operation-not-supported
for some old kernels, it does not support online or offline memory block.

operation-failed
the operation of online/offline memory block fails, because of some errors happen.

Since

2.3

GuestMemoryBlockResponse (Object)

Members

phys-index: int
same with the 'phys-index' member of GuestMemoryBlock.

response: GuestMemoryBlockResponseType
the result of memory block operation.

error-code: int (optional)
the error number. When memory block operation fails, we assign the value of 'errno' to this member, it indicates what goes wrong. When the operation succeeds, it will be omitted.

Since

2.3

guest-set-memory-blocks (Command)

Attempt to reconfigure (currently: enable/disable) state of memory blocks inside the guest.

Arguments

mem-blks: array of GuestMemoryBlock
The memory blocks to be reconfigured. This list is processed node by node in order. In each node phys-index is used to look up the guest MEMORY BLOCK, for which online specifies the requested state. The set of distinct phys-index's is only required to be a subset of the guest-supported identifiers. There's no restriction on list length or on repeating the same phys-index (with possibly different online field). Preferably the input list should describe a modified subset of guest-get-memory-blocks' return value.

Returns

The operation results, it is a list of GuestMemoryBlockResponse, which is corresponding to the input list.

Note: it will return an empty list if the mem-blks list was empty on input, or there is an error, and in this case, guest state will not be changed.

Since

2.3

GuestMemoryBlockInfo (Object)

Members

size: int
the size (in bytes) of the guest memory blocks, which are the minimal units of memory block online/offline operations (also called Logical Memory Hotplug).

Since

2.3

guest-get-memory-block-info (Command)

Get information relating to guest memory blocks.

Returns

GuestMemoryBlockInfo

Since

2.3

GuestExecStatus (Object)

Members

exited: boolean
true if process has already terminated.

exitcode: int (optional)
process exit code if it was normally terminated.

signal: int (optional)
signal number (linux) or unhandled exception code (windows) if the process was abnormally terminated.

out-data: string (optional)
base64-encoded stdout of the process. This field will only be populated after the process exits.

err-data: string (optional)
base64-encoded stderr of the process. Note: out-data and err-data are present only if 'capture-output' was specified for 'guest-exec'. This field will only be populated after the process exits.

out-truncated: boolean (optional)
true if stdout was not fully captured due to size limitation.

err-truncated: boolean (optional)
true if stderr was not fully captured due to size limitation.

Since

2.5

guest-exec-status (Command)

Check status of process associated with PID retrieved via guest-exec. Reap the process and associated metadata if it has exited.

Arguments

pid: int
pid returned from guest-exec

Returns

GuestExecStatus

Since

2.5

GuestExec (Object)

Members

pid: int
pid of child process in guest OS

Since

2.5

GuestExecCaptureOutputMode (Enum)

An enumeration of guest-exec capture modes.

Values

none
: do not capture any output

stdout
: only capture stdout

stderr
: only capture stderr

separated
: capture both stdout and stderr, but separated into GuestExecStatus out-data and err-data, respectively

merged (**If: not CONFIG_WIN32**)
: capture both stdout and stderr, but merge together into out-data. Not effective on windows guests.

Since

8.0

GuestExecCaptureOutput (Alternate)

Controls what guest-exec output gets captures.

Members

flag: boolean
: captures both stdout and stderr if true. Equivalent to GuestExecCaptureOutputMode::all. (since 2.5)

mode: GuestExecCaptureOutputMode
: capture mode; preferred interface

Since

8.0

guest-exec (Command)

Execute a command in the guest

Arguments

path: string
: path or executable name to execute

arg: array of string (**optional**)
: argument list to pass to executable

env: array of string (**optional**)
: environment variables to pass to executable

input-data: string (optional)
data to be passed to process stdin (base64 encoded)

capture-output: GuestExecCaptureOutput (optional)
bool flag to enable capture of stdout/stderr of running process. Defaults to false.

Returns

PID

Since

2.5

GuestHostName (Object)

Members

host-name: string
Fully qualified domain name of the guest OS

Since

2.10

guest-get-host-name (Command)

Return a name for the machine.

The returned name is not necessarily a fully-qualified domain name, or even present in DNS or some other name service at all. It need not even be unique on your local network or site, but usually it is.

Returns

the host name of the machine

Since

2.10

`GuestUser` (Object)

Members

`user: string`
Username

`domain: string` (optional)
Logon domain (windows only)

`login-time: number`
Time of login of this user on the computer. If multiple instances of the user are logged in, the earliest login time is reported. The value is in fractional seconds since epoch time.

Since

2.10

`guest-get-users` (Command)

Retrieves a list of currently active users on the VM.

Returns

A unique list of users.

Since

2.10

`GuestTimezone` (Object)

Members

`zone: string` (optional)
Timezone name. These values may differ depending on guest/OS and should only be used for informational purposes.

`offset: int`
Offset to UTC in seconds, negative numbers for time zones west of GMT, positive numbers for east

Since

2.10

guest-get-timezone (Command)

Retrieves the timezone information from the guest.

Returns

A GuestTimezone dictionary.

Since

2.10

GuestOSInfo (Object)

Members

kernel-release: string (optional)

- POSIX: release field returned by uname(2)
- Windows: build number of the OS

kernel-version: string (optional)

- POSIX: version field returned by uname(2)
- Windows: version number of the OS

machine: string (optional)

- POSIX: machine field returned by uname(2)
- Windows: one of x86, x86_64, arm, ia64

id: string (optional)

- POSIX: as defined by os-release(5)
- Windows: contains string "mswindows"

name: string (optional)

- POSIX: as defined by os-release(5)
- Windows: contains string "Microsoft Windows"

pretty-name: string (optional)

- POSIX: as defined by os-release(5)
- Windows: product name, e.g. "Microsoft Windows 10 Enterprise"

version: string (optional)

- POSIX: as defined by os-release(5)

- Windows: long version string, e.g. “Microsoft Windows Server 2008”

version-id: string (optional)

- POSIX: as defined by os-release(5)
- Windows: short version identifier, e.g. “7” or “20012r2”

variant: string (optional)

- POSIX: as defined by os-release(5)
- Windows: contains string “server” or “client”

variant-id: string (optional)

- POSIX: as defined by os-release(5)
- Windows: contains string “server” or “client”

Notes

On POSIX systems the fields `id`, `name`, `pretty-name`, `version`, `version-id`, `variant` and `variant-id` follow the definition specified in os-release(5). Refer to the manual page for exact description of the fields. Their values are taken from the os-release file. If the file is not present in the system, or the values are not present in the file, the fields are not included.

On Windows the values are filled from information gathered from the system.

Since

2.10

guest-get-osinfo (Command)

Retrieve guest operating system information

Returns

`GuestOSInfo`

Since

2.10

GuestDeviceType (Enum)

Values

pci
PCI device

GuestDeviceIdPCI (Object)

Members

vendor-id: int
vendor ID

device-id: int
device ID

Since

5.2

GuestDeviceId (Object)

Id of the device

Members

type: GuestDeviceType
device type

The members of GuestDeviceIdPCI when type is "pci"

Since

5.2

GuestDeviceInfo (Object)

Members

driver-name: string
name of the associated driver

driver-date: int (optional)
driver release date, in nanoseconds since the epoch

driver-version: string (optional)
driver version

id: GuestDeviceId (optional)
device ID

Since

5.2

guest-get-devices (Command)

Retrieve information about device drivers in Windows guest

Returns

GuestDeviceInfo

Since

5.2

GuestAuthorizedKeys (Object)

Members

keys: array of string
public keys (in OpenSSH/sshd(8) authorized_keys format)

Since

5.2

guest-ssh-get-authorized-keys (Command)

Return the public keys from user .ssh/authorized_keys on Unix systems (not implemented for other systems).

Arguments

username: string
the user account to add the authorized keys

Returns

GuestAuthorizedKeys

Since

5.2

guest-ssh-add-authorized-keys (Command)

Append public keys to user .ssh/authorized_keys on Unix systems (not implemented for other systems).

Arguments

username: string
: the user account to add the authorized keys

keys: array of string
: the public keys to add (in OpenSSH/sshd(8) authorized_keys format)

reset: boolean (optional)
: ignore the existing content, set it with the given keys only

Since

5.2

guest-ssh-remove-authorized-keys (Command)

Remove public keys from the user .ssh/authorized_keys on Unix systems (not implemented for other systems). It's not an error if the key is already missing.

Arguments

username: string
: the user account to remove the authorized keys

keys: array of string
: the public keys to remove (in OpenSSH/sshd(8) authorized_keys format)

Since

5.2

`GuestDiskStats` (Object)

Members

`read-sectors`: `int` (optional)
sectors read

`read-ios`: `int` (optional)
reads completed successfully

`read-merges`: `int` (optional)
read requests merged

`write-sectors`: `int` (optional)
sectors written

`write-ios`: `int` (optional)
writes completed

`write-merges`: `int` (optional)
write requests merged

`discard-sectors`: `int` (optional)
sectors discarded

`discard-ios`: `int` (optional)
discards completed successfully

`discard-merges`: `int` (optional)
discard requests merged

`flush-ios`: `int` (optional)
flush requests completed successfully

`read-ticks`: `int` (optional)
time spent reading(ms)

`write-ticks`: `int` (optional)
time spent writing(ms)

`discard-ticks`: `int` (optional)
time spent discarding(ms)

`flush-ticks`: `int` (optional)
time spent flushing(ms)

`ios-pgr`: `int` (optional)
number of I/Os currently in flight

`total-ticks`: `int` (optional)
time spent doing I/Os (ms)

`weight-ticks`: `int` (optional)
weighted time spent doing I/Os since the last update of this field(ms)

Since

7.1

GuestDiskStatsInfo (Object)

Members

name: string
disk name

major: int
major device number of disk

minor: int
minor device number of disk

stats: GuestDiskStats
I/O statistics

guest-get-diskstats (Command)

Retrieve information about disk stats.

Returns

List of disk stats of guest.

Since

7.1

GuestCpuStatsType (Enum)

Guest operating systems supporting CPU statistics

Values

linux
Linux

Since

7.1

GuestLinuxCpuStats (Object)

CPU statistics of Linux

Members

cpu: int
CPU index in guest OS

user: int
Time spent in user mode

nice: int
Time spent in user mode with low priority (nice)

system: int
Time spent in system mode

idle: int
Time spent in the idle task

iowait: int (optional)
Time waiting for I/O to complete (since Linux 2.5.41)

irq: int (optional)
Time servicing interrupts (since Linux 2.6.0-test4)

softirq: int (optional)
Time servicing softirqs (since Linux 2.6.0-test4)

steal: int (optional)
Stolen time by host (since Linux 2.6.11)

guest: int (optional)
ime spent running a virtual CPU for guest operating systems under the control of the Linux kernel (since Linux 2.6.24)

guestnice: int (optional)
Time spent running a niced guest (since Linux 2.6.33)

Since

7.1

`GuestCpuStats` (Object)

Get statistics of each CPU in millisecond.

Members

`type`: `GuestCpuStatsType`
guest operating system

The members of `GuestLinuxCpuStats` when `type` is `"linux"`

Since

7.1

`guest-get-cpustats` (Command)

Retrieve information about CPU stats.

Returns

List of CPU stats of guest.

Since

7.1

5.11 QEMU QMP Reference Manual

Contents

* *ChardevInfo (Object)*
* *query-chardev (Command)*
* *ChardevBackendInfo (Object)*
* *query-chardev-backends (Command)*
* *DataFormat (Enum)*
* *ringbuf-write (Command)*
* *ringbuf-read (Command)*
* *ChardevCommon (Object)*
* *ChardevFile (Object)*
* *ChardevHostdev (Object)*
* *ChardevSocket (Object)*
* *ChardevUdp (Object)*
* *ChardevMux (Object)*
* *ChardevStdio (Object)*
* *ChardevSpiceChannel (Object)*
* *ChardevSpicePort (Object)*
* *ChardevDBus (Object)*
* *ChardevVC (Object)*
* *ChardevRingbuf (Object)*
* *ChardevQemuVDAgent (Object)*
* *ChardevBackendKind (Enum)*
* *ChardevFileWrapper (Object)*
* *ChardevHostdevWrapper (Object)*
* *ChardevSocketWrapper (Object)*
* *ChardevUdpWrapper (Object)*
* *ChardevCommonWrapper (Object)*
* *ChardevMuxWrapper (Object)*
* *ChardevStdioWrapper (Object)*
* *ChardevSpiceChannelWrapper (Object)*
* *ChardevSpicePortWrapper (Object)*
* *ChardevQemuVDAgentWrapper (Object)*
* *ChardevDBusWrapper (Object)*
* *ChardevVCWrapper (Object)*
* *ChardevRingbufWrapper (Object)*
* *ChardevBackend (Object)*

* *COLO_EXIT (Event)*
* *COLOExitReason (Enum)*
* *x-colo-lost-heartbeat (Command)*
* *migrate_cancel (Command)*
* *migrate-continue (Command)*
* *MigrationAddressType (Enum)*
* *FileMigrationArgs (Object)*
* *MigrationExecCommand (Object)*
* *MigrationAddress (Object)*
* *MigrationChannelType (Enum)*
* *MigrationChannel (Object)*
* *migrate (Command)*
* *migrate-incoming (Command)*
* *xen-save-devices-state (Command)*
* *xen-set-global-dirty-log (Command)*
* *xen-load-devices-state (Command)*
* *xen-set-replication (Command)*
* *ReplicationStatus (Object)*
* *query-xen-replication-status (Command)*
* *xen-colo-do-checkpoint (Command)*
* *COLOStatus (Object)*
* *query-colo-status (Command)*
* *migrate-recover (Command)*
* *migrate-pause (Command)*
* *UNPLUG_PRIMARY (Event)*
* *DirtyRateVcpu (Object)*
* *DirtyRateStatus (Enum)*
* *DirtyRateMeasureMode (Enum)*
* *TimeUnit (Enum)*
* *DirtyRateInfo (Object)*
* *calc-dirty-rate (Command)*
* *query-dirty-rate (Command)*
* *DirtyLimitInfo (Object)*
* *set-vcpu-dirty-limit (Command)*
* *cancel-vcpu-dirty-limit (Command)*

* *X86CPURegister32 (Enum)*
* *X86CPUFeatureWordInfo (Object)*
* *DummyForceArrays (Object)*
* *NumaCpuOptions (Object)*
* *HmatLBMemoryHierarchy (Enum)*
* *HmatLBDataType (Enum)*
* *NumaHmatLBOptions (Object)*
* *HmatCacheAssociativity (Enum)*
* *HmatCacheWritePolicy (Enum)*
* *NumaHmatCacheOptions (Object)*
* *memsave (Command)*
* *pmemsave (Command)*
* *Memdev (Object)*
* *query-memdev (Command)*
* *CpuInstanceProperties (Object)*
* *HotpluggableCPU (Object)*
* *query-hotpluggable-cpus (Command)*
* *set-numa-node (Command)*
* *balloon (Command)*
* *BalloonInfo (Object)*
* *query-balloon (Command)*
* *BALLOON_CHANGE (Event)*
* *HvBalloonInfo (Object)*
* *query-hv-balloon-status-report (Command)*
* *HV_BALLOON_STATUS_REPORT (Event)*
* *MemoryInfo (Object)*
* *query-memory-size-summary (Command)*
* *PCDIMMDeviceInfo (Object)*
* *VirtioPMEMDeviceInfo (Object)*
* *VirtioMEMDeviceInfo (Object)*
* *SgxEPCDeviceInfo (Object)*
* *HvBalloonDeviceInfo (Object)*
* *MemoryDeviceInfoKind (Enum)*
* *PCDIMMDeviceInfoWrapper (Object)*
* *VirtioPMEMDeviceInfoWrapper (Object)*

* *VirtioMEMDeviceInfoWrapper (Object)*
* *SgxEPCDeviceInfoWrapper (Object)*
* *HvBalloonDeviceInfoWrapper (Object)*
* *MemoryDeviceInfo (Object)*
* *SgxEPC (Object)*
* *SgxEPCProperties (Object)*
* *query-memory-devices (Command)*
* *MEMORY_DEVICE_SIZE_CHANGE (Event)*
* *MEM_UNPLUG_ERROR (Event)*
* *BootConfiguration (Object)*
* *SMPConfiguration (Object)*
* *x-query-irq (Command)*
* *x-query-jit (Command)*
* *x-query-numa (Command)*
* *x-query-opcount (Command)*
* *x-query-ramblock (Command)*
* *x-query-roms (Command)*
* *x-query-usb (Command)*
* *SmbiosEntryPointType (Enum)*
* *MemorySizeConfiguration (Object)*
* *dumpdtb (Command)*
* *CpuModelInfo (Object)*
* *CpuModelExpansionType (Enum)*
* *CpuModelCompareResult (Enum)*
* *CpuModelBaselineInfo (Object)*
* *CpuModelCompareInfo (Object)*
* *query-cpu-model-comparison (Command)*
* *query-cpu-model-baseline (Command)*
* *CpuModelExpansionInfo (Object)*
* *query-cpu-model-expansion (Command)*
* *CpuDefinitionInfo (Object)*
* *query-cpu-definitions (Command)*
* *CpuS390Polarization (Enum)*
* *set-cpu-topology (Command)*
* *CPU_POLARIZATION_CHANGE (Event)*

        * *CommandLineParameterType (Enum)*
        * *CommandLineParameterInfo (Object)*
        * *CommandLineOptionInfo (Object)*
        * *query-command-line-options (Command)*
        * *RTC_CHANGE (Event)*
        * *VFU_CLIENT_HANGUP (Event)*
        * *rtc-reset-reinjection (Command)*
        * *SevState (Enum)*
        * *SevInfo (Object)*
        * *query-sev (Command)*
        * *SevLaunchMeasureInfo (Object)*
        * *query-sev-launch-measure (Command)*
        * *SevCapability (Object)*
        * *query-sev-capabilities (Command)*
        * *sev-inject-launch-secret (Command)*
        * *SevAttestationReport (Object)*
        * *query-sev-attestation-report (Command)*
        * *dump-skeys (Command)*
        * *GICCapability (Object)*
        * *query-gic-capabilities (Command)*
        * *SGXEPCSection (Object)*
        * *SGXInfo (Object)*
        * *query-sgx (Command)*
        * *query-sgx-capabilities (Command)*
        * *EvtchnPortType (Enum)*
        * *EvtchnInfo (Object)*
        * *xen-event-list (Command)*
        * *xen-event-inject (Command)*
    - *Audio*
        * *AudiodevPerDirectionOptions (Object)*
        * *AudiodevGenericOptions (Object)*
        * *AudiodevAlsaPerDirectionOptions (Object)*
        * *AudiodevAlsaOptions (Object)*
        * *AudiodevSndioOptions (Object)*
        * *AudiodevCoreaudioPerDirectionOptions (Object)*

5.11.1 Introduction

This document describes all commands currently supported by QMP.

Most of the time their usage is exactly the same as in the user Monitor, this means that any other document which also describe commands (the manpage, QEMU's manual, etc) can and should be consulted.

QMP has two types of commands: regular and query commands. Regular commands usually change the Virtual Machine's state someway, while query commands just return information. The sections below are divided accordingly.

It's important to observe that all communication examples are formatted in a reader-friendly way, so that they're easier to understand. However, in real protocol usage, they're emitted as a single line.

Also, the following notation is used to denote data flow:

Example:

```
-> data issued by the Client
<- Server data response
```

Please refer to the *QEMU Machine Protocol Specification* for detailed information on the Server command and response formats.

5.11.2 QMP errors

QapiErrorClass (Enum)

QEMU error classes

Values

GenericError
: this is used for errors that don't require a specific error class. This should be the default case for most errors

CommandNotFound
: the requested command has not been found

DeviceNotActive
: a device has failed to be become active

DeviceNotFound
: the requested device has not been found

KVMMissingCap
: the requested operation can't be fulfilled because a required KVM capability is missing

Since

1.2

5.11.3 Common data types

IoOperationType (Enum)

An enumeration of the I/O operation types

Values

read
: read operation

write
: write operation

Since

2.1

`OnOffAuto` (Enum)

An enumeration of three options: on, off, and auto

Values

`auto`
QEMU selects the value between on and off

`on`
Enabled

`off`
Disabled

Since

2.2

`OnOffSplit` (Enum)

An enumeration of three values: on, off, and split

Values

`on`
Enabled

`off`
Disabled

`split`
Mixed

Since

2.6

`StrOrNull` (Alternate)

This is a string value or the explicit lack of a string (null pointer in C). Intended for cases when 'optional absent' already has a different meaning.

Members

s: string
the string value

n: null
no string value

Since

2.10

OffAutoPCIBAR (Enum)

An enumeration of options for specifying a PCI BAR

Values

off
The specified feature is disabled

auto
The PCI BAR for the feature is automatically selected

bar0
PCI BAR0 is used for the feature

bar1
PCI BAR1 is used for the feature

bar2
PCI BAR2 is used for the feature

bar3
PCI BAR3 is used for the feature

bar4
PCI BAR4 is used for the feature

bar5
PCI BAR5 is used for the feature

Since

2.12

PCIELinkSpeed (Enum)

An enumeration of PCIe link speeds in units of GT/s

Values

2_5
: 2.5GT/s

5
: 5.0GT/s

8
: 8.0GT/s

16
: 16.0GT/s

32
: 32.0GT/s (since 9.0)

64
: 64.0GT/s (since 9.0)

Since

4.0

PCIELinkWidth (Enum)

An enumeration of PCIe link width

Values

1
: x1

2
: x2

4
: x4

8
: x8

12
: x12

16
: x16

32
: x32

Since

4.0

HostMemPolicy (Enum)

Host memory policy types

Values

default
: restore default policy, remove any nondefault policy

preferred
: set the preferred host nodes for allocation

bind
: a strict policy that restricts memory allocation to the host nodes specified

interleave
: memory allocations are interleaved across the set of host nodes specified

Since

2.1

NetFilterDirection (Enum)

Indicates whether a netfilter is attached to a netdev's transmit queue or receive queue or both.

Values

all
: the filter is attached both to the receive and the transmit queue of the netdev (default).

rx
: the filter is attached to the receive queue of the netdev, where it will receive packets sent to the netdev.

tx
: the filter is attached to the transmit queue of the netdev, where it will receive packets sent by the netdev.

Since

2.5

GrabToggleKeys (Enum)

Keys to toggle input-linux between host and guest.

Values

ctrl-ctrl
Not documented

alt-alt
Not documented

shift-shift
Not documented

meta-meta
Not documented

scrolllock
Not documented

ctrl-scrolllock
Not documented

Since

4.0

HumanReadableText (Object)

Members

human-readable-text: string
Formatted output intended for humans.

Since

6.2

5.11.4 Socket data types

NetworkAddressFamily (Enum)

The network address family

Values

ipv4
IPV4 family

ipv6
IPV6 family

unix
unix socket

vsock
vsock family (since 2.8)

unknown
otherwise

Since

2.1

InetSocketAddressBase (Object)

Members

host: string
host part of the address

port: string
port part of the address

InetSocketAddress (Object)

Captures a socket address or address range in the Internet namespace.

Members

numeric: boolean (optional)
true if the host/port are guaranteed to be numeric, false if name resolution should be attempted. Defaults to false. (Since 2.9)

to: int (optional)
If present, this is range of possible addresses, with port between `port` and `to`.

ipv4: boolean (optional)
whether to accept IPv4 addresses, default try both IPv4 and IPv6

ipv6: boolean (optional)
whether to accept IPv6 addresses, default try both IPv4 and IPv6

keep-alive: boolean (optional)
enable keep-alive when connecting to this socket. Not supported for passive sockets. (Since 4.2)

mptcp: boolean (optional) (If: HAVE_IPPROTO_MPTCP)
enable multi-path TCP. (Since 6.1)

The members of `InetSocketAddressBase`

Since

1.3

`UnixSocketAddress` (Object)

Captures a socket address in the local ("Unix socket") namespace.

Members

`path`: `string`
filesystem path to use

`abstract`: `boolean` (optional) (If: `CONFIG_LINUX`)
if true, this is a Linux abstract socket address. `path` will be prefixed by a null byte, and optionally padded with null bytes. Defaults to false. (Since 5.1)

`tight`: `boolean` (optional) (If: `CONFIG_LINUX`)
if false, pad an abstract socket address with enough null bytes to make it fill struct sockaddr_un member sun_path. Defaults to true. (Since 5.1)

Since

1.3

`VsockSocketAddress` (Object)

Captures a socket address in the vsock namespace.

Members

`cid`: `string`
unique host identifier

`port`: `string`
port

Note

string types are used to allow for possible future hostname or service resolution support.

Since

2.8

FdSocketAddress (Object)

A file descriptor name or number.

Members

str: string
decimal is for file descriptor number, otherwise it's a file descriptor name. Named file descriptors are permitted in monitor commands, in combination with the 'getfd' command. Decimal file descriptors are permitted at startup or other contexts where no monitor context is active.

Since

1.2

InetSocketAddressWrapper (Object)

Members

data: InetSocketAddress
internet domain socket address

Since

1.3

UnixSocketAddressWrapper (Object)

Members

data: UnixSocketAddress
UNIX domain socket address

Since

1.3

VsockSocketAddressWrapper (Object)

Members

data: VsockSocketAddress
VSOCK domain socket address

Since

2.8

FdSocketAddressWrapper (Object)

Members

data: FdSocketAddress
file descriptor name or number

Since

1.3

SocketAddressLegacy (Object)

Captures the address of a socket, which could also be a named file descriptor

Members

type: SocketAddressType
Transport type

The members of InetSocketAddressWrapper when type is "inet"
The members of UnixSocketAddressWrapper when type is "unix"
The members of VsockSocketAddressWrapper when type is "vsock"
The members of FdSocketAddressWrapper when type is "fd"

Note

This type is deprecated in favor of SocketAddress. The difference between SocketAddressLegacy and SocketAddress is that the latter has fewer {} on the wire.

Since

1.3

SocketAddressType (Enum)

Available SocketAddress types

Values

inet
: Internet address

unix
: Unix domain socket

vsock
: VMCI address

fd
: Socket file descriptor

Since

2.9

SocketAddress (Object)

Captures the address of a socket, which could also be a socket file descriptor

Members

type: SocketAddressType
: Transport type

The members of InetSocketAddress when type is "inet"
The members of UnixSocketAddress when type is "unix"
The members of VsockSocketAddress when type is "vsock"
The members of FdSocketAddress when type is "fd"

Since

2.9

5.11.5 VM run state

RunState (Enum)

An enumeration of VM run states.

Values

debug
QEMU is running on a debugger

finish-migrate
guest is paused to finish the migration process

inmigrate
guest is paused waiting for an incoming migration. Note that this state does not tell whether the machine will start at the end of the migration. This depends on the command-line -S option and any invocation of 'stop' or 'cont' that has happened since QEMU was started.

internal-error
An internal error that prevents further guest execution has occurred

io-error
the last IOP has failed and the device is configured to pause on I/O errors

paused
guest has been paused via the 'stop' command

postmigrate
guest is paused following a successful 'migrate'

prelaunch
QEMU was started with -S and guest has not started

restore-vm
guest is paused to restore VM state

running
guest is actively running

save-vm
guest is paused to save the VM state

shutdown
guest is shut down (and -no-shutdown is in use)

suspended
guest is suspended (ACPI S3)

watchdog
the watchdog action is configured to pause and has been triggered

guest-panicked
guest has been panicked as a result of guest OS panic

colo
guest is paused to save/restore VM state under colo checkpoint, VM can not get into this state unless colo capability is enabled for migration. (since 2.8)

ShutdownCause (Enum)

An enumeration of reasons for a Shutdown.

Values

none
No shutdown request pending

host-error
An error prevents further use of guest

host-qmp-quit
Reaction to the QMP command 'quit'

host-qmp-system-reset
Reaction to the QMP command 'system_reset'

host-signal
Reaction to a signal, such as SIGINT

host-ui
Reaction to a UI event, like window close

guest-shutdown
Guest shutdown/suspend request, via ACPI or other hardware-specific means

guest-reset
Guest reset request, and command line turns that into a shutdown

guest-panic
Guest panicked, and command line turns that into a shutdown

subsystem-reset
Partial guest reset that does not trigger QMP events and ignores –no-reboot. This is useful for sanitizing hypercalls on s390 that are used during kexec/kdump/boot

snapshot-load
A snapshot is being loaded by the record & replay subsystem. This value is used only within QEMU. It doesn't occur in QMP. (since 7.2)

StatusInfo (Object)

Information about VM run state

Members

running: boolean
true if all VCPUs are runnable, false if not runnable

status: RunState
the virtual machine RunState

Since

0.14

query-status (Command)

Query the run status of the VM

Returns

StatusInfo reflecting the VM

Since

0.14

Example

```
-> { "execute": "query-status" }
<- { "return": { "running": true,
                 "status": "running" } }
```

SHUTDOWN (Event)

Emitted when the virtual machine has shut down, indicating that qemu is about to exit.

Arguments

guest: boolean
: If true, the shutdown was triggered by a guest request (such as a guest-initiated ACPI shutdown request or other hardware-specific action) rather than a host request (such as sending qemu a SIGINT). (since 2.10)

reason: ShutdownCause
: The ShutdownCause which resulted in the SHUTDOWN. (since 4.0)

Note

If the command-line option "-no-shutdown" has been specified, qemu will not exit, and a STOP event will eventually follow the SHUTDOWN event

Since

0.12

Example

```
<- { "event": "SHUTDOWN",
     "data": { "guest": true, "reason": "guest-shutdown" },
     "timestamp": { "seconds": 1267040730, "microseconds": 682951 } }
```

POWERDOWN (Event)

Emitted when the virtual machine is powered down through the power control system, such as via ACPI.

Since

0.12

Example

```
<- { "event": "POWERDOWN",
     "timestamp": { "seconds": 1267040730, "microseconds": 682951 } }
```

RESET (Event)

Emitted when the virtual machine is reset

Arguments

guest: boolean
: If true, the reset was triggered by a guest request (such as a guest-initiated ACPI reboot request or other hardware-specific action) rather than a host request (such as the QMP command system_reset). (since 2.10)

reason: ShutdownCause
: The ShutdownCause of the RESET. (since 4.0)

Since

0.12

Example

```
<- { "event": "RESET",
     "data": { "guest": false, "reason": "guest-reset" },
     "timestamp": { "seconds": 1267041653, "microseconds": 9518 } }
```

STOP (Event)

Emitted when the virtual machine is stopped

Since

0.12

Example

```
<- { "event": "STOP",
     "timestamp": { "seconds": 1267041730, "microseconds": 281295 } }
```

RESUME (Event)

Emitted when the virtual machine resumes execution

Since

0.12

Example

```
<- { "event": "RESUME",
     "timestamp": { "seconds": 1271770767, "microseconds": 582542 } }
```

SUSPEND (Event)

Emitted when guest enters a hardware suspension state, for example, S3 state, which is sometimes called standby state

Since

1.1

Example

```
<- { "event": "SUSPEND",
     "timestamp": { "seconds": 1344456160, "microseconds": 309119 } }
```

SUSPEND_DISK (Event)

Emitted when guest enters a hardware suspension state with data saved on disk, for example, S4 state, which is sometimes called hibernate state

Note

QEMU shuts down (similar to event SHUTDOWN) when entering this state

Since

1.2

Example

```
<- { "event": "SUSPEND_DISK",
     "timestamp": { "seconds": 1344456160, "microseconds": 309119 } }
```

WAKEUP (Event)

Emitted when the guest has woken up from suspend state and is running

Since

1.1

Example

```
<- { "event": "WAKEUP",
     "timestamp": { "seconds": 1344522075, "microseconds": 745528 } }
```

WATCHDOG (Event)

Emitted when the watchdog device's timer is expired

Arguments

action: WatchdogAction
action that has been taken

Note

If action is "reset", "shutdown", or "pause" the WATCHDOG event is followed respectively by the RESET, SHUTDOWN, or STOP events

Note

This event is rate-limited.

Since

0.13

Example

```
<- { "event": "WATCHDOG",
     "data": { "action": "reset" },
     "timestamp": { "seconds": 1267061043, "microseconds": 959568 } }
```

WatchdogAction (Enum)

An enumeration of the actions taken when the watchdog device's timer is expired

Values

reset
system resets

shutdown
system shutdown, note that it is similar to `powerdown`, which tries to set to system status and notify guest

poweroff
system poweroff, the emulator program exits

pause
system pauses, similar to `stop`

debug
system enters debug state

none
nothing is done

inject-nmi
a non-maskable interrupt is injected into the first VCPU (all VCPUS on x86) (since 2.4)

Since

2.1

RebootAction (Enum)

Possible QEMU actions upon guest reboot

Values

reset
Reset the VM

shutdown
Shutdown the VM and exit, according to the shutdown action

Since

6.0

ShutdownAction (Enum)

Possible QEMU actions upon guest shutdown

Values

poweroff
Shutdown the VM and exit

pause
pause the VM

Since

6.0

PanicAction (Enum)

Values

none
Continue VM execution

pause
Pause the VM

shutdown
Shutdown the VM and exit, according to the shutdown action

exit-failure
Shutdown the VM and exit with nonzero status (since 7.1)

Since

6.0

watchdog-set-action (Command)

Set watchdog action.

Arguments

action: WatchdogAction
WatchdogAction action taken when watchdog timer expires.

Since

2.11

Example

```
-> { "execute": "watchdog-set-action",
     "arguments": { "action": "inject-nmi" } }
<- { "return": {} }
```

set-action (Command)

Set the actions that will be taken by the emulator in response to guest events.

Arguments

reboot: RebootAction (optional)
RebootAction action taken on guest reboot.

shutdown: ShutdownAction (optional)
ShutdownAction action taken on guest shutdown.

panic: PanicAction (optional)
PanicAction action taken on guest panic.

watchdog: WatchdogAction (optional)
WatchdogAction action taken when watchdog timer expires.

Since

6.0

Example

```
-> { "execute": "set-action",
     "arguments": { "reboot": "shutdown",
                    "shutdown" : "pause",
                    "panic": "pause",
                    "watchdog": "inject-nmi" } }
<- { "return": {} }
```

GUEST_PANICKED (Event)

Emitted when guest OS panic is detected

Arguments

action: GuestPanicAction
action that has been taken, currently always “pause”

info: GuestPanicInformation (optional)
information about a panic (since 2.9)

Since

1.5

Example

```
<- { "event": "GUEST_PANICKED",
     "data": { "action": "pause" },
     "timestamp": { "seconds": 1648245231, "microseconds": 900001 } }
```

GUEST_CRASHLOADED (Event)

Emitted when guest OS crash loaded is detected

Arguments

action: GuestPanicAction
: action that has been taken, currently always "run"

info: GuestPanicInformation (optional)
: information about a panic

Since

5.0

Example

```
<- { "event": "GUEST_CRASHLOADED",
     "data": { "action": "run" },
     "timestamp": { "seconds": 1648245259, "microseconds": 893771 } }
```

GuestPanicAction (Enum)

An enumeration of the actions taken when guest OS panic is detected

Values

pause
: system pauses

poweroff
: system powers off (since 2.8)

run
: system continues to run (since 5.0)

Since

2.1

GuestPanicInformationType (Enum)

An enumeration of the guest panic information types

Values

hyper-v
: hyper-v guest panic information type

s390
: s390 guest panic information type (Since: 2.12)

Since

2.9

GuestPanicInformation (Object)

Information about a guest panic

Members

type: GuestPanicInformationType
: Crash type that defines the hypervisor specific information

The members of GuestPanicInformationHyperV when type is "hyper-v"
The members of GuestPanicInformationS390 when type is "s390"

Since

2.9

GuestPanicInformationHyperV (Object)

Hyper-V specific guest panic information (HV crash MSRs)

Members

arg1: int
for Windows, STOP code for the guest crash. For Linux, an error code.

arg2: int
for Windows, first argument of the STOP. For Linux, the guest OS ID, which has the kernel version in bits 16-47 and 0x8100 in bits 48-63.

arg3: int
for Windows, second argument of the STOP. For Linux, the program counter of the guest.

arg4: int
for Windows, third argument of the STOP. For Linux, the RAX register (x86) or the stack pointer (aarch64) of the guest.

arg5: int
for Windows, fourth argument of the STOP. For x86 Linux, the stack pointer of the guest.

Since

2.9

S390CrashReason (Enum)

Reason why the CPU is in a crashed state.

Values

unknown
no crash reason was set

disabled-wait
the CPU has entered a disabled wait state

extint-loop
clock comparator or cpu timer interrupt with new PSW enabled for external interrupts

pgmint-loop
program interrupt with BAD new PSW

opint-loop
operation exception interrupt with invalid code at the program interrupt new PSW

Since

2.12

GuestPanicInformationS390 (Object)

S390 specific guest panic information (PSW)

Members

core: int
core id of the CPU that crashed

psw-mask: int
control fields of guest PSW

psw-addr: int
guest instruction address

reason: S390CrashReason
guest crash reason

Since

2.12

MEMORY_FAILURE (Event)

Emitted when a memory failure occurs on host side.

Arguments

recipient: MemoryFailureRecipient
recipient is defined as `MemoryFailureRecipient`.

action: MemoryFailureAction
action that has been taken.

flags: MemoryFailureFlags
flags for MemoryFailureAction.

Since

5.2

Example

```
<- { "event": "MEMORY_FAILURE",
     "data": { "recipient": "hypervisor",
               "action": "fatal",
               "flags": { "action-required": false,
                          "recursive": false } },
     "timestamp": { "seconds": 1267061043, "microseconds": 959568 } }
```

MemoryFailureRecipient (Enum)

Hardware memory failure occurs, handled by recipient.

Values

hypervisor
memory failure at QEMU process address space. (none guest memory, but used by QEMU itself).

guest
memory failure at guest memory,

Since

5.2

MemoryFailureAction (Enum)

Actions taken by QEMU in response to a hardware memory failure.

Values

ignore
the memory failure could be ignored. This will only be the case for action-optional failures.

inject
memory failure occurred in guest memory, the guest enabled MCE handling mechanism, and QEMU could inject the MCE into the guest successfully.

fatal
the failure is unrecoverable. This occurs for action-required failures if the recipient is the hypervisor; QEMU will exit.

reset
the failure is unrecoverable but confined to the guest. This occurs if the recipient is a guest guest which is not ready to handle memory failures.

Since

5.2

MemoryFailureFlags (Object)

Additional information on memory failures.

Members

action-required: boolean
whether a memory failure event is action-required or action-optional (e.g. a failure during memory scrub).

recursive: boolean
whether the failure occurred while the previous failure was still in progress.

Since

5.2

NotifyVmexitOption (Enum)

An enumeration of the options specified when enabling notify VM exit

Values

run
enable the feature, do nothing and continue if the notify VM exit happens.

internal-error
enable the feature, raise a internal error if the notify VM exit happens.

disable
disable the feature.

Since

7.2

5.11.6 Cryptography

QCryptoTLSCredsEndpoint (Enum)

The type of network endpoint that will be using the credentials. Most types of credential require different setup / structures depending on whether they will be used in a server versus a client.

Values

client
the network endpoint is acting as the client

server
the network endpoint is acting as the server

Since

2.5

QCryptoSecretFormat (Enum)

The data format that the secret is provided in

Values

raw
raw bytes. When encoded in JSON only valid UTF-8 sequences can be used

base64
arbitrary base64 encoded binary data

Since

2.6

QCryptoHashAlgorithm (Enum)

The supported algorithms for computing content digests

Values

md5
MD5. Should not be used in any new code, legacy compat only

sha1
SHA-1. Should not be used in any new code, legacy compat only

sha224
SHA-224. (since 2.7)

sha256
SHA-256. Current recommended strong hash.

sha384
SHA-384. (since 2.7)

sha512
SHA-512. (since 2.7)

ripemd160
RIPEMD-160. (since 2.7)

Since

2.6

QCryptoCipherAlgorithm (Enum)

The supported algorithms for content encryption ciphers

Values

aes-128
: AES with 128 bit / 16 byte keys

aes-192
: AES with 192 bit / 24 byte keys

aes-256
: AES with 256 bit / 32 byte keys

des
: DES with 56 bit / 8 byte keys. Do not use except in VNC. (since 6.1)

3des
: 3DES(EDE) with 192 bit / 24 byte keys (since 2.9)

cast5-128
: Cast5 with 128 bit / 16 byte keys

serpent-128
: Serpent with 128 bit / 16 byte keys

serpent-192
: Serpent with 192 bit / 24 byte keys

serpent-256
: Serpent with 256 bit / 32 byte keys

twofish-128
: Twofish with 128 bit / 16 byte keys

twofish-192
: Twofish with 192 bit / 24 byte keys

twofish-256
: Twofish with 256 bit / 32 byte keys

sm4
: SM4 with 128 bit / 16 byte keys (since 9.0)

Since

2.6

QCryptoCipherMode (Enum)

The supported modes for content encryption ciphers

Values

ecb
Electronic Code Book

cbc
Cipher Block Chaining

xts
XEX with tweaked code book and ciphertext stealing

ctr
Counter (Since 2.8)

Since

2.6

QCryptoIVGenAlgorithm (Enum)

The supported algorithms for generating initialization vectors for full disk encryption. The 'plain' generator should not be used for disks with sector numbers larger than 2^32, except where compatibility with pre-existing Linux dm-crypt volumes is required.

Values

plain
64-bit sector number truncated to 32-bits

plain64
64-bit sector number

essiv
64-bit sector number encrypted with a hash of the encryption key

Since

2.6

QCryptoBlockFormat (Enum)

The supported full disk encryption formats

Values

qcow
: QCow/QCow2 built-in AES-CBC encryption. Use only for liberating data from old images.

luks
: LUKS encryption format. Recommended for new images

Since

2.6

QCryptoBlockOptionsBase (Object)

The common options that apply to all full disk encryption formats

Members

format: QCryptoBlockFormat
: the encryption format

Since

2.6

QCryptoBlockOptionsQCow (Object)

The options that apply to QCow/QCow2 AES-CBC encryption format

Members

key-secret: string (optional)
: the ID of a QCryptoSecret object providing the decryption key. Mandatory except when probing image for metadata only.

Since

2.6

QCryptoBlockOptionsLUKS (Object)

The options that apply to LUKS encryption format

Members

key-secret: string (optional)
the ID of a QCryptoSecret object providing the decryption key. Mandatory except when probing image for metadata only.

Since

2.6

QCryptoBlockCreateOptionsLUKS (Object)

The options that apply to LUKS encryption format initialization

Members

cipher-alg: QCryptoCipherAlgorithm (optional)
the cipher algorithm for data encryption Currently defaults to 'aes-256'.

cipher-mode: QCryptoCipherMode (optional)
the cipher mode for data encryption Currently defaults to 'xts'

ivgen-alg: QCryptoIVGenAlgorithm (optional)
the initialization vector generator Currently defaults to 'plain64'

ivgen-hash-alg: QCryptoHashAlgorithm (optional)
the initialization vector generator hash Currently defaults to 'sha256'

hash-alg: QCryptoHashAlgorithm (optional)
the master key hash algorithm Currently defaults to 'sha256'

iter-time: int (optional)
number of milliseconds to spend in PBKDF passphrase processing. Currently defaults to 2000. (since 2.8)

detached-header: boolean (optional)
create a detached LUKS header. (since 9.0)

The members of QCryptoBlockOptionsLUKS

Since

2.6

QCryptoBlockOpenOptions (Object)

The options that are available for all encryption formats when opening an existing volume

Members

The members of QCryptoBlockOptionsBase
The members of QCryptoBlockOptionsQCow when format is "qcow"
The members of QCryptoBlockOptionsLUKS when format is "luks"

Since

2.6

QCryptoBlockCreateOptions (Object)

The options that are available for all encryption formats when initializing a new volume

Members

The members of QCryptoBlockOptionsBase
The members of QCryptoBlockOptionsQCow when format is "qcow"
The members of QCryptoBlockCreateOptionsLUKS when format is "luks"

Since

2.6

QCryptoBlockInfoBase (Object)

The common information that applies to all full disk encryption formats

Members

format: QCryptoBlockFormat
: the encryption format

Since

2.7

QCryptoBlockInfoLUKSSlot (Object)

Information about the LUKS block encryption key slot options

Members

active: boolean
whether the key slot is currently in use

key-offset: int
offset to the key material in bytes

iters: int (optional)
number of PBKDF2 iterations for key material

stripes: int (optional)
number of stripes for splitting key material

Since

2.7

QCryptoBlockInfoLUKS (Object)

Information about the LUKS block encryption options

Members

cipher-alg: QCryptoCipherAlgorithm
the cipher algorithm for data encryption

cipher-mode: QCryptoCipherMode
the cipher mode for data encryption

ivgen-alg: QCryptoIVGenAlgorithm
the initialization vector generator

ivgen-hash-alg: QCryptoHashAlgorithm (optional)
the initialization vector generator hash

hash-alg: QCryptoHashAlgorithm
the master key hash algorithm

detached-header: boolean
whether the LUKS header is detached (Since 9.0)

payload-offset: int
offset to the payload data in bytes

master-key-iters: int
number of PBKDF2 iterations for key material

uuid: string
unique identifier for the volume

slots: array of QCryptoBlockInfoLUKSSlot
information about each key slot

Since

2.7

QCryptoBlockInfo (Object)

Information about the block encryption options

Members

The members of QCryptoBlockInfoBase
The members of QCryptoBlockInfoLUKS when format is "luks"

Since

2.7

QCryptoBlockLUKSKeyslotState (Enum)

Defines state of keyslots that are affected by the update

Values

active
The slots contain the given password and marked as active

inactive
The slots are erased (contain garbage) and marked as inactive

Since

5.1

QCryptoBlockAmendOptionsLUKS (Object)

This struct defines the update parameters that activate/de-activate set of keyslots

Members

state: QCryptoBlockLUKSKeyslotState
the desired state of the keyslots

new-secret: string (optional)
The ID of a QCryptoSecret object providing the password to be written into added active keyslots

old-secret: string (optional)
Optional (for deactivation only) If given will deactivate all keyslots that match password located in QCryptoSecret with this ID

iter-time: int (optional)
Optional (for activation only) Number of milliseconds to spend in PBKDF passphrase processing for the newly activated keyslot. Currently defaults to 2000.

keyslot: int (optional)
Optional. ID of the keyslot to activate/deactivate. For keyslot activation, keyslot should not be active already (this is unsafe to update an active keyslot), but possible if 'force' parameter is given. If keyslot is not given, first free keyslot will be written.

For keyslot deactivation, this parameter specifies the exact keyslot to deactivate

secret: string (optional)
Optional. The ID of a QCryptoSecret object providing the password to use to retrieve current master key. Defaults to the same secret that was used to open the image

Since

5.1

QCryptoBlockAmendOptions (Object)

The options that are available for all encryption formats when amending encryption settings

Members

The members of QCryptoBlockOptionsBase
The members of QCryptoBlockAmendOptionsLUKS when format is "luks"

Since

5.1

SecretCommonProperties (Object)

Properties for objects of classes derived from secret-common.

Members

loaded: boolean (optional)
: if true, the secret is loaded immediately when applying this option and will probably fail when processing the next option. Don't use; only provided for compatibility. (default: false)

format: QCryptoSecretFormat (optional)
: the data format that the secret is provided in (default: raw)

keyid: string (optional)
: the name of another secret that should be used to decrypt the provided data. If not present, the data is assumed to be unencrypted.

iv: string (optional)
: the random initialization vector used for encryption of this particular secret. Should be a base64 encrypted string of the 16-byte IV. Mandatory if `keyid` is given. Ignored if `keyid` is absent.

Features

deprecated
: Member `loaded` is deprecated. Setting true doesn't make sense, and false is already the default.

Since

2.6

SecretProperties (Object)

Properties for secret objects.

Either `data` or `file` must be provided, but not both.

Members

data: string (optional)
: the associated with the secret from

file: string (optional)
: the filename to load the data associated with the secret from

The members of SecretCommonProperties

Since

2.6

SecretKeyringProperties (Object)

Properties for secret_keyring objects.

Members

serial: int
serial number that identifies a key to get from the kernel

The members of SecretCommonProperties

Since

5.1

TlsCredsProperties (Object)

Properties for objects of classes derived from tls-creds.

Members

verify-peer: boolean (optional)
if true the peer credentials will be verified once the handshake is completed. This is a no-op for anonymous credentials. (default: true)

dir: string (optional)
the path of the directory that contains the credential files

endpoint: QCryptoTLSCredsEndpoint (optional)
whether the QEMU network backend that uses the credentials will be acting as a client or as a server (default: client)

priority: string (optional)
a gnutls priority string as described at https://gnutls.org/manual/html_node/Priority-Strings.html

Since

2.5

TlsCredsAnonProperties (Object)

Properties for tls-creds-anon objects.

Members

loaded: boolean (optional)
if true, the credentials are loaded immediately when applying this option and will ignore options that are processed later. Don't use; only provided for compatibility. (default: false)

The members of TlsCredsProperties

Features

deprecated
Member loaded is deprecated. Setting true doesn't make sense, and false is already the default.

Since

2.5

TlsCredsPskProperties (Object)

Properties for tls-creds-psk objects.

Members

loaded: boolean (optional)
if true, the credentials are loaded immediately when applying this option and will ignore options that are processed later. Don't use; only provided for compatibility. (default: false)

username: string (optional)
the username which will be sent to the server. For clients only. If absent, "qemu" is sent and the property will read back as an empty string.

The members of TlsCredsProperties

Features

deprecated
Member loaded is deprecated. Setting true doesn't make sense, and false is already the default.

Since

3.0

TlsCredsX509Properties (Object)

Properties for tls-creds-x509 objects.

Members

loaded: boolean (optional)
if true, the credentials are loaded immediately when applying this option and will ignore options that are processed later. Don't use; only provided for compatibility. (default: false)

sanity-check: boolean (optional)
if true, perform some sanity checks before using the credentials (default: true)

passwordid: string (optional)
For the server-key.pem and client-key.pem files which contain sensitive private keys, it is possible to use an encrypted version by providing the passwordid parameter. This provides the ID of a previously created secret object containing the password for decryption.

The members of TlsCredsProperties

Features

deprecated
Member loaded is deprecated. Setting true doesn't make sense, and false is already the default.

Since

2.5

QCryptoAkCipherAlgorithm (Enum)

The supported algorithms for asymmetric encryption ciphers

Values

rsa
RSA algorithm

Since

7.1

QCryptoAkCipherKeyType (Enum)

The type of asymmetric keys.

Values

public
Not documented

private
Not documented

Since

7.1

QCryptoRSAPaddingAlgorithm (Enum)

The padding algorithm for RSA.

Values

raw
no padding used

pkcs1
pkcs1#v1.5

Since

7.1

QCryptoAkCipherOptionsRSA (Object)

Specific parameters for RSA algorithm.

Members

hash-alg: QCryptoHashAlgorithm
QCryptoHashAlgorithm

padding-alg: QCryptoRSAPaddingAlgorithm
QCryptoRSAPaddingAlgorithm

Since

7.1

QCryptoAkCipherOptions (Object)

The options that are available for all asymmetric key algorithms when creating a new QCryptoAkCipher.

Members

alg: QCryptoAkCipherAlgorithm
encryption cipher algorithm

The members of QCryptoAkCipherOptionsRSA when alg is "rsa"

Since

7.1

5.11.7 Background jobs

JobType (Enum)

Type of a background job.

Values

commit
block commit job type, see "block-commit"

stream
block stream job type, see "block-stream"

mirror
drive mirror job type, see "drive-mirror"

backup
drive backup job type, see "drive-backup"

create
image creation job type, see "blockdev-create" (since 3.0)

amend
image options amend job type, see "x-blockdev-amend" (since 5.1)

snapshot-load
snapshot load job type, see "snapshot-load" (since 6.0)

snapshot-save
snapshot save job type, see "snapshot-save" (since 6.0)

snapshot-delete
snapshot delete job type, see "snapshot-delete" (since 6.0)

Since

1.7

JobStatus (Enum)

Indicates the present state of a given job in its lifetime.

Values

undefined
Erroneous, default state. Should not ever be visible.

created
The job has been created, but not yet started.

running
The job is currently running.

paused
The job is running, but paused. The pause may be requested by either the QMP user or by internal processes.

ready
The job is running, but is ready for the user to signal completion. This is used for long-running jobs like mirror that are designed to run indefinitely.

standby
The job is ready, but paused. This is nearly identical to `paused`. The job may return to `ready` or otherwise be canceled.

waiting
The job is waiting for other jobs in the transaction to converge to the waiting state. This status will likely not be visible for the last job in a transaction.

pending
The job has finished its work, but has finalization steps that it needs to make prior to completing. These changes will require manual intervention via `job-finalize` if auto-finalize was set to false. These pending changes may still fail.

aborting
The job is in the process of being aborted, and will finish with an error. The job will afterwards report that it is `concluded`. This status may not be visible to the management process.

concluded
The job has finished all work. If auto-dismiss was set to false, the job will remain in the query list until it is dismissed via `job-dismiss`.

null
The job is in the process of being dismantled. This state should not ever be visible externally.

Since

2.12

`JobVerb` (Enum)

Represents command verbs that can be applied to a job.

Values

cancel
see `job-cancel`

pause
see `job-pause`

resume
see `job-resume`

set-speed
see `block-job-set-speed`

complete
see `job-complete`

dismiss
see `job-dismiss`

finalize
see `job-finalize`

change
see `block-job-change` (since 8.2)

Since

2.12

JOB_STATUS_CHANGE (Event)

Emitted when a job transitions to a different status.

Arguments

id: string
The job identifier

status: JobStatus
The new job status

Since

3.0

job-pause (Command)

Pause an active job.

This command returns immediately after marking the active job for pausing. Pausing an already paused job is an error.

The job will pause as soon as possible, which means transitioning into the PAUSED state if it was RUNNING, or into STANDBY if it was READY. The corresponding JOB_STATUS_CHANGE event will be emitted.

Cancelling a paused job automatically resumes it.

Arguments

id: string
The job identifier.

Since

3.0

job-resume (Command)

Resume a paused job.

This command returns immediately after resuming a paused job. Resuming an already running job is an error.

Arguments

id: string
The job identifier.

Since

3.0

job-cancel (Command)

Instruct an active background job to cancel at the next opportunity. This command returns immediately after marking the active job for cancellation.

The job will cancel as soon as possible and then emit a JOB_STATUS_CHANGE event. Usually, the status will change to ABORTING, but it is possible that a job successfully completes (e.g. because it was almost done and there was no opportunity to cancel earlier than completing the job) and transitions to PENDING instead.

Arguments

id: string
The job identifier.

Since

3.0

job-complete (Command)

Manually trigger completion of an active job in the READY state.

Arguments

id: string
The job identifier.

Since

3.0

job-dismiss (Command)

Deletes a job that is in the CONCLUDED state. This command only needs to be run explicitly for jobs that don't have automatic dismiss enabled.

This command will refuse to operate on any job that has not yet reached its terminal state, JOB_STATUS_CONCLUDED. For jobs that make use of JOB_READY event, job-cancel or job-complete will still need to be used as appropriate.

Arguments

id: string
The job identifier.

Since

3.0

job-finalize (Command)

Instructs all jobs in a transaction (or a single job if it is not part of any transaction) to finalize any graph changes and do any necessary cleanup. This command requires that all involved jobs are in the PENDING state.

For jobs in a transaction, instructing one job to finalize will force ALL jobs in the transaction to finalize, so it is only necessary to instruct a single member job to finalize.

Arguments

id: string
The identifier of any job in the transaction, or of a job that is not part of any transaction.

Since

3.0

JobInfo (Object)

Information about a job.

Members

id: string
The job identifier

type: JobType
The kind of job that is being performed

status: JobStatus
Current job state/status

current-progress: int
Progress made until now. The unit is arbitrary and the value can only meaningfully be used for the ratio of `current-progress` to `total-progress`. The value is monotonically increasing.

total-progress: int
Estimated `current-progress` value at the completion of the job. This value can arbitrarily change while the job is running, in both directions.

error: string (optional)
If this field is present, the job failed; if it is still missing in the CONCLUDED state, this indicates successful completion.

The value is a human-readable error message to describe the reason for the job failure. It should not be parsed by applications.

Since

3.0

`query-jobs` (Command)

Return information about jobs.

Returns

a list with a `JobInfo` for each active job

Since

3.0

5.11.8 Block devices

Block core (VM unrelated)

`SnapshotInfo` (Object)

Members

id: string
unique snapshot id

name: string
user chosen name

vm-state-size: int
size of the VM state

date-sec: int
UTC date of the snapshot in seconds

date-nsec: int
fractional part in nano seconds to be used with date-sec

vm-clock-sec: int
VM clock relative to boot in seconds

vm-clock-nsec: int
fractional part in nano seconds to be used with vm-clock-sec

icount: int (optional)
Current instruction count. Appears when execution record/replay is enabled. Used for "time-traveling" to match the moment in the recorded execution with the snapshots. This counter may be obtained through `query-replay` command (since 5.2)

Since

1.3

ImageInfoSpecificQCow2EncryptionBase (Object)

Members

format: BlockdevQcow2EncryptionFormat
The encryption format

Since

2.10

ImageInfoSpecificQCow2Encryption (Object)

Members

The members of ImageInfoSpecificQCow2EncryptionBase
The members of QCryptoBlockInfoLUKS when format is "luks"

Since

2.10

ImageInfoSpecificQCow2 (Object)

Members

compat: string
compatibility level

data-file: string (optional)
the filename of the external data file that is stored in the image and used as a default for opening the image (since: 4.0)

data-file-raw: boolean (optional)
True if the external data file must stay valid as a standalone (read-only) raw image without looking at qcow2 metadata (since: 4.0)

extended-l2: boolean (optional)
true if the image has extended L2 entries; only valid for compat >= 1.1 (since 5.2)

lazy-refcounts: boolean (optional)
on or off; only valid for compat >= 1.1

corrupt: boolean (optional)
true if the image has been marked corrupt; only valid for compat >= 1.1 (since 2.2)

refcount-bits: int
width of a refcount entry in bits (since 2.3)

encrypt: ImageInfoSpecificQCow2Encryption (optional)
details about encryption parameters; only set if image is encrypted (since 2.10)

bitmaps: array of Qcow2BitmapInfo (optional)
A list of qcow2 bitmap details (since 4.0)

compression-type: Qcow2CompressionType
the image cluster compression method (since 5.1)

Since

1.7

ImageInfoSpecificVmdk (Object)

Members

create-type: string
The create type of VMDK image

cid: int
Content id of image

parent-cid: int
Parent VMDK image's cid

extents: array of VmdkExtentInfo
List of extent files

Since

1.7

VmdkExtentInfo (Object)

Information about a VMDK extent file

Members

filename: string
Name of the extent file

format: string
Extent type (e.g. FLAT or SPARSE)

virtual-size: int
Number of bytes covered by this extent

cluster-size: int (optional)
Cluster size in bytes (for non-flat extents)

compressed: boolean (optional)
Whether this extent contains compressed data

Since

8.0

ImageInfoSpecificRbd (Object)

Members

encryption-format: RbdImageEncryptionFormat (optional)
Image encryption format

Since

6.1

ImageInfoSpecificFile (Object)

Members

extent-size-hint: int (optional)
Extent size hint (if available)

Since

8.0

ImageInfoSpecificKind (Enum)

Values

luks
Since 2.7

rbd
Since 6.1

file
Since 8.0

qcow2
Not documented

vmdk
Not documented

Since

1.7

ImageInfoSpecificQCow2Wrapper (Object)

Members

data: ImageInfoSpecificQCow2
image information specific to QCOW2

Since

1.7

ImageInfoSpecificVmdkWrapper (Object)

Members

data: ImageInfoSpecificVmdk
image information specific to VMDK

Since

6.1

ImageInfoSpecificLUKSWrapper (Object)

Members

data: QCryptoBlockInfoLUKS
image information specific to LUKS

Since

2.7

ImageInfoSpecificRbdWrapper (Object)

Members

data: ImageInfoSpecificRbd
image information specific to RBD

Since

6.1

ImageInfoSpecificFileWrapper (Object)

Members

data: ImageInfoSpecificFile
image information specific to files

Since

8.0

ImageInfoSpecific (Object)

A discriminated record of image format specific information structures.

Members

type: ImageInfoSpecificKind
block driver name

The members of ImageInfoSpecificQCow2Wrapper when type is "qcow2"
The members of ImageInfoSpecificVmdkWrapper when type is "vmdk"
The members of ImageInfoSpecificLUKSWrapper when type is "luks"
The members of ImageInfoSpecificRbdWrapper when type is "rbd"
The members of ImageInfoSpecificFileWrapper when type is "file"

Since

1.7

BlockNodeInfo (Object)

Information about a QEMU image file

Members

filename: string
name of the image file

format: string
format of the image file

virtual-size: int
maximum capacity in bytes of the image

actual-size: int (optional)
actual size on disk in bytes of the image

dirty-flag: boolean (optional)
true if image is not cleanly closed

cluster-size: int (optional)
size of a cluster in bytes

encrypted: boolean (optional)
true if the image is encrypted

compressed: boolean (optional)
true if the image is compressed (Since 1.7)

backing-filename: string (optional)
name of the backing file

full-backing-filename: string (optional)
full path of the backing file

backing-filename-format: string (optional)
the format of the backing file

snapshots: array of SnapshotInfo (optional)
list of VM snapshots

format-specific: ImageInfoSpecific (optional)
structure supplying additional format-specific information (since 1.7)

Since

8.0

ImageInfo (Object)

Information about a QEMU image file, and potentially its backing image

Members

backing-image: ImageInfo (optional)
info of the backing image

The members of BlockNodeInfo

Since

1.3

BlockChildInfo (Object)

Information about all nodes in the block graph starting at some node, annotated with information about that node in relation to its parent.

Members

name: string
Child name of the root node in the BlockGraphInfo struct, in its role as the child of some undescribed parent node

info: BlockGraphInfo
Block graph information starting at this node

Since

8.0

BlockGraphInfo (Object)

Information about all nodes in a block (sub)graph in the form of BlockNodeInfo data. The base BlockNodeInfo struct contains the information for the (sub)graph's root node.

Members

children: array of BlockChildInfo
Array of links to this node's child nodes' information

The members of BlockNodeInfo

Since

8.0

ImageCheck (Object)

Information about a QEMU image file check

Members

filename: string
name of the image file checked

format: string
format of the image file checked

check-errors: int
number of unexpected errors occurred during check

image-end-offset: int (optional)
offset (in bytes) where the image ends, this field is present if the driver for the image format supports it

corruptions: int (optional)
number of corruptions found during the check if any

leaks: int (optional)
number of leaks found during the check if any

corruptions-fixed: int (optional)
number of corruptions fixed during the check if any

leaks-fixed: int (optional)
number of leaks fixed during the check if any

total-clusters: int (optional)
total number of clusters, this field is present if the driver for the image format supports it

allocated-clusters: int (optional)
total number of allocated clusters, this field is present if the driver for the image format supports it

fragmented-clusters: int (optional)
total number of fragmented clusters, this field is present if the driver for the image format supports it

compressed-clusters: int (optional)
total number of compressed clusters, this field is present if the driver for the image format supports it

Since

1.4

MapEntry (Object)

Mapping information from a virtual block range to a host file range

Members

start: int
virtual (guest) offset of the first byte described by this entry

length: int
the number of bytes of the mapped virtual range

data: boolean
reading the image will actually read data from a file (in particular, if `offset` is present this means that the sectors are not simply preallocated, but contain actual data in raw format)

zero: boolean
whether the virtual blocks read as zeroes

compressed: boolean
true if the data is stored compressed (since 8.2)

depth: int
number of layers (0 = top image, 1 = top image's backing file, ..., n - 1 = bottom image (where n is the number of images in the chain)) before reaching one for which the range is allocated

present: boolean
true if this layer provides the data, false if adding a backing layer could impact this region (since 6.1)

offset: int (optional)
if present, the image file stores the data for this range in raw format at the given (host) offset

filename: string (optional)
filename that is referred to by `offset`

Since

2.6

BlockdevCacheInfo (Object)

Cache mode information for a block device

Members

writeback: boolean
true if writeback mode is enabled

direct: boolean
true if the host page cache is bypassed (O_DIRECT)

no-flush: boolean
true if flush requests are ignored for the device

Since

2.3

BlockDeviceInfo (Object)

Information about the backing device for a block device.

Members

file: string
the filename of the backing device

node-name: string (optional)
the name of the block driver node (Since 2.0)

ro: boolean
true if the backing device was open read-only

drv: string
the name of the block format used to open the backing device. As of 0.14 this can be: 'blkdebug', 'bochs', 'cloop', 'cow', 'dmg', 'file', 'file', 'ftp', 'ftps', 'host_cdrom', 'host_device', 'http', 'https', 'luks', 'nbd', 'parallels', 'qcow', 'qcow2', 'raw', 'vdi', 'vmdk', 'vpc', 'vvfat' 2.2: 'archipelago' added, 'cow' dropped 2.3: 'host_floppy' deprecated 2.5: 'host_floppy' dropped 2.6: 'luks' added 2.8: 'replication' added, 'tftp' dropped 2.9: 'archipelago' dropped

backing_file: string (optional)
the name of the backing file (for copy-on-write)

backing_file_depth: int
number of files in the backing file chain (since: 1.2)

encrypted: boolean
true if the backing device is encrypted

detect_zeroes: BlockdevDetectZeroesOptions
: detect and optimize zero writes (Since 2.1)

bps: int
: total throughput limit in bytes per second is specified

bps_rd: int
: read throughput limit in bytes per second is specified

bps_wr: int
: write throughput limit in bytes per second is specified

iops: int
: total I/O operations per second is specified

iops_rd: int
: read I/O operations per second is specified

iops_wr: int
: write I/O operations per second is specified

image: ImageInfo
: the info of image used (since: 1.6)

bps_max: int (optional)
: total throughput limit during bursts, in bytes (Since 1.7)

bps_rd_max: int (optional)
: read throughput limit during bursts, in bytes (Since 1.7)

bps_wr_max: int (optional)
: write throughput limit during bursts, in bytes (Since 1.7)

iops_max: int (optional)
: total I/O operations per second during bursts, in bytes (Since 1.7)

iops_rd_max: int (optional)
: read I/O operations per second during bursts, in bytes (Since 1.7)

iops_wr_max: int (optional)
: write I/O operations per second during bursts, in bytes (Since 1.7)

bps_max_length: int (optional)
: maximum length of the bps_max burst period, in seconds. (Since 2.6)

bps_rd_max_length: int (optional)
: maximum length of the bps_rd_max burst period, in seconds. (Since 2.6)

bps_wr_max_length: int (optional)
: maximum length of the bps_wr_max burst period, in seconds. (Since 2.6)

iops_max_length: int (optional)
: maximum length of the iops burst period, in seconds. (Since 2.6)

iops_rd_max_length: int (optional)
: maximum length of the iops_rd_max burst period, in seconds. (Since 2.6)

iops_wr_max_length: int (optional)
: maximum length of the iops_wr_max burst period, in seconds. (Since 2.6)

iops_size: int (optional)
: an I/O size in bytes (Since 1.7)

group: string (optional)
throttle group name (Since 2.4)

cache: BlockdevCacheInfo
the cache mode used for the block device (since: 2.3)

write_threshold: int
configured write threshold for the device. 0 if disabled. (Since 2.3)

dirty-bitmaps: array of BlockDirtyInfo (optional)
dirty bitmaps information (only present if node has one or more dirty bitmaps) (Since 4.2)

Since

0.14

BlockDeviceIoStatus (Enum)

An enumeration of block device I/O status.

Values

ok
The last I/O operation has succeeded

failed
The last I/O operation has failed

nospace
The last I/O operation has failed due to a no-space condition

Since

1.0

BlockDirtyInfo (Object)

Block dirty bitmap information.

Members

name: string (optional)
the name of the dirty bitmap (Since 2.4)

count: int
number of dirty bytes according to the dirty bitmap

granularity: int
granularity of the dirty bitmap in bytes (since 1.4)

recording: boolean
true if the bitmap is recording new writes from the guest. (since 4.0)

busy: boolean
true if the bitmap is in-use by some operation (NBD or jobs) and cannot be modified via QMP or used by another operation. (since 4.0)

persistent: boolean
true if the bitmap was stored on disk, is scheduled to be stored on disk, or both. (since 4.0)

inconsistent: boolean (optional)
true if this is a persistent bitmap that was improperly stored. Implies `persistent` to be true; `recording` and `busy` to be false. This bitmap cannot be used. To remove it, use `block-dirty-bitmap-remove`. (Since 4.0)

Since

1.3

`Qcow2BitmapInfoFlags` (Enum)

An enumeration of flags that a bitmap can report to the user.

Values

in-use
This flag is set by any process actively modifying the qcow2 file, and cleared when the updated bitmap is flushed to the qcow2 image. The presence of this flag in an offline image means that the bitmap was not saved correctly after its last usage, and may contain inconsistent data.

auto
The bitmap must reflect all changes of the virtual disk by any application that would write to this qcow2 file.

Since

4.0

`Qcow2BitmapInfo` (Object)

Qcow2 bitmap information.

Members

name: string
the name of the bitmap

granularity: int
granularity of the bitmap in bytes

flags: array of Qcow2BitmapInfoFlags
flags of the bitmap

Since

4.0

BlockLatencyHistogramInfo (Object)

Block latency histogram.

Members

boundaries: array of int
: list of interval boundary values in nanoseconds, all greater than zero and in ascending order. For example, the list [10, 50, 100] produces the following histogram intervals: [0, 10), [10, 50), [50, 100), [100, +inf).

bins: array of int
: list of io request counts corresponding to histogram intervals, one more element than `boundaries` has. For the example above, `bins` may be something like [3, 1, 5, 2], and corresponding histogram looks like:

```
5|           *
4|           *
3| *         *
2| *         *    *
1| *    *    *    *
 +------------------
     10   50   100
```

Since

4.0

BlockInfo (Object)

Block device information. This structure describes a virtual device and the backing device associated with it.

Members

device: string
: The device name associated with the virtual device.

qdev: string (optional)
: The qdev ID, or if no ID is assigned, the QOM path of the block device. (since 2.10)

type: string
: This field is returned only for compatibility reasons, it should not be used (always returns 'unknown')

removable: boolean
: True if the device supports removable media.

locked: boolean
: True if the guest has locked this device from having its media removed

tray_open: boolean (optional)
True if the device's tray is open (only present if it has a tray)

io-status: BlockDeviceIoStatus (optional)
`BlockDeviceIoStatus`. Only present if the device supports it and the VM is configured to stop on errors (supported device models: virtio-blk, IDE, SCSI except scsi-generic)

inserted: BlockDeviceInfo (optional)
`BlockDeviceInfo` describing the device if media is present

Since

0.14

BlockMeasureInfo (Object)

Image file size calculation information. This structure describes the size requirements for creating a new image file.

The size requirements depend on the new image file format. File size always equals virtual disk size for the 'raw' format, even for sparse POSIX files. Compact formats such as 'qcow2' represent unallocated and zero regions efficiently so file size may be smaller than virtual disk size.

The values are upper bounds that are guaranteed to fit the new image file. Subsequent modification, such as internal snapshot or further bitmap creation, may require additional space and is not covered here.

Members

required: int
Size required for a new image file, in bytes, when copying just allocated guest-visible contents.

fully-allocated: int
Image file size, in bytes, once data has been written to all sectors, when copying just guest-visible contents.

bitmaps: int (optional)
Additional size required if all the top-level bitmap metadata in the source image were to be copied to the destination, present only when source and destination both support persistent bitmaps. (since 5.1)

Since

2.10

query-block (Command)

Get a list of BlockInfo for all virtual block devices.

Returns

a list of `BlockInfo` describing each virtual block device. Filter nodes that were created implicitly are skipped over.

Since

0.14

Example

```
-> { "execute": "query-block" }
<- {
      "return":[
         {
            "io-status": "ok",
            "device":"ide0-hd0",
            "locked":false,
            "removable":false,
            "inserted":{
               "ro":false,
               "drv":"qcow2",
               "encrypted":false,
               "file":"disks/test.qcow2",
               "backing_file_depth":1,
               "bps":1000000,
               "bps_rd":0,
               "bps_wr":0,
               "iops":1000000,
               "iops_rd":0,
               "iops_wr":0,
               "bps_max": 8000000,
               "bps_rd_max": 0,
               "bps_wr_max": 0,
               "iops_max": 0,
               "iops_rd_max": 0,
               "iops_wr_max": 0,
               "iops_size": 0,
               "detect_zeroes": "on",
               "write_threshold": 0,
               "image":{
                  "filename":"disks/test.qcow2",
                  "format":"qcow2",
                  "virtual-size":2048000,
                  "backing_file":"base.qcow2",
                  "full-backing-filename":"disks/base.qcow2",
                  "backing-filename-format":"qcow2",
                  "snapshots":[
                     {
                        "id": "1",
                        "name": "snapshot1",
                        "vm-state-size": 0,
```

(continues on next page)

(continued from previous page)

```
                        "date-sec": 10000200,
                        "date-nsec": 12,
                        "vm-clock-sec": 206,
                        "vm-clock-nsec": 30
                     }
                  ],
                  "backing-image":{
                      "filename":"disks/base.qcow2",
                      "format":"qcow2",
                      "virtual-size":2048000
                  }
               }
            },
            "qdev": "ide_disk",
            "type":"unknown"
         },
         {
            "io-status": "ok",
            "device":"ide1-cd0",
            "locked":false,
            "removable":true,
            "qdev": "/machine/unattached/device[23]",
            "tray_open": false,
            "type":"unknown"
         },
         {
            "device":"floppy0",
            "locked":false,
            "removable":true,
            "qdev": "/machine/unattached/device[20]",
            "type":"unknown"
         },
         {
            "device":"sd0",
            "locked":false,
            "removable":true,
            "type":"unknown"
         }
      ]
   }
```

BlockDeviceTimedStats (Object)

Statistics of a block device during a given interval of time.

Members

interval_length: int
Interval used for calculating the statistics, in seconds.

min_rd_latency_ns: int
Minimum latency of read operations in the defined interval, in nanoseconds.

min_wr_latency_ns: int
Minimum latency of write operations in the defined interval, in nanoseconds.

min_zone_append_latency_ns: int
Minimum latency of zone append operations in the defined interval, in nanoseconds (since 8.1)

min_flush_latency_ns: int
Minimum latency of flush operations in the defined interval, in nanoseconds.

max_rd_latency_ns: int
Maximum latency of read operations in the defined interval, in nanoseconds.

max_wr_latency_ns: int
Maximum latency of write operations in the defined interval, in nanoseconds.

max_zone_append_latency_ns: int
Maximum latency of zone append operations in the defined interval, in nanoseconds (since 8.1)

max_flush_latency_ns: int
Maximum latency of flush operations in the defined interval, in nanoseconds.

avg_rd_latency_ns: int
Average latency of read operations in the defined interval, in nanoseconds.

avg_wr_latency_ns: int
Average latency of write operations in the defined interval, in nanoseconds.

avg_zone_append_latency_ns: int
Average latency of zone append operations in the defined interval, in nanoseconds (since 8.1)

avg_flush_latency_ns: int
Average latency of flush operations in the defined interval, in nanoseconds.

avg_rd_queue_depth: number
Average number of pending read operations in the defined interval.

avg_wr_queue_depth: number
Average number of pending write operations in the defined interval.

avg_zone_append_queue_depth: number
Average number of pending zone append operations in the defined interval (since 8.1).

Since

2.5

BlockDeviceStats (Object)

Statistics of a virtual block device or a block backing device.

Members

rd_bytes: int
The number of bytes read by the device.

wr_bytes: int
The number of bytes written by the device.

zone_append_bytes: int
The number of bytes appended by the zoned devices (since 8.1)

unmap_bytes: int
The number of bytes unmapped by the device (Since 4.2)

rd_operations: int
The number of read operations performed by the device.

wr_operations: int
The number of write operations performed by the device.

zone_append_operations: int
The number of zone append operations performed by the zoned devices (since 8.1)

flush_operations: int
The number of cache flush operations performed by the device (since 0.15)

unmap_operations: int
The number of unmap operations performed by the device (Since 4.2)

rd_total_time_ns: int
Total time spent on reads in nanoseconds (since 0.15).

wr_total_time_ns: int
Total time spent on writes in nanoseconds (since 0.15).

zone_append_total_time_ns: int
Total time spent on zone append writes in nanoseconds (since 8.1)

flush_total_time_ns: int
Total time spent on cache flushes in nanoseconds (since 0.15).

unmap_total_time_ns: int
Total time spent on unmap operations in nanoseconds (Since 4.2)

wr_highest_offset: int
The offset after the greatest byte written to the device. The intended use of this information is for growable sparse files (like qcow2) that are used on top of a physical device.

rd_merged: int
Number of read requests that have been merged into another request (Since 2.3).

wr_merged: int
Number of write requests that have been merged into another request (Since 2.3).

zone_append_merged: int
Number of zone append requests that have been merged into another request (since 8.1)

unmap_merged: int
Number of unmap requests that have been merged into another request (Since 4.2)

idle_time_ns: int (optional)
Time since the last I/O operation, in nanoseconds. If the field is absent it means that there haven't been any operations yet (Since 2.5).

failed_rd_operations: int
The number of failed read operations performed by the device (Since 2.5)

failed_wr_operations: int
The number of failed write operations performed by the device (Since 2.5)

failed_zone_append_operations: int
The number of failed zone append write operations performed by the zoned devices (since 8.1)

failed_flush_operations: int
The number of failed flush operations performed by the device (Since 2.5)

failed_unmap_operations: int
The number of failed unmap operations performed by the device (Since 4.2)

invalid_rd_operations: int
The number of invalid read operations performed by the device (Since 2.5)

invalid_wr_operations: int
The number of invalid write operations performed by the device (Since 2.5)

invalid_zone_append_operations: int
The number of invalid zone append operations performed by the zoned device (since 8.1)

invalid_flush_operations: int
The number of invalid flush operations performed by the device (Since 2.5)

invalid_unmap_operations: int
The number of invalid unmap operations performed by the device (Since 4.2)

account_invalid: boolean
Whether invalid operations are included in the last access statistics (Since 2.5)

account_failed: boolean
Whether failed operations are included in the latency and last access statistics (Since 2.5)

timed_stats: array of BlockDeviceTimedStats
Statistics specific to the set of previously defined intervals of time (Since 2.5)

rd_latency_histogram: BlockLatencyHistogramInfo (optional)
`BlockLatencyHistogramInfo`. (Since 4.0)

wr_latency_histogram: BlockLatencyHistogramInfo (optional)
`BlockLatencyHistogramInfo`. (Since 4.0)

zone_append_latency_histogram: BlockLatencyHistogramInfo (optional)
`BlockLatencyHistogramInfo`. (since 8.1)

flush_latency_histogram: BlockLatencyHistogramInfo (optional)
`BlockLatencyHistogramInfo`. (Since 4.0)

Since

0.14

BlockStatsSpecificFile (Object)

File driver statistics

Members

discard-nb-ok: int
The number of successful discard operations performed by the driver.

discard-nb-failed: int
The number of failed discard operations performed by the driver.

discard-bytes-ok: int
The number of bytes discarded by the driver.

Since

4.2

BlockStatsSpecificNvme (Object)

NVMe driver statistics

Members

completion-errors: int
The number of completion errors.

aligned-accesses: int
The number of aligned accesses performed by the driver.

unaligned-accesses: int
The number of unaligned accesses performed by the driver.

Since

5.2

BlockStatsSpecific (Object)

Block driver specific statistics

Members

driver: BlockdevDriver
block driver name

The members of BlockStatsSpecificFile when driver is "file"
The members of BlockStatsSpecificFile when driver is "host_device" (If: HAVE_HOST_BLOCK_DEVICE)
The members of BlockStatsSpecificNvme when driver is "nvme"

Since

4.2

BlockStats (Object)

Statistics of a virtual block device or a block backing device.

Members

device: string (optional)
If the stats are for a virtual block device, the name corresponding to the virtual block device.

node-name: string (optional)
The node name of the device. (Since 2.3)

qdev: string (optional)
The qdev ID, or if no ID is assigned, the QOM path of the block device. (since 3.0)

stats: BlockDeviceStats
A BlockDeviceStats for the device.

driver-specific: BlockStatsSpecific (optional)
Optional driver-specific stats. (Since 4.2)

parent: BlockStats (optional)
This describes the file block device if it has one. Contains recursively the statistics of the underlying protocol (e.g. the host file for a qcow2 image). If there is no underlying protocol, this field is omitted

backing: BlockStats (optional)
This describes the backing block device if it has one. (Since 2.0)

Since

0.14

query-blockstats (Command)

Query the BlockStats for all virtual block devices.

Arguments

query-nodes: boolean (optional)
: If true, the command will query all the block nodes that have a node name, in a list which will include "parent" information, but not "backing". If false or omitted, the behavior is as before - query all the device backends, recursively including their "parent" and "backing". Filter nodes that were created implicitly are skipped over in this mode. (Since 2.3)

Returns

A list of BlockStats for each virtual block devices.

Since

0.14

Example

```
-> { "execute": "query-blockstats" }
<- {
      "return":[
         {
            "device":"ide0-hd0",
            "parent":{
               "stats":{
                  "wr_highest_offset":3686448128,
                  "wr_bytes":9786368,
                  "wr_operations":751,
                  "rd_bytes":122567168,
                  "rd_operations":36772
                  "wr_total_times_ns":313253456
                  "rd_total_times_ns":3465673657
                  "flush_total_times_ns":49653
                  "flush_operations":61,
                  "rd_merged":0,
                  "wr_merged":0,
                  "idle_time_ns":2953431879,
                  "account_invalid":true,
                  "account_failed":false
               }
```

(continues on next page)

(continued from previous page)

```
            },
            "stats":{
               "wr_highest_offset":2821110784,
               "wr_bytes":9786368,
               "wr_operations":692,
               "rd_bytes":122739200,
               "rd_operations":36604
               "flush_operations":51,
               "wr_total_times_ns":313253456
               "rd_total_times_ns":3465673657
               "flush_total_times_ns":49653,
               "rd_merged":0,
               "wr_merged":0,
               "idle_time_ns":2953431879,
               "account_invalid":true,
               "account_failed":false
            },
            "qdev": "/machine/unattached/device[23]"
         },
         {
            "device":"ide1-cd0",
            "stats":{
               "wr_highest_offset":0,
               "wr_bytes":0,
               "wr_operations":0,
               "rd_bytes":0,
               "rd_operations":0
               "flush_operations":0,
               "wr_total_times_ns":0
               "rd_total_times_ns":0
               "flush_total_times_ns":0,
               "rd_merged":0,
               "wr_merged":0,
               "account_invalid":false,
               "account_failed":false
            },
            "qdev": "/machine/unattached/device[24]"
         },
         {
            "device":"floppy0",
            "stats":{
               "wr_highest_offset":0,
               "wr_bytes":0,
               "wr_operations":0,
               "rd_bytes":0,
               "rd_operations":0
               "flush_operations":0,
               "wr_total_times_ns":0
               "rd_total_times_ns":0
               "flush_total_times_ns":0,
               "rd_merged":0,
               "wr_merged":0,
```

(continues on next page)

(continued from previous page)

```
               "account_invalid":false,
               "account_failed":false
            },
            "qdev": "/machine/unattached/device[16]"
         },
         {
            "device":"sd0",
            "stats":{
               "wr_highest_offset":0,
               "wr_bytes":0,
               "wr_operations":0,
               "rd_bytes":0,
               "rd_operations":0
               "flush_operations":0,
               "wr_total_times_ns":0
               "rd_total_times_ns":0
               "flush_total_times_ns":0,
               "rd_merged":0,
               "wr_merged":0,
               "account_invalid":false,
               "account_failed":false
            }
         }
      ]
   }
```

BlockdevOnError (Enum)

An enumeration of possible behaviors for errors on I/O operations. The exact meaning depends on whether the I/O was initiated by a guest or by a block job

Values

report
: for guest operations, report the error to the guest; for jobs, cancel the job

ignore
: ignore the error, only report a QMP event (BLOCK_IO_ERROR or BLOCK_JOB_ERROR). The backup, mirror and commit block jobs retry the failing request later and may still complete successfully. The stream block job continues to stream and will complete with an error.

enospc
: same as `stop` on ENOSPC, same as `report` otherwise.

stop
: for guest operations, stop the virtual machine; for jobs, pause the job

auto
: inherit the error handling policy of the backend (since: 2.7)

Since

1.3

MirrorSyncMode (Enum)

An enumeration of possible behaviors for the initial synchronization phase of storage mirroring.

Values

top
copies data in the topmost image to the destination

full
copies data from all images to the destination

none
only copy data written from now on

incremental
only copy data described by the dirty bitmap. (since: 2.4)

bitmap
only copy data described by the dirty bitmap. (since: 4.2) Behavior on completion is determined by the BitmapSyncMode.

Since

1.3

BitmapSyncMode (Enum)

An enumeration of possible behaviors for the synchronization of a bitmap when used for data copy operations.

Values

on-success
The bitmap is only synced when the operation is successful. This is the behavior always used for 'INCREMENTAL' backups.

never
The bitmap is never synchronized with the operation, and is treated solely as a read-only manifest of blocks to copy.

always
The bitmap is always synchronized with the operation, regardless of whether or not the operation was successful.

Since

4.2

MirrorCopyMode (Enum)

An enumeration whose values tell the mirror block job when to trigger writes to the target.

Values

background
: copy data in background only.

write-blocking
: when data is written to the source, write it (synchronously) to the target as well. In addition, data is copied in background just like in `background` mode.

Since

3.0

BlockJobInfoMirror (Object)

Information specific to mirror block jobs.

Members

actively-synced: boolean
: Whether the source is actively synced to the target, i.e. same data and new writes are done synchronously to both.

Since

8.2

BlockJobInfo (Object)

Information about a long-running block device operation.

Members

type: JobType
the job type ('stream' for image streaming)

device: string
The job identifier. Originally the device name but other values are allowed since QEMU 2.7

len: int
Estimated `offset` value at the completion of the job. This value can arbitrarily change while the job is running, in both directions.

offset: int
Progress made until now. The unit is arbitrary and the value can only meaningfully be used for the ratio of `offset` to `len`. The value is monotonically increasing.

busy: boolean
false if the job is known to be in a quiescent state, with no pending I/O. (Since 1.3)

paused: boolean
whether the job is paused or, if `busy` is true, will pause itself as soon as possible. (Since 1.3)

speed: int
the rate limit, bytes per second

io-status: BlockDeviceIoStatus
the status of the job (since 1.3)

ready: boolean
true if the job may be completed (since 2.2)

status: JobStatus
Current job state/status (since 2.12)

auto-finalize: boolean
Job will finalize itself when PENDING, moving to the CONCLUDED state. (since 2.12)

auto-dismiss: boolean
Job will dismiss itself when CONCLUDED, moving to the NULL state and disappearing from the query list. (since 2.12)

error: string (optional)
Error information if the job did not complete successfully. Not set if the job completed successfully. (since 2.12.1)

The members of `BlockJobInfoMirror` when `type` is `"mirror"`

Since

1.1

query-block-jobs (Command)

Return information about long-running block device operations.

Returns

a list of `BlockJobInfo` for each active block job

Since

1.1

block_resize (Command)

Resize a block image while a guest is running.

Either `device` or `node-name` must be set but not both.

Arguments

`device`: `string` (optional)
: the name of the device to get the image resized

`node-name`: `string` (optional)
: graph node name to get the image resized (Since 2.0)

`size`: `int`
: new image size in bytes

Errors

- If `device` is not a valid block device, DeviceNotFound

Since

0.14

Example

```
-> { "execute": "block_resize",
     "arguments": { "device": "scratch", "size": 1073741824 } }
<- { "return": {} }
```

NewImageMode (Enum)

An enumeration that tells QEMU how to set the backing file path in a new image file.

Values

existing
QEMU should look for an existing image file.

absolute-paths
QEMU should create a new image with absolute paths for the backing file. If there is no backing file available, the new image will not be backed either.

Since

1.1

BlockdevSnapshotSync (Object)

Either `device` or `node-name` must be set but not both.

Members

device: string (optional)
the name of the device to take a snapshot of.

node-name: string (optional)
graph node name to generate the snapshot from (Since 2.0)

snapshot-file: string
the target of the new overlay image. If the file exists, or if it is a device, the overlay will be created in the existing file/device. Otherwise, a new file will be created.

snapshot-node-name: string (optional)
the graph node name of the new image (Since 2.0)

format: string (optional)
the format of the overlay image, default is 'qcow2'.

mode: NewImageMode (optional)
whether and how QEMU should create a new image, default is 'absolute-paths'.

BlockdevSnapshot (Object)

Members

node: string
device or node name that will have a snapshot taken.

overlay: string
reference to the existing block device that will become the overlay of `node`, as part of taking the snapshot. It must not have a current backing file (this can be achieved by passing "backing": null to blockdev-add).

Since

2.5

BackupPerf (Object)

Optional parameters for backup. These parameters don't affect functionality, but may significantly affect performance.

Members

use-copy-range: boolean (optional)
Use copy offloading. Default false.

max-workers: int (optional)
Maximum number of parallel requests for the sustained background copying process. Doesn't influence copy-before-write operations. Default 64.

max-chunk: int (optional)
Maximum request length for the sustained background copying process. Doesn't influence copy-before-write operations. 0 means unlimited. If max-chunk is non-zero then it should not be less than job cluster size which is calculated as maximum of target image cluster size and 64k. Default 0.

Since

6.0

BackupCommon (Object)

Members

job-id: string (optional)
identifier for the newly-created block job. If omitted, the device name will be used. (Since 2.7)

device: string
the device name or node-name of a root node which should be copied.

sync: MirrorSyncMode
what parts of the disk image should be copied to the destination (all the disk, only the sectors allocated in the topmost image, from a dirty bitmap, or only new I/O).

speed: int (optional)
the maximum speed, in bytes per second. The default is 0, for unlimited.

bitmap: string (optional)
The name of a dirty bitmap to use. Must be present if sync is "bitmap" or "incremental". Can be present if sync is "full" or "top". Must not be present otherwise. (Since 2.4 (drive-backup), 3.1 (blockdev-backup))

bitmap-mode: BitmapSyncMode (optional)
Specifies the type of data the bitmap should contain after the operation concludes. Must be present if a bitmap was provided, Must NOT be present otherwise. (Since 4.2)

compress: boolean (optional)
true to compress data, if the target format supports it. (default: false) (since 2.8)

on-source-error: BlockdevOnError (optional)
the action to take on an error on the source, default 'report'. 'stop' and 'enospc' can only be used if the block device supports io-status (see BlockInfo).

on-target-error: BlockdevOnError (optional)
the action to take on an error on the target, default 'report' (no limitations, since this applies to a different block device than `device`).

auto-finalize: boolean (optional)
When false, this job will wait in a PENDING state after it has finished its work, waiting for `block-job-finalize` before making any block graph changes. When true, this job will automatically perform its abort or commit actions. Defaults to true. (Since 2.12)

auto-dismiss: boolean (optional)
When false, this job will wait in a CONCLUDED state after it has completely ceased all work, and awaits `block-job-dismiss`. When true, this job will automatically disappear from the query list without user intervention. Defaults to true. (Since 2.12)

filter-node-name: string (optional)
the node name that should be assigned to the filter driver that the backup job inserts into the graph above node specified by `drive`. If this option is not given, a node name is autogenerated. (Since: 4.2)

x-perf: BackupPerf (optional)
Performance options. (Since 6.0)

Features

unstable
Member `x-perf` is experimental.

Note

`on-source-error` and `on-target-error` only affect background I/O. If an error occurs during a guest write request, the device's rerror/werror actions will be used.

Since

4.2

DriveBackup (Object)

Members

target: string
the target of the new image. If the file exists, or if it is a device, the existing file/device will be used as the new destination. If it does not exist, a new file will be created.

format: string (optional)
the format of the new destination, default is to probe if `mode` is 'existing', else the format of the source

mode: NewImageMode (optional)
whether and how QEMU should create a new image, default is 'absolute-paths'.

The members of BackupCommon

Since

1.6

BlockdevBackup (Object)

Members

target: string
: the device name or node-name of the backup target node.

The members of BackupCommon

Since

2.3

blockdev-snapshot-sync (Command)

Takes a synchronous snapshot of a block device.

For the arguments, see the documentation of BlockdevSnapshotSync.

Errors

- If `device` is not a valid block device, DeviceNotFound

Since

0.14

Example

```
-> { "execute": "blockdev-snapshot-sync",
     "arguments": { "device": "ide-hd0",
                    "snapshot-file":
                    "/some/place/my-image",
                    "format": "qcow2" } }
<- { "return": {} }
```

blockdev-snapshot (Command)

Takes a snapshot of a block device.

Take a snapshot, by installing 'node' as the backing image of 'overlay'. Additionally, if 'node' is associated with a block device, the block device changes to using 'overlay' as its new active image.

For the arguments, see the documentation of BlockdevSnapshot.

Features

allow-write-only-overlay
: If present, the check whether this operation is safe was relaxed so that it can be used to change backing file of a destination of a blockdev-mirror. (since 5.0)

Since

2.5

Example

```
-> { "execute": "blockdev-add",
     "arguments": { "driver": "qcow2",
                    "node-name": "node1534",
                    "file": { "driver": "file",
                              "filename": "hd1.qcow2" },
                    "backing": null } }

<- { "return": {} }

-> { "execute": "blockdev-snapshot",
     "arguments": { "node": "ide-hd0",
                    "overlay": "node1534" } }
<- { "return": {} }
```

change-backing-file (Command)

Change the backing file in the image file metadata. This does not cause QEMU to reopen the image file to reparse the backing filename (it may, however, perform a reopen to change permissions from r/o -> r/w -> r/o, if needed). The new backing file string is written into the image file metadata, and the QEMU internal strings are updated.

Arguments

image-node-name: string
The name of the block driver state node of the image to modify. The "device" argument is used to verify "image-node-name" is in the chain described by "device".

device: string
The device name or node-name of the root node that owns image-node-name.

backing-file: string
The string to write as the backing file. This string is not validated, so care should be taken when specifying the string or the image chain may not be able to be reopened again.

Errors

- If "device" does not exist or cannot be determined, DeviceNotFound

Since

2.1

block-commit (Command)

Live commit of data from overlay image nodes into backing nodes - i.e., writes data between 'top' and 'base' into 'base'.

If top == base, that is an error. If top has no overlays on top of it, or if it is in use by a writer, the job will not be completed by itself. The user needs to complete the job with the block-job-complete command after getting the ready event. (Since 2.0)

If the base image is smaller than top, then the base image will be resized to be the same size as top. If top is smaller than the base image, the base will not be truncated. If you want the base image size to match the size of the smaller top, you can safely truncate it yourself once the commit operation successfully completes.

Arguments

job-id: string (optional)
identifier for the newly-created block job. If omitted, the device name will be used. (Since 2.7)

device: string
the device name or node-name of a root node

base-node: string (optional)
The node name of the backing image to write data into. If not specified, this is the deepest backing image. (since: 3.1)

base: string (optional)
Same as base-node, except that it is a file name rather than a node name. This must be the exact filename string that was used to open the node; other strings, even if addressing the same file, are not accepted

top-node: string (optional)
The node name of the backing image within the image chain which contains the topmost data to be committed down. If not specified, this is the active layer. (since: 3.1)

top: string (optional)
Same as top-node, except that it is a file name rather than a node name. This must be the exact filename string that was used to open the node; other strings, even if addressing the same file, are not accepted

backing-file: string (optional)
The backing file string to write into the overlay image of 'top'. If 'top' does not have an overlay image, or if 'top' is in use by a writer, specifying a backing file string is an error.

This filename is not validated. If a pathname string is such that it cannot be resolved by QEMU, that means that subsequent QMP or HMP commands must use node-names for the image in question, as filename lookup methods will fail.

If not specified, QEMU will automatically determine the backing file string to use, or error out if there is no obvious choice. Care should be taken when specifying the string, to specify a valid filename or protocol. (Since 2.1)

backing-mask-protocol: boolean (optional)
If true, replace any protocol mentioned in the 'backing file format' with 'raw', rather than storing the protocol name as the backing format. Can be used even when no image header will be updated (default false; since 9.0).

speed: int (optional)
the maximum speed, in bytes per second

on-error: BlockdevOnError (optional)
the action to take on an error. 'ignore' means that the request should be retried. (default: report; Since: 5.0)

filter-node-name: string (optional)
the node name that should be assigned to the filter driver that the commit job inserts into the graph above top. If this option is not given, a node name is autogenerated. (Since: 2.9)

auto-finalize: boolean (optional)
When false, this job will wait in a PENDING state after it has finished its work, waiting for block-job-finalize before making any block graph changes. When true, this job will automatically perform its abort or commit actions. Defaults to true. (Since 3.1)

auto-dismiss: boolean (optional)
When false, this job will wait in a CONCLUDED state after it has completely ceased all work, and awaits block-job-dismiss. When true, this job will automatically disappear from the query list without user intervention. Defaults to true. (Since 3.1)

Features

deprecated
Members base and top are deprecated. Use base-node and top-node instead.

Errors

- If device does not exist, DeviceNotFound
- Any other error returns a GenericError.

Since

1.3

Example

```
-> { "execute": "block-commit",
     "arguments": { "device": "virtio0",
                    "top": "/tmp/snap1.qcow2" } }
<- { "return": {} }
```

drive-backup (Command)

Start a point-in-time copy of a block device to a new destination. The status of ongoing drive-backup operations can be checked with query-block-jobs where the BlockJobInfo.type field has the value 'backup'. The operation can be stopped before it has completed using the block-job-cancel command.

Arguments

The members of DriveBackup

Features

deprecated
: This command is deprecated. Use blockdev-backup instead.

Errors

- If device is not a valid block device, GenericError

Since

1.6

Example

```
-> { "execute": "drive-backup",
     "arguments": { "device": "drive0",
                    "sync": "full",
                    "target": "backup.img" } }
<- { "return": {} }
```

blockdev-backup (Command)

Start a point-in-time copy of a block device to a new destination. The status of ongoing blockdev-backup operations can be checked with query-block-jobs where the BlockJobInfo.type field has the value 'backup'. The operation can be stopped before it has completed using the block-job-cancel command.

Arguments

The members of BlockdevBackup

Errors

- If device is not a valid block device, DeviceNotFound

Since

2.3

Example

```
-> { "execute": "blockdev-backup",
     "arguments": { "device": "src-id",
                    "sync": "full",
                    "target": "tgt-id" } }
<- { "return": {} }
```

query-named-block-nodes (Command)

Get the named block driver list

Arguments

flat: boolean (optional)
Omit the nested data about backing image ("backing-image" key) if true. Default is false (Since 5.0)

Returns

the list of BlockDeviceInfo

Since

2.0

Example

```
-> { "execute": "query-named-block-nodes" }
<- { "return": [ { "ro":false,
                   "drv":"qcow2",
                   "encrypted":false,
                   "file":"disks/test.qcow2",
                   "node-name": "my-node",
                   "backing_file_depth":1,
                   "detect_zeroes":"off",
                   "bps":1000000,
                   "bps_rd":0,
                   "bps_wr":0,
                   "iops":1000000,
                   "iops_rd":0,
                   "iops_wr":0,
                   "bps_max": 8000000,
                   "bps_rd_max": 0,
                   "bps_wr_max": 0,
                   "iops_max": 0,
                   "iops_rd_max": 0,
                   "iops_wr_max": 0,
                   "iops_size": 0,
                   "write_threshold": 0,
                   "image":{
                      "filename":"disks/test.qcow2",
                      "format":"qcow2",
                      "virtual-size":2048000,
                      "backing_file":"base.qcow2",
                      "full-backing-filename":"disks/base.qcow2",
                      "backing-filename-format":"qcow2",
                      "snapshots":[
                         {
                            "id": "1",
                            "name": "snapshot1",
                            "vm-state-size": 0,
                            "date-sec": 10000200,
                            "date-nsec": 12,
                            "vm-clock-sec": 206,
                            "vm-clock-nsec": 30
                         }
                      ],
                      "backing-image":{
                          "filename":"disks/base.qcow2",
                          "format":"qcow2",
                          "virtual-size":2048000
                      }
                   } } ] }
```

XDbgBlockGraphNodeType (Enum)

Values

block-backend
corresponds to BlockBackend

block-job
corresponds to BlockJob

block-driver
corresponds to BlockDriverState

Since

4.0

XDbgBlockGraphNode (Object)

Members

id: int
Block graph node identifier. This `id` is generated only for x-debug-query-block-graph and does not relate to any other identifiers in Qemu.

type: XDbgBlockGraphNodeType
Type of graph node. Can be one of block-backend, block-job or block-driver-state.

name: string
Human readable name of the node. Corresponds to node-name for block-driver-state nodes; is not guaranteed to be unique in the whole graph (with block-jobs and block-backends).

Since

4.0

BlockPermission (Enum)

Enum of base block permissions.

Values

consistent-read
A user that has the "permission" of consistent reads is guaranteed that their view of the contents of the block device is complete and self-consistent, representing the contents of a disk at a specific point. For most block devices (including their backing files) this is true, but the property cannot be maintained in a few situations like for intermediate nodes of a commit block job.

write
This permission is required to change the visible disk contents.

write-unchanged
: This permission (which is weaker than BLK_PERM_WRITE) is both enough and required for writes to the block node when the caller promises that the visible disk content doesn't change. As the BLK_PERM_WRITE permission is strictly stronger, either is sufficient to perform an unchanging write.

resize
: This permission is required to change the size of a block node.

Since

4.0

XDbgBlockGraphEdge (Object)

Block Graph edge description for x-debug-query-block-graph.

Members

parent: int
: parent id

child: int
: child id

name: string
: name of the relation (examples are 'file' and 'backing')

perm: array of BlockPermission
: granted permissions for the parent operating on the child

shared-perm: array of BlockPermission
: permissions that can still be granted to other users of the child while it is still attached to this parent

Since

4.0

XDbgBlockGraph (Object)

Block Graph - list of nodes and list of edges.

Members

nodes: array of XDbgBlockGraphNode
: Not documented

edges: array of XDbgBlockGraphEdge
: Not documented

Since

4.0

x-debug-query-block-graph (Command)

Get the block graph.

Features

unstable
: This command is meant for debugging.

Since

4.0

drive-mirror (Command)

Start mirroring a block device's writes to a new destination. target specifies the target of the new image. If the file exists, or if it is a device, it will be used as the new destination for writes. If it does not exist, a new file will be created. `format` specifies the format of the mirror image, default is to probe if mode='existing', else the format of the source.

Arguments

The members of `DriveMirror`

Errors

- If `device` is not a valid block device, GenericError

Since

1.3

Example

```
-> { "execute": "drive-mirror",
     "arguments": { "device": "ide-hd0",
                    "target": "/some/place/my-image",
                    "sync": "full",
                    "format": "qcow2" } }
<- { "return": {} }
```

DriveMirror (Object)

A set of parameters describing drive mirror setup.

Members

job-id: string (optional)
identifier for the newly-created block job. If omitted, the device name will be used. (Since 2.7)

device: string
the device name or node-name of a root node whose writes should be mirrored.

target: string
the target of the new image. If the file exists, or if it is a device, the existing file/device will be used as the new destination. If it does not exist, a new file will be created.

format: string (optional)
the format of the new destination, default is to probe if `mode` is 'existing', else the format of the source

node-name: string (optional)
the new block driver state node name in the graph (Since 2.1)

replaces: string (optional)
with sync=full graph node name to be replaced by the new image when a whole image copy is done. This can be used to repair broken Quorum files. By default, `device` is replaced, although implicitly created filters on it are kept. (Since 2.1)

mode: NewImageMode (optional)
whether and how QEMU should create a new image, default is 'absolute-paths'.

speed: int (optional)
the maximum speed, in bytes per second

sync: MirrorSyncMode
what parts of the disk image should be copied to the destination (all the disk, only the sectors allocated in the topmost image, or only new I/O).

granularity: int (optional)
granularity of the dirty bitmap, default is 64K if the image format doesn't have clusters, 4K if the clusters are smaller than that, else the cluster size. Must be a power of 2 between 512 and 64M (since 1.4).

buf-size: int (optional)
maximum amount of data in flight from source to target (since 1.4).

on-source-error: BlockdevOnError (optional)
the action to take on an error on the source, default 'report'. 'stop' and 'enospc' can only be used if the block device supports io-status (see BlockInfo).

on-target-error: BlockdevOnError (optional)
the action to take on an error on the target, default 'report' (no limitations, since this applies to a different block device than `device`).

unmap: boolean (optional)
Whether to try to unmap target sectors where source has only zero. If true, and target unallocated sectors will read as zero, target image sectors will be unmapped; otherwise, zeroes will be written. Both will result in identical contents. Default is true. (Since 2.4)

copy-mode: MirrorCopyMode (optional)
when to copy data to the destination; defaults to 'background' (Since: 3.0)

auto-finalize: boolean (optional)
When false, this job will wait in a PENDING state after it has finished its work, waiting for `block-job-finalize` before making any block graph changes. When true, this job will automatically perform its abort or commit actions. Defaults to true. (Since 3.1)

auto-dismiss: boolean (optional)
When false, this job will wait in a CONCLUDED state after it has completely ceased all work, and awaits `block-job-dismiss`. When true, this job will automatically disappear from the query list without user intervention. Defaults to true. (Since 3.1)

Since

1.3

BlockDirtyBitmap (Object)

Members

node: string
name of device/node which the bitmap is tracking

name: string
name of the dirty bitmap

Since

2.4

BlockDirtyBitmapAdd (Object)

Members

node: string
name of device/node which the bitmap is tracking

name: string
name of the dirty bitmap (must be less than 1024 bytes)

granularity: int (optional)
the bitmap granularity, default is 64k for block-dirty-bitmap-add

persistent: boolean (optional)
the bitmap is persistent, i.e. it will be saved to the corresponding block device image file on its close. For now only Qcow2 disks support persistent bitmaps. Default is false for block-dirty-bitmap-add. (Since: 2.10)

disabled: boolean (optional)
the bitmap is created in the disabled state, which means that it will not track drive changes. The bitmap may be enabled with block-dirty-bitmap-enable. Default is false. (Since: 4.0)

Since

2.4

BlockDirtyBitmapOrStr (Alternate)

Members

local: string
name of the bitmap, attached to the same node as target bitmap.

external: BlockDirtyBitmap
bitmap with specified node

Since

4.1

BlockDirtyBitmapMerge (Object)

Members

node: string
name of device/node which the `target` bitmap is tracking

target: string
name of the destination dirty bitmap

bitmaps: array of BlockDirtyBitmapOrStr
name(s) of the source dirty bitmap(s) at `node` and/or fully specified BlockDirtyBitmap elements. The latter are supported since 4.1.

Since

4.0

block-dirty-bitmap-add (Command)

Create a dirty bitmap with a name on the node, and start tracking the writes.

Errors

- If node is not a valid block device or node, DeviceNotFound
- If name is already taken, GenericError with an explanation

Since

2.4

Example

```
-> { "execute": "block-dirty-bitmap-add",
     "arguments": { "node": "drive0", "name": "bitmap0" } }
<- { "return": {} }
```

block-dirty-bitmap-remove (Command)

Stop write tracking and remove the dirty bitmap that was created with block-dirty-bitmap-add. If the bitmap is persistent, remove it from its storage too.

Errors

- If node is not a valid block device or node, DeviceNotFound
- If name is not found, GenericError with an explanation
- if name is frozen by an operation, GenericError

Since

2.4

Example

```
-> { "execute": "block-dirty-bitmap-remove",
     "arguments": { "node": "drive0", "name": "bitmap0" } }
<- { "return": {} }
```

block-dirty-bitmap-clear (Command)

Clear (reset) a dirty bitmap on the device, so that an incremental backup from this point in time forward will only backup clusters modified after this clear operation.

Errors

- If node is not a valid block device, DeviceNotFound
- If name is not found, GenericError with an explanation

Since

2.4

Example

```
-> { "execute": "block-dirty-bitmap-clear",
     "arguments": { "node": "drive0", "name": "bitmap0" } }
<- { "return": {} }
```

block-dirty-bitmap-enable (Command)

Enables a dirty bitmap so that it will begin tracking disk changes.

Errors

- If node is not a valid block device, DeviceNotFound
- If name is not found, GenericError with an explanation

Since

4.0

Example

```
-> { "execute": "block-dirty-bitmap-enable",
     "arguments": { "node": "drive0", "name": "bitmap0" } }
<- { "return": {} }
```

block-dirty-bitmap-disable (Command)

Disables a dirty bitmap so that it will stop tracking disk changes.

Errors

- If `node` is not a valid block device, DeviceNotFound
- If `name` is not found, GenericError with an explanation

Since

4.0

Example

```
-> { "execute": "block-dirty-bitmap-disable",
     "arguments": { "node": "drive0", "name": "bitmap0" } }
<- { "return": {} }
```

block-dirty-bitmap-merge (Command)

Merge dirty bitmaps listed in `bitmaps` to the `target` dirty bitmap. Dirty bitmaps in `bitmaps` will be unchanged, except if it also appears as the `target` bitmap. Any bits already set in `target` will still be set after the merge, i.e., this operation does not clear the target. On error, `target` is unchanged.

The resulting bitmap will count as dirty any clusters that were dirty in any of the source bitmaps. This can be used to achieve backup checkpoints, or in simpler usages, to copy bitmaps.

Errors

- If `node` is not a valid block device, DeviceNotFound
- If any bitmap in `bitmaps` or `target` is not found, GenericError
- If any of the bitmaps have different sizes or granularities, GenericError

Since

4.0

Example

```
-> { "execute": "block-dirty-bitmap-merge",
     "arguments": { "node": "drive0", "target": "bitmap0",
                    "bitmaps": ["bitmap1"] } }
<- { "return": {} }
```

BlockDirtyBitmapSha256 (Object)

SHA256 hash of dirty bitmap data

Members

sha256: string
ASCII representation of SHA256 bitmap hash

Since

2.10

x-debug-block-dirty-bitmap-sha256 (Command)

Get bitmap SHA256.

Features

unstable
This command is meant for debugging.

Returns

BlockDirtyBitmapSha256

Errors

- If node is not a valid block device, DeviceNotFound
- If name is not found or if hashing has failed, GenericError with an explanation

Since

2.10

blockdev-mirror (Command)

Start mirroring a block device's writes to a new destination.

Arguments

job-id: string (optional)
identifier for the newly-created block job. If omitted, the device name will be used. (Since 2.7)

device: string
The device name or node-name of a root node whose writes should be mirrored.

target: string
the id or node-name of the block device to mirror to. This mustn't be attached to guest.

replaces: string (optional)
with sync=full graph node name to be replaced by the new image when a whole image copy is done. This can be used to repair broken Quorum files. By default, `device` is replaced, although implicitly created filters on it are kept.

speed: int (optional)
the maximum speed, in bytes per second

sync: MirrorSyncMode
what parts of the disk image should be copied to the destination (all the disk, only the sectors allocated in the topmost image, or only new I/O).

granularity: int (optional)
granularity of the dirty bitmap, default is 64K if the image format doesn't have clusters, 4K if the clusters are smaller than that, else the cluster size. Must be a power of 2 between 512 and 64M

buf-size: int (optional)
maximum amount of data in flight from source to target

on-source-error: BlockdevOnError (optional)
the action to take on an error on the source, default 'report'. 'stop' and 'enospc' can only be used if the block device supports io-status (see BlockInfo).

on-target-error: BlockdevOnError (optional)
the action to take on an error on the target, default 'report' (no limitations, since this applies to a different block device than `device`).

filter-node-name: string (optional)
the node name that should be assigned to the filter driver that the mirror job inserts into the graph above `device`. If this option is not given, a node name is autogenerated. (Since: 2.9)

copy-mode: MirrorCopyMode (optional)
when to copy data to the destination; defaults to 'background' (Since: 3.0)

auto-finalize: boolean (optional)
When false, this job will wait in a PENDING state after it has finished its work, waiting for `block-job-finalize` before making any block graph changes. When true, this job will automatically perform its abort or commit actions. Defaults to true. (Since 3.1)

auto-dismiss: boolean (optional)
When false, this job will wait in a CONCLUDED state after it has completely ceased all work, and awaits `block-job-dismiss`. When true, this job will automatically disappear from the query list without user intervention. Defaults to true. (Since 3.1)

Since

2.6

Example

```
-> { "execute": "blockdev-mirror",
     "arguments": { "device": "ide-hd0",
                    "target": "target0",
                    "sync": "full" } }
<- { "return": {} }
```

`BlockIOThrottle` (Object)

A set of parameters describing block throttling.

Members

device: string (optional)
Block device name

id: string (optional)
The name or QOM path of the guest device (since: 2.8)

bps: int
total throughput limit in bytes per second

bps_rd: int
read throughput limit in bytes per second

bps_wr: int
write throughput limit in bytes per second

iops: int
total I/O operations per second

iops_rd: int
read I/O operations per second

iops_wr: int
write I/O operations per second

bps_max: int (optional)
total throughput limit during bursts, in bytes (Since 1.7)

bps_rd_max: int (optional)
read throughput limit during bursts, in bytes (Since 1.7)

bps_wr_max: int (optional)
write throughput limit during bursts, in bytes (Since 1.7)

iops_max: int (optional)
total I/O operations per second during bursts, in bytes (Since 1.7)

iops_rd_max: int (optional)
read I/O operations per second during bursts, in bytes (Since 1.7)

iops_wr_max: int (optional)
write I/O operations per second during bursts, in bytes (Since 1.7)

bps_max_length: int (optional)
maximum length of the bps_max burst period, in seconds. It must only be set if bps_max is set as well. Defaults to 1. (Since 2.6)

bps_rd_max_length: int (optional)
maximum length of the bps_rd_max burst period, in seconds. It must only be set if bps_rd_max is set as well. Defaults to 1. (Since 2.6)

bps_wr_max_length: int (optional)
maximum length of the bps_wr_max burst period, in seconds. It must only be set if bps_wr_max is set as well. Defaults to 1. (Since 2.6)

iops_max_length: int (optional)
maximum length of the iops burst period, in seconds. It must only be set if iops_max is set as well. Defaults to 1. (Since 2.6)

iops_rd_max_length: int (optional)
maximum length of the iops_rd_max burst period, in seconds. It must only be set if iops_rd_max is set as well. Defaults to 1. (Since 2.6)

iops_wr_max_length: int (optional)
maximum length of the iops_wr_max burst period, in seconds. It must only be set if iops_wr_max is set as well. Defaults to 1. (Since 2.6)

iops_size: int (optional)
an I/O size in bytes (Since 1.7)

group: string (optional)
throttle group name (Since 2.4)

Features

deprecated
Member device is deprecated. Use id instead.

Since

1.1

`ThrottleLimits` (Object)

Limit parameters for throttling. Since some limit combinations are illegal, limits should always be set in one transaction. All fields are optional. When setting limits, if a field is missing the current value is not changed.

Members

`iops-total`: `int` (optional)
limit total I/O operations per second

`iops-total-max`: `int` (optional)
I/O operations burst

`iops-total-max-length`: `int` (optional)
length of the iops-total-max burst period, in seconds It must only be set if `iops-total-max` is set as well.

`iops-read`: `int` (optional)
limit read operations per second

`iops-read-max`: `int` (optional)
I/O operations read burst

`iops-read-max-length`: `int` (optional)
length of the iops-read-max burst period, in seconds It must only be set if `iops-read-max` is set as well.

`iops-write`: `int` (optional)
limit write operations per second

`iops-write-max`: `int` (optional)
I/O operations write burst

`iops-write-max-length`: `int` (optional)
length of the iops-write-max burst period, in seconds It must only be set if `iops-write-max` is set as well.

`bps-total`: `int` (optional)
limit total bytes per second

`bps-total-max`: `int` (optional)
total bytes burst

`bps-total-max-length`: `int` (optional)
length of the bps-total-max burst period, in seconds. It must only be set if `bps-total-max` is set as well.

`bps-read`: `int` (optional)
limit read bytes per second

`bps-read-max`: `int` (optional)
total bytes read burst

`bps-read-max-length`: `int` (optional)
length of the bps-read-max burst period, in seconds It must only be set if `bps-read-max` is set as well.

`bps-write`: `int` (optional)
limit write bytes per second

`bps-write-max`: `int` (optional)
total bytes write burst

`bps-write-max-length`: `int` (optional)
length of the bps-write-max burst period, in seconds It must only be set if `bps-write-max` is set as well.

iops-size: int (optional)
when limiting by iops max size of an I/O in bytes

Since

2.11

ThrottleGroupProperties (Object)

Properties for throttle-group objects.

Members

limits: ThrottleLimits (optional)
limits to apply for this throttle group

x-iops-total: int (optional)
Not documented

x-iops-total-max: int (optional)
Not documented

x-iops-total-max-length: int (optional)
Not documented

x-iops-read: int (optional)
Not documented

x-iops-read-max: int (optional)
Not documented

x-iops-read-max-length: int (optional)
Not documented

x-iops-write: int (optional)
Not documented

x-iops-write-max: int (optional)
Not documented

x-iops-write-max-length: int (optional)
Not documented

x-bps-total: int (optional)
Not documented

x-bps-total-max: int (optional)
Not documented

x-bps-total-max-length: int (optional)
Not documented

x-bps-read: int (optional)
Not documented

x-bps-read-max: int (optional)
Not documented

x-bps-read-max-length: int (optional)
Not documented

x-bps-write: int (optional)
Not documented

x-bps-write-max: int (optional)
Not documented

x-bps-write-max-length: int (optional)
Not documented

x-iops-size: int (optional)
Not documented

Features

unstable
All members starting with x- are aliases for the same key without x- in the `limits` object. This is not a stable interface and may be removed or changed incompatibly in the future. Use `limits` for a supported stable interface.

Since

2.11

block-stream (Command)

Copy data from a backing file into a block device.

The block streaming operation is performed in the background until the entire backing file has been copied. This command returns immediately once streaming has started. The status of ongoing block streaming operations can be checked with query-block-jobs. The operation can be stopped before it has completed using the block-job-cancel command.

The node that receives the data is called the top image, can be located in any part of the chain (but always above the base image; see below) and can be specified using its device or node name. Earlier qemu versions only allowed 'device' to name the top level node; presence of the 'base-node' parameter during introspection can be used as a witness of the enhanced semantics of 'device'.

If a base file is specified then sectors are not copied from that base file and its backing chain. This can be used to stream a subset of the backing file chain instead of flattening the entire image. When streaming completes the image file will have the base file as its backing file, unless that node was changed while the job was running. In that case, base's parent's backing (or filtered, whichever exists) child (i.e., base at the beginning of the job) will be the new backing file.

On successful completion the image file is updated to drop the backing file and the BLOCK_JOB_COMPLETED event is emitted.

In case `device` is a filter node, block-stream modifies the first non-filter overlay node below it to point to the new backing node instead of modifying `device` itself.

Arguments

job-id: string (optional)
identifier for the newly-created block job. If omitted, the device name will be used. (Since 2.7)

device: string
the device or node name of the top image

base: string (optional)
the common backing file name. It cannot be set if base-node or bottom is also set.

base-node: string (optional)
the node name of the backing file. It cannot be set if base or bottom is also set. (Since 2.8)

bottom: string (optional)
the last node in the chain that should be streamed into top. It cannot be set if base or base-node is also set. It cannot be filter node. (Since 6.0)

backing-file: string (optional)
The backing file string to write into the top image. This filename is not validated.

If a pathname string is such that it cannot be resolved by QEMU, that means that subsequent QMP or HMP commands must use node-names for the image in question, as filename lookup methods will fail.

If not specified, QEMU will automatically determine the backing file string to use, or error out if there is no obvious choice. Care should be taken when specifying the string, to specify a valid filename or protocol. (Since 2.1)

backing-mask-protocol: boolean (optional)
If true, replace any protocol mentioned in the 'backing file format' with 'raw', rather than storing the protocol name as the backing format. Can be used even when no image header will be updated (default false; since 9.0).

speed: int (optional)
the maximum speed, in bytes per second

on-error: BlockdevOnError (optional)
the action to take on an error (default report). 'stop' and 'enospc' can only be used if the block device supports io-status (see BlockInfo). (Since 1.3)

filter-node-name: string (optional)
the node name that should be assigned to the filter driver that the stream job inserts into the graph above device. If this option is not given, a node name is autogenerated. (Since: 6.0)

auto-finalize: boolean (optional)
When false, this job will wait in a PENDING state after it has finished its work, waiting for block-job-finalize before making any block graph changes. When true, this job will automatically perform its abort or commit actions. Defaults to true. (Since 3.1)

auto-dismiss: boolean (optional)
When false, this job will wait in a CONCLUDED state after it has completely ceased all work, and awaits block-job-dismiss. When true, this job will automatically disappear from the query list without user intervention. Defaults to true. (Since 3.1)

Errors

- If `device` does not exist, DeviceNotFound.

Since

1.1

Example

```
-> { "execute": "block-stream",
     "arguments": { "device": "virtio0",
                    "base": "/tmp/master.qcow2" } }
<- { "return": {} }
```

`block-job-set-speed` (Command)

Set maximum speed for a background block operation.

This command can only be issued when there is an active block job.

Throttling can be disabled by setting the speed to 0.

Arguments

`device`: `string`
: The job identifier. This used to be a device name (hence the name of the parameter), but since QEMU 2.7 it can have other values.

`speed`: `int`
: the maximum speed, in bytes per second, or 0 for unlimited. Defaults to 0.

Errors

- If no background operation is active on this device, DeviceNotActive

Since

1.1

block-job-cancel (Command)

Stop an active background block operation.

This command returns immediately after marking the active background block operation for cancellation. It is an error to call this command if no operation is in progress.

The operation will cancel as soon as possible and then emit the BLOCK_JOB_CANCELLED event. Before that happens the job is still visible when enumerated using query-block-jobs.

Note that if you issue 'block-job-cancel' after 'drive-mirror' has indicated (via the event BLOCK_JOB_READY) that the source and destination are synchronized, then the event triggered by this command changes to BLOCK_JOB_COMPLETED, to indicate that the mirroring has ended and the destination now has a point-in-time copy tied to the time of the cancellation.

For streaming, the image file retains its backing file unless the streaming operation happens to complete just as it is being cancelled. A new streaming operation can be started at a later time to finish copying all data from the backing file.

Arguments

device: string
: The job identifier. This used to be a device name (hence the name of the parameter), but since QEMU 2.7 it can have other values.

force: boolean (optional)
: If true, and the job has already emitted the event BLOCK_JOB_READY, abandon the job immediately (even if it is paused) instead of waiting for the destination to complete its final synchronization (since 1.3)

Errors

- If no background operation is active on this device, DeviceNotActive

Since

1.1

block-job-pause (Command)

Pause an active background block operation.

This command returns immediately after marking the active background block operation for pausing. It is an error to call this command if no operation is in progress or if the job is already paused.

The operation will pause as soon as possible. No event is emitted when the operation is actually paused. Cancelling a paused job automatically resumes it.

Arguments

device: string
The job identifier. This used to be a device name (hence the name of the parameter), but since QEMU 2.7 it can have other values.

Errors

- If no background operation is active on this device, DeviceNotActive

Since

1.3

block-job-resume (Command)

Resume an active background block operation.

This command returns immediately after resuming a paused background block operation. It is an error to call this command if no operation is in progress or if the job is not paused.

This command also clears the error status of the job.

Arguments

device: string
The job identifier. This used to be a device name (hence the name of the parameter), but since QEMU 2.7 it can have other values.

Errors

- If no background operation is active on this device, DeviceNotActive

Since

1.3

block-job-complete (Command)

Manually trigger completion of an active background block operation. This is supported for drive mirroring, where it also switches the device to write to the target path only. The ability to complete is signaled with a BLOCK_JOB_READY event.

This command completes an active background block operation synchronously. The ordering of this command's return with the BLOCK_JOB_COMPLETED event is not defined. Note that if an I/O error occurs during the processing of this command: 1) the command itself will fail; 2) the error will be processed according to the rerror/werror arguments that were specified when starting the operation.

A cancelled or paused job cannot be completed.

Arguments

device: string
The job identifier. This used to be a device name (hence the name of the parameter), but since QEMU 2.7 it can have other values.

Errors

- If no background operation is active on this device, DeviceNotActive

Since

1.3

block-job-dismiss (Command)

For jobs that have already concluded, remove them from the block-job-query list. This command only needs to be run for jobs which were started with QEMU 2.12+ job lifetime management semantics.

This command will refuse to operate on any job that has not yet reached its terminal state, JOB_STATUS_CONCLUDED. For jobs that make use of the BLOCK_JOB_READY event, block-job-cancel or block-job-complete will still need to be used as appropriate.

Arguments

id: string
The job identifier.

Since

2.12

block-job-finalize (Command)

Once a job that has manual=true reaches the pending state, it can be instructed to finalize any graph changes and do any necessary cleanup via this command. For jobs in a transaction, instructing one job to finalize will force ALL jobs in the transaction to finalize, so it is only necessary to instruct a single member job to finalize.

Arguments

id: string
The job identifier.

Since

2.12

BlockJobChangeOptionsMirror (Object)

Members

copy-mode: MirrorCopyMode
Switch to this copy mode. Currently, only the switch from 'background' to 'write-blocking' is implemented.

Since

8.2

BlockJobChangeOptions (Object)

Block job options that can be changed after job creation.

Members

id: string
The job identifier

type: JobType
The job type

The members of BlockJobChangeOptionsMirror when type is "mirror"

Since

8.2

block-job-change (Command)

Change the block job's options.

Arguments

The members of `BlockJobChangeOptions`

Since

8.2

`BlockdevDiscardOptions` (Enum)

Determines how to handle discard requests.

Values

`ignore`
: Ignore the request

`unmap`
: Forward as an unmap request

Since

2.9

`BlockdevDetectZeroesOptions` (Enum)

Describes the operation mode for the automatic conversion of plain zero writes by the OS to driver specific optimized zero write commands.

Values

`off`
: Disabled (default)

`on`
: Enabled

`unmap`
: Enabled and even try to unmap blocks if possible. This requires also that `BlockdevDiscardOptions` is set to unmap for this device.

Since

2.1

BlockdevAioOptions (Enum)

Selects the AIO backend to handle I/O requests

Values

threads
Use qemu's thread pool

native
Use native AIO backend (only Linux and Windows)

io_uring (If: CONFIG_LINUX_IO_URING)
Use linux io_uring (since 5.0)

Since

2.9

BlockdevCacheOptions (Object)

Includes cache-related options for block devices

Members

direct: boolean (optional)
enables use of O_DIRECT (bypass the host page cache; default: false)

no-flush: boolean (optional)
ignore any flush requests for the device (default: false)

Since

2.9

BlockdevDriver (Enum)

Drivers that are supported in block device operations.

Values

throttle
Since 2.11

nvme
Since 2.12

copy-on-read
Since 3.0

blklogwrites
Since 3.0

blkreplay
Since 4.2

compress
Since 5.0

copy-before-write
Since 6.2

snapshot-access
Since 7.0

blkdebug
Not documented

blkverify
Not documented

bochs
Not documented

cloop
Not documented

dmg
Not documented

file
Not documented

ftp
Not documented

ftps
Not documented

gluster
Not documented

host_cdrom (**If: HAVE_HOST_BLOCK_DEVICE**)
Not documented

host_device (**If: HAVE_HOST_BLOCK_DEVICE**)
Not documented

http
Not documented

https
Not documented

io_uring (**If:** CONFIG_BLKIO)
Not documented

iscsi
Not documented

luks
Not documented

nbd
Not documented

nfs
Not documented

null-aio
Not documented

null-co
Not documented

nvme-io_uring (**If:** CONFIG_BLKIO)
Not documented

parallels
Not documented

preallocate
Not documented

qcow
Not documented

qcow2
Not documented

qed
Not documented

quorum
Not documented

raw
Not documented

rbd
Not documented

replication (**If:** CONFIG_REPLICATION)
Not documented

ssh
Not documented

vdi
Not documented

vhdx
Not documented

virtio-blk-vfio-pci (**If:** CONFIG_BLKIO)
Not documented

virtio-blk-vhost-user (**If:** CONFIG_BLKIO)
Not documented

virtio-blk-vhost-vdpa (**If:** CONFIG_BLKIO)
Not documented

vmdk
Not documented

vpc
Not documented

vvfat
Not documented

Since

2.9

BlockdevOptionsFile (Object)

Driver specific block device options for the file backend.

Members

filename: string
path to the image file

pr-manager: string (optional)
the id for the object that will handle persistent reservations for this device (default: none, forward the commands via SG_IO; since 2.11)

aio: BlockdevAioOptions (optional)
AIO backend (default: threads) (since: 2.8)

aio-max-batch: int (optional)
maximum number of requests to batch together into a single submission in the AIO backend. The smallest value between this and the aio-max-batch value of the IOThread object is chosen. 0 means that the AIO backend will handle it automatically. (default: 0, since 6.2)

locking: OnOffAuto (optional)
whether to enable file locking. If set to 'auto', only enable when Open File Descriptor (OFD) locking API is available (default: auto, since 2.10)

drop-cache: boolean (optional) (If: CONFIG_LINUX**)**
invalidate page cache during live migration. This prevents stale data on the migration destination with cache.direct=off. Currently only supported on Linux hosts. (default: on, since: 4.0)

x-check-cache-dropped: boolean (optional)
whether to check that page cache was dropped on live migration. May cause noticeable delays if the image file is large, do not use in production. (default: off) (since: 3.0)

Features

dynamic-auto-read-only
: If present, enabled auto-read-only means that the driver will open the image read-only at first, dynamically reopen the image file read-write when the first writer is attached to the node and reopen read-only when the last writer is detached. This allows giving QEMU write permissions only on demand when an operation actually needs write access.

unstable
: Member x-check-cache-dropped is meant for debugging.

Since

2.9

BlockdevOptionsNull (Object)

Driver specific block device options for the null backend.

Members

size: int (optional)
: size of the device in bytes.

latency-ns: int (optional)
: emulated latency (in nanoseconds) in processing requests. Default to zero which completes requests immediately. (Since 2.4)

read-zeroes: boolean (optional)
: if true, reads from the device produce zeroes; if false, the buffer is left unchanged. (default: false; since: 4.1)

Since

2.9

BlockdevOptionsNVMe (Object)

Driver specific block device options for the NVMe backend.

Members

device: string
: PCI controller address of the NVMe device in format hhhh:bb:ss.f (host:bus:slot.function)

namespace: int
: namespace number of the device, starting from 1.

Note that the PCI `device` must have been unbound from any host kernel driver before instructing QEMU to add the blockdev.

Since

2.12

BlockdevOptionsVVFAT (Object)

Driver specific block device options for the vvfat protocol.

Members

dir: string
directory to be exported as FAT image

fat-type: int (optional)
FAT type: 12, 16 or 32

floppy: boolean (optional)
whether to export a floppy image (true) or partitioned hard disk (false; default)

label: string (optional)
set the volume label, limited to 11 bytes. FAT16 and FAT32 traditionally have some restrictions on labels, which are ignored by most operating systems. Defaults to "QEMU VVFAT". (since 2.4)

rw: boolean (optional)
whether to allow write operations (default: false)

Since

2.9

BlockdevOptionsGenericFormat (Object)

Driver specific block device options for image format that have no option besides their data source.

Members

file: BlockdevRef
reference to or definition of the data source block device

Since

2.9

BlockdevOptionsLUKS (Object)

Driver specific block device options for LUKS.

Members

key-secret: string (optional)
the ID of a QCryptoSecret object providing the decryption key (since 2.6). Mandatory except when doing a metadata-only probe of the image.

header: BlockdevRef (optional)
block device holding a detached LUKS header. (since 9.0)

The members of BlockdevOptionsGenericFormat

Since

2.9

BlockdevOptionsGenericCOWFormat (Object)

Driver specific block device options for image format that have no option besides their data source and an optional backing file.

Members

backing: BlockdevRefOrNull (optional)
reference to or definition of the backing file block device, null disables the backing file entirely. Defaults to the backing file stored the image file.

The members of BlockdevOptionsGenericFormat

Since

2.9

Qcow2OverlapCheckMode (Enum)

General overlap check modes.

Values

none
Do not perform any checks

constant
Perform only checks which can be done in constant time and without reading anything from disk

cached
Perform only checks which can be done without reading anything from disk

all
Perform all available overlap checks

Since

2.9

Qcow2OverlapCheckFlags (Object)

Structure of flags for each metadata structure. Setting a field to 'true' makes QEMU guard that Qcow2 format structure against unintended overwriting. See Qcow2 format specification for detailed information on these structures. The default value is chosen according to the template given.

Members

template: Qcow2OverlapCheckMode (optional)
Specifies a template mode which can be adjusted using the other flags, defaults to 'cached'

main-header: boolean (optional)
Qcow2 format header

active-l1: boolean (optional)
Qcow2 active L1 table

active-l2: boolean (optional)
Qcow2 active L2 table

refcount-table: boolean (optional)
Qcow2 refcount table

refcount-block: boolean (optional)
Qcow2 refcount blocks

snapshot-table: boolean (optional)
Qcow2 snapshot table

inactive-l1: boolean (optional)
Qcow2 inactive L1 tables

inactive-l2: boolean (optional)
Qcow2 inactive L2 tables

bitmap-directory: boolean (optional)
Qcow2 bitmap directory (since 3.0)

Since

2.9

Qcow2OverlapChecks (Alternate)

Specifies which metadata structures should be guarded against unintended overwriting.

Members

flags: Qcow2OverlapCheckFlags
set of flags for separate specification of each metadata structure type

mode: Qcow2OverlapCheckMode
named mode which chooses a specific set of flags

Since

2.9

BlockdevQcowEncryptionFormat (Enum)

Values

aes
AES-CBC with plain64 initialization vectors

Since

2.10

BlockdevQcowEncryption (Object)

Members

format: BlockdevQcowEncryptionFormat
encryption format

The members of QCryptoBlockOptionsQCow when format is "aes"

Since

2.10

BlockdevOptionsQcow (Object)

Driver specific block device options for qcow.

Members

encrypt: BlockdevQcowEncryption (optional)
Image decryption options. Mandatory for encrypted images, except when doing a metadata-only probe of the image.

The members of BlockdevOptionsGenericCOWFormat

Since

2.10

BlockdevQcow2EncryptionFormat (Enum)

Values

aes
AES-CBC with plain64 initialization vectors

luks
Not documented

Since

2.10

BlockdevQcow2Encryption (Object)

Members

format: BlockdevQcow2EncryptionFormat
encryption format

The members of QCryptoBlockOptionsQCow when format is "aes"
The members of QCryptoBlockOptionsLUKS when format is "luks"

Since

2.10

BlockdevOptionsPreallocate (Object)

Filter driver intended to be inserted between format and protocol node and do preallocation in protocol node on write.

Members

prealloc-align: int (optional)
on preallocation, align file length to this number, default 1048576 (1M)

prealloc-size: int (optional)
how much to preallocate, default 134217728 (128M)

The members of BlockdevOptionsGenericFormat

Since

6.0

BlockdevOptionsQcow2 (Object)

Driver specific block device options for qcow2.

Members

lazy-refcounts: boolean (optional)
whether to enable the lazy refcounts feature (default is taken from the image file)

pass-discard-request: boolean (optional)
whether discard requests to the qcow2 device should be forwarded to the data source

pass-discard-snapshot: boolean (optional)
whether discard requests for the data source should be issued when a snapshot operation (e.g. deleting a snapshot) frees clusters in the qcow2 file

pass-discard-other: boolean (optional)
whether discard requests for the data source should be issued on other occasions where a cluster gets freed

discard-no-unref: boolean (optional)
when enabled, data clusters will remain preallocated when they are no longer used, e.g. because they are discarded or converted to zero clusters. As usual, whether the old data is discarded or kept on the protocol level (i.e. in the image file) depends on the setting of the pass-discard-request option. Keeping the clusters preallocated prevents qcow2 fragmentation that would otherwise be caused by freeing and re-allocating them later. Besides potential performance degradation, such fragmentation can lead to increased allocation of clusters past the end of the image file, resulting in image files whose file length can grow much larger than their guest disk size would suggest. If image file length is of concern (e.g. when storing qcow2 images directly on block devices), you should consider enabling this option. (since 8.1)

overlap-check: Qcow2OverlapChecks (optional)
which overlap checks to perform for writes to the image, defaults to 'cached' (since 2.2)

cache-size: int (optional)
the maximum total size of the L2 table and refcount block caches in bytes (since 2.2)

l2-cache-size: int (optional)
the maximum size of the L2 table cache in bytes (since 2.2)

l2-cache-entry-size: int (optional)
the size of each entry in the L2 cache in bytes. It must be a power of two between 512 and the cluster size. The default value is the cluster size (since 2.12)

refcount-cache-size: int (optional)
the maximum size of the refcount block cache in bytes (since 2.2)

cache-clean-interval: int (optional)
clean unused entries in the L2 and refcount caches. The interval is in seconds. The default value is 600 on supporting platforms, and 0 on other platforms. 0 disables this feature. (since 2.5)

encrypt: BlockdevQcow2Encryption (optional)
Image decryption options. Mandatory for encrypted images, except when doing a metadata-only probe of the image. (since 2.10)

data-file: BlockdevRef (optional)
reference to or definition of the external data file. This may only be specified for images that require an external data file. If it is not specified for such an image, the data file name is loaded from the image file. (since 4.0)

The members of BlockdevOptionsGenericCOWFormat

Since

2.9

SshHostKeyCheckMode (Enum)

Values

none
Don't check the host key at all

hash
Compare the host key with a given hash

known_hosts
Check the host key against the known_hosts file

Since

2.12

SshHostKeyCheckHashType (Enum)

Values

md5
The given hash is an md5 hash

sha1
The given hash is an sha1 hash

sha256
The given hash is an sha256 hash

Since

2.12

SshHostKeyHash (Object)

Members

type: SshHostKeyCheckHashType
The hash algorithm used for the hash

hash: string
The expected hash value

Since

2.12

SshHostKeyCheck (Object)

Members

mode: SshHostKeyCheckMode
How to check the host key

The members of SshHostKeyHash when mode is "hash"

Since

2.12

BlockdevOptionsSsh (Object)

Members

server: InetSocketAddress
host address

path: string
path to the image on the host

user: string (optional)
user as which to connect, defaults to current local user name

host-key-check: SshHostKeyCheck (optional)
Defines how and what to check the host key against (default: known_hosts)

Since

2.9

BlkdebugEvent (Enum)

Trigger events supported by blkdebug.

Values

l1_shrink_write_table
write zeros to the l1 table to shrink image. (since 2.11)

l1_shrink_free_l2_clusters
discard the l2 tables. (since 2.11)

cor_write
a write due to copy-on-read (since 2.11)

cluster_alloc_space
an allocation of file space for a cluster (since 4.1)

none
triggers once at creation of the blkdebug node (since 4.1)

l1_update
Not documented

l1_grow_alloc_table
Not documented

l1_grow_write_table
Not documented

l1_grow_activate_table
: Not documented

l2_load
: Not documented

l2_update
: Not documented

l2_update_compressed
: Not documented

l2_alloc_cow_read
: Not documented

l2_alloc_write
: Not documented

read_aio
: Not documented

read_backing_aio
: Not documented

read_compressed
: Not documented

write_aio
: Not documented

write_compressed
: Not documented

vmstate_load
: Not documented

vmstate_save
: Not documented

cow_read
: Not documented

cow_write
: Not documented

reftable_load
: Not documented

reftable_grow
: Not documented

reftable_update
: Not documented

refblock_load
: Not documented

refblock_update
: Not documented

refblock_update_part
: Not documented

refblock_alloc
: Not documented

refblock_alloc_hookup
: Not documented

refblock_alloc_write
: Not documented

refblock_alloc_write_blocks
: Not documented

refblock_alloc_write_table
: Not documented

refblock_alloc_switch_table
: Not documented

cluster_alloc
: Not documented

cluster_alloc_bytes
: Not documented

cluster_free
: Not documented

flush_to_os
: Not documented

flush_to_disk
: Not documented

pwritev_rmw_head
: Not documented

pwritev_rmw_after_head
: Not documented

pwritev_rmw_tail
: Not documented

pwritev_rmw_after_tail
: Not documented

pwritev
: Not documented

pwritev_zero
: Not documented

pwritev_done
: Not documented

empty_image_prepare
: Not documented

Since

2.9

BlkdebugIOType (Enum)

Kinds of I/O that blkdebug can inject errors in.

Values

read
: .bdrv_co_preadv()

write
: .bdrv_co_pwritev()

write-zeroes
: .bdrv_co_pwrite_zeroes()

discard
: .bdrv_co_pdiscard()

flush
: .bdrv_co_flush_to_disk()

block-status
: .bdrv_co_block_status()

Since

4.1

BlkdebugInjectErrorOptions (Object)

Describes a single error injection for blkdebug.

Members

event: BlkdebugEvent
: trigger event

state: int (optional)
: the state identifier blkdebug needs to be in to actually trigger the event; defaults to "any"

iotype: BlkdebugIOType (optional)
: the type of I/O operations on which this error should be injected; defaults to "all read, write, write-zeroes, discard, and flush operations" (since: 4.1)

errno: int (optional)
: error identifier (errno) to be returned; defaults to EIO

sector: int (optional)
: specifies the sector index which has to be affected in order to actually trigger the event; defaults to "any sector"

once: boolean (optional)
disables further events after this one has been triggered; defaults to false

immediately: boolean (optional)
fail immediately; defaults to false

Since

2.9

BlkdebugSetStateOptions (Object)

Describes a single state-change event for blkdebug.

Members

event: BlkdebugEvent
trigger event

state: int (optional)
the current state identifier blkdebug needs to be in; defaults to "any"

new_state: int
the state identifier blkdebug is supposed to assume if this event is triggered

Since

2.9

BlockdevOptionsBlkdebug (Object)

Driver specific block device options for blkdebug.

Members

image: BlockdevRef
underlying raw block device (or image file)

config: string (optional)
filename of the configuration file

align: int (optional)
required alignment for requests in bytes, must be positive power of 2, or 0 for default

max-transfer: int (optional)
maximum size for I/O transfers in bytes, must be positive multiple of `align` and of the underlying file's request alignment (but need not be a power of 2), or 0 for default (since 2.10)

opt-write-zero: int (optional)
preferred alignment for write zero requests in bytes, must be positive multiple of `align` and of the underlying file's request alignment (but need not be a power of 2), or 0 for default (since 2.10)

max-write-zero: int (optional)
maximum size for write zero requests in bytes, must be positive multiple of `align`, of `opt-write-zero`, and of the underlying file's request alignment (but need not be a power of 2), or 0 for default (since 2.10)

opt-discard: int (optional)
preferred alignment for discard requests in bytes, must be positive multiple of `align` and of the underlying file's request alignment (but need not be a power of 2), or 0 for default (since 2.10)

max-discard: int (optional)
maximum size for discard requests in bytes, must be positive multiple of `align`, of `opt-discard`, and of the underlying file's request alignment (but need not be a power of 2), or 0 for default (since 2.10)

inject-error: array of BlkdebugInjectErrorOptions (optional)
array of error injection descriptions

set-state: array of BlkdebugSetStateOptions (optional)
array of state-change descriptions

take-child-perms: array of BlockPermission (optional)
Permissions to take on `image` in addition to what is necessary anyway (which depends on how the blkdebug node is used). Defaults to none. (since 5.0)

unshare-child-perms: array of BlockPermission (optional)
Permissions not to share on `image` in addition to what cannot be shared anyway (which depends on how the blkdebug node is used). Defaults to none. (since 5.0)

Since

2.9

BlockdevOptionsBlklogwrites (Object)

Driver specific block device options for blklogwrites.

Members

file: BlockdevRef
block device

log: BlockdevRef
block device used to log writes to `file`

log-sector-size: int (optional)
sector size used in logging writes to `file`, determines granularity of offsets and sizes of writes (default: 512)

log-append: boolean (optional)
append to an existing log (default: false)

log-super-update-interval: int (optional)
interval of write requests after which the log super block is updated to disk (default: 4096)

Since

3.0

BlockdevOptionsBlkverify (Object)

Driver specific block device options for blkverify.

Members

test: BlockdevRef
block device to be tested

raw: BlockdevRef
raw image used for verification

Since

2.9

BlockdevOptionsBlkreplay (Object)

Driver specific block device options for blkreplay.

Members

image: BlockdevRef
disk image which should be controlled with blkreplay

Since

4.2

QuorumReadPattern (Enum)

An enumeration of quorum read patterns.

Values

quorum
read all the children and do a quorum vote on reads

fifo
read only from the first child that has not failed

Since

2.9

BlockdevOptionsQuorum (Object)

Driver specific block device options for Quorum

Members

blkverify: boolean (optional)
true if the driver must print content mismatch set to false by default

children: array of BlockdevRef
the children block devices to use

vote-threshold: int
the vote limit under which a read will fail

rewrite-corrupted: boolean (optional)
rewrite corrupted data when quorum is reached (Since 2.1)

read-pattern: QuorumReadPattern (optional)
choose read pattern and set to quorum by default (Since 2.2)

Since

2.9

BlockdevOptionsGluster (Object)

Driver specific block device options for Gluster

Members

volume: string
name of gluster volume where VM image resides

path: string
absolute path to image file in gluster volume

server: array of SocketAddress
gluster servers description

debug: int (optional)
libgfapi log level (default '4' which is Error) (Since 2.8)

logfile: string (optional)
libgfapi log file (default /dev/stderr) (Since 2.8)

Since

2.9

BlockdevOptionsIoUring (Object)

Driver specific block device options for the io_uring backend.

Members

filename: string
path to the image file

Since

7.2

If

CONFIG_BLKIO

BlockdevOptionsNvmeIoUring (Object)

Driver specific block device options for the nvme-io_uring backend.

Members

path: string
path to the NVMe namespace's character device (e.g. /dev/ng0n1).

Since

7.2

If

CONFIG_BLKIO

BlockdevOptionsVirtioBlkVfioPci (Object)

Driver specific block device options for the virtio-blk-vfio-pci backend.

Members

path: string
path to the PCI device's sysfs directory (e.g. /sys/bus/pci/devices/0000:00:01.0).

Since

7.2

If

CONFIG_BLKIO

BlockdevOptionsVirtioBlkVhostUser (Object)

Driver specific block device options for the virtio-blk-vhost-user backend.

Members

path: string
path to the vhost-user UNIX domain socket.

Since

7.2

If

CONFIG_BLKIO

BlockdevOptionsVirtioBlkVhostVdpa (Object)

Driver specific block device options for the virtio-blk-vhost-vdpa backend.

Members

path: string
path to the vhost-vdpa character device.

Features

fdset
Member path supports the special "/dev/fdset/N" path (since 8.1)

Since

7.2

If

CONFIG_BLKIO

IscsiTransport (Enum)

An enumeration of libiscsi transport types

Values

tcp
Not documented

iser
Not documented

Since

2.9

IscsiHeaderDigest (Enum)

An enumeration of header digests supported by libiscsi

Values

crc32c
Not documented

none
Not documented

crc32c-none
Not documented

none-crc32c
Not documented

Since

2.9

BlockdevOptionsIscsi (Object)

Driver specific block device options for iscsi

Members

transport: IscsiTransport
The iscsi transport type

portal: string
The address of the iscsi portal

target: string
The target iqn name

lun: int (optional)
LUN to connect to. Defaults to 0.

user: string (optional)
User name to log in with. If omitted, no CHAP authentication is performed.

password-secret: string (optional)
The ID of a QCryptoSecret object providing the password for the login. This option is required if `user` is specified.

initiator-name: string (optional)
The iqn name we want to identify to the target as. If this option is not specified, an initiator name is generated automatically.

header-digest: IscsiHeaderDigest (optional)
The desired header digest. Defaults to none-crc32c.

timeout: int (optional)
Timeout in seconds after which a request will timeout. 0 means no timeout and is the default.

Since

2.9

RbdAuthMode (Enum)

Values

cephx
Not documented

none
Not documented

Since

3.0

RbdImageEncryptionFormat (Enum)

Values

luks-any
Used for opening either luks or luks2 (Since 8.0)

luks
Not documented

luks2
Not documented

Since

6.1

RbdEncryptionOptionsLUKSBase (Object)

Members

key-secret: string
ID of a QCryptoSecret object providing a passphrase for unlocking the encryption

Since

6.1

RbdEncryptionCreateOptionsLUKSBase (Object)

Members

cipher-alg: QCryptoCipherAlgorithm (optional)
The encryption algorithm

The members of RbdEncryptionOptionsLUKSBase

Since

6.1

RbdEncryptionOptionsLUKS (Object)

Members

The members of RbdEncryptionOptionsLUKSBase

Since

6.1

RbdEncryptionOptionsLUKS2 (Object)

Members

The members of RbdEncryptionOptionsLUKSBase

Since

6.1

RbdEncryptionOptionsLUKSAny (Object)

Members

The members of RbdEncryptionOptionsLUKSBase

Since

8.0

RbdEncryptionCreateOptionsLUKS (Object)

Members

The members of RbdEncryptionCreateOptionsLUKSBase

Since

6.1

RbdEncryptionCreateOptionsLUKS2 (Object)

Members

The members of RbdEncryptionCreateOptionsLUKSBase

Since

6.1

RbdEncryptionOptions (Object)

Members

format: RbdImageEncryptionFormat
Encryption format.

parent: RbdEncryptionOptions (optional)
Parent image encryption options (for cloned images). Can be left unspecified if this cloned image is encrypted using the same format and secret as its parent image (i.e. not explicitly formatted) or if its parent image is not encrypted. (Since 8.0)

The members of RbdEncryptionOptionsLUKS when format is "luks"
The members of RbdEncryptionOptionsLUKS2 when format is "luks2"
The members of RbdEncryptionOptionsLUKSAny when format is "luks-any"

Since

6.1

RbdEncryptionCreateOptions (Object)

Members

format: RbdImageEncryptionFormat
Encryption format.

The members of RbdEncryptionCreateOptionsLUKS when format is "luks"
The members of RbdEncryptionCreateOptionsLUKS2 when format is "luks2"

Since

6.1

BlockdevOptionsRbd (Object)

Members

pool: string
Ceph pool name.

namespace: string (optional)
Rados namespace name in the Ceph pool. (Since 5.0)

image: string
Image name in the Ceph pool.

conf: string (optional)
path to Ceph configuration file. Values in the configuration file will be overridden by options specified via QAPI.

snapshot: string (optional)
Ceph snapshot name.

encrypt: RbdEncryptionOptions (optional)
Image encryption options. (Since 6.1)

user: string (optional)
Ceph id name.

auth-client-required: array of RbdAuthMode (optional)
Acceptable authentication modes. This maps to Ceph configuration option "auth_client_required". (Since 3.0)

key-secret: string (optional)
ID of a QCryptoSecret object providing a key for cephx authentication. This maps to Ceph configuration option "key". (Since 3.0)

server: array of InetSocketAddressBase (optional)
Monitor host address and port. This maps to the "mon_host" Ceph option.

Since

2.9

ReplicationMode (Enum)

An enumeration of replication modes.

Values

primary
Primary mode, the vm's state will be sent to secondary QEMU.

secondary
Secondary mode, receive the vm's state from primary QEMU.

Since

2.9

If

CONFIG_REPLICATION

BlockdevOptionsReplication (Object)

Driver specific block device options for replication

Members

mode: ReplicationMode
the replication mode

top-id: string (optional)
In secondary mode, node name or device ID of the root node who owns the replication node chain. Must not be given in primary mode.

The members of BlockdevOptionsGenericFormat

Since

2.9

If

`CONFIG_REPLICATION`

`NFSTransport` (Enum)

An enumeration of NFS transport types

Values

`inet`
: TCP transport

Since

2.9

`NFSServer` (Object)

Captures the address of the socket

Members

`type: NFSTransport`
: transport type used for NFS (only TCP supported)

`host: string`
: host address for NFS server

Since

2.9

`BlockdevOptionsNfs` (Object)

Driver specific block device option for NFS

Members

server: NFSServer
host address

path: string
path of the image on the host

user: int (optional)
UID value to use when talking to the server (defaults to 65534 on Windows and getuid() on unix)

group: int (optional)
GID value to use when talking to the server (defaults to 65534 on Windows and getgid() in unix)

tcp-syn-count: int (optional)
number of SYNs during the session establishment (defaults to libnfs default)

readahead-size: int (optional)
set the readahead size in bytes (defaults to libnfs default)

page-cache-size: int (optional)
set the pagecache size in bytes (defaults to libnfs default)

debug: int (optional)
set the NFS debug level (max 2) (defaults to libnfs default)

Since

2.9

BlockdevOptionsCurlBase (Object)

Driver specific block device options shared by all protocols supported by the curl backend.

Members

url: string
URL of the image file

readahead: int (optional)
Size of the read-ahead cache; must be a multiple of 512 (defaults to 256 kB)

timeout: int (optional)
Timeout for connections, in seconds (defaults to 5)

username: string (optional)
Username for authentication (defaults to none)

password-secret: string (optional)
ID of a QCryptoSecret object providing a password for authentication (defaults to no password)

proxy-username: string (optional)
Username for proxy authentication (defaults to none)

proxy-password-secret: string (optional)
ID of a QCryptoSecret object providing a password for proxy authentication (defaults to no password)

Since

2.9

BlockdevOptionsCurlHttp (Object)

Driver specific block device options for HTTP connections over the curl backend. URLs must start with "http://".

Members

cookie: string (optional)
List of cookies to set; format is "name1=content1; name2=content2;" as explained by CURLOPT_COOKIE(3). Defaults to no cookies.

cookie-secret: string (optional)
ID of a QCryptoSecret object providing the cookie data in a secure way. See `cookie` for the format. (since 2.10)

The members of BlockdevOptionsCurlBase

Since

2.9

BlockdevOptionsCurlHttps (Object)

Driver specific block device options for HTTPS connections over the curl backend. URLs must start with "https://".

Members

cookie: string (optional)
List of cookies to set; format is "name1=content1; name2=content2;" as explained by CURLOPT_COOKIE(3). Defaults to no cookies.

sslverify: boolean (optional)
Whether to verify the SSL certificate's validity (defaults to true)

cookie-secret: string (optional)
ID of a QCryptoSecret object providing the cookie data in a secure way. See `cookie` for the format. (since 2.10)

The members of BlockdevOptionsCurlBase

Since

2.9

BlockdevOptionsCurlFtp (Object)

Driver specific block device options for FTP connections over the curl backend. URLs must start with "ftp://".

Members

The members of BlockdevOptionsCurlBase

Since

2.9

BlockdevOptionsCurlFtps (Object)

Driver specific block device options for FTPS connections over the curl backend. URLs must start with "ftps://".

Members

sslverify: boolean (optional)
Whether to verify the SSL certificate's validity (defaults to true)

The members of BlockdevOptionsCurlBase

Since

2.9

BlockdevOptionsNbd (Object)

Driver specific block device options for NBD.

Members

server: SocketAddress
NBD server address

export: string (optional)
export name

tls-creds: string (optional)
TLS credentials ID

tls-hostname: string (optional)
TLS hostname override for certificate validation (Since 7.0)

x-dirty-bitmap: string (optional)
A metadata context name such as "qemu:dirty-bitmap:NAME" or "qemu:allocation-depth" to query in place of the traditional "base:allocation" block status (see NBD_OPT_LIST_META_CONTEXT in the NBD protocol; and yes, naming this option x-context would have made more sense) (since 3.0)

reconnect-delay: int (optional)
On an unexpected disconnect, the nbd client tries to connect again until succeeding or encountering a serious error. During the first `reconnect-delay` seconds, all requests are paused and will be rerun on a successful reconnect. After that time, any delayed requests and all future requests before a successful reconnect will immediately fail. Default 0 (Since 4.2)

open-timeout: int (optional)
In seconds. If zero, the nbd driver tries the connection only once, and fails to open if the connection fails. If non-zero, the nbd driver will repeat connection attempts until successful or until `open-timeout` seconds have elapsed. Default 0 (Since 7.0)

Features

unstable
Member `x-dirty-bitmap` is experimental.

Since

2.9

BlockdevOptionsRaw (Object)

Driver specific block device options for the raw driver.

Members

offset: int (optional)
position where the block device starts

size: int (optional)
the assumed size of the device

The members of BlockdevOptionsGenericFormat

Since

2.9

BlockdevOptionsThrottle (Object)

Driver specific block device options for the throttle driver

Members

throttle-group: string
the name of the throttle-group object to use. It must already exist.

file: BlockdevRef
reference to or definition of the data source block device

Since

2.11

BlockdevOptionsCor (Object)

Driver specific block device options for the copy-on-read driver.

Members

bottom: string (optional)
The name of a non-filter node (allocation-bearing layer) that limits the COR operations in the backing chain (inclusive), so that no data below this node will be copied by this filter. If option is absent, the limit is not applied, so that data from all backing layers may be copied.

The members of BlockdevOptionsGenericFormat

Since

6.0

OnCbwError (Enum)

An enumeration of possible behaviors for copy-before-write operation failures.

Values

break-guest-write
report the error to the guest. This way, the guest will not be able to overwrite areas that cannot be backed up, so the backup process remains valid.

break-snapshot
continue guest write. Doing so will make the provided snapshot state invalid and any backup or export process based on it will finally fail.

Since

7.1

BlockdevOptionsCbw (Object)

Driver specific block device options for the copy-before-write driver, which does so called copy-before-write operations: when data is written to the filter, the filter first reads corresponding blocks from its file child and copies them to `target` child. After successfully copying, the write request is propagated to file child. If copying fails, the original write request is failed too and no data is written to file child.

Members

target: BlockdevRef
The target for copy-before-write operations.

bitmap: BlockDirtyBitmap (optional)
If specified, copy-before-write filter will do copy-before-write operations only for dirty regions of the bitmap. Bitmap size must be equal to length of file and target child of the filter. Note also, that bitmap is used only to initialize internal bitmap of the process, so further modifications (or removing) of specified bitmap doesn't influence the filter. (Since 7.0)

on-cbw-error: OnCbwError (optional)
Behavior on failure of copy-before-write operation. Default is `break-guest-write`. (Since 7.1)

cbw-timeout: int (optional)
Zero means no limit. Non-zero sets the timeout in seconds for copy-before-write operation. When a timeout occurs, the respective copy-before-write operation will fail, and the `on-cbw-error` parameter will decide how this failure is handled. Default 0. (Since 7.1)

The members of BlockdevOptionsGenericFormat

Since

6.2

BlockdevOptions (Object)

Options for creating a block device. Many options are available for all block devices, independent of the block driver:

Members

driver: BlockdevDriver
block driver name

node-name: string (optional)
the node name of the new node (Since 2.0). This option is required on the top level of blockdev-add. Valid node names start with an alphabetic character and may contain only alphanumeric characters, '-', '.' and '_'. Their maximum length is 31 characters.

discard: BlockdevDiscardOptions (optional)
discard-related options (default: ignore)

cache: BlockdevCacheOptions (optional)
cache-related options

read-only: boolean (optional)
whether the block device should be read-only (default: false). Note that some block drivers support only read-only access, either generally or in certain configurations. In this case, the default value does not work and the option must be specified explicitly.

auto-read-only: boolean (optional)
if true and `read-only` is false, QEMU may automatically decide not to open the image read-write as requested, but fall back to read-only instead (and switch between the modes later), e.g. depending on whether the image file is writable or whether a writing user is attached to the node (default: false, since 3.1)

detect-zeroes: BlockdevDetectZeroesOptions (optional)
detect and optimize zero writes (Since 2.1) (default: off)

force-share: boolean (optional)
force share all permission on added nodes. Requires read-only=true. (Since 2.10)

The members of BlockdevOptionsBlkdebug when driver is "blkdebug"
The members of BlockdevOptionsBlklogwrites when driver is "blklogwrites"
The members of BlockdevOptionsBlkverify when driver is "blkverify"
The members of BlockdevOptionsBlkreplay when driver is "blkreplay"
The members of BlockdevOptionsGenericFormat when driver is "bochs"
The members of BlockdevOptionsGenericFormat when driver is "cloop"
The members of BlockdevOptionsGenericFormat when driver is "compress"
The members of BlockdevOptionsCbw when driver is "copy-before-write"
The members of BlockdevOptionsCor when driver is "copy-on-read"
The members of BlockdevOptionsGenericFormat when driver is "dmg"
The members of BlockdevOptionsFile when driver is "file"
The members of BlockdevOptionsCurlFtp when driver is "ftp"
The members of BlockdevOptionsCurlFtps when driver is "ftps"
The members of BlockdevOptionsGluster when driver is "gluster"
The members of BlockdevOptionsFile when driver is "host_cdrom" (If: HAVE_HOST_BLOCK_DEVICE)
The members of BlockdevOptionsFile when driver is "host_device" (If: HAVE_HOST_BLOCK_DEVICE)
The members of BlockdevOptionsCurlHttp when driver is "http"
The members of BlockdevOptionsCurlHttps when driver is "https"
The members of BlockdevOptionsIoUring when driver is "io_uring" (If: CONFIG_BLKIO)
The members of BlockdevOptionsIscsi when driver is "iscsi"
The members of BlockdevOptionsLUKS when driver is "luks"
The members of BlockdevOptionsNbd when driver is "nbd"
The members of BlockdevOptionsNfs when driver is "nfs"
The members of BlockdevOptionsNull when driver is "null-aio"
The members of BlockdevOptionsNull when driver is "null-co"
The members of BlockdevOptionsNVMe when driver is "nvme"
The members of BlockdevOptionsNvmeIoUring when driver is "nvme-io_uring" (If: CONFIG_BLKIO)
The members of BlockdevOptionsGenericFormat when driver is "parallels"
The members of BlockdevOptionsPreallocate when driver is "preallocate"
The members of BlockdevOptionsQcow2 when driver is "qcow2"
The members of BlockdevOptionsQcow when driver is "qcow"
The members of BlockdevOptionsGenericCOWFormat when driver is "qed"
The members of BlockdevOptionsQuorum when driver is "quorum"
The members of BlockdevOptionsRaw when driver is "raw"
The members of BlockdevOptionsRbd when driver is "rbd"
The members of BlockdevOptionsReplication when driver is "replication" (If: CONFIG_REPLICATION)
The members of BlockdevOptionsGenericFormat when driver is "snapshot-access"
The members of BlockdevOptionsSsh when driver is "ssh"
The members of BlockdevOptionsThrottle when driver is "throttle"
The members of BlockdevOptionsGenericFormat when driver is "vdi"
The members of BlockdevOptionsGenericFormat when driver is "vhdx"
The members of BlockdevOptionsVirtioBlkVfioPci when driver is "virtio-blk-vfio-pci" (If: CONFIG_BLKIO)
The members of BlockdevOptionsVirtioBlkVhostUser when driver is "virtio-blk-vhost-user" (If: CONFIG_BLKIO)
The members of BlockdevOptionsVirtioBlkVhostVdpa when driver is "virtio-blk-vhost-vdpa" (If: CONFIG_BLKIO)
The members of BlockdevOptionsGenericCOWFormat when driver is "vmdk"
The members of BlockdevOptionsGenericFormat when driver is "vpc"
The members of BlockdevOptionsVVFAT when driver is "vvfat"

Since

2.9

BlockdevRef (Alternate)

Reference to a block device.

Members

definition: BlockdevOptions
defines a new block device inline

reference: string
references the ID of an existing block device

Since

2.9

BlockdevRefOrNull (Alternate)

Reference to a block device.

Members

definition: BlockdevOptions
defines a new block device inline

reference: string
references the ID of an existing block device. An empty string means that no block device should be referenced. Deprecated; use null instead.

null: null
No block device should be referenced (since 2.10)

Since

2.9

blockdev-add (Command)

Creates a new block device.

Arguments

The members of BlockdevOptions

Since

2.9

Examples

```
-> { "execute": "blockdev-add",
     "arguments": {
          "driver": "qcow2",
          "node-name": "test1",
          "file": {
              "driver": "file",
              "filename": "test.qcow2"
           }
      }
    }
<- { "return": {} }

-> { "execute": "blockdev-add",
     "arguments": {
          "driver": "qcow2",
          "node-name": "node0",
          "discard": "unmap",
          "cache": {
             "direct": true
           },
           "file": {
             "driver": "file",
             "filename": "/tmp/test.qcow2"
           },
           "backing": {
              "driver": "raw",
              "file": {
                 "driver": "file",
                 "filename": "/dev/fdset/4"
               }
           }
       }
     }

<- { "return": {} }
```

blockdev-reopen (Command)

Reopens one or more block devices using the given set of options. Any option not specified will be reset to its default value regardless of its previous status. If an option cannot be changed or a particular driver does not support reopening then the command will return an error. All devices in the list are reopened in one transaction, so if one of them fails then the whole transaction is cancelled.

The command receives a list of block devices to reopen. For each one of them, the top-level node-name option (from BlockdevOptions) must be specified and is used to select the block device to be reopened. Other node-name options must be either omitted or set to the current name of the appropriate node. This command won't change any node name and any attempt to do it will result in an error.

In the case of options that refer to child nodes, the behavior of this command depends on the value:

1) A set of options (BlockdevOptions): the child is reopened with the specified set of options.

2) A reference to the current child: the child is reopened using its existing set of options.

3) A reference to a different node: the current child is replaced with the specified one.

4) NULL: the current child (if any) is detached.

Options (1) and (2) are supported in all cases. Option (3) is supported for file and backing, and option (4) for backing only.

Unlike with blockdev-add, the backing option must always be present unless the node being reopened does not have a backing file and its image does not have a default backing file name as part of its metadata.

Arguments

options: array of BlockdevOptions
Not documented

Since

6.1

blockdev-del (Command)

Deletes a block device that has been added using blockdev-add. The command will fail if the node is attached to a device or is otherwise being used.

Arguments

node-name: string
Name of the graph node to delete.

Since

2.9

Example

```
-> { "execute": "blockdev-add",
     "arguments": {
          "driver": "qcow2",
          "node-name": "node0",
          "file": {
              "driver": "file",
              "filename": "test.qcow2"
          }
     }
   }
<- { "return": {} }

-> { "execute": "blockdev-del",
     "arguments": { "node-name": "node0" }
   }
<- { "return": {} }
```

BlockdevCreateOptionsFile (Object)

Driver specific image creation options for file.

Members

filename: string
Filename for the new image file

size: int
Size of the virtual disk in bytes

preallocation: PreallocMode (optional)
Preallocation mode for the new image (default: off; allowed values: off, falloc (if CONFIG_POSIX_FALLOCATE), full (if CONFIG_POSIX))

nocow: boolean (optional)
Turn off copy-on-write (valid only on btrfs; default: off)

extent-size-hint: int (optional)
Extent size hint to add to the image file; 0 for not adding an extent size hint (default: 1 MB, since 5.1)

Since

2.12

BlockdevCreateOptionsGluster (Object)

Driver specific image creation options for gluster.

Members

location: BlockdevOptionsGluster
Where to store the new image file

size: int
Size of the virtual disk in bytes

preallocation: PreallocMode (optional)
Preallocation mode for the new image (default: off; allowed values: off, falloc (if CONFIG_GLUSTERFS_FALLOCATE), full (if CONFIG_GLUSTERFS_ZEROFILL))

Since

2.12

BlockdevCreateOptionsLUKS (Object)

Driver specific image creation options for LUKS.

Members

file: BlockdevRef (optional)
Node to create the image format on, mandatory except when 'preallocation' is not requested

header: BlockdevRef (optional)
Block device holding a detached LUKS header. (since 9.0)

size: int
Size of the virtual disk in bytes

preallocation: PreallocMode (optional)
Preallocation mode for the new image (since: 4.2) (default: off; allowed values: off, metadata, falloc, full)

The members of QCryptoBlockCreateOptionsLUKS

Since

2.12

BlockdevCreateOptionsNfs (Object)

Driver specific image creation options for NFS.

Members

location: BlockdevOptionsNfs
Where to store the new image file

size: int
Size of the virtual disk in bytes

Since

2.12

BlockdevCreateOptionsParallels (Object)

Driver specific image creation options for parallels.

Members

file: BlockdevRef
Node to create the image format on

size: int
Size of the virtual disk in bytes

cluster-size: int (optional)
Cluster size in bytes (default: 1 MB)

Since

2.12

BlockdevCreateOptionsQcow (Object)

Driver specific image creation options for qcow.

Members

file: BlockdevRef
Node to create the image format on

size: int
Size of the virtual disk in bytes

backing-file: string (optional)
File name of the backing file if a backing file should be used

encrypt: QCryptoBlockCreateOptions (optional)
Encryption options if the image should be encrypted

Since

2.12

BlockdevQcow2Version (Enum)

Values

v2
The original QCOW2 format as introduced in qemu 0.10 (version 2)

v3
The extended QCOW2 format as introduced in qemu 1.1 (version 3)

Since

2.12

Qcow2CompressionType (Enum)

Compression type used in qcow2 image file

Values

zlib
zlib compression, see <http://zlib.net/>

zstd (If: CONFIG_ZSTD)
zstd compression, see <http://github.com/facebook/zstd>

Since

5.1

BlockdevCreateOptionsQcow2 (Object)

Driver specific image creation options for qcow2.

Members

file: BlockdevRef
Node to create the image format on

data-file: BlockdevRef (optional)
Node to use as an external data file in which all guest data is stored so that only metadata remains in the qcow2 file (since: 4.0)

data-file-raw: boolean (optional)
True if the external data file must stay valid as a standalone (read-only) raw image without looking at qcow2 metadata (default: false; since: 4.0)

extended-l2: boolean (optional)
True to make the image have extended L2 entries (default: false; since 5.2)

size: int
Size of the virtual disk in bytes

version: BlockdevQcow2Version (optional)
Compatibility level (default: v3)

backing-file: string (optional)
File name of the backing file if a backing file should be used

backing-fmt: BlockdevDriver (optional)
Name of the block driver to use for the backing file

encrypt: QCryptoBlockCreateOptions (optional)
Encryption options if the image should be encrypted

cluster-size: int (optional)
qcow2 cluster size in bytes (default: 65536)

preallocation: PreallocMode (optional)
Preallocation mode for the new image (default: off; allowed values: off, falloc, full, metadata)

lazy-refcounts: boolean (optional)
True if refcounts may be updated lazily (default: off)

refcount-bits: int (optional)
Width of reference counts in bits (default: 16)

compression-type: Qcow2CompressionType (optional)
The image cluster compression method (default: zlib, since 5.1)

Since

2.12

BlockdevCreateOptionsQed (Object)

Driver specific image creation options for qed.

Members

file: BlockdevRef
Node to create the image format on

size: int
Size of the virtual disk in bytes

backing-file: string (optional)
File name of the backing file if a backing file should be used

backing-fmt: BlockdevDriver (optional)
Name of the block driver to use for the backing file

cluster-size: int (optional)
Cluster size in bytes (default: 65536)

table-size: int (optional)
L1/L2 table size (in clusters)

Since

2.12

BlockdevCreateOptionsRbd (Object)

Driver specific image creation options for rbd/Ceph.

Members

location: BlockdevOptionsRbd
Where to store the new image file. This location cannot point to a snapshot.

size: int
Size of the virtual disk in bytes

cluster-size: int (optional)
RBD object size

encrypt: RbdEncryptionCreateOptions (optional)
Image encryption options. (Since 6.1)

Since

2.12

BlockdevVmdkSubformat (Enum)

Subformat options for VMDK images

Values

monolithicSparse
: Single file image with sparse cluster allocation

monolithicFlat
: Single flat data image and a descriptor file

twoGbMaxExtentSparse
: Data is split into 2GB (per virtual LBA) sparse extent files, in addition to a descriptor file

twoGbMaxExtentFlat
: Data is split into 2GB (per virtual LBA) flat extent files, in addition to a descriptor file

streamOptimized
: Single file image sparse cluster allocation, optimized for streaming over network.

Since

4.0

BlockdevVmdkAdapterType (Enum)

Adapter type info for VMDK images

Values

ide
: Not documented

buslogic
: Not documented

lsilogic
: Not documented

legacyESX
: Not documented

Since

4.0

BlockdevCreateOptionsVmdk (Object)

Driver specific image creation options for VMDK.

Members

file: BlockdevRef
Where to store the new image file. This refers to the image file for monolithcSparse and streamOptimized format, or the descriptor file for other formats.

size: int
Size of the virtual disk in bytes

extents: array of BlockdevRef (optional)
Where to store the data extents. Required for monolithcFlat, twoGbMaxExtentSparse and twoGbMaxExtentFlat formats. For monolithicFlat, only one entry is required; for twoGbMaxExtent* formats, the number of entries required is calculated as extent_number = virtual_size / 2GB. Providing more extents than will be used is an error.

subformat: BlockdevVmdkSubformat (optional)
The subformat of the VMDK image. Default: "monolithicSparse".

backing-file: string (optional)
The path of backing file. Default: no backing file is used.

adapter-type: BlockdevVmdkAdapterType (optional)
The adapter type used to fill in the descriptor. Default: ide.

hwversion: string (optional)
Hardware version. The meaningful options are "4" or "6". Default: "4".

toolsversion: string (optional)
VMware guest tools version. Default: "2147483647" (Since 6.2)

zeroed-grain: boolean (optional)
Whether to enable zeroed-grain feature for sparse subformats. Default: false.

Since

4.0

BlockdevCreateOptionsSsh (Object)

Driver specific image creation options for SSH.

Members

location: BlockdevOptionsSsh
Where to store the new image file

size: int
Size of the virtual disk in bytes

Since

2.12

BlockdevCreateOptionsVdi (Object)

Driver specific image creation options for VDI.

Members

file: BlockdevRef
Node to create the image format on

size: int
Size of the virtual disk in bytes

preallocation: PreallocMode (optional)
Preallocation mode for the new image (default: off; allowed values: off, metadata)

Since

2.12

BlockdevVhdxSubformat (Enum)

Values

dynamic
Growing image file

fixed
Preallocated fixed-size image file

Since

2.12

BlockdevCreateOptionsVhdx (Object)

Driver specific image creation options for vhdx.

Members

file: BlockdevRef
Node to create the image format on

size: int
Size of the virtual disk in bytes

log-size: int (optional)
Log size in bytes, must be a multiple of 1 MB (default: 1 MB)

block-size: int (optional)
Block size in bytes, must be a multiple of 1 MB and not larger than 256 MB (default: automatically choose a block size depending on the image size)

subformat: BlockdevVhdxSubformat (optional)
vhdx subformat (default: dynamic)

block-state-zero: boolean (optional)
Force use of payload blocks of type 'ZERO'. Non-standard, but default. Do not set to 'off' when using 'qemu-img convert' with subformat=dynamic.

Since

2.12

BlockdevVpcSubformat (Enum)

Values

dynamic
Growing image file

fixed
Preallocated fixed-size image file

Since

2.12

BlockdevCreateOptionsVpc (Object)

Driver specific image creation options for vpc (VHD).

Members

file: BlockdevRef
Node to create the image format on

size: int
Size of the virtual disk in bytes

subformat: BlockdevVpcSubformat (optional)
vhdx subformat (default: dynamic)

force-size: boolean (optional)
Force use of the exact byte size instead of rounding to the next size that can be represented in CHS geometry (default: false)

Since

2.12

BlockdevCreateOptions (Object)

Options for creating an image format on a given node.

Members

driver: BlockdevDriver
block driver to create the image format

The members of BlockdevCreateOptionsFile when driver is "file"
The members of BlockdevCreateOptionsGluster when driver is "gluster"
The members of BlockdevCreateOptionsLUKS when driver is "luks"
The members of BlockdevCreateOptionsNfs when driver is "nfs"
The members of BlockdevCreateOptionsParallels when driver is "parallels"
The members of BlockdevCreateOptionsQcow when driver is "qcow"
The members of BlockdevCreateOptionsQcow2 when driver is "qcow2"
The members of BlockdevCreateOptionsQed when driver is "qed"
The members of BlockdevCreateOptionsRbd when driver is "rbd"
The members of BlockdevCreateOptionsSsh when driver is "ssh"
The members of BlockdevCreateOptionsVdi when driver is "vdi"
The members of BlockdevCreateOptionsVhdx when driver is "vhdx"
The members of BlockdevCreateOptionsVmdk when driver is "vmdk"
The members of BlockdevCreateOptionsVpc when driver is "vpc"

Since

2.12

blockdev-create (Command)

Starts a job to create an image format on a given node. The job is automatically finalized, but a manual job-dismiss is required.

Arguments

job-id: string
Identifier for the newly created job.

options: BlockdevCreateOptions
Options for the image creation.

Since

3.0

BlockdevAmendOptionsLUKS (Object)

Driver specific image amend options for LUKS.

Members

The members of QCryptoBlockAmendOptionsLUKS

Since

5.1

BlockdevAmendOptionsQcow2 (Object)

Driver specific image amend options for qcow2. For now, only encryption options can be amended

Members

encrypt: QCryptoBlockAmendOptions (optional)
Encryption options to be amended

Since

5.1

BlockdevAmendOptions (Object)

Options for amending an image format

Members

driver: BlockdevDriver
Block driver of the node to amend.

The members of BlockdevAmendOptionsLUKS when driver is "luks"
The members of BlockdevAmendOptionsQcow2 when driver is "qcow2"

Since

5.1

x-blockdev-amend (Command)

Starts a job to amend format specific options of an existing open block device The job is automatically finalized, but a manual job-dismiss is required.

Arguments

job-id: string
Identifier for the newly created job.

node-name: string
Name of the block node to work on

options: BlockdevAmendOptions
Options (driver specific)

force: boolean (optional)
Allow unsafe operations, format specific For luks that allows erase of the last active keyslot (permanent loss of data), and replacement of an active keyslot (possible loss of data if IO error happens)

Features

unstable
This command is experimental.

Since

5.1

BlockErrorAction (Enum)

An enumeration of action that has been taken when a DISK I/O occurs

Values

ignore
error has been ignored

report
error has been reported to the device

stop
error caused VM to be stopped

Since

2.1

BLOCK_IMAGE_CORRUPTED (Event)

Emitted when a disk image is being marked corrupt. The image can be identified by its device or node name. The 'device' field is always present for compatibility reasons, but it can be empty ("") if the image does not have a device name associated.

Arguments

device: string
device name. This is always present for compatibility reasons, but it can be empty ("") if the image does not have a device name associated.

node-name: string (optional)
node name (Since: 2.4)

msg: string
informative message for human consumption, such as the kind of corruption being detected. It should not be parsed by machine as it is not guaranteed to be stable

offset: int (optional)
if the corruption resulted from an image access, this is the host's access offset into the image

size: int (optional)
if the corruption resulted from an image access, this is the access size

fatal: boolean
if set, the image is marked corrupt and therefore unusable after this event and must be repaired (Since 2.2; before, every BLOCK_IMAGE_CORRUPTED event was fatal)

Note

If action is "stop", a STOP event will eventually follow the BLOCK_IO_ERROR event.

Example

```
<- { "event": "BLOCK_IMAGE_CORRUPTED",
     "data": { "device": "", "node-name": "drive", "fatal": false,
               "msg": "L2 table offset 0x2a2a2a00 unaligned (L1 index: 0)" },
     "timestamp": { "seconds": 1648243240, "microseconds": 906060 } }
```

Since

1.7

BLOCK_IO_ERROR (Event)

Emitted when a disk I/O error occurs

Arguments

device: string
device name. This is always present for compatibility reasons, but it can be empty ("") if the image does not have a device name associated.

node-name: string (optional)
node name. Note that errors may be reported for the root node that is directly attached to a guest device rather than for the node where the error occurred. The node name is not present if the drive is empty. (Since: 2.8)

operation: IoOperationType
I/O operation

action: BlockErrorAction
action that has been taken

nospace: boolean (optional)
true if I/O error was caused due to a no-space condition. This key is only present if query-block's io-status is present, please see query-block documentation for more information (since: 2.2)

reason: string
human readable string describing the error cause. (This field is a debugging aid for humans, it should not be parsed by applications) (since: 2.2)

Note

If action is “stop”, a STOP event will eventually follow the BLOCK_IO_ERROR event

Since

0.13

Example

```
<- { "event": "BLOCK_IO_ERROR",
     "data": { "device": "ide0-hd1",
               "node-name": "#block212",
               "operation": "write",
               "action": "stop",
               "reason": "No space left on device" },
     "timestamp": { "seconds": 1265044230, "microseconds": 450486 } }
```

BLOCK_JOB_COMPLETED (Event)

Emitted when a block job has completed

Arguments

type: JobType
job type

device: string
The job identifier. Originally the device name but other values are allowed since QEMU 2.7

len: int
maximum progress value

offset: int
current progress value. On success this is equal to len. On failure this is less than len

speed: int
rate limit, bytes per second

error: string (optional)
error message. Only present on failure. This field contains a human-readable error message. There are no semantics other than that streaming has failed and clients should not try to interpret the error string

Since

1.1

Example

```
<- { "event": "BLOCK_JOB_COMPLETED",
     "data": { "type": "stream", "device": "virtio-disk0",
               "len": 10737418240, "offset": 10737418240,
               "speed": 0 },
     "timestamp": { "seconds": 1267061043, "microseconds": 959568 } }
```

BLOCK_JOB_CANCELLED (Event)

Emitted when a block job has been cancelled

Arguments

type: JobType
job type

device: string
The job identifier. Originally the device name but other values are allowed since QEMU 2.7

len: int
maximum progress value

offset: int
current progress value. On success this is equal to len. On failure this is less than len

speed: int
rate limit, bytes per second

Since

1.1

Example

```
<- { "event": "BLOCK_JOB_CANCELLED",
     "data": { "type": "stream", "device": "virtio-disk0",
               "len": 10737418240, "offset": 134217728,
               "speed": 0 },
     "timestamp": { "seconds": 1267061043, "microseconds": 959568 } }
```

BLOCK_JOB_ERROR (Event)

Emitted when a block job encounters an error

Arguments

device: string
The job identifier. Originally the device name but other values are allowed since QEMU 2.7

operation: IoOperationType
I/O operation

action: BlockErrorAction
action that has been taken

Since

1.3

Example

```
<- { "event": "BLOCK_JOB_ERROR",
     "data": { "device": "ide0-hd1",
               "operation": "write",
               "action": "stop" },
     "timestamp": { "seconds": 1265044230, "microseconds": 450486 } }
```

BLOCK_JOB_READY (Event)

Emitted when a block job is ready to complete

Arguments

type: JobType
job type

device: string
The job identifier. Originally the device name but other values are allowed since QEMU 2.7

len: int
maximum progress value

offset: int
current progress value. On success this is equal to len. On failure this is less than len

speed: int
rate limit, bytes per second

Note

The "ready to complete" status is always reset by a BLOCK_JOB_ERROR event

Since

1.3

Example

```
<- { "event": "BLOCK_JOB_READY",
     "data": { "device": "drive0", "type": "mirror", "speed": 0,
               "len": 2097152, "offset": 2097152 },
     "timestamp": { "seconds": 1265044230, "microseconds": 450486 } }
```

BLOCK_JOB_PENDING (Event)

Emitted when a block job is awaiting explicit authorization to finalize graph changes via block-job-finalize. If this job is part of a transaction, it will not emit this event until the transaction has converged first.

Arguments

type: JobType
job type

id: string
The job identifier.

Since

2.12

Example

```
<- { "event": "BLOCK_JOB_PENDING",
     "data": { "type": "mirror", "id": "backup_1" },
     "timestamp": { "seconds": 1265044230, "microseconds": 450486 } }
```

PreallocMode (Enum)

Preallocation mode of QEMU image file

Values

off
no preallocation

metadata
preallocate only for metadata

falloc
like `full` preallocation but allocate disk space by posix_fallocate() rather than writing data.

full
preallocate all data by writing it to the device to ensure disk space is really available. This data may or may not be zero, depending on the image format and storage. `full` preallocation also sets up metadata correctly.

Since

2.2

BLOCK_WRITE_THRESHOLD (Event)

Emitted when writes on block device reaches or exceeds the configured write threshold. For thin-provisioned devices, this means the device should be extended to avoid pausing for disk exhaustion. The event is one shot. Once triggered, it needs to be re-registered with another block-set-write-threshold command.

Arguments

node-name: string
graph node name on which the threshold was exceeded.

amount-exceeded: int
amount of data which exceeded the threshold, in bytes.

write-threshold: int
last configured threshold, in bytes.

Since

2.3

block-set-write-threshold (Command)

Change the write threshold for a block drive. An event will be delivered if a write to this block drive crosses the configured threshold. The threshold is an offset, thus must be non-negative. Default is no write threshold. Setting the threshold to zero disables it.

This is useful to transparently resize thin-provisioned drives without the guest OS noticing.

Arguments

node-name: string
graph node name on which the threshold must be set.

write-threshold: int
configured threshold for the block device, bytes. Use 0 to disable the threshold.

Since

2.3

Example

```
-> { "execute": "block-set-write-threshold",
     "arguments": { "node-name": "mydev",
                    "write-threshold": 17179869184 } }
<- { "return": {} }
```

x-blockdev-change (Command)

Dynamically reconfigure the block driver state graph. It can be used to add, remove, insert or replace a graph node. Currently only the Quorum driver implements this feature to add or remove its child. This is useful to fix a broken quorum child.

If `node` is specified, it will be inserted under `parent`. `child` may not be specified in this case. If both `parent` and `child` are specified but `node` is not, `child` will be detached from `parent`.

Arguments

parent: string
the id or name of the parent node.

child: string (optional)
the name of a child under the given parent node.

node: string (optional)
the name of the node that will be added.

Features

unstable

This command is experimental, and its API is not stable. It does not support all kinds of operations, all kinds of children, nor all block drivers.

FIXME Removing children from a quorum node means introducing gaps in the child indices. This cannot be represented in the 'children' list of BlockdevOptionsQuorum, as returned by .bdrv_refresh_filename().

Warning: The data in a new quorum child MUST be consistent with that of the rest of the array.

Since

2.7

Examples

```
1. Add a new node to a quorum

-> { "execute": "blockdev-add",
     "arguments": {
         "driver": "raw",
         "node-name": "new_node",
         "file": { "driver": "file",
                   "filename": "test.raw" } } }
<- { "return": {} }
-> { "execute": "x-blockdev-change",
     "arguments": { "parent": "disk1",
                    "node": "new_node" } }
<- { "return": {} }

2. Delete a quorum's node

-> { "execute": "x-blockdev-change",
     "arguments": { "parent": "disk1",
                    "child": "children.1" } }
<- { "return": {} }
```

x-blockdev-set-iothread (Command)

Move `node` and its children into the `iothread`. If `iothread` is null then move `node` and its children into the main loop.

The node must not be attached to a BlockBackend.

Arguments

node-name: string
the name of the block driver node

iothread: StrOrNull
the name of the IOThread object or null for the main loop

force: boolean (optional)
true if the node and its children should be moved when a BlockBackend is already attached

Features

unstable
This command is experimental and intended for test cases that need control over IOThreads only.

Since

2.12

Examples

```
1. Move a node into an IOThread

-> { "execute": "x-blockdev-set-iothread",
     "arguments": { "node-name": "disk1",
                    "iothread": "iothread0" } }
<- { "return": {} }

2. Move a node into the main loop

-> { "execute": "x-blockdev-set-iothread",
     "arguments": { "node-name": "disk1",
                    "iothread": null } }
<- { "return": {} }
```

QuorumOpType (Enum)

An enumeration of the quorum operation types

Values

read
read operation

write
write operation

flush
flush operation

Since

2.6

QUORUM_FAILURE (Event)

Emitted by the Quorum block driver if it fails to establish a quorum

Arguments

reference: string
device name if defined else node name

sector-num: int
number of the first sector of the failed read operation

sectors-count: int
failed read operation sector count

Note

This event is rate-limited.

Since

2.0

Example

```
<- { "event": "QUORUM_FAILURE",
     "data": { "reference": "usr1", "sector-num": 345435, "sectors-count": 5 },
     "timestamp": { "seconds": 1344522075, "microseconds": 745528 } }
```

QUORUM_REPORT_BAD (Event)

Emitted to report a corruption of a Quorum file

Arguments

type: QuorumOpType
quorum operation type (Since 2.6)

error: string (optional)
error message. Only present on failure. This field contains a human-readable error message. There are no semantics other than that the block layer reported an error and clients should not try to interpret the error string.

node-name: string
the graph node name of the block driver state

sector-num: int
number of the first sector of the failed read operation

sectors-count: int
failed read operation sector count

Note

This event is rate-limited.

Since

2.0

Examples

```
1. Read operation

<- { "event": "QUORUM_REPORT_BAD",
     "data": { "node-name": "node0", "sector-num": 345435, "sectors-count": 5,
               "type": "read" },
     "timestamp": { "seconds": 1344522075, "microseconds": 745528 } }

2. Flush operation

<- { "event": "QUORUM_REPORT_BAD",
     "data": { "node-name": "node0", "sector-num": 0, "sectors-count": 2097120,
               "type": "flush", "error": "Broken pipe" },
     "timestamp": { "seconds": 1456406829, "microseconds": 291763 } }
```

BlockdevSnapshotInternal (Object)

Members

device: string
the device name or node-name of a root node to generate the snapshot from

name: string
the name of the internal snapshot to be created

Notes

In transaction, if name is empty, or any snapshot matching name exists, the operation will fail. Only some image formats support it, for example, qcow2, and rbd.

Since

1.7

blockdev-snapshot-internal-sync (Command)

Synchronously take an internal snapshot of a block device, when the format of the image used supports it. If the name is an empty string, or a snapshot with name already exists, the operation will fail.

For the arguments, see the documentation of BlockdevSnapshotInternal.

Errors

- If device is not a valid block device, GenericError
- If any snapshot matching name exists, or name is empty, GenericError
- If the format of the image used does not support it, GenericError

Since

1.7

Example

```
-> { "execute": "blockdev-snapshot-internal-sync",
     "arguments": { "device": "ide-hd0",
                    "name": "snapshot0" }
   }
<- { "return": {} }
```

blockdev-snapshot-delete-internal-sync (Command)

Synchronously delete an internal snapshot of a block device, when the format of the image used support it. The snapshot is identified by name or id or both. One of the name or id is required. Return SnapshotInfo for the successfully deleted snapshot.

Arguments

device: string
the device name or node-name of a root node to delete the snapshot from

id: string (optional)
optional the snapshot's ID to be deleted

name: string (optional)
optional the snapshot's name to be deleted

Returns

SnapshotInfo

Errors

- If `device` is not a valid block device, GenericError
- If snapshot not found, GenericError
- If the format of the image used does not support it, GenericError
- If `id` and `name` are both not specified, GenericError

Since

1.7

Example

```
-> { "execute": "blockdev-snapshot-delete-internal-sync",
     "arguments": { "device": "ide-hd0",
                    "name": "snapshot0" }
   }
<- { "return": {
                    "id": "1",
                    "name": "snapshot0",
                    "vm-state-size": 0,
                    "date-sec": 1000012,
                    "date-nsec": 10,
                    "vm-clock-sec": 100,
                    "vm-clock-nsec": 20,
                    "icount": 220414
```

(continues on next page)

(continued from previous page)

```
    }
  }
```

DummyBlockCoreForceArrays (Object)

Not used by QMP; hack to let us use BlockGraphInfoList internally

Members

unused-block-graph-info: array of BlockGraphInfo
Not documented

Since

8.0

Additional block stuff (VM related)

BiosAtaTranslation (Enum)

Policy that BIOS should use to interpret cylinder/head/sector addresses. Note that Bochs BIOS and SeaBIOS will not actually translate logical CHS to physical; instead, they will use logical block addressing.

Values

auto
If cylinder/heads/sizes are passed, choose between none and LBA depending on the size of the disk. If they are not passed, choose none if QEMU can guess that the disk had 16 or fewer heads, large if QEMU can guess that the disk had 131072 or fewer tracks across all heads (i.e. cylinders*heads<131072), otherwise LBA.

none
The physical disk geometry is equal to the logical geometry.

lba
Assume 63 sectors per track and one of 16, 32, 64, 128 or 255 heads (if fewer than 255 are enough to cover the whole disk with 1024 cylinders/head). The number of cylinders/head is then computed based on the number of sectors and heads.

large
The number of cylinders per head is scaled down to 1024 by correspondingly scaling up the number of heads.

rechs
Same as `large`, but first convert a 16-head geometry to 15-head, by proportionally scaling up the number of cylinders/head.

Since

2.0

FloppyDriveType (Enum)

Type of Floppy drive to be emulated by the Floppy Disk Controller.

Values

144
: 1.44MB 3.5” drive

288
: 2.88MB 3.5” drive

120
: 1.2MB 5.25” drive

none
: No drive connected

auto
: Automatically determined by inserted media at boot

Since

2.6

PRManagerInfo (Object)

Information about a persistent reservation manager

Members

id: string
: the identifier of the persistent reservation manager

connected: boolean
: true if the persistent reservation manager is connected to the underlying storage or helper

Since

3.0

query-pr-managers (Command)

Returns a list of information about each persistent reservation manager.

Returns

a list of PRManagerInfo for each persistent reservation manager

Since

3.0

eject (Command)

Ejects the medium from a removable drive.

Arguments

device: string (optional)
Block device name

id: string (optional)
The name or QOM path of the guest device (since: 2.8)

force: boolean (optional)
If true, eject regardless of whether the drive is locked. If not specified, the default value is false.

Features

deprecated
Member device is deprecated. Use id instead.

Errors

- If device is not a valid block device, DeviceNotFound

Notes

Ejecting a device with no media results in success

Since

0.14

Example

```
-> { "execute": "eject", "arguments": { "id": "ide1-0-1" } }
<- { "return": {} }
```

blockdev-open-tray (Command)

Opens a block device's tray. If there is a block driver state tree inserted as a medium, it will become inaccessible to the guest (but it will remain associated to the block device, so closing the tray will make it accessible again).

If the tray was already open before, this will be a no-op.

Once the tray opens, a DEVICE_TRAY_MOVED event is emitted. There are cases in which no such event will be generated, these include:

- if the guest has locked the tray, `force` is false and the guest does not respond to the eject request
- if the BlockBackend denoted by `device` does not have a guest device attached to it
- if the guest device does not have an actual tray

Arguments

`device`: `string` (optional)
Block device name

`id`: `string` (optional)
The name or QOM path of the guest device (since: 2.8)

`force`: `boolean` (optional)
if false (the default), an eject request will be sent to the guest if it has locked the tray (and the tray will not be opened immediately); if true, the tray will be opened regardless of whether it is locked

Features

`deprecated`
Member `device` is deprecated. Use `id` instead.

Since

2.5

Example

```
-> { "execute": "blockdev-open-tray",
     "arguments": { "id": "ide0-1-0" } }

<- { "timestamp": { "seconds": 1418751016,
                    "microseconds": 716996 },
     "event": "DEVICE_TRAY_MOVED",
     "data": { "device": "ide1-cd0",
               "id": "ide0-1-0",
               "tray-open": true } }

<- { "return": {} }
```

blockdev-close-tray (Command)

Closes a block device's tray. If there is a block driver state tree associated with the block device (which is currently ejected), that tree will be loaded as the medium.

If the tray was already closed before, this will be a no-op.

Arguments

device: string (optional)
: Block device name

id: string (optional)
: The name or QOM path of the guest device (since: 2.8)

Features

deprecated
: Member `device` is deprecated. Use `id` instead.

Since

2.5

Example

```
-> { "execute": "blockdev-close-tray",
     "arguments": { "id": "ide0-1-0" } }

<- { "timestamp": { "seconds": 1418751345,
                    "microseconds": 272147 },
     "event": "DEVICE_TRAY_MOVED",
     "data": { "device": "ide1-cd0",
               "id": "ide0-1-0",
               "tray-open": false } }

<- { "return": {} }
```

blockdev-remove-medium (Command)

Removes a medium (a block driver state tree) from a block device. That block device's tray must currently be open (unless there is no attached guest device).

If the tray is open and there is no medium inserted, this will be a no-op.

Arguments

id: string
The name or QOM path of the guest device

Since

2.12

Example

```
-> { "execute": "blockdev-remove-medium",
     "arguments": { "id": "ide0-1-0" } }

<- { "error": { "class": "GenericError",
                "desc": "Tray of device 'ide0-1-0' is not open" } }

-> { "execute": "blockdev-open-tray",
     "arguments": { "id": "ide0-1-0" } }

<- { "timestamp": { "seconds": 1418751627,
                    "microseconds": 549958 },
     "event": "DEVICE_TRAY_MOVED",
     "data": { "device": "ide1-cd0",
               "id": "ide0-1-0",
               "tray-open": true } }
```

(continues on next page)

(continued from previous page)

```
<- { "return": {} }

-> { "execute": "blockdev-remove-medium",
     "arguments": { "id": "ide0-1-0" } }

<- { "return": {} }
```

`blockdev-insert-medium` (Command)

Inserts a medium (a block driver state tree) into a block device. That block device's tray must currently be open (unless there is no attached guest device) and there must be no medium inserted already.

Arguments

`id: string`
: The name or QOM path of the guest device

`node-name: string`
: name of a node in the block driver state graph

Since

2.12

Example

```
-> { "execute": "blockdev-add",
     "arguments": {
         "node-name": "node0",
         "driver": "raw",
         "file": { "driver": "file",
                   "filename": "fedora.iso" } } }
<- { "return": {} }

-> { "execute": "blockdev-insert-medium",
     "arguments": { "id": "ide0-1-0",
                    "node-name": "node0" } }

<- { "return": {} }
```

BlockdevChangeReadOnlyMode (Enum)

Specifies the new read-only mode of a block device subject to the `blockdev-change-medium` command.

Values

retain
Retains the current read-only mode

read-only
Makes the device read-only

read-write
Makes the device writable

Since

2.3

blockdev-change-medium (Command)

Changes the medium inserted into a block device by ejecting the current medium and loading a new image file which is inserted as the new medium (this command combines blockdev-open-tray, blockdev-remove-medium, blockdev-insert-medium and blockdev-close-tray).

Arguments

device: string (optional)
Block device name

id: string (optional)
The name or QOM path of the guest device (since: 2.8)

filename: string
filename of the new image to be loaded

format: string (optional)
format to open the new image with (defaults to the probed format)

read-only-mode: BlockdevChangeReadOnlyMode (optional)
change the read-only mode of the device; defaults to 'retain'

force: boolean (optional)
if false (the default), an eject request through blockdev-open-tray will be sent to the guest if it has locked the tray (and the tray will not be opened immediately); if true, the tray will be opened regardless of whether it is locked. (since 7.1)

Features

deprecated
Member `device` is deprecated. Use `id` instead.

Since

2.5

Examples

```
1. Change a removable medium

-> { "execute": "blockdev-change-medium",
     "arguments": { "id": "ide0-1-0",
                    "filename": "/srv/images/Fedora-12-x86_64-DVD.iso",
                    "format": "raw" } }
<- { "return": {} }

2. Load a read-only medium into a writable drive

-> { "execute": "blockdev-change-medium",
     "arguments": { "id": "floppyA",
                    "filename": "/srv/images/ro.img",
                    "format": "raw",
                    "read-only-mode": "retain" } }

<- { "error":
     { "class": "GenericError",
       "desc": "Could not open '/srv/images/ro.img': Permission denied" } }

-> { "execute": "blockdev-change-medium",
     "arguments": { "id": "floppyA",
                    "filename": "/srv/images/ro.img",
                    "format": "raw",
                    "read-only-mode": "read-only" } }

<- { "return": {} }
```

DEVICE_TRAY_MOVED (Event)

Emitted whenever the tray of a removable device is moved by the guest or by HMP/QMP commands

Arguments

device: string
Block device name. This is always present for compatibility reasons, but it can be empty (“”) if the image does not have a device name associated.

id: string
The name or QOM path of the guest device (since 2.8)

tray-open: boolean
true if the tray has been opened or false if it has been closed

Since

1.1

Example

```
<- { "event": "DEVICE_TRAY_MOVED",
     "data": { "device": "ide1-cd0",
               "id": "/machine/unattached/device[22]",
               "tray-open": true
     },
     "timestamp": { "seconds": 1265044230, "microseconds": 450486 } }
```

PR_MANAGER_STATUS_CHANGED (Event)

Emitted whenever the connected status of a persistent reservation manager changes.

Arguments

id: string
The id of the PR manager object

connected: boolean
true if the PR manager is connected to a backend

Since

3.0

Example

```
<- { "event": "PR_MANAGER_STATUS_CHANGED",
     "data": { "id": "pr-helper0",
               "connected": true
     },
     "timestamp": { "seconds": 1519840375, "microseconds": 450486 } }
```

block_set_io_throttle (Command)

Change I/O throttle limits for a block drive.

Since QEMU 2.4, each device with I/O limits is member of a throttle group.

If two or more devices are members of the same group, the limits will apply to the combined I/O of the whole group in a round-robin fashion. Therefore, setting new I/O limits to a device will affect the whole group.

The name of the group can be specified using the 'group' parameter. If the parameter is unset, it is assumed to be the current group of that device. If it's not in any group yet, the name of the device will be used as the name for its group.

The 'group' parameter can also be used to move a device to a different group. In this case the limits specified in the parameters will be applied to the new group only.

I/O limits can be disabled by setting all of them to 0. In this case the device will be removed from its group and the rest of its members will not be affected. The 'group' parameter is ignored.

Arguments

The members of `BlockIOThrottle`

Errors

- If `device` is not a valid block device, DeviceNotFound

Since

1.1

Examples

```
-> { "execute": "block_set_io_throttle",
     "arguments": { "id": "virtio-blk-pci0/virtio-backend",
                    "bps": 0,
                    "bps_rd": 0,
                    "bps_wr": 0,
                    "iops": 512,
                    "iops_rd": 0,
                    "iops_wr": 0,
                    "bps_max": 0,
```

(continues on next page)

(continued from previous page)

```
                     "bps_rd_max": 0,
                     "bps_wr_max": 0,
                     "iops_max": 0,
                     "iops_rd_max": 0,
                     "iops_wr_max": 0,
                     "bps_max_length": 0,
                     "iops_size": 0 } }
<- { "return": {} }

-> { "execute": "block_set_io_throttle",
     "arguments": { "id": "ide0-1-0",
                     "bps": 1000000,
                     "bps_rd": 0,
                     "bps_wr": 0,
                     "iops": 0,
                     "iops_rd": 0,
                     "iops_wr": 0,
                     "bps_max": 8000000,
                     "bps_rd_max": 0,
                     "bps_wr_max": 0,
                     "iops_max": 0,
                     "iops_rd_max": 0,
                     "iops_wr_max": 0,
                     "bps_max_length": 60,
                     "iops_size": 0 } }
<- { "return": {} }
```

block-latency-histogram-set (Command)

Manage read, write and flush latency histograms for the device.

If only `id` parameter is specified, remove all present latency histograms for the device. Otherwise, add/reset some of (or all) latency histograms.

Arguments

id: string
: The name or QOM path of the guest device.

boundaries: array of int (optional)
: list of interval boundary values (see description in BlockLatencyHistogramInfo definition). If specified, all latency histograms are removed, and empty ones created for all io types with intervals corresponding to `boundaries` (except for io types, for which specific boundaries are set through the following parameters).

boundaries-read: array of int (optional)
: list of interval boundary values for read latency histogram. If specified, old read latency histogram is removed, and empty one created with intervals corresponding to `boundaries-read`. The parameter has higher priority then `boundaries`.

boundaries-write: array of int (optional)
: list of interval boundary values for write latency histogram.

boundaries-zap: array of int (optional)
list of interval boundary values for zone append write latency histogram.

boundaries-flush: array of int (optional)
list of interval boundary values for flush latency histogram.

Errors

- if device is not found or any boundary arrays are invalid.

Since

4.0

Example

```
Set new histograms for all io types with intervals
[0, 10), [10, 50), [50, 100), [100, +inf):

-> { "execute": "block-latency-histogram-set",
     "arguments": { "id": "drive0",
                    "boundaries": [10, 50, 100] } }
<- { "return": {} }
```

Example

```
Set new histogram only for write, other histograms will remain
not changed (or not created):

-> { "execute": "block-latency-histogram-set",
     "arguments": { "id": "drive0",
                    "boundaries-write": [10, 50, 100] } }
<- { "return": {} }
```

Example

```
Set new histograms with the following intervals:
  read, flush: [0, 10), [10, 50), [50, 100), [100, +inf)
  write: [0, 1000), [1000, 5000), [5000, +inf)

-> { "execute": "block-latency-histogram-set",
     "arguments": { "id": "drive0",
                    "boundaries": [10, 50, 100],
                    "boundaries-write": [1000, 5000] } }
<- { "return": {} }
```

Example

```
Remove all latency histograms:

-> { "execute": "block-latency-histogram-set",
     "arguments": { "id": "drive0" } }
<- { "return": {} }
```

Block device exports

NbdServerOptions (Object)

Keep this type consistent with the nbd-server-start arguments. The only intended difference is using SocketAddress instead of SocketAddressLegacy.

Members

addr: SocketAddress
Address on which to listen.

tls-creds: string (optional)
ID of the TLS credentials object (since 2.6).

tls-authz: string (optional)
ID of the QAuthZ authorization object used to validate the client's x509 distinguished name. This object is is only resolved at time of use, so can be deleted and recreated on the fly while the NBD server is active. If missing, it will default to denying access (since 4.0).

max-connections: int (optional)
The maximum number of connections to allow at the same time, 0 for unlimited. Setting this to 1 also stops the server from advertising multiple client support (since 5.2; default: 0)

Since

4.2

nbd-server-start (Command)

Start an NBD server listening on the given host and port. Block devices can then be exported using nbd-server-add. The NBD server will present them as named exports; for example, another QEMU instance could refer to them as "nbd:HOST:PORT:exportname=NAME".

Keep this type consistent with the NbdServerOptions type. The only intended difference is using SocketAddressLegacy instead of SocketAddress.

Arguments

addr: SocketAddressLegacy
Address on which to listen.

tls-creds: string (optional)
ID of the TLS credentials object (since 2.6).

tls-authz: string (optional)
ID of the QAuthZ authorization object used to validate the client's x509 distinguished name. This object is is only resolved at time of use, so can be deleted and recreated on the fly while the NBD server is active. If missing, it will default to denying access (since 4.0).

max-connections: int (optional)
The maximum number of connections to allow at the same time, 0 for unlimited. Setting this to 1 also stops the server from advertising multiple client support (since 5.2; default: 0).

Errors

- if the server is already running

Since

1.3

BlockExportOptionsNbdBase (Object)

An NBD block export (common options shared between nbd-server-add and the NBD branch of block-export-add).

Members

name: string (optional)
Export name. If unspecified, the `device` parameter is used as the export name. (Since 2.12)

description: string (optional)
Free-form description of the export, up to 4096 bytes. (Since 5.0)

Since

5.0

BlockExportOptionsNbd (Object)

An NBD block export (distinct options used in the NBD branch of block-export-add).

Members

bitmaps: array of BlockDirtyBitmapOrStr (optional)
Also export each of the named dirty bitmaps reachable from device, so the NBD client can use NBD_OPT_SET_META_CONTEXT with the metadata context name "qemu:dirty-bitmap:BITMAP" to inspect each bitmap. Since 7.1 bitmap may be specified by node/name pair.

allocation-depth: boolean (optional)
Also export the allocation depth map for device, so the NBD client can use NBD_OPT_SET_META_CONTEXT with the metadata context name "qemu:allocation-depth" to inspect allocation details. (since 5.2)

The members of BlockExportOptionsNbdBase

Since

5.2

BlockExportOptionsVhostUserBlk (Object)

A vhost-user-blk block export.

Members

addr: SocketAddress
The vhost-user socket on which to listen. Both 'unix' and 'fd' SocketAddress types are supported. Passed fds must be UNIX domain sockets.

logical-block-size: int (optional)
Logical block size in bytes. Defaults to 512 bytes.

num-queues: int (optional)
Number of request virtqueues. Must be greater than 0. Defaults to 1.

Since

5.2

FuseExportAllowOther (Enum)

Possible allow_other modes for FUSE exports.

Values

off
: Do not pass allow_other as a mount option.

on
: Pass allow_other as a mount option.

auto
: Try mounting with allow_other first, and if that fails, retry without allow_other.

Since

6.1

BlockExportOptionsFuse (Object)

Options for exporting a block graph node on some (file) mountpoint as a raw image.

Members

mountpoint: string
: Path on which to export the block device via FUSE. This must point to an existing regular file.

growable: boolean (optional)
: Whether writes beyond the EOF should grow the block node accordingly. (default: false)

allow-other: FuseExportAllowOther (optional)
: If this is off, only qemu's user is allowed access to this export. That cannot be changed even with chmod or chown. Enabling this option will allow other users access to the export with the FUSE mount option "allow_other". Note that using allow_other as a non-root user requires user_allow_other to be enabled in the global fuse.conf configuration file. In auto mode (the default), the FUSE export driver will first attempt to mount the export with allow_other, and if that fails, try again without. (since 6.1; default: auto)

Since

6.0

If

CONFIG_FUSE

BlockExportOptionsVduseBlk (Object)

A vduse-blk block export.

Members

name: string
the name of VDUSE device (must be unique across the host).

num-queues: int (optional)
the number of virtqueues. Defaults to 1.

queue-size: int (optional)
the size of virtqueue. Defaults to 256.

logical-block-size: int (optional)
Logical block size in bytes. Range [512, PAGE_SIZE] and must be power of 2. Defaults to 512 bytes.

serial: string (optional)
the serial number of virtio block device. Defaults to empty string.

Since

7.1

NbdServerAddOptions (Object)

An NBD block export, per legacy nbd-server-add command.

Members

device: string
The device name or node name of the node to be exported

writable: boolean (optional)
Whether clients should be able to write to the device via the NBD connection (default false).

bitmap: string (optional)
Also export a single dirty bitmap reachable from `device`, so the NBD client can use NBD_OPT_SET_META_CONTEXT with the metadata context name "qemu:dirty-bitmap:BITMAP" to inspect the bitmap (since 4.0).

The members of BlockExportOptionsNbdBase

Since

5.0

nbd-server-add (Command)

Export a block node to QEMU's embedded NBD server.

The export name will be used as the id for the resulting block export.

Arguments

The members of NbdServerAddOptions

Features

deprecated
: This command is deprecated. Use block-export-add instead.

Errors

- if the server is not running
- if an export with the same name already exists

Since

1.3

BlockExportRemoveMode (Enum)

Mode for removing a block export.

Values

safe
: Remove export if there are no existing connections, fail otherwise.

hard
: Drop all connections immediately and remove export.

Since

2.12

nbd-server-remove (Command)

Remove NBD export by name.

Arguments

name: string
Block export id.

mode: BlockExportRemoveMode (optional)
Mode of command operation. See BlockExportRemoveMode description. Default is 'safe'.

Features

deprecated
This command is deprecated. Use block-export-del instead.

Errors

- if the server is not running
- if export is not found
- if mode is 'safe' and there are existing connections

Since

2.12

nbd-server-stop (Command)

Stop QEMU's embedded NBD server, and unregister all devices previously added via nbd-server-add.

Since

1.3

BlockExportType (Enum)

An enumeration of block export types

Values

nbd
NBD export

vhost-user-blk (If: CONFIG_VHOST_USER_BLK_SERVER)
vhost-user-blk export (since 5.2)

fuse (If: CONFIG_FUSE)
FUSE export (since: 6.0)

vduse-blk (If: CONFIG_VDUSE_BLK_EXPORT)
vduse-blk export (since 7.1)

Since

4.2

BlockExportOptions (Object)

Describes a block export, i.e. how single node should be exported on an external interface.

Members

type: BlockExportType
Block export type

id: string
A unique identifier for the block export (across all export types)

node-name: string
The node name of the block node to be exported (since: 5.2)

writable: boolean (optional)
True if clients should be able to write to the export (default false)

writethrough: boolean (optional)
If true, caches are flushed after every write request to the export before completion is signalled. (since: 5.2; default: false)

iothread: string (optional)
The name of the iothread object where the export will run. The default is to use the thread currently associated with the block node. (since: 5.2)

fixed-iothread: boolean (optional)
True prevents the block node from being moved to another thread while the export is active. If true and `iothread` is given, export creation fails if the block node cannot be moved to the iothread. The default is false. (since: 5.2)

The members of BlockExportOptionsNbd when type is "nbd"
The members of BlockExportOptionsVhostUserBlk when type is "vhost-user-blk" (If: CONFIG_VHOST_USER_BLK_SERVER)
The members of BlockExportOptionsFuse when type is "fuse" (If: CONFIG_FUSE)
The members of BlockExportOptionsVduseBlk when type is "vduse-blk" (If: CONFIG_VDUSE_BLK_EXPORT)

Since

4.2

block-export-add (Command)

Creates a new block export.

Arguments

The members of BlockExportOptions

Since

5.2

block-export-del (Command)

Request to remove a block export. This drops the user's reference to the export, but the export may still stay around after this command returns until the shutdown of the export has completed.

Arguments

id: string
Block export id.

mode: BlockExportRemoveMode (optional)
Mode of command operation. See BlockExportRemoveMode description. Default is 'safe'.

Errors

- if the export is not found
- if mode is 'safe' and the export is still in use (e.g. by existing client connections)

Since

5.2

BLOCK_EXPORT_DELETED (Event)

Emitted when a block export is removed and its id can be reused.

Arguments

id: string
Block export id.

Since

5.2

BlockExportInfo (Object)

Information about a single block export.

Members

id: string
The unique identifier for the block export

type: BlockExportType
The block export type

node-name: string
The node name of the block node that is exported

shutting-down: boolean
True if the export is shutting down (e.g. after a block-export-del command, but before the shutdown has completed)

Since

5.2

query-block-exports (Command)

Returns

A list of BlockExportInfo describing all block exports

Since

5.2

5.11.9 Character devices

ChardevInfo (Object)

Information about a character device.

Members

label: string
: the label of the character device

filename: string
: the filename of the character device

frontend-open: boolean
: shows whether the frontend device attached to this backend (e.g. with the chardev=... option) is in open or closed state (since 2.1)

Notes

filename is encoded using the QEMU command line character device encoding. See the QEMU man page for details.

Since

0.14

query-chardev (Command)

Returns information about current character devices.

Returns

a list of `ChardevInfo`

Since

0.14

Example

```
-> { "execute": "query-chardev" }
<- {
      "return": [
         {
            "label": "charchannel0",
            "filename": "unix:/var/lib/libvirt/qemu/seabios.rhel6.agent,server=on",
            "frontend-open": false
         },
         {
            "label": "charmonitor",
            "filename": "unix:/var/lib/libvirt/qemu/seabios.rhel6.monitor,server=on",
            "frontend-open": true
         },
         {
            "label": "charserial0",
            "filename": "pty:/dev/pts/2",
            "frontend-open": true
         }
      ]
   }
```

`ChardevBackendInfo` (Object)

Information about a character device backend

Members

`name: string`
The backend name

Since

2.0

query-chardev-backends (Command)

Returns information about character device backends.

Returns

a list of `ChardevBackendInfo`

Since

2.0

Example

```
-> { "execute": "query-chardev-backends" }
<- {
      "return":[
         {
            "name":"udp"
         },
         {
            "name":"tcp"
         },
         {
            "name":"unix"
         },
         {
            "name":"spiceport"
         }
      ]
   }
```

DataFormat (Enum)

An enumeration of data format.

Values

utf8
Data is a UTF-8 string (RFC 3629)

base64
Data is Base64 encoded binary (RFC 3548)

Since

1.4

ringbuf-write (Command)

Write to a ring buffer character device.

Arguments

device: string
the ring buffer character device name

data: string
data to write

format: DataFormat (optional)
data encoding (default 'utf8').

- base64: data must be base64 encoded text. Its binary decoding gets written.
- utf8: data's UTF-8 encoding is written
- data itself is always Unicode regardless of format, like any other string.

Since

1.4

Example

```
-> { "execute": "ringbuf-write",
     "arguments": { "device": "foo",
                    "data": "abcdefgh",
                    "format": "utf8" } }
<- { "return": {} }
```

ringbuf-read (Command)

Read from a ring buffer character device.

Arguments

device: string
the ring buffer character device name

size: int
how many bytes to read at most

format: DataFormat (optional)
data encoding (default 'utf8').

- base64: the data read is returned in base64 encoding.
- utf8: the data read is interpreted as UTF-8. Bug: can screw up when the buffer contains invalid UTF-8 sequences, NUL characters, after the ring buffer lost data, and when reading stops because the size limit is reached.
- The return value is always Unicode regardless of format, like any other string.

Returns

data read from the device

Since

1.4

Example

```
-> { "execute": "ringbuf-read",
     "arguments": { "device": "foo",
                    "size": 1000,
                    "format": "utf8" } }
<- { "return": "abcdefgh" }
```

ChardevCommon (Object)

Configuration shared across all chardev backends

Members

logfile: string (optional)
The name of a logfile to save output

logappend: boolean (optional)
true to append instead of truncate (default to false to truncate)

Since

2.6

ChardevFile (Object)

Configuration info for file chardevs.

Members

in: string (optional)
The name of the input file

out: string
The name of the output file

append: boolean (optional)
Open the file in append mode (default false to truncate) (Since 2.6)

The members of ChardevCommon

Since

1.4

ChardevHostdev (Object)

Configuration info for device and pipe chardevs.

Members

device: string
The name of the special file for the device, i.e. /dev/ttyS0 on Unix or COM1: on Windows

The members of ChardevCommon

Since

1.4

ChardevSocket (Object)

Configuration info for (stream) socket chardevs.

Members

addr: SocketAddressLegacy
socket address to listen on (server=true) or connect to (server=false)

tls-creds: string (optional)
the ID of the TLS credentials object (since 2.6)

tls-authz: string (optional)
the ID of the QAuthZ authorization object against which the client's x509 distinguished name will be validated. This object is only resolved at time of use, so can be deleted and recreated on the fly while the chardev server is active. If missing, it will default to denying access (since 4.0)

server: boolean (optional)
create server socket (default: true)

wait: boolean (optional)
wait for incoming connection on server sockets (default: false). Silently ignored with server: false. This use is deprecated.

nodelay: boolean (optional)
set TCP_NODELAY socket option (default: false)

telnet: boolean (optional)
enable telnet protocol on server sockets (default: false)

tn3270: boolean (optional)
enable tn3270 protocol on server sockets (default: false) (Since: 2.10)

websocket: boolean (optional)
enable websocket protocol on server sockets (default: false) (Since: 3.1)

reconnect: int (optional)
For a client socket, if a socket is disconnected, then attempt a reconnect after the given number of seconds. Setting this to zero disables this function. (default: 0) (Since: 2.2)

The members of ChardevCommon

Since

1.4

ChardevUdp (Object)

Configuration info for datagram socket chardevs.

Members

remote: SocketAddressLegacy
remote address

local: SocketAddressLegacy (optional)
local address

The members of ChardevCommon

Since

1.5

ChardevMux (Object)

Configuration info for mux chardevs.

Members

chardev: string
name of the base chardev.

The members of ChardevCommon

Since

1.5

ChardevStdio (Object)

Configuration info for stdio chardevs.

Members

signal: boolean (optional)
Allow signals (such as SIGINT triggered by ^C) be delivered to qemu. Default: true.

The members of ChardevCommon

Since

1.5

ChardevSpiceChannel (Object)

Configuration info for spice vm channel chardevs.

Members

type: string
kind of channel (for example vdagent).

The members of ChardevCommon

Since

1.5

If

CONFIG_SPICE

ChardevSpicePort (Object)

Configuration info for spice port chardevs.

Members

fqdn: string
name of the channel (see docs/spice-port-fqdn.txt)

The members of ChardevCommon

Since

1.5

If

CONFIG_SPICE

ChardevDBus (Object)

Configuration info for DBus chardevs.

Members

name: string
name of the channel (following docs/spice-port-fqdn.txt)

The members of ChardevCommon

Since

7.0

If

CONFIG_DBUS_DISPLAY

ChardevVC (Object)

Configuration info for virtual console chardevs.

Members

width: int (optional)
console width, in pixels

height: int (optional)
console height, in pixels

cols: int (optional)
console width, in chars

rows: int (optional)
console height, in chars

The members of ChardevCommon

Note

the options are only effective when the VNC or SDL graphical display backend is active. They are ignored with the GTK, Spice, VNC and D-Bus display backends.

Since

1.5

ChardevRingbuf (Object)

Configuration info for ring buffer chardevs.

Members

size: int (optional)
ring buffer size, must be power of two, default is 65536

The members of ChardevCommon

Since

1.5

ChardevQemuVDAgent (Object)

Configuration info for qemu vdagent implementation.

Members

mouse: boolean (optional)
enable/disable mouse, default is enabled.

clipboard: boolean (optional)
enable/disable clipboard, default is disabled.

The members of ChardevCommon

Since

6.1

If

CONFIG_SPICE_PROTOCOL

ChardevBackendKind (Enum)

Values

pipe
Since 1.5

udp
Since 1.5

mux
Since 1.5

msmouse
Since 1.5

wctablet
Since 2.9

braille (**If:** CONFIG_BRLAPI)
Since 1.5

testdev
Since 2.2

stdio
Since 1.5

console (**If:** CONFIG_WIN32)
Since 1.5

spicevmc (**If:** CONFIG_SPICE)
Since 1.5

spiceport (**If:** CONFIG_SPICE)
Since 1.5

qemu-vdagent (**If:** CONFIG_SPICE_PROTOCOL)
Since 6.1

dbus (**If:** CONFIG_DBUS_DISPLAY)
Since 7.0

vc
v1.5

ringbuf
Since 1.6

memory
Since 1.5

file
Not documented

serial (**If:** HAVE_CHARDEV_SERIAL)
Not documented

parallel (**If:** HAVE_CHARDEV_PARALLEL)
Not documented

socket
Not documented

pty
Not documented

null
Not documented

Features

deprecated
Member `memory` is deprecated. Use `ringbuf` instead.

Since

1.4

`ChardevFileWrapper` (Object)

Members

data: ChardevFile
Configuration info for file chardevs

Since

1.4

`ChardevHostdevWrapper` (Object)

Members

data: ChardevHostdev
Configuration info for device and pipe chardevs

Since

1.4

`ChardevSocketWrapper` (Object)

Members

data: ChardevSocket
Configuration info for (stream) socket chardevs

Since

1.4

ChardevUdpWrapper (Object)

Members

data: ChardevUdp
Configuration info for datagram socket chardevs

Since

1.5

ChardevCommonWrapper (Object)

Members

data: ChardevCommon
Configuration shared across all chardev backends

Since

2.6

ChardevMuxWrapper (Object)

Members

data: ChardevMux
Configuration info for mux chardevs

Since

1.5

ChardevStdioWrapper (Object)

Members

data: ChardevStdio
Configuration info for stdio chardevs

Since

1.5

ChardevSpiceChannelWrapper (Object)

Members

data: ChardevSpiceChannel
Configuration info for spice vm channel chardevs

Since

1.5

If

CONFIG_SPICE

ChardevSpicePortWrapper (Object)

Members

data: ChardevSpicePort
Configuration info for spice port chardevs

Since

1.5

If

CONFIG_SPICE

ChardevQemuVDAgentWrapper (Object)

Members

data: ChardevQemuVDAgent
Configuration info for qemu vdagent implementation

Since

6.1

If

CONFIG_SPICE_PROTOCOL

ChardevDBusWrapper (Object)

Members

data: ChardevDBus
Configuration info for DBus chardevs

Since

7.0

If

CONFIG_DBUS_DISPLAY

ChardevVCWrapper (Object)

Members

data: ChardevVC
Configuration info for virtual console chardevs

Since

1.5

ChardevRingbufWrapper (Object)

Members

data: ChardevRingbuf
Configuration info for ring buffer chardevs

Since

1.5

ChardevBackend (Object)

Configuration info for the new chardev backend.

Members

type: ChardevBackendKind
backend type

The members of ChardevFileWrapper when type is "file"
The members of ChardevHostdevWrapper when type is "serial" (If: HAVE_CHARDEV_SERIAL)
The members of ChardevHostdevWrapper when type is "parallel" (If: HAVE_CHARDEV_PARALLEL)
The members of ChardevHostdevWrapper when type is "pipe"
The members of ChardevSocketWrapper when type is "socket"
The members of ChardevUdpWrapper when type is "udp"
The members of ChardevCommonWrapper when type is "pty"
The members of ChardevCommonWrapper when type is "null"
The members of ChardevMuxWrapper when type is "mux"
The members of ChardevCommonWrapper when type is "msmouse"
The members of ChardevCommonWrapper when type is "wctablet"
The members of ChardevCommonWrapper when type is "braille" (If: CONFIG_BRLAPI)
The members of ChardevCommonWrapper when type is "testdev"
The members of ChardevStdioWrapper when type is "stdio"
The members of ChardevCommonWrapper when type is "console" (If: CONFIG_WIN32)
The members of ChardevSpiceChannelWrapper when type is "spicevmc" (If: CONFIG_SPICE)
The members of ChardevSpicePortWrapper when type is "spiceport" (If: CONFIG_SPICE)
The members of ChardevQemuVDAgentWrapper when type is "qemu-vdagent" (If: CONFIG_SPICE_PROTOCOL)
The members of ChardevDBusWrapper when type is "dbus" (If: CONFIG_DBUS_DISPLAY)
The members of ChardevVCWrapper when type is "vc"
The members of ChardevRingbufWrapper when type is "ringbuf"
The members of ChardevRingbufWrapper when type is "memory"

Since

1.4

ChardevReturn (Object)

Return info about the chardev backend just created.

Members

pty: string (optional)
name of the slave pseudoterminal device, present if and only if a chardev of type 'pty' was created

Since

1.4

chardev-add (Command)

Add a character device backend

Arguments

id: string
the chardev's ID, must be unique

backend: ChardevBackend
backend type and parameters

Returns

ChardevReturn.

Since

1.4

Examples

```
-> { "execute" : "chardev-add",
     "arguments" : { "id" : "foo",
                     "backend" : { "type" : "null", "data" : {} } } }
<- { "return": {} }

-> { "execute" : "chardev-add",
     "arguments" : { "id" : "bar",
                     "backend" : { "type" : "file",
                                   "data" : { "out" : "/tmp/bar.log" } } } }
<- { "return": {} }

-> { "execute" : "chardev-add",
     "arguments" : { "id" : "baz",
                     "backend" : { "type" : "pty", "data" : {} } } }
<- { "return": { "pty" : "/dev/pty/42" } }
```

chardev-change (Command)

Change a character device backend

Arguments

id: string
the chardev's ID, must exist

backend: ChardevBackend
new backend type and parameters

Returns

ChardevReturn.

Since

2.10

Examples

```
-> { "execute" : "chardev-change",
     "arguments" : { "id" : "baz",
                     "backend" : { "type" : "pty", "data" : {} } } }
<- { "return": { "pty" : "/dev/pty/42" } }

-> {"execute" : "chardev-change",
    "arguments" : {
        "id" : "charchannel2",
        "backend" : {
            "type" : "socket",
            "data" : {
                "addr" : {
                    "type" : "unix" ,
                    "data" : {
                        "path" : "/tmp/charchannel2.socket"
                    }
                },
                "server" : true,
                "wait" : false }}}}
<- {"return": {}}
```

chardev-remove (Command)

Remove a character device backend

Arguments

id: string
: the chardev's ID, must exist and not be in use

Since

1.4

Example

```
-> { "execute": "chardev-remove", "arguments": { "id" : "foo" } }
<- { "return": {} }
```

chardev-send-break (Command)

Send a break to a character device

Arguments

id: string
: the chardev's ID, must exist

Since

2.10

Example

```
-> { "execute": "chardev-send-break", "arguments": { "id" : "foo" } }
<- { "return": {} }
```

VSERPORT_CHANGE (Event)

Emitted when the guest opens or closes a virtio-serial port.

Arguments

id: string
device identifier of the virtio-serial port

open: boolean
true if the guest has opened the virtio-serial port

Note

This event is rate-limited.

Since

2.1

Example

```
<- { "event": "VSERPORT_CHANGE",
     "data": { "id": "channel0", "open": true },
     "timestamp": { "seconds": 1401385907, "microseconds": 422329 } }
```

5.11.10 Dump guest memory

DumpGuestMemoryFormat (Enum)

An enumeration of guest-memory-dump's format.

Values

elf
elf format

kdump-zlib
makedumpfile flattened, kdump-compressed format with zlib compression

kdump-lzo
makedumpfile flattened, kdump-compressed format with lzo compression

kdump-snappy
makedumpfile flattened, kdump-compressed format with snappy compression

kdump-raw-zlib
raw assembled kdump-compressed format with zlib compression (since 8.2)

kdump-raw-lzo
raw assembled kdump-compressed format with lzo compression (since 8.2)

kdump-raw-snappy
raw assembled kdump-compressed format with snappy compression (since 8.2)

win-dmp
Windows full crashdump format, can be used instead of ELF converting (since 2.13)

Since

2.0

dump-guest-memory (Command)

Dump guest's memory to vmcore. It is a synchronous operation that can take very long depending on the amount of guest memory.

Arguments

paging: boolean
if true, do paging to get guest's memory mapping. This allows using gdb to process the core file.

IMPORTANT: this option can make QEMU allocate several gigabytes of RAM. This can happen for a large guest, or a malicious guest pretending to be large.

Also, paging=true has the following limitations:

1. The guest may be in a catastrophic state or can have corrupted memory, which cannot be trusted
2. The guest can be in real-mode even if paging is enabled. For example, the guest uses ACPI to sleep, and ACPI sleep state goes in real-mode
3. Currently only supported on i386 and x86_64.

protocol: string
the filename or file descriptor of the vmcore. The supported protocols are:

1. file: the protocol starts with "file:", and the following string is the file's path.
2. fd: the protocol starts with "fd:", and the following string is the fd's name.

detach: boolean (optional)
if true, QMP will return immediately rather than waiting for the dump to finish. The user can track progress using "query-dump". (since 2.6).

begin: int (optional)
if specified, the starting physical address.

length: int (optional)
if specified, the memory size, in bytes. If you don't want to dump all guest's memory, please specify the start `begin` and `length`

format: DumpGuestMemoryFormat (optional)
if specified, the format of guest memory dump. But non-elf format is conflict with paging and filter, ie. `paging`, `begin` and `length` is not allowed to be specified with non-elf `format` at the same time (since 2.0)

Note

All boolean arguments default to false

Since

1.2

Example

```
-> { "execute": "dump-guest-memory",
     "arguments": { "paging": false, "protocol": "fd:dump" } }
<- { "return": {} }
```

DumpStatus (Enum)

Describe the status of a long-running background guest memory dump.

Values

none
: no dump-guest-memory has started yet.

active
: there is one dump running in background.

completed
: the last dump has finished successfully.

failed
: the last dump has failed.

Since

2.6

DumpQueryResult (Object)

The result format for 'query-dump'.

Members

status: DumpStatus
enum of DumpStatus, which shows current dump status

completed: int
bytes written in latest dump (uncompressed)

total: int
total bytes to be written in latest dump (uncompressed)

Since

2.6

query-dump (Command)

Query latest dump status.

Returns

A DumpStatus object showing the dump status.

Since

2.6

Example

```
-> { "execute": "query-dump" }
<- { "return": { "status": "active", "completed": 1024000,
                 "total": 2048000 } }
```

DUMP_COMPLETED (Event)

Emitted when background dump has completed

Arguments

result: DumpQueryResult
final dump status

error: string (optional)
human-readable error string that provides hint on why dump failed. Only presents on failure. The user should not try to interpret the error string.

Since

2.6

Example

```
<- { "event": "DUMP_COMPLETED",
     "data": { "result": { "total": 1090650112, "status": "completed",
                           "completed": 1090650112 } },
     "timestamp": { "seconds": 1648244171, "microseconds": 950316 } }
```

DumpGuestMemoryCapability (Object)

Members

formats: array of DumpGuestMemoryFormat
the available formats for dump-guest-memory

Since

2.0

query-dump-guest-memory-capability (Command)

Returns the available formats for dump-guest-memory

Returns

A DumpGuestMemoryCapability object listing available formats for dump-guest-memory

Since

2.0

Example

```
-> { "execute": "query-dump-guest-memory-capability" }
<- { "return": { "formats":
                 ["elf", "kdump-zlib", "kdump-lzo", "kdump-snappy"] } }
```

5.11.11 Net devices

set_link (Command)

Sets the link status of a virtual network adapter.

Arguments

name: string
: the device name of the virtual network adapter

up: boolean
: true to set the link status to be up

Errors

- If name is not a valid network device, DeviceNotFound

Since

0.14

Notes

Not all network adapters support setting link status. This command will succeed even if the network adapter does not support link status notification.

Example

```
-> { "execute": "set_link",
     "arguments": { "name": "e1000.0", "up": false } }
<- { "return": {} }
```

netdev_add (Command)

Add a network backend.

Additional arguments depend on the type.

Arguments

The members of `Netdev`

Since

0.14

Errors

- If `type` is not a valid network backend, DeviceNotFound

Example

```
-> { "execute": "netdev_add",
     "arguments": { "type": "user", "id": "netdev1",
                    "dnssearch": [ { "str": "example.org" } ] } }
<- { "return": {} }
```

`netdev_del` (Command)

Remove a network backend.

Arguments

`id`: `string`
: the name of the network backend to remove

Errors

- If `id` is not a valid network backend, DeviceNotFound

Since

0.14

Example

```
-> { "execute": "netdev_del", "arguments": { "id": "netdev1" } }
<- { "return": {} }
```

NetLegacyNicOptions (Object)

Create a new Network Interface Card.

Members

netdev: string (optional)
id of -netdev to connect to

macaddr: string (optional)
MAC address

model: string (optional)
device model (e1000, rtl8139, virtio etc.)

addr: string (optional)
PCI device address

vectors: int (optional)
number of MSI-x vectors, 0 to disable MSI-X

Since

1.2

String (Object)

A fat type wrapping 'str', to be embedded in lists.

Members

str: string
Not documented

Since

1.2

NetdevUserOptions (Object)

Use the user mode network stack which requires no administrator privilege to run.

Members

hostname: string (optional)
client hostname reported by the builtin DHCP server

restrict: boolean (optional)
isolate the guest from the host

ipv4: boolean (optional)
whether to support IPv4, default true for enabled (since 2.6)

ipv6: boolean (optional)
whether to support IPv6, default true for enabled (since 2.6)

ip: string (optional)
legacy parameter, use net= instead

net: string (optional)
IP network address that the guest will see, in the form addr[/netmask] The netmask is optional, and can be either in the form a.b.c.d or as a number of valid top-most bits. Default is 10.0.2.0/24.

host: string (optional)
guest-visible address of the host

tftp: string (optional)
root directory of the built-in TFTP server

bootfile: string (optional)
BOOTP filename, for use with tftp=

dhcpstart: string (optional)
the first of the 16 IPs the built-in DHCP server can assign

dns: string (optional)
guest-visible address of the virtual nameserver

dnssearch: array of String (optional)
list of DNS suffixes to search, passed as DHCP option to the guest

domainname: string (optional)
guest-visible domain name of the virtual nameserver (since 3.0)

ipv6-prefix: string (optional)
IPv6 network prefix (default is fec0::) (since 2.6). The network prefix is given in the usual hexadecimal IPv6 address notation.

ipv6-prefixlen: int (optional)
IPv6 network prefix length (default is 64) (since 2.6)

ipv6-host: string (optional)
guest-visible IPv6 address of the host (since 2.6)

ipv6-dns: string (optional)
guest-visible IPv6 address of the virtual nameserver (since 2.6)

smb: string (optional)
root directory of the built-in SMB server

smbserver: string (optional)
IP address of the built-in SMB server

hostfwd: array of String (optional)
redirect incoming TCP or UDP host connections to guest endpoints

guestfwd: array of String (optional)
forward guest TCP connections

tftp-server-name: string (optional)
RFC2132 "TFTP server name" string (Since 3.1)

Since

1.2

NetdevTapOptions (Object)

Used to configure a host TAP network interface backend.

Members

ifname: string (optional)
interface name

fd: string (optional)
file descriptor of an already opened tap

fds: string (optional)
multiple file descriptors of already opened multiqueue capable tap

script: string (optional)
script to initialize the interface

downscript: string (optional)
script to shut down the interface

br: string (optional)
bridge name (since 2.8)

helper: string (optional)
command to execute to configure bridge

sndbuf: int (optional)
send buffer limit. Understands [TGMKkb] suffixes.

vnet_hdr: boolean (optional)
enable the IFF_VNET_HDR flag on the tap interface

vhost: boolean (optional)
enable vhost-net network accelerator

vhostfd: string (optional)
file descriptor of an already opened vhost net device

vhostfds: string (optional)
file descriptors of multiple already opened vhost net devices

vhostforce: boolean (optional)
vhost on for non-MSIX virtio guests

queues: int (optional)
number of queues to be created for multiqueue capable tap

poll-us: int (optional)
maximum number of microseconds that could be spent on busy polling for tap (since 2.7)

Since

1.2

NetdevSocketOptions (Object)

Socket netdevs are used to establish a network connection to another QEMU virtual machine via a TCP socket.

Members

fd: string (optional)
file descriptor of an already opened socket

listen: string (optional)
port number, and optional hostname, to listen on

connect: string (optional)
port number, and optional hostname, to connect to

mcast: string (optional)
UDP multicast address and port number

localaddr: string (optional)
source address and port for multicast and udp packets

udp: string (optional)
UDP unicast address and port number

Since

1.2

NetdevL2TPv3Options (Object)

Configure an Ethernet over L2TPv3 tunnel.

Members

src: string
source address

dst: string
destination address

srcport: string (optional)
source port - mandatory for udp, optional for ip

dstport: string (optional)
destination port - mandatory for udp, optional for ip

ipv6: boolean (optional)
force the use of ipv6

udp: boolean (optional)
use the udp version of l2tpv3 encapsulation

cookie64: boolean (optional)
use 64 bit cookies

counter: boolean (optional)
have sequence counter

pincounter: boolean (optional)
pin sequence counter to zero - workaround for buggy implementations or networks with packet reorder

txcookie: int (optional)
32 or 64 bit transmit cookie

rxcookie: int (optional)
32 or 64 bit receive cookie

txsession: int
32 bit transmit session

rxsession: int (optional)
32 bit receive session - if not specified set to the same value as transmit

offset: int (optional)
additional offset - allows the insertion of additional application-specific data before the packet payload

Since

2.1

NetdevVdeOptions (Object)

Connect to a vde switch running on the host.

Members

sock: string (optional)
socket path

port: int (optional)
port number

group: string (optional)
group owner of socket

mode: int (optional)
permissions for socket

Since

1.2

NetdevBridgeOptions (Object)

Connect a host TAP network interface to a host bridge device.

Members

br: string (optional)
bridge name

helper: string (optional)
command to execute to configure bridge

Since

1.2

NetdevHubPortOptions (Object)

Connect two or more net clients through a software hub.

Members

hubid: int
hub identifier number

netdev: string (optional)
used to connect hub to a netdev instead of a device (since 2.12)

Since

1.2

NetdevNetmapOptions (Object)

Connect a client to a netmap-enabled NIC or to a VALE switch port

Members

ifname: string
Either the name of an existing network interface supported by netmap, or the name of a VALE port (created on the fly). A VALE port name is in the form 'valeXXX:YYY', where XXX and YYY are non-negative integers. XXX identifies a switch and YYY identifies a port of the switch. VALE ports having the same XXX are therefore connected to the same switch.

devname: string (optional)
path of the netmap device (default: '/dev/netmap').

Since

2.0

AFXDPMode (Enum)

Attach mode for a default XDP program

Values

skb
generic mode, no driver support necessary

native
DRV mode, program is attached to a driver, packets are passed to the socket without allocation of skb.

Since

8.2

If

CONFIG_AF_XDP

NetdevAFXDPOptions (Object)

AF_XDP network backend

Members

ifname: string
The name of an existing network interface.

mode: AFXDPMode (optional)
Attach mode for a default XDP program. If not specified, then 'native' will be tried first, then 'skb'.

force-copy: boolean (optional)
Force XDP copy mode even if device supports zero-copy. (default: false)

queues: int (optional)
number of queues to be used for multiqueue interfaces (default: 1).

start-queue: int (optional)
Use queues starting from this queue number (default: 0).

inhibit: boolean (optional)
Don't load a default XDP program, use one already loaded to the interface (default: false). Requires sock-fds.

sock-fds: string (optional)
A colon (:) separated list of file descriptors for already open but not bound AF_XDP sockets in the queue order. One fd per queue. These descriptors should already be added into XDP socket map for corresponding queues. Requires inhibit.

Since

8.2

If

CONFIG_AF_XDP

NetdevVhostUserOptions (Object)

Vhost-user network backend

Members

chardev: string
name of a unix socket chardev

vhostforce: boolean (optional)
vhost on for non-MSIX virtio guests (default: false).

queues: int (optional)
number of queues to be created for multiqueue vhost-user (default: 1) (Since 2.5)

Since

2.1

NetdevVhostVDPAOptions (Object)

Vhost-vdpa network backend

vDPA device is a device that uses a datapath which complies with the virtio specifications with a vendor specific control path.

Members

vhostdev: string (optional)
path of vhost-vdpa device (default:'/dev/vhost-vdpa-0')

vhostfd: string (optional)
file descriptor of an already opened vhost vdpa device

queues: int (optional)
number of queues to be created for multiqueue vhost-vdpa (default: 1)

x-svq: boolean (optional)
Start device with (experimental) shadow virtqueue. (Since 7.1) (default: false)

Features

unstable
Member `x-svq` is experimental.

Since

5.1

NetdevVmnetHostOptions (Object)

vmnet (host mode) network backend.

Allows the vmnet interface to communicate with other vmnet interfaces that are in host mode and also with the host.

Members

start-address: string (optional)
The starting IPv4 address to use for the interface. Must be in the private IP range (RFC 1918). Must be specified along with `end-address` and `subnet-mask`. This address is used as the gateway address. The subsequent address up to and including end-address are placed in the DHCP pool.

end-address: string (optional)
The DHCP IPv4 range end address to use for the interface. Must be in the private IP range (RFC 1918). Must be specified along with `start-address` and `subnet-mask`.

subnet-mask: string (optional)
The IPv4 subnet mask to use on the interface. Must be specified along with `start-address` and `subnet-mask`.

isolated: boolean (optional)
Enable isolation for this interface. Interface isolation ensures that vmnet interface is not able to communicate with any other vmnet interfaces. Only communication with host is allowed. Requires at least macOS Big Sur 11.0.

net-uuid: string (optional)
The identifier (UUID) to uniquely identify the isolated network vmnet interface should be added to. If set, no DHCP service is provided for this interface and network communication is allowed only with other interfaces added to this network identified by the UUID. Requires at least macOS Big Sur 11.0.

Since

7.1

If

CONFIG_VMNET

NetdevVmnetSharedOptions (Object)

vmnet (shared mode) network backend.

Allows traffic originating from the vmnet interface to reach the Internet through a network address translator (NAT). The vmnet interface can communicate with the host and with other shared mode interfaces on the same subnet. If no DHCP settings, subnet mask and IPv6 prefix specified, the interface can communicate with any of other interfaces in shared mode.

Members

start-address: string (optional)
The starting IPv4 address to use for the interface. Must be in the private IP range (RFC 1918). Must be specified along with `end-address` and `subnet-mask`. This address is used as the gateway address. The subsequent address up to and including end-address are placed in the DHCP pool.

end-address: string (optional)
The DHCP IPv4 range end address to use for the interface. Must be in the private IP range (RFC 1918). Must be specified along with `start-address` and `subnet-mask`.

subnet-mask: string (optional)
The IPv4 subnet mask to use on the interface. Must be specified along with `start-address` and `subnet-mask`.

isolated: boolean (optional)
Enable isolation for this interface. Interface isolation ensures that vmnet interface is not able to communicate with any other vmnet interfaces. Only communication with host is allowed. Requires at least macOS Big Sur 11.0.

nat66-prefix: string (optional)
The IPv6 prefix to use into guest network. Must be a unique local address i.e. start with fd00::/8 and have length of 64.

Since

7.1

If

CONFIG_VMNET

NetdevVmnetBridgedOptions (Object)

vmnet (bridged mode) network backend.

Bridges the vmnet interface with a physical network interface.

Members

ifname: string
The name of the physical interface to be bridged.

isolated: boolean (optional)
Enable isolation for this interface. Interface isolation ensures that vmnet interface is not able to communicate with any other vmnet interfaces. Only communication with host is allowed. Requires at least macOS Big Sur 11.0.

Since

7.1

If

CONFIG_VMNET

NetdevStreamOptions (Object)

Configuration info for stream socket netdev

Members

addr: SocketAddress
socket address to listen on (server=true) or connect to (server=false)

server: boolean (optional)
create server socket (default: false)

reconnect: int (optional)
For a client socket, if a socket is disconnected, then attempt a reconnect after the given number of seconds. Setting this to zero disables this function. (default: 0) (since 8.0)

Only SocketAddress types 'unix', 'inet' and 'fd' are supported.

Since

7.2

NetdevDgramOptions (Object)

Configuration info for datagram socket netdev.

Members

remote: SocketAddress (optional)
remote address

local: SocketAddress (optional)
local address

Only SocketAddress types 'unix', 'inet' and 'fd' are supported.

If remote address is present and it's a multicast address, local address is optional. Otherwise local address is required and remote address is optional.

Table 1: Valid parameters combination table

remote	local	okay?
absent	absent	no
absent	not fd	no
absent	fd	yes
multicast	absent	yes
multicast	present	yes
not multicast	absent	no
not multicast	present	yes

Since

7.2

NetClientDriver (Enum)

Available netdev drivers.

Values

l2tpv3
since 2.1

vhost-vdpa
since 5.1

vmnet-host (If: CONFIG_VMNET)
since 7.1

vmnet-shared (**If:** CONFIG_VMNET)
: since 7.1

vmnet-bridged (**If:** CONFIG_VMNET)
: since 7.1

stream
: since 7.2

dgram
: since 7.2

af-xdp (**If:** CONFIG_AF_XDP)
: since 8.2

none
: Not documented

nic
: Not documented

user
: Not documented

tap
: Not documented

socket
: Not documented

vde
: Not documented

bridge
: Not documented

hubport
: Not documented

netmap
: Not documented

vhost-user
: Not documented

Since

2.7

Netdev (Object)

Captures the configuration of a network device.

Members

id: string
identifier for monitor commands.

type: NetClientDriver
Specify the driver used for interpreting remaining arguments.

The members of NetLegacyNicOptions when type is "nic"
The members of NetdevUserOptions when type is "user"
The members of NetdevTapOptions when type is "tap"
The members of NetdevL2TPv3Options when type is "l2tpv3"
The members of NetdevSocketOptions when type is "socket"
The members of NetdevStreamOptions when type is "stream"
The members of NetdevDgramOptions when type is "dgram"
The members of NetdevVdeOptions when type is "vde"
The members of NetdevBridgeOptions when type is "bridge"
The members of NetdevHubPortOptions when type is "hubport"
The members of NetdevNetmapOptions when type is "netmap"
The members of NetdevAFXDPOptions when type is "af-xdp" (If: CONFIG_AF_XDP)
The members of NetdevVhostUserOptions when type is "vhost-user"
The members of NetdevVhostVDPAOptions when type is "vhost-vdpa"
The members of NetdevVmnetHostOptions when type is "vmnet-host" (If: CONFIG_VMNET)
The members of NetdevVmnetSharedOptions when type is "vmnet-shared" (If: CONFIG_VMNET)
The members of NetdevVmnetBridgedOptions when type is "vmnet-bridged" (If: CONFIG_VMNET)

Since

1.2

RxState (Enum)

Packets receiving state

Values

normal
filter assigned packets according to the mac-table

none
don't receive any assigned packet

all
receive all assigned packets

Since

1.6

RxFilterInfo (Object)

Rx-filter information for a NIC.

Members

name: string
net client name

promiscuous: boolean
whether promiscuous mode is enabled

multicast: RxState
multicast receive state

unicast: RxState
unicast receive state

vlan: RxState
vlan receive state (Since 2.0)

broadcast-allowed: boolean
whether to receive broadcast

multicast-overflow: boolean
multicast table is overflowed or not

unicast-overflow: boolean
unicast table is overflowed or not

main-mac: string
the main macaddr string

vlan-table: array of int
a list of active vlan id

unicast-table: array of string
a list of unicast macaddr string

multicast-table: array of string
a list of multicast macaddr string

Since

1.6

`query-rx-filter` (Command)

Return rx-filter information for all NICs (or for the given NIC).

Arguments

`name: string` (optional)
net client name

Returns

list of `RxFilterInfo` for all NICs (or for the given NIC).

Errors

- if the given `name` doesn't exist
- if the given NIC doesn't support rx-filter querying
- if the given net client isn't a NIC

Since

1.6

Example

```
-> { "execute": "query-rx-filter", "arguments": { "name": "vnet0" } }
<- { "return": [
        {
            "promiscuous": true,
            "name": "vnet0",
            "main-mac": "52:54:00:12:34:56",
            "unicast": "normal",
            "vlan": "normal",
            "vlan-table": [
                4,
                0
            ],
            "unicast-table": [
            ],
            "multicast": "normal",
            "multicast-overflow": false,
            "unicast-overflow": false,
            "multicast-table": [
                "01:00:5e:00:00:01",
                "33:33:00:00:00:01",
                "33:33:ff:12:34:56"
            ],
```

(continues on next page)

(continued from previous page)

```
            "broadcast-allowed": false
        }
      ]
  }
```

NIC_RX_FILTER_CHANGED (Event)

Emitted once until the 'query-rx-filter' command is executed, the first event will always be emitted

Arguments

name: string (optional)
net client name

path: string
device path

Since

1.6

Example

```
<- { "event": "NIC_RX_FILTER_CHANGED",
     "data": { "name": "vnet0",
               "path": "/machine/peripheral/vnet0/virtio-backend" },
     "timestamp": { "seconds": 1368697518, "microseconds": 326866 } }
```

AnnounceParameters (Object)

Parameters for self-announce timers

Members

initial: int
Initial delay (in ms) before sending the first GARP/RARP announcement

max: int
Maximum delay (in ms) between GARP/RARP announcement packets

rounds: int
Number of self-announcement attempts

step: int
Delay increase (in ms) after each self-announcement attempt

interfaces: array of string (optional)
An optional list of interface names, which restricts the announcement to the listed interfaces. (Since 4.1)

id: string (optional)
A name to be used to identify an instance of announce-timers and to allow it to modified later. Not for use as part of the migration parameters. (Since 4.1)

Since

4.0

announce-self (Command)

Trigger generation of broadcast RARP frames to update network switches. This can be useful when network bonds fail-over the active slave.

Arguments

The members of AnnounceParameters

Example

```
-> { "execute": "announce-self",
     "arguments": {
         "initial": 50, "max": 550, "rounds": 10, "step": 50,
         "interfaces": ["vn2", "vn3"], "id": "bob" } }
<- { "return": {} }
```

Since

4.0

FAILOVER_NEGOTIATED (Event)

Emitted when VIRTIO_NET_F_STANDBY was enabled during feature negotiation. Failover primary devices which were hidden (not hotplugged when requested) before will now be hotplugged by the virtio-net standby device.

Arguments

device-id: string
QEMU device id of the unplugged device

Since

4.2

Example

```
<- { "event": "FAILOVER_NEGOTIATED",
     "data": { "device-id": "net1" },
     "timestamp": { "seconds": 1368697518, "microseconds": 326866 } }
```

NETDEV_STREAM_CONNECTED (Event)

Emitted when the netdev stream backend is connected

Arguments

netdev-id: string
: QEMU netdev id that is connected

addr: SocketAddress
: The destination address

Since

7.2

Examples

```
<- { "event": "NETDEV_STREAM_CONNECTED",
     "data": { "netdev-id": "netdev0",
               "addr": { "port": "47666", "ipv6": true,
                         "host": "::1", "type": "inet" } },
     "timestamp": { "seconds": 1666269863, "microseconds": 311222 } }

<- { "event": "NETDEV_STREAM_CONNECTED",
     "data": { "netdev-id": "netdev0",
               "addr": { "path": "/tmp/qemu0", "type": "unix" } },
     "timestamp": { "seconds": 1666269706, "microseconds": 413651 } }
```

NETDEV_STREAM_DISCONNECTED (Event)

Emitted when the netdev stream backend is disconnected

Arguments

netdev-id: string
QEMU netdev id that is disconnected

Since

7.2

Example

```
<- { 'event': 'NETDEV_STREAM_DISCONNECTED',
     'data': {'netdev-id': 'netdev0'},
     'timestamp': {'seconds': 1663330937, 'microseconds': 526695} }
```

5.11.12 eBPF Objects

eBPF object is an ELF binary that contains the eBPF program and eBPF map description(BTF). Overall, eBPF object should contain the program and enough metadata to create/load eBPF with libbpf. As the eBPF maps/program should correspond to QEMU, the eBPF can't be used from different QEMU build.

Currently, there is a possible eBPF for receive-side scaling (RSS).

EbpfObject (Object)

An eBPF ELF object.

Members

object: string
the eBPF object encoded in base64

Since

9.0

If

CONFIG_EBPF

EbpfProgramID (Enum)

The eBPF programs that can be gotten with request-ebpf.

Values

rss
: Receive side scaling, technology that allows steering traffic between queues by calculation hash. Users may set up indirection table and hash/packet types configurations. Used with virtio-net.

Since

9.0

If

CONFIG_EBPF

request-ebpf (Command)

Retrieve an eBPF object that can be loaded with libbpf. Management applications (e.g. libvirt) may load it and pass file descriptors to QEMU, so they can run running QEMU without BPF capabilities.

Arguments

id: EbpfProgramID
: The ID of the program to return.

Returns

eBPF object encoded in base64.

Since

9.0

If

CONFIG_EBPF

5.11.13 Rocker switch device

RockerSwitch (Object)

Rocker switch information.

Members

name: string
switch name

id: int
switch ID

ports: int
number of front-panel ports

Since

2.4

query-rocker (Command)

Return rocker switch information.

Arguments

name: string
Not documented

Returns

Rocker information

Since

2.4

Example

```
-> { "execute": "query-rocker", "arguments": { "name": "sw1" } }
<- { "return": {"name": "sw1", "ports": 2, "id": 1327446905938}}
```

RockerPortDuplex (Enum)

An eumeration of port duplex states.

Values

half
: half duplex

full
: full duplex

Since

2.4

RockerPortAutoneg (Enum)

An eumeration of port autoneg states.

Values

off
: autoneg is off

on
: autoneg is on

Since

2.4

RockerPort (Object)

Rocker switch port information.

Members

name: string
port name

enabled: boolean
port is enabled for I/O

link-up: boolean
physical link is UP on port

speed: int
port link speed in Mbps

duplex: RockerPortDuplex
port link duplex

autoneg: RockerPortAutoneg
port link autoneg

Since

2.4

query-rocker-ports (Command)

Return rocker switch port information.

Arguments

name: string
Not documented

Returns

a list of `RockerPort` information

Since

2.4

Example

```
-> { "execute": "query-rocker-ports", "arguments": { "name": "sw1" } }
<- { "return": [ {"duplex": "full", "enabled": true, "name": "sw1.1",
                  "autoneg": "off", "link-up": true, "speed": 10000},
                 {"duplex": "full", "enabled": true, "name": "sw1.2",
                  "autoneg": "off", "link-up": true, "speed": 10000}
   ]}
```

RockerOfDpaFlowKey (Object)

Rocker switch OF-DPA flow key

Members

priority: int
key priority, 0 being lowest priority

tbl-id: int
flow table ID

in-pport: int (optional)
physical input port

tunnel-id: int (optional)
tunnel ID

vlan-id: int (optional)
VLAN ID

eth-type: int (optional)
Ethernet header type

eth-src: string (optional)
Ethernet header source MAC address

eth-dst: string (optional)
Ethernet header destination MAC address

ip-proto: int (optional)
IP Header protocol field

ip-tos: int (optional)
IP header TOS field

ip-dst: string (optional)
IP header destination address

Note

optional members may or may not appear in the flow key depending if they're relevant to the flow key.

Since

2.4

RockerOfDpaFlowMask (Object)

Rocker switch OF-DPA flow mask

Members

in-pport: int (optional)
physical input port

tunnel-id: int (optional)
tunnel ID

vlan-id: int (optional)
VLAN ID

eth-src: string (optional)
Ethernet header source MAC address

eth-dst: string (optional)
Ethernet header destination MAC address

ip-proto: int (optional)
IP Header protocol field

ip-tos: int (optional)
IP header TOS field

Note

optional members may or may not appear in the flow mask depending if they're relevant to the flow mask.

Since

2.4

RockerOfDpaFlowAction (Object)

Rocker switch OF-DPA flow action

Members

goto-tbl: int (optional)
next table ID

group-id: int (optional)
group ID

tunnel-lport: int (optional)
tunnel logical port ID

vlan-id: int (optional)
VLAN ID

new-vlan-id: int (optional)
new VLAN ID

out-pport: int (optional)
physical output port

Note

optional members may or may not appear in the flow action depending if they're relevant to the flow action.

Since

2.4

RockerOfDpaFlow (Object)

Rocker switch OF-DPA flow

Members

cookie: int
flow unique cookie ID

hits: int
count of matches (hits) on flow

key: RockerOfDpaFlowKey
flow key

mask: RockerOfDpaFlowMask
flow mask

action: RockerOfDpaFlowAction
flow action

Since

2.4

query-rocker-of-dpa-flows (Command)

Return rocker OF-DPA flow information.

Arguments

name: string
switch name

tbl-id: int (optional)
flow table ID. If tbl-id is not specified, returns flow information for all tables.

Returns

rocker OF-DPA flow information

Since

2.4

Example

```
-> { "execute": "query-rocker-of-dpa-flows",
     "arguments": { "name": "sw1" } }
<- { "return": [ {"key": {"in-pport": 0, "priority": 1, "tbl-id": 0},
                  "hits": 138,
                  "cookie": 0,
                  "action": {"goto-tbl": 10},
                  "mask": {"in-pport": 4294901760}
                 },
                 {...more...},
   ]}
```

RockerOfDpaGroup (Object)

Rocker switch OF-DPA group

Members

id: int
group unique ID

type: int
group type

vlan-id: int (optional)
VLAN ID

pport: int (optional)
physical port number

index: int (optional)
group index, unique with group type

out-pport: int (optional)
output physical port number

group-id: int (optional)
next group ID

set-vlan-id: int (optional)
VLAN ID to set

pop-vlan: int (optional)
pop VLAN headr from packet

group-ids: array of int (optional)
list of next group IDs

set-eth-src: string (optional)
set source MAC address in Ethernet header

set-eth-dst: string (optional)
set destination MAC address in Ethernet header

ttl-check: int (optional)
perform TTL check

Note

optional members may or may not appear in the group depending if they're relevant to the group type.

Since

2.4

query-rocker-of-dpa-groups (Command)

Return rocker OF-DPA group information.

Arguments

name: string
switch name

type: int (optional)
group type. If type is not specified, returns group information for all group types.

Returns

rocker OF-DPA group information

Since

2.4

Example

```
-> { "execute": "query-rocker-of-dpa-groups",
     "arguments": { "name": "sw1" } }
<- { "return": [ {"type": 0, "out-pport": 2,
                  "pport": 2, "vlan-id": 3841,
                  "pop-vlan": 1, "id": 251723778},
                 {"type": 0, "out-pport": 0,
                  "pport": 0, "vlan-id": 3841,
                  "pop-vlan": 1, "id": 251723776},
                 {"type": 0, "out-pport": 1,
                  "pport": 1, "vlan-id": 3840,
                  "pop-vlan": 1, "id": 251658241},
                 {"type": 0, "out-pport": 0,
                  "pport": 0, "vlan-id": 3840,
                  "pop-vlan": 1, "id": 251658240}
   ]}
```

5.11.14 TPM (trusted platform module) devices

`TpmModel` (Enum)

An enumeration of TPM models

Values

`tpm-tis`
: TPM TIS model

`tpm-crb`
: TPM CRB model (since 2.12)

`tpm-spapr`
: TPM SPAPR model (since 5.0)

Since

1.5

If

CONFIG_TPM

query-tpm-models (Command)

Return a list of supported TPM models

Returns

a list of TpmModel

Since

1.5

Example

```
-> { "execute": "query-tpm-models" }
<- { "return": [ "tpm-tis", "tpm-crb", "tpm-spapr" ] }
```

If

CONFIG_TPM

TpmType (Enum)

An enumeration of TPM types

Values

passthrough
: TPM passthrough type

emulator
: Software Emulator TPM type (since 2.11)

Since

1.5

If

CONFIG_TPM

query-tpm-types (Command)

Return a list of supported TPM types

Returns

a list of TpmType

Since

1.5

Example

```
-> { "execute": "query-tpm-types" }
<- { "return": [ "passthrough", "emulator" ] }
```

If

CONFIG_TPM

TPMPassthroughOptions (Object)

Information about the TPM passthrough type

Members

path: string (optional)
: string describing the path used for accessing the TPM device

cancel-path: string (optional)
: string showing the TPM's sysfs cancel file for cancellation of TPM commands while they are executing

Since

1.5

If

CONFIG_TPM

TPMEmulatorOptions (Object)

Information about the TPM emulator type

Members

chardev: string
Name of a unix socket chardev

Since

2.11

If

CONFIG_TPM

TPMPassthroughOptionsWrapper (Object)

Members

data: TPMPassthroughOptions
Information about the TPM passthrough type

Since

1.5

If

CONFIG_TPM

TPMEmulatorOptionsWrapper (Object)

Members

data: TPMEmulatorOptions
Information about the TPM emulator type

Since

2.11

If

CONFIG_TPM

TpmTypeOptions (Object)

A union referencing different TPM backend types' configuration options

Members

type: TpmType

- 'passthrough' The configuration options for the TPM passthrough type
- 'emulator' The configuration options for TPM emulator backend type

The members of TPMPassthroughOptionsWrapper when type is "passthrough"
The members of TPMEmulatorOptionsWrapper when type is "emulator"

Since

1.5

If

CONFIG_TPM

TPMInfo (Object)

Information about the TPM

Members

id: string
The Id of the TPM

model: TpmModel
The TPM frontend model

options: TpmTypeOptions
The TPM (backend) type configuration options

Since

1.5

If

CONFIG_TPM

query-tpm (Command)

Return information about the TPM device

Since

1.5

Example

```
-> { "execute": "query-tpm" }
<- { "return":
     [
       { "model": "tpm-tis",
         "options":
           { "type": "passthrough",
             "data":
               { "cancel-path": "/sys/class/misc/tpm0/device/cancel",
                 "path": "/dev/tpm0"
               }
           },
         "id": "tpm0"
       }
     ]
   }
```

If

`CONFIG_TPM`

5.11.15 Remote desktop

`DisplayProtocol` (Enum)

Display protocols which support changing password options.

Values

`vnc`
Not documented

`spice`
Not documented

Since

7.0

`SetPasswordAction` (Enum)

An action to take on changing a password on a connection with active clients.

Values

`keep`
maintain existing clients

`fail`
fail the command if clients are connected

`disconnect`
disconnect existing clients

Since

7.0

SetPasswordOptions (Object)

Options for set_password.

Members

protocol: DisplayProtocol

- 'vnc' to modify the VNC server password
- 'spice' to modify the Spice server password

password: string
the new password

connected: SetPasswordAction (optional)
How to handle existing clients when changing the password. If nothing is specified, defaults to 'keep'. For VNC, only 'keep' is currently implemented.

The members of SetPasswordOptionsVnc when protocol is "vnc"

Since

7.0

SetPasswordOptionsVnc (Object)

Options for set_password specific to the VNC protocol.

Members

display: string (optional)
The id of the display where the password should be changed. Defaults to the first.

Since

7.0

set_password (Command)

Set the password of a remote display server.

Arguments

The members of `SetPasswordOptions`

Errors

- If Spice is not enabled, DeviceNotFound

Since

0.14

Example

```
-> { "execute": "set_password", "arguments": { "protocol": "vnc",
                                               "password": "secret" } }
<- { "return": {} }
```

`ExpirePasswordOptions` (Object)

General options for expire_password.

Members

`protocol`: `DisplayProtocol`

- 'vnc' to modify the VNC server expiration
- 'spice' to modify the Spice server expiration

`time`: `string`
when to expire the password.

- 'now' to expire the password immediately
- 'never' to cancel password expiration
- '+INT' where INT is the number of seconds from now (integer)
- 'INT' where INT is the absolute time in seconds

The members of `ExpirePasswordOptionsVnc` when `protocol` is `"vnc"`

Notes

Time is relative to the server and currently there is no way to coordinate server time with client time. It is not recommended to use the absolute time version of the `time` parameter unless you're sure you are on the same machine as the QEMU instance.

Since

7.0

`ExpirePasswordOptionsVnc` (Object)

Options for expire_password specific to the VNC protocol.

Members

`display`: `string` (optional)
The id of the display where the expiration should be changed. Defaults to the first.

Since

7.0

`expire_password` (Command)

Expire the password of a remote display server.

Arguments

The members of `ExpirePasswordOptions`

Errors

- If `protocol` is 'spice' and Spice is not active, DeviceNotFound

Since

0.14

Example

```
-> { "execute": "expire_password", "arguments": { "protocol": "vnc",
                                                  "time": "+60" } }
<- { "return": {} }
```

ImageFormat (Enum)

Supported image format types.

Values

png
PNG format

ppm
PPM format

Since

7.1

screendump (Command)

Capture the contents of a screen and write it to a file.

Arguments

filename: string
the path of a new file to store the image

device: string (optional)
ID of the display device that should be dumped. If this parameter is missing, the primary display will be used. (Since 2.12)

head: int (optional)
head to use in case the device supports multiple heads. If this parameter is missing, head #0 will be used. Also note that the head can only be specified in conjunction with the device ID. (Since 2.12)

format: ImageFormat (optional)
image format for screendump. (default: ppm) (Since 7.1)

Since

0.14

Example

```
-> { "execute": "screendump",
     "arguments": { "filename": "/tmp/image" } }
<- { "return": {} }
```

If

CONFIG_PIXMAN

Spice

SpiceBasicInfo (Object)

The basic information for SPICE network connection

Members

host: string
IP address

port: string
port number

family: NetworkAddressFamily
address family

Since

2.1

If

CONFIG_SPICE

SpiceServerInfo (Object)

Information about a SPICE server

Members

auth: string (optional)
authentication method

The members of SpiceBasicInfo

Since

2.1

If

CONFIG_SPICE

SpiceChannel (Object)

Information about a SPICE client channel.

Members

connection-id: int
SPICE connection id number. All channels with the same id belong to the same SPICE session.

channel-type: int
SPICE channel type number. "1" is the main control channel, filter for this one if you want to track spice sessions only

channel-id: int
SPICE channel ID number. Usually "0", might be different when multiple channels of the same type exist, such as multiple display channels in a multihead setup

tls: boolean
true if the channel is encrypted, false otherwise.

The members of SpiceBasicInfo

Since

0.14

If

CONFIG_SPICE

SpiceQueryMouseMode (Enum)

An enumeration of Spice mouse states.

Values

client
: Mouse cursor position is determined by the client.

server
: Mouse cursor position is determined by the server.

unknown
: No information is available about mouse mode used by the spice server.

Note

spice/enums.h has a SpiceMouseMode already, hence the name.

Since

1.1

If

CONFIG_SPICE

SpiceInfo (Object)

Information about the SPICE session.

Members

enabled: boolean
true if the SPICE server is enabled, false otherwise

migrated: boolean
true if the last guest migration completed and spice migration had completed as well, false otherwise (since 1.4)

host: string (optional)
The hostname the SPICE server is bound to. This depends on the name resolution on the host and may be an IP address.

port: int (optional)
The SPICE server's port number.

compiled-version: string (optional)
SPICE server version.

tls-port: int (optional)
The SPICE server's TLS port number.

auth: string (optional)
the current authentication type used by the server

- 'none' if no authentication is being used
- 'spice' uses SASL or direct TLS authentication, depending on command line options

mouse-mode: SpiceQueryMouseMode
The mode in which the mouse cursor is displayed currently. Can be determined by the client or the server, or unknown if spice server doesn't provide this information. (since: 1.1)

channels: array of SpiceChannel (optional)
a list of `SpiceChannel` for each active spice channel

Since

0.14

If

`CONFIG_SPICE`

query-spice (Command)

Returns information about the current SPICE server

Returns

SpiceInfo

Since

0.14

Example

```
-> { "execute": "query-spice" }
<- { "return": {
         "enabled": true,
         "auth": "spice",
         "port": 5920,
         "migrated":false,
         "tls-port": 5921,
         "host": "0.0.0.0",
         "mouse-mode":"client",
         "channels": [
            {
               "port": "54924",
               "family": "ipv4",
               "channel-type": 1,
               "connection-id": 1804289383,
               "host": "127.0.0.1",
               "channel-id": 0,
               "tls": true
            },
            {
               "port": "36710",
               "family": "ipv4",
               "channel-type": 4,
               "connection-id": 1804289383,
               "host": "127.0.0.1",
               "channel-id": 0,
               "tls": false
            },
            [ ... more channels follow ... ]
         ]
      }
   }
```

If

CONFIG_SPICE

SPICE_CONNECTED (Event)

Emitted when a SPICE client establishes a connection

Arguments

server: SpiceBasicInfo
server information

client: SpiceBasicInfo
client information

Since

0.14

Example

```
<- { "timestamp": {"seconds": 1290688046, "microseconds": 388707},
     "event": "SPICE_CONNECTED",
     "data": {
       "server": { "port": "5920", "family": "ipv4", "host": "127.0.0.1"},
       "client": {"port": "52873", "family": "ipv4", "host": "127.0.0.1"}
   }}
```

If

CONFIG_SPICE

SPICE_INITIALIZED (Event)

Emitted after initial handshake and authentication takes place (if any) and the SPICE channel is up and running

Arguments

server: SpiceServerInfo
server information

client: SpiceChannel
client information

Since

0.14

Example

```
<- { "timestamp": {"seconds": 1290688046, "microseconds": 417172},
     "event": "SPICE_INITIALIZED",
     "data": {"server": {"auth": "spice", "port": "5921",
                         "family": "ipv4", "host": "127.0.0.1"},
              "client": {"port": "49004", "family": "ipv4", "channel-type": 3,
                         "connection-id": 1804289383, "host": "127.0.0.1",
                         "channel-id": 0, "tls": true}
   }}
```

If

CONFIG_SPICE

SPICE_DISCONNECTED (Event)

Emitted when the SPICE connection is closed

Arguments

server: SpiceBasicInfo
server information

client: SpiceBasicInfo
client information

Since

0.14

Example

```
<- { "timestamp": {"seconds": 1290688046, "microseconds": 388707},
     "event": "SPICE_DISCONNECTED",
     "data": {
       "server": { "port": "5920", "family": "ipv4", "host": "127.0.0.1"},
       "client": {"port": "52873", "family": "ipv4", "host": "127.0.0.1"}
   }}
```

If

CONFIG_SPICE

SPICE_MIGRATE_COMPLETED (Event)

Emitted when SPICE migration has completed

Since

1.3

Example

```
<- { "timestamp": {"seconds": 1290688046, "microseconds": 417172},
     "event": "SPICE_MIGRATE_COMPLETED" }
```

If

CONFIG_SPICE

VNC

VncBasicInfo (Object)

The basic information for vnc network connection

Members

host: string
IP address

service: string
The service name of the vnc port. This may depend on the host system's service database so symbolic names should not be relied on.

family: NetworkAddressFamily
address family

websocket: boolean
true in case the socket is a websocket (since 2.3).

Since

2.1

If

CONFIG_VNC

VncServerInfo (Object)

The network connection information for server

Members

auth: string (optional)
authentication method used for the plain (non-websocket) VNC server

The members of VncBasicInfo

Since

2.1

If

CONFIG_VNC

VncClientInfo (Object)

Information about a connected VNC client.

Members

x509_dname: string (optional)
If x509 authentication is in use, the Distinguished Name of the client.

sasl_username: string (optional)
If SASL authentication is in use, the SASL username used for authentication.

The members of VncBasicInfo

Since

0.14

If

CONFIG_VNC

VncInfo (Object)

Information about the VNC session.

Members

enabled: boolean
true if the VNC server is enabled, false otherwise

host: string (optional)
The hostname the VNC server is bound to. This depends on the name resolution on the host and may be an IP address.

family: NetworkAddressFamily (optional)

- 'ipv6' if the host is listening for IPv6 connections
- 'ipv4' if the host is listening for IPv4 connections
- 'unix' if the host is listening on a unix domain socket
- 'unknown' otherwise

service: string (optional)
The service name of the server's port. This may depends on the host system's service database so symbolic names should not be relied on.

auth: string (optional)
the current authentication type used by the server

- 'none' if no authentication is being used
- 'vnc' if VNC authentication is being used
- 'vencrypt+plain' if VEncrypt is used with plain text authentication
- 'vencrypt+tls+none' if VEncrypt is used with TLS and no authentication
- 'vencrypt+tls+vnc' if VEncrypt is used with TLS and VNC authentication
- 'vencrypt+tls+plain' if VEncrypt is used with TLS and plain text auth
- 'vencrypt+x509+none' if VEncrypt is used with x509 and no auth
- 'vencrypt+x509+vnc' if VEncrypt is used with x509 and VNC auth
- 'vencrypt+x509+plain' if VEncrypt is used with x509 and plain text auth
- 'vencrypt+tls+sasl' if VEncrypt is used with TLS and SASL auth
- 'vencrypt+x509+sasl' if VEncrypt is used with x509 and SASL auth

clients: array of VncClientInfo (optional)
a list of VncClientInfo of all currently connected clients

Since

0.14

If

CONFIG_VNC

VncPrimaryAuth (Enum)

vnc primary authentication method.

Values

none
Not documented

vnc
Not documented

ra2
Not documented

ra2ne
Not documented

tight
Not documented

ultra
Not documented

tls
Not documented

vencrypt
Not documented

sasl
Not documented

Since

2.3

If

CONFIG_VNC

VncVencryptSubAuth (Enum)

vnc sub authentication method with vencrypt.

Values

plain
Not documented

tls-none
Not documented

x509-none
Not documented

tls-vnc
Not documented

x509-vnc
Not documented

tls-plain
Not documented

x509-plain
Not documented

tls-sasl
Not documented

x509-sasl
Not documented

Since

2.3

If

CONFIG_VNC

VncServerInfo2 (Object)

The network connection information for server

Members

auth: VncPrimaryAuth
The current authentication type used by the servers

vencrypt: VncVencryptSubAuth (optional)
The vencrypt sub authentication type used by the servers, only specified in case auth == vencrypt.

The members of VncBasicInfo

Since

2.9

If

CONFIG_VNC

VncInfo2 (Object)

Information about a vnc server

Members

id: string
vnc server name.

server: array of VncServerInfo2
A list of VncBasincInfo describing all listening sockets. The list can be empty (in case the vnc server is disabled). It also may have multiple entries: normal + websocket, possibly also ipv4 + ipv6 in the future.

clients: array of VncClientInfo
A list of VncClientInfo of all currently connected clients. The list can be empty, for obvious reasons.

auth: VncPrimaryAuth
The current authentication type used by the non-websockets servers

vencrypt: VncVencryptSubAuth (optional)
The vencrypt authentication type used by the servers, only specified in case auth == vencrypt.

display: string (optional)
The display device the vnc server is linked to.

Since

2.3

If

CONFIG_VNC

query-vnc (Command)

Returns information about the current VNC server

Returns

VncInfo

Since

0.14

Example

```
-> { "execute": "query-vnc" }
<- { "return": {
         "enabled":true,
         "host":"0.0.0.0",
         "service":"50402",
         "auth":"vnc",
         "family":"ipv4",
         "clients":[
            {
               "host":"127.0.0.1",
               "service":"50401",
               "family":"ipv4",
               "websocket":false
            }
         ]
      }
   }
```

If

`CONFIG_VNC`

`query-vnc-servers` (Command)

Returns a list of vnc servers. The list can be empty.

Returns

a list of `VncInfo2`

Since

2.3

If

`CONFIG_VNC`

`change-vnc-password` (Command)

Change the VNC server password.

Arguments

`password`: `string`
: the new password to use with VNC authentication

Since

1.1

Notes

An empty password in this command will set the password to the empty string. Existing clients are unaffected by executing this command.

If

CONFIG_VNC

VNC_CONNECTED (Event)

Emitted when a VNC client establishes a connection

Arguments

server: VncServerInfo
server information

client: VncBasicInfo
client information

Note

This event is emitted before any authentication takes place, thus the authentication ID is not provided

Since

0.13

Example

```
<- { "event": "VNC_CONNECTED",
     "data": {
           "server": { "auth": "sasl", "family": "ipv4", "websocket": false,
                       "service": "5901", "host": "0.0.0.0" },
           "client": { "family": "ipv4", "service": "58425",
                       "host": "127.0.0.1", "websocket": false } },
     "timestamp": { "seconds": 1262976601, "microseconds": 975795 } }
```

If

CONFIG_VNC

VNC_INITIALIZED (Event)

Emitted after authentication takes place (if any) and the VNC session is made active

Arguments

server: VncServerInfo
server information

client: VncClientInfo
client information

Since

0.13

Example

```
<-  { "event": "VNC_INITIALIZED",
      "data": {
           "server": { "auth": "sasl", "family": "ipv4", "websocket": false,
                       "service": "5901", "host": "0.0.0.0"},
           "client": { "family": "ipv4", "service": "46089", "websocket": false,
                       "host": "127.0.0.1", "sasl_username": "luiz" } },
      "timestamp": { "seconds": 1263475302, "microseconds": 150772 } }
```

If

CONFIG_VNC

VNC_DISCONNECTED (Event)

Emitted when the connection is closed

Arguments

server: VncServerInfo
server information

client: VncClientInfo
client information

Since

0.13

Example

```
<- { "event": "VNC_DISCONNECTED",
     "data": {
           "server": { "auth": "sasl", "family": "ipv4", "websocket": false,
                       "service": "5901", "host": "0.0.0.0" },
           "client": { "family": "ipv4", "service": "58425", "websocket": false,
                       "host": "127.0.0.1", "sasl_username": "luiz" } },
     "timestamp": { "seconds": 1262976601, "microseconds": 975795 } }
```

If

CONFIG_VNC

5.11.16 Input

MouseInfo (Object)

Information about a mouse device.

Members

name: string
: the name of the mouse device

index: int
: the index of the mouse device

current: boolean
: true if this device is currently receiving mouse events

absolute: boolean
: true if this device supports absolute coordinates as input

Since

0.14

query-mice (Command)

Returns information about each active mouse device

Returns

a list of MouseInfo for each device

Since

0.14

Example

```
-> { "execute": "query-mice" }
<- { "return": [
          {
             "name":"QEMU Microsoft Mouse",
             "index":0,
             "current":false,
             "absolute":false
          },
          {
             "name":"QEMU PS/2 Mouse",
             "index":1,
             "current":true,
             "absolute":true
          }
       ]
   }
```

QKeyCode (Enum)

An enumeration of key name.

This is used by the send-key command.

Values

unmapped
since 2.0

pause
since 2.0

ro
since 2.4

kp_comma
since 2.4

kp_equals
since 2.6

power
since 2.6

hiragana
since 2.9

henkan
since 2.9

yen
since 2.9

sleep
since 2.10

wake
since 2.10

audionext
since 2.10

audioprev
since 2.10

audiostop
since 2.10

audioplay
since 2.10

audiomute
since 2.10

volumeup
since 2.10

volumedown
since 2.10

mediaselect
since 2.10

mail
since 2.10

calculator
since 2.10

computer
since 2.10

ac_home
since 2.10

ac_back
since 2.10

ac_forward
since 2.10

ac_refresh
since 2.10

ac_bookmarks
since 2.10

muhenkan
since 2.12

katakanahiragana
since 2.12

lang1
since 6.1

lang2
since 6.1

f13
since 8.0

f14
since 8.0

f15
since 8.0

f16
since 8.0

f17
since 8.0

f18
since 8.0

f19
since 8.0

f20
since 8.0

f21
since 8.0

f22
since 8.0

f23
since 8.0

f24
since 8.0

shift
Not documented

shift_r
Not documented

alt
Not documented

alt_r
Not documented

ctrl
Not documented

ctrl_r
Not documented

menu
Not documented

esc
Not documented

1
Not documented

2
Not documented

3
Not documented

4
Not documented

5
Not documented

6
Not documented

7
Not documented

8
Not documented

9
Not documented

0
Not documented

minus
Not documented

equal
Not documented

backspace
Not documented

tab
Not documented

q
Not documented

w
Not documented

e
Not documented

r
Not documented

t
Not documented

y
Not documented

u
Not documented

i
Not documented

o
Not documented

p
Not documented

bracket_left
Not documented

bracket_right
Not documented

ret
Not documented

a
Not documented

s
Not documented

d
Not documented

f
Not documented

g
Not documented

h
Not documented

j
Not documented

k
Not documented

l
Not documented

semicolon
Not documented

apostrophe
Not documented

grave_accent
Not documented

backslash
Not documented

z
Not documented

x
Not documented

c
Not documented

v
Not documented

b
Not documented

n
Not documented

m
Not documented

comma
Not documented

dot
Not documented

slash
Not documented

asterisk
Not documented

spc
Not documented

caps_lock
Not documented

f1
Not documented

f2
Not documented

f3
Not documented

f4
Not documented

f5
Not documented

f6
Not documented

f7
Not documented

f8
Not documented

f9
Not documented

f10
Not documented

num_lock
Not documented

scroll_lock
Not documented

kp_divide
Not documented

kp_multiply
Not documented

kp_subtract
Not documented

kp_add
Not documented

kp_enter
Not documented

kp_decimal
Not documented

sysrq
Not documented

kp_0
Not documented

kp_1
Not documented

kp_2
Not documented

kp_3
Not documented

kp_4
Not documented

kp_5
Not documented

kp_6
Not documented

kp_7
Not documented

kp_8
Not documented

kp_9
Not documented

less
Not documented

f11
Not documented

f12
Not documented

print
Not documented

home
Not documented

pgup
Not documented

pgdn
Not documented

end
Not documented

left
Not documented

up
Not documented

down
Not documented

right
Not documented

insert
Not documented

delete
Not documented

stop
Not documented

again
Not documented

props
Not documented

undo
Not documented

front
Not documented

copy
Not documented

open
Not documented

paste
Not documented

find
Not documented

cut
Not documented

lf
Not documented

help
Not documented

meta_l
Not documented

meta_r
Not documented

compose
Not documented

'sysrq' was mistakenly added to hack around the fact that the ps2 driver was not generating correct scancodes sequences when 'alt+print' was pressed. This flaw is now fixed and the 'sysrq' key serves no further purpose. Any further use of 'sysrq' will be transparently changed to 'print', so they are effectively synonyms.

Since

1.3

KeyValueKind (Enum)

Values

number
Not documented

qcode
Not documented

Since

1.3

IntWrapper (Object)

Members

data: int
a numeric key code

Since

1.3

QKeyCodeWrapper (Object)

Members

data: QKeyCode
An enumeration of key name

Since

1.3

KeyValue (Object)

Represents a keyboard key.

Members

type: KeyValueKind
key encoding

The members of IntWrapper when type is "number"
The members of QKeyCodeWrapper when type is "qcode"

Since

1.3

send-key (Command)

Send keys to guest.

Arguments

keys: array of KeyValue
An array of KeyValue elements. All KeyValues in this array are simultaneously sent to the guest. A KeyValue.number value is sent directly to the guest, while KeyValue.qcode must be a valid QKeyCode value

hold-time: int (optional)
time to delay key up events, milliseconds. Defaults to 100

Errors

- If key is unknown or redundant, GenericError

Since

1.3

Example

```
-> { "execute": "send-key",
     "arguments": { "keys": [ { "type": "qcode", "data": "ctrl" },
                              { "type": "qcode", "data": "alt" },
                              { "type": "qcode", "data": "delete" } ] } }
<- { "return": {} }
```

InputButton (Enum)

Button of a pointer input device (mouse, tablet).

Values

side
front side button of a 5-button mouse (since 2.9)

extra
rear side button of a 5-button mouse (since 2.9)

touch
screen contact on a multi-touch device (since 8.1)

left
Not documented

middle
Not documented

right
Not documented

wheel-up
Not documented

wheel-down
Not documented

wheel-left
Not documented

wheel-right
Not documented

Since

2.0

InputAxis (Enum)

Position axis of a pointer input device (mouse, tablet).

Values

x
Not documented

y
Not documented

Since

2.0

InputMultiTouchType (Enum)

Type of a multi-touch event.

Values

begin
A new touch event sequence has just started.

update
A touch event sequence has been updated.

end
A touch event sequence has finished.

cancel
A touch event sequence has been canceled.

data
Absolute position data.

Since

8.1

InputKeyEvent (Object)

Keyboard input event.

Members

key: KeyValue
Which key this event is for.

down: boolean
True for key-down and false for key-up events.

Since

2.0

InputBtnEvent (Object)

Pointer button input event.

Members

button: InputButton
Which button this event is for.

down: boolean
True for key-down and false for key-up events.

Since

2.0

InputMoveEvent (Object)

Pointer motion input event.

Members

axis: InputAxis
Which axis is referenced by `value`.

value: int
Pointer position. For absolute coordinates the valid range is 0 -> 0x7fff

Since

2.0

InputMultiTouchEvent (Object)

MultiTouch input event.

Members

type: InputMultiTouchType
The type of multi-touch event.

slot: int
Which slot has generated the event.

tracking-id: int
ID to correlate this event with previously generated events.

axis: InputAxis
Which axis is referenced by `value`.

value: int
Contact position.

Since

8.1

InputEventKind (Enum)

Values

key
a keyboard input event

btn
a pointer button input event

rel
a relative pointer motion input event

abs
an absolute pointer motion input event

mtt
a multi-touch input event

Since

2.0

InputKeyEventWrapper (Object)

Members

data: InputKeyEvent
Keyboard input event

Since

2.0

InputBtnEventWrapper (Object)

Members

data: InputBtnEvent
Pointer button input event

Since

2.0

InputMoveEventWrapper (Object)

Members

data: InputMoveEvent
Pointer motion input event

Since

2.0

InputMultiTouchEventWrapper (Object)

Members

data: InputMultiTouchEvent
MultiTouch input event

Since

8.1

InputEvent (Object)

Input event union.

Members

type: InputEventKind
the type of input event

The members of InputKeyEventWrapper when type is "key"
The members of InputBtnEventWrapper when type is "btn"
The members of InputMoveEventWrapper when type is "rel"
The members of InputMoveEventWrapper when type is "abs"
The members of InputMultiTouchEventWrapper when type is "mtt"

Since

2.0

`input-send-event` (Command)

Send input event(s) to guest.

The `device` and `head` parameters can be used to send the input event to specific input devices in case (a) multiple input devices of the same kind are added to the virtual machine and (b) you have configured input routing (see docs/multiseat.txt) for those input devices. The parameters work exactly like the device and head properties of input devices. If `device` is missing, only devices that have no input routing config are admissible. If `device` is specified, both input devices with and without input routing config are admissible, but devices with input routing config take precedence.

Arguments

`device`: `string` (optional)
display device to send event(s) to.

`head`: `int` (optional)
head to send event(s) to, in case the display device supports multiple scanouts.

`events`: `array of InputEvent`
List of InputEvent union.

Since

2.6

Note

The consoles are visible in the qom tree, under /backend/console[$index]. They have a device link and head property, so it is possible to map which console belongs to which device and display.

Examples

```
1. Press left mouse button.

-> { "execute": "input-send-event",
    "arguments": { "device": "video0",
                   "events": [ { "type": "btn",
                   "data" : { "down": true, "button": "left" } } ] } }
<- { "return": {} }

-> { "execute": "input-send-event",
    "arguments": { "device": "video0",
                   "events": [ { "type": "btn",
                   "data" : { "down": false, "button": "left" } } ] } }
<- { "return": {} }

2. Press ctrl-alt-del.

-> { "execute": "input-send-event",
```

(continues on next page)

(continued from previous page)

```
     "arguments": { "events": [
        { "type": "key", "data" : { "down": true,
          "key": {"type": "qcode", "data": "ctrl" } } },
        { "type": "key", "data" : { "down": true,
          "key": {"type": "qcode", "data": "alt" } } },
        { "type": "key", "data" : { "down": true,
          "key": {"type": "qcode", "data": "delete" } } } ] } }
<- { "return": {} }

3. Move mouse pointer to absolute coordinates (20000, 400).

-> { "execute": "input-send-event" ,
  "arguments": { "events": [
                { "type": "abs", "data" : { "axis": "x", "value" : 20000 } },
                { "type": "abs", "data" : { "axis": "y", "value" : 400 } } ] } }
<- { "return": {} }
```

DisplayGTK (Object)

GTK display options.

Members

grab-on-hover: boolean (optional)
Grab keyboard input on mouse hover.

zoom-to-fit: boolean (optional)
Zoom guest display to fit into the host window. When turned off the host window will be resized instead. In case the display device can notify the guest on window resizes (virtio-gpu) this will default to "on", assuming the guest will resize the display to match the window size then. Otherwise it defaults to "off". (Since 3.1)

show-tabs: boolean (optional)
Display the tab bar for switching between the various graphical interfaces (e.g. VGA and virtual console character devices) by default. (Since 7.1)

show-menubar: boolean (optional)
Display the main window menubar. Defaults to "on". (Since 8.0)

Since

2.12

DisplayEGLHeadless (Object)

EGL headless display options.

Members

rendernode: string (optional)
Which DRM render node should be used. Default is the first available node on the host.

Since

3.1

DisplayDBus (Object)

DBus display options.

Members

addr: string (optional)
The D-Bus bus address (default to the session bus).

rendernode: string (optional)
Which DRM render node should be used. Default is the first available node on the host.

p2p: boolean (optional)
Whether to use peer-to-peer connections (accepted through `add_client`).

audiodev: string (optional)
Use the specified DBus audiodev to export audio.

Since

7.0

DisplayGLMode (Enum)

Display OpenGL mode.

Values

off
Disable OpenGL (default).

on
Use OpenGL, pick context type automatically. Would better be named 'auto' but is called 'on' for backward compatibility with bool type.

core
Use OpenGL with Core (desktop) Context.

es
Use OpenGL with ES (embedded systems) Context.

Since

3.0

DisplayCurses (Object)

Curses display options.

Members

charset: string (optional)
Font charset used by guest (default: CP437).

Since

4.0

DisplayCocoa (Object)

Cocoa display options.

Members

left-command-key: boolean (optional)
Enable/disable forwarding of left command key to guest. Allows command-tab window switching on the host without sending this key to the guest when "off". Defaults to "on"

full-grab: boolean (optional)
Capture all key presses, including system combos. This requires accessibility permissions, since it performs a global grab on key events. (default: off) See https://support.apple.com/en-in/guide/mac-help/mh32356/mac

swap-opt-cmd: boolean (optional)
Swap the Option and Command keys so that their key codes match their position on non-Mac keyboards and you can use Meta/Super and Alt where you expect them. (default: off)

zoom-to-fit: boolean (optional)
Zoom guest display to fit into the host window. When turned off the host window will be resized instead. Defaults to "off". (Since 8.2)

zoom-interpolation: boolean (optional)
Apply interpolation to smooth output when zoom-to-fit is enabled. Defaults to "off". (Since 9.0)

Since

7.0

HotKeyMod (Enum)

Set of modifier keys that need to be held for shortcut key actions.

Values

lctrl-lalt
Not documented

lshift-lctrl-lalt
Not documented

rctrl
Not documented

Since

7.1

DisplaySDL (Object)

SDL2 display options.

Members

grab-mod: HotKeyMod (optional)
Modifier keys that should be pressed together with the "G" key to release the mouse grab.

Since

7.1

DisplayType (Enum)

Display (user interface) type.

Values

default
The default user interface, selecting from the first available of gtk, sdl, cocoa, and vnc.

none
No user interface or video output display. The guest will still see an emulated graphics card, but its output will not be displayed to the QEMU user.

gtk (If: CONFIG_GTK)
The GTK user interface.

sdl (If: CONFIG_SDL)
The SDL user interface.

egl-headless (If: CONFIG_OPENGL)
No user interface, offload GL operations to a local DRI device. Graphical display need to be paired with VNC or Spice. (Since 3.1)

curses (If: CONFIG_CURSES)
Display video output via curses. For graphics device models which support a text mode, QEMU can display this output using a curses/ncurses interface. Nothing is displayed when the graphics device is in graphical mode or if the graphics device does not support a text mode. Generally only the VGA device models support text mode.

cocoa (If: CONFIG_COCOA)
The Cocoa user interface.

spice-app (If: CONFIG_SPICE)
Set up a Spice server and run the default associated application to connect to it. The server will redirect the serial console and QEMU monitors. (Since 4.0)

dbus (If: CONFIG_DBUS_DISPLAY)
Start a D-Bus service for the display. (Since 7.0)

Since

2.12

DisplayOptions (Object)

Display (user interface) options.

Members

type: DisplayType
Which DisplayType qemu should use.

full-screen: boolean (optional)
Start user interface in fullscreen mode (default: off).

window-close: boolean (optional)
Allow to quit qemu with window close button (default: on).

show-cursor: boolean (optional)
Force showing the mouse cursor (default: off). (since: 5.0)

gl: DisplayGLMode (optional)
Enable OpenGL support (default: off).

The members of DisplayGTK when type is "gtk" (If: CONFIG_GTK)
The members of DisplayCocoa when type is "cocoa" (If: CONFIG_COCOA)
The members of DisplayCurses when type is "curses" (If: CONFIG_CURSES)
The members of DisplayEGLHeadless when type is "egl-headless" (If: CONFIG_OPENGL)
The members of DisplayDBus when type is "dbus" (If: CONFIG_DBUS_DISPLAY)
The members of DisplaySDL when type is "sdl" (If: CONFIG_SDL)

Since

2.12

query-display-options (Command)

Returns information about display configuration

Returns

DisplayOptions

Since

3.1

DisplayReloadType (Enum)

Available DisplayReload types.

Values

vnc
VNC display

Since

6.0

DisplayReloadOptionsVNC (Object)

Specify the VNC reload options.

Members

tls-certs: boolean (optional)
reload tls certs or not.

Since

6.0

DisplayReloadOptions (Object)

Options of the display configuration reload.

Members

type: DisplayReloadType
Specify the display type.

The members of DisplayReloadOptionsVNC when type is "vnc"

Since

6.0

display-reload (Command)

Reload display configuration.

Arguments

The members of DisplayReloadOptions

Since

6.0

Example

```
-> { "execute": "display-reload",
     "arguments": { "type": "vnc", "tls-certs": true  } }
<- { "return": {} }
```

DisplayUpdateType (Enum)

Available DisplayUpdate types.

Values

vnc
: VNC display

Since

7.1

DisplayUpdateOptionsVNC (Object)

Specify the VNC reload options.

Members

addresses: array of SocketAddress (optional)
: If specified, change set of addresses to listen for connections. Addresses configured for websockets are not touched.

Since

7.1

DisplayUpdateOptions (Object)

Options of the display configuration reload.

Members

type: DisplayUpdateType
Specify the display type.

The members of DisplayUpdateOptionsVNC when type is "vnc"

Since

7.1

display-update (Command)

Update display configuration.

Arguments

The members of DisplayUpdateOptions

Since

7.1

Example

```
-> { "execute": "display-update",
     "arguments": { "type": "vnc", "addresses":
                    [ { "type": "inet", "host": "0.0.0.0",
                        "port": "5901" } ] } }
<- { "return": {} }
```

client_migrate_info (Command)

Set migration information for remote display. This makes the server ask the client to automatically reconnect using the new parameters once migration finished successfully. Only implemented for SPICE.

Arguments

protocol: string
must be "spice"

hostname: string
migration target hostname

port: int (optional)
spice tcp port for plaintext channels

tls-port: int (optional)
spice tcp port for tls-secured channels

cert-subject: string (optional)
server certificate subject

Since

0.14

Example

```
-> { "execute": "client_migrate_info",
     "arguments": { "protocol": "spice",
                    "hostname": "virt42.lab.kraxel.org",
                    "port": 1234 } }
<- { "return": {} }
```

5.11.17 User authorization

QAuthZListPolicy (Enum)

The authorization policy result

Values

deny
deny access

allow
allow access

Since

4.0

QAuthZListFormat (Enum)

The authorization policy match format

Values

exact
an exact string match

glob
string with ? and * shell wildcard support

Since

4.0

QAuthZListRule (Object)

A single authorization rule.

Members

match: string
a string or glob to match against a user identity

policy: QAuthZListPolicy
the result to return if `match` evaluates to true

format: QAuthZListFormat (optional)
the format of the `match` rule (default 'exact')

Since

4.0

AuthZListProperties (Object)

Properties for authz-list objects.

Members

policy: QAuthZListPolicy (optional)
Default policy to apply when no rule matches (default: deny)

rules: array of QAuthZListRule (optional)
Authorization rules based on matching user

Since

4.0

AuthZListFileProperties (Object)

Properties for authz-listfile objects.

Members

filename: string
File name to load the configuration from. The file must contain valid JSON for AuthZListProperties.

refresh: boolean (optional)
If true, inotify is used to monitor the file, automatically reloading changes. If an error occurs during reloading, all authorizations will fail until the file is next successfully loaded. (default: true if the binary was built with CONFIG_INOTIFY1, false otherwise)

Since

4.0

AuthZPAMProperties (Object)

Properties for authz-pam objects.

Members

service: string
PAM service name to use for authorization

Since

4.0

AuthZSimpleProperties (Object)

Properties for authz-simple objects.

Members

identity: string
Identifies the allowed user. Its format depends on the network service that authorization object is associated with. For authorizing based on TLS x509 certificates, the identity must be the x509 distinguished name.

Since

4.0

5.11.18 Migration

MigrationStats (Object)

Detailed migration status.

Members

transferred: int
amount of bytes already transferred to the target VM

remaining: int
amount of bytes remaining to be transferred to the target VM

total: int
total amount of bytes involved in the migration process

duplicate: int
number of duplicate (zero) pages (since 1.2)

normal: int
number of normal pages (since 1.2)

normal-bytes: int
number of normal bytes sent (since 1.2)

dirty-pages-rate: int
number of pages dirtied by second by the guest (since 1.3)

mbps: number
throughput in megabits/sec. (since 1.6)

dirty-sync-count: int
number of times that dirty ram was synchronized (since 2.1)

postcopy-requests: int
The number of page requests received from the destination (since 2.7)

page-size: int
The number of bytes per page for the various page-based statistics (since 2.10)

multifd-bytes: int
The number of bytes sent through multifd (since 3.0)

pages-per-second: int
the number of memory pages transferred per second (Since 4.0)

precopy-bytes: int
The number of bytes sent in the pre-copy phase (since 7.0).

downtime-bytes: int
The number of bytes sent while the guest is paused (since 7.0).

postcopy-bytes: int
The number of bytes sent during the post-copy phase (since 7.0).

dirty-sync-missed-zero-copy: int
Number of times dirty RAM synchronization could not avoid copying dirty pages. This is between 0 and `dirty-sync-count` * `multifd-channels`. (since 7.1)

Since

0.14

XBZRLECacheStats (Object)

Detailed XBZRLE migration cache statistics

Members

cache-size: int
XBZRLE cache size

bytes: int
amount of bytes already transferred to the target VM

pages: int
amount of pages transferred to the target VM

cache-miss: int
number of cache miss

cache-miss-rate: number
rate of cache miss (since 2.1)

encoding-rate: number
rate of encoded bytes (since 5.1)

overflow: int
number of overflows

Since

1.2

CompressionStats (Object)

Detailed migration compression statistics

Members

pages: int
amount of pages compressed and transferred to the target VM

busy: int
count of times that no free thread was available to compress data

busy-rate: number
rate of thread busy

compressed-size: int
amount of bytes after compression

compression-rate: number
rate of compressed size

Since

3.1

MigrationStatus (Enum)

An enumeration of migration status.

Values

none
no migration has ever happened.

setup
migration process has been initiated.

cancelling
in the process of cancelling migration.

cancelled
cancelling migration is finished.

active
in the process of doing migration.

postcopy-active
like active, but now in postcopy mode. (since 2.5)

postcopy-paused
during postcopy but paused. (since 3.0)

postcopy-recover
trying to recover from a paused postcopy. (since 3.0)

completed
migration is finished.

failed
some error occurred during migration process.

colo
VM is in the process of fault tolerance, VM can not get into this state unless colo capability is enabled for migration. (since 2.8)

pre-switchover
Paused before device serialisation. (since 2.11)

device
During device serialisation when pause-before-switchover is enabled (since 2.11)

wait-unplug
wait for device unplug request by guest OS to be completed. (since 4.2)

Since

2.3

VfioStats (Object)

Detailed VFIO devices migration statistics

Members

transferred: int
amount of bytes transferred to the target VM by VFIO devices

Since

5.2

MigrationInfo (Object)

Information about current migration process.

Members

status: MigrationStatus (optional)
MigrationStatus describing the current migration status. If this field is not returned, no migration process has been initiated

ram: MigrationStats (optional)
MigrationStats containing detailed migration status, only returned if status is 'active' or 'completed'(since 1.2)

xbzrle-cache: XBZRLECacheStats (optional)
XBZRLECacheStats containing detailed XBZRLE migration statistics, only returned if XBZRLE feature is on and status is 'active' or 'completed' (since 1.2)

total-time: int (optional)
total amount of milliseconds since migration started. If migration has ended, it returns the total migration time. (since 1.2)

downtime: int (optional)
only present when migration finishes correctly total downtime in milliseconds for the guest. (since 1.3)

expected-downtime: int (optional)
only present while migration is active expected downtime in milliseconds for the guest in last walk of the dirty bitmap. (since 1.3)

setup-time: int (optional)
amount of setup time in milliseconds *before* the iterations begin but *after* the QMP command is issued. This is designed to provide an accounting of any activities (such as RDMA pinning) which may be expensive, but do not actually occur during the iterative migration rounds themselves. (since 1.6)

cpu-throttle-percentage: int (optional)
percentage of time guest cpus are being throttled during auto-converge. This is only present when auto-converge has started throttling guest cpus. (Since 2.7)

error-desc: string (optional)
the human readable error description string. Clients should not attempt to parse the error strings. (Since 2.7)

postcopy-blocktime: int (optional)
total time when all vCPU were blocked during postcopy live migration. This is only present when the postcopy-blocktime migration capability is enabled. (Since 3.0)

postcopy-vcpu-blocktime: array of int (optional)
list of the postcopy blocktime per vCPU. This is only present when the postcopy-blocktime migration capability is enabled. (Since 3.0)

socket-address: array of SocketAddress (optional)
Only used for tcp, to know what the real port is (Since 4.0)

vfio: VfioStats (optional)
`VfioStats` containing detailed VFIO devices migration statistics, only returned if VFIO device is present, migration is supported by all VFIO devices and status is 'active' or 'completed' (since 5.2)

blocked-reasons: array of string (optional)
A list of reasons an outgoing migration is blocked. Present and non-empty when migration is blocked. (since 6.0)

dirty-limit-throttle-time-per-round: int (optional)
Maximum throttle time (in microseconds) of virtual CPUs each dirty ring full round, which shows how MigrationCapability dirty-limit affects the guest during live migration. (Since 8.1)

dirty-limit-ring-full-time: int (optional)
Estimated average dirty ring full time (in microseconds) for each dirty ring full round. The value equals the dirty ring memory size divided by the average dirty page rate of the virtual CPU, which can be used to observe the average memory load of the virtual CPU indirectly. Note that zero means guest doesn't dirty memory. (Since 8.1)

Since

0.14

query-migrate (Command)

Returns information about current migration process. If migration is active there will be another json-object with RAM migration status.

Returns

`MigrationInfo`

Since

0.14

Examples

```
1. Before the first migration

-> { "execute": "query-migrate" }
<- { "return": {} }

2. Migration is done and has succeeded

-> { "execute": "query-migrate" }
<- { "return": {
        "status": "completed",
        "total-time":12345,
        "setup-time":12345,
        "downtime":12345,
        "ram":{
          "transferred":123,
          "remaining":123,
          "total":246,
          "duplicate":123,
          "normal":123,
          "normal-bytes":123456,
          "dirty-sync-count":15
        }
     }
   }

3. Migration is done and has failed

-> { "execute": "query-migrate" }
<- { "return": { "status": "failed" } }

4. Migration is being performed:

-> { "execute": "query-migrate" }
<- {
      "return":{
         "status":"active",
         "total-time":12345,
         "setup-time":12345,
         "expected-downtime":12345,
         "ram":{
            "transferred":123,
            "remaining":123,
            "total":246,
            "duplicate":123,
            "normal":123,
            "normal-bytes":123456,
```

(continues on next page)

(continued from previous page)

```
            "dirty-sync-count":15
         }
      }
   }

5. Migration is being performed and XBZRLE is active:

-> { "execute": "query-migrate" }
<- {
      "return":{
         "status":"active",
         "total-time":12345,
         "setup-time":12345,
         "expected-downtime":12345,
         "ram":{
            "total":1057024,
            "remaining":1053304,
            "transferred":3720,
            "duplicate":10,
            "normal":3333,
            "normal-bytes":3412992,
            "dirty-sync-count":15
         },
         "xbzrle-cache":{
            "cache-size":67108864,
            "bytes":20971520,
            "pages":2444343,
            "cache-miss":2244,
            "cache-miss-rate":0.123,
            "encoding-rate":80.1,
            "overflow":34434
         }
      }
   }
```

`MigrationCapability` (Enum)

Migration capabilities enumeration

Values

xbzrle
: Migration supports xbzrle (Xor Based Zero Run Length Encoding). This feature allows us to minimize migration traffic for certain work loads, by sending compressed difference of the pages

rdma-pin-all
: Controls whether or not the entire VM memory footprint is mlock()'d on demand or all at once. Refer to docs/rdma.txt for usage. Disabled by default. (since 2.0)

zero-blocks
: During storage migration encode blocks of zeroes efficiently. This essentially saves 1MB of zeroes per block on

the wire. Enabling requires source and target VM to support this feature. To enable it is sufficient to enable the capability on the source VM. The feature is disabled by default. (since 1.6)

events
generate events for each migration state change (since 2.4)

auto-converge
If enabled, QEMU will automatically throttle down the guest to speed up convergence of RAM migration. (since 1.6)

postcopy-ram
Start executing on the migration target before all of RAM has been migrated, pulling the remaining pages along as needed. The capacity must have the same setting on both source and target or migration will not even start. NOTE: If the migration fails during postcopy the VM will fail. (since 2.6)

x-colo
If enabled, migration will never end, and the state of the VM on the primary side will be migrated continuously to the VM on secondary side, this process is called COarse-Grain LOck Stepping (COLO) for Non-stop Service. (since 2.8)

release-ram
if enabled, qemu will free the migrated ram pages on the source during postcopy-ram migration. (since 2.9)

return-path
If enabled, migration will use the return path even for precopy. (since 2.10)

pause-before-switchover
Pause outgoing migration before serialising device state and before disabling block IO (since 2.11)

multifd
Use more than one fd for migration (since 4.0)

dirty-bitmaps
If enabled, QEMU will migrate named dirty bitmaps. (since 2.12)

postcopy-blocktime
Calculate downtime for postcopy live migration (since 3.0)

late-block-activate
If enabled, the destination will not activate block devices (and thus take locks) immediately at the end of migration. (since 3.0)

x-ignore-shared
If enabled, QEMU will not migrate shared memory that is accessible on the destination machine. (since 4.0)

validate-uuid
Send the UUID of the source to allow the destination to ensure it is the same. (since 4.2)

background-snapshot
If enabled, the migration stream will be a snapshot of the VM exactly at the point when the migration procedure starts. The VM RAM is saved with running VM. (since 6.0)

zero-copy-send
Controls behavior on sending memory pages on migration. When true, enables a zero-copy mechanism for sending memory pages, if host supports it. Requires that QEMU be permitted to use locked memory for guest RAM pages. (since 7.1)

postcopy-preempt
If enabled, the migration process will allow postcopy requests to preempt precopy stream, so postcopy requests will be handled faster. This is a performance feature and should not affect the correctness of postcopy migration. (since 7.1)

switchover-ack
: If enabled, migration will not stop the source VM and complete the migration until an ACK is received from the destination that it's OK to do so. Exactly when this ACK is sent depends on the migrated devices that use this feature. For example, a device can use it to make sure some of its data is sent and loaded in the destination before doing switchover. This can reduce downtime if devices that support this capability are present. 'return-path' capability must be enabled to use it. (since 8.1)

dirty-limit
: If enabled, migration will throttle vCPUs as needed to keep their dirty page rate within `vcpu-dirty-limit`. This can improve responsiveness of large guests during live migration, and can result in more stable read performance. Requires KVM with accelerator property "dirty-ring-size" set. (Since 8.1)

mapped-ram
: Migrate using fixed offsets in the migration file for each RAM page. Requires a migration URI that supports seeking, such as a file. (since 9.0)

Features

unstable
: Members `x-colo` and `x-ignore-shared` are experimental.

Since

1.2

`MigrationCapabilityStatus` (Object)

Migration capability information

Members

capability: `MigrationCapability`
: capability enum

state: `boolean`
: capability state bool

Since

1.2

migrate-set-capabilities (Command)

Enable/Disable the following migration capabilities (like xbzrle)

Arguments

capabilities: array of MigrationCapabilityStatus
json array of capability modifications to make

Since

1.2

Example

```
-> { "execute": "migrate-set-capabilities" , "arguments":
     { "capabilities": [ { "capability": "xbzrle", "state": true } ] } }
<- { "return": {} }
```

query-migrate-capabilities (Command)

Returns information about the current migration capabilities status

Returns

`MigrationCapabilityStatus`

Since

1.2

Example

```
-> { "execute": "query-migrate-capabilities" }
<- { "return": [
      {"state": false, "capability": "xbzrle"},
      {"state": false, "capability": "rdma-pin-all"},
      {"state": false, "capability": "auto-converge"},
      {"state": false, "capability": "zero-blocks"},
      {"state": true, "capability": "events"},
      {"state": false, "capability": "postcopy-ram"},
      {"state": false, "capability": "x-colo"}
   ]}
```

MultiFDCompression (Enum)

An enumeration of multifd compression methods.

Values

none
no compression.

zlib
use zlib compression method.

zstd (If: CONFIG_ZSTD)
use zstd compression method.

Since

5.0

MigMode (Enum)

Values

normal
the original form of migration. (since 8.2)

cpr-reboot
The migrate command stops the VM and saves state to the URI. After quitting QEMU, the user resumes by running QEMU -incoming.

This mode allows the user to quit QEMU, optionally update and reboot the OS, and restart QEMU. If the user reboots, the URI must persist across the reboot, such as by using a file.

Unlike normal mode, the use of certain local storage options does not block the migration, but the user must not modify the contents of guest block devices between the quit and restart.

This mode supports VFIO devices provided the user first puts the guest in the suspended runstate, such as by issuing guest-suspend-ram to the QEMU guest agent.

Best performance is achieved when the memory backend is shared and the `x-ignore-shared` migration capability is set, but this is not required. Further, if the user reboots before restarting such a configuration, the shared memory must persist across the reboot, such as by backing it with a dax device.

`cpr-reboot` may not be used with postcopy, background-snapshot, or COLO.

(since 8.2)

ZeroPageDetection (Enum)

Values

none
Do not perform zero page checking.

legacy
Perform zero page checking in main migration thread.

multifd
Perform zero page checking in multifd sender thread if multifd migration is enabled, else in the main migration thread as for `legacy`.

Since

9.0

BitmapMigrationBitmapAliasTransform (Object)

Members

persistent: boolean (optional)
If present, the bitmap will be made persistent or transient depending on this parameter.

Since

6.0

BitmapMigrationBitmapAlias (Object)

Members

name: string
The name of the bitmap.

alias: string
An alias name for migration (for example the bitmap name on the opposite site).

transform: BitmapMigrationBitmapAliasTransform (optional)
Allows the modification of the migrated bitmap. (since 6.0)

Since

5.2

BitmapMigrationNodeAlias (Object)

Maps a block node name and the bitmaps it has to aliases for dirty bitmap migration.

Members

node-name: string
A block node name.

alias: string
An alias block node name for migration (for example the node name on the opposite site).

bitmaps: array of BitmapMigrationBitmapAlias
Mappings for the bitmaps on this node.

Since

5.2

MigrationParameter (Enum)

Migration parameters enumeration

Values

announce-initial
Initial delay (in milliseconds) before sending the first announce (Since 4.0)

announce-max
Maximum delay (in milliseconds) between packets in the announcement (Since 4.0)

announce-rounds
Number of self-announce packets sent after migration (Since 4.0)

announce-step
Increase in delay (in milliseconds) between subsequent packets in the announcement (Since 4.0)

throttle-trigger-threshold
The ratio of bytes_dirty_period and bytes_xfer_period to trigger throttling. It is expressed as percentage. The default value is 50. (Since 5.0)

cpu-throttle-initial
Initial percentage of time guest cpus are throttled when migration auto-converge is activated. The default value is 20. (Since 2.7)

cpu-throttle-increment
throttle percentage increase each time auto-converge detects that migration is not making progress. The default value is 10. (Since 2.7)

cpu-throttle-tailslow
Make CPU throttling slower at tail stage At the tail stage of throttling, the Guest is very sensitive to CPU percentage while the cpu-throttle -increment is excessive usually at tail stage. If this parameter is true, we will compute the ideal CPU percentage used by the Guest, which may exactly make the dirty rate match the dirty rate threshold. Then we will choose a smaller throttle increment between the one specified by cpu-throttle-increment and the one generated by ideal CPU percentage. Therefore, it is compatible to traditional throttling, meanwhile the throttle increment won't be excessive at tail stage. The default value is false. (Since 5.1)

tls-creds
ID of the 'tls-creds' object that provides credentials for establishing a TLS connection over the migration data channel. On the outgoing side of the migration, the credentials must be for a 'client' endpoint, while for the incoming side the credentials must be for a 'server' endpoint. Setting this to a non-empty string enables TLS for all migrations. An empty string means that QEMU will use plain text mode for migration, rather than TLS. (Since 2.7)

tls-hostname
migration target's hostname for validating the server's x509 certificate identity. If empty, QEMU will use the hostname from the migration URI, if any. A non-empty value is required when using x509 based TLS credentials and the migration URI does not include a hostname, such as fd: or exec: based migration. (Since 2.7)

Note: empty value works only since 2.9.

tls-authz
ID of the 'authz' object subclass that provides access control checking of the TLS x509 certificate distinguished name. This object is only resolved at time of use, so can be deleted and recreated on the fly while the migration server is active. If missing, it will default to denying access (Since 4.0)

max-bandwidth
maximum speed for migration, in bytes per second. (Since 2.8)

avail-switchover-bandwidth
to set the available bandwidth that migration can use during switchover phase. NOTE! This does not limit the bandwidth during switchover, but only for calculations when making decisions to switchover. By default, this value is zero, which means QEMU will estimate the bandwidth automatically. This can be set when the estimated value is not accurate, while the user is able to guarantee such bandwidth is available when switching over. When specified correctly, this can make the switchover decision much more accurate. (Since 8.2)

downtime-limit
set maximum tolerated downtime for migration. maximum downtime in milliseconds (Since 2.8)

x-checkpoint-delay
The delay time (in ms) between two COLO checkpoints in periodic mode. (Since 2.8)

multifd-channels
Number of channels used to migrate data in parallel. This is the same number that the number of sockets used for migration. The default value is 2 (since 4.0)

xbzrle-cache-size
cache size to be used by XBZRLE migration. It needs to be a multiple of the target page size and a power of 2 (Since 2.11)

max-postcopy-bandwidth
Background transfer bandwidth during postcopy. Defaults to 0 (unlimited). In bytes per second. (Since 3.0)

max-cpu-throttle
maximum cpu throttle percentage. Defaults to 99. (Since 3.1)

multifd-compression
Which compression method to use. Defaults to none. (Since 5.0)

multifd-zlib-level
Set the compression level to be used in live migration, the compression level is an integer between 0 and 9, where 0 means no compression, 1 means the best compression speed, and 9 means best compression ratio which will consume more CPU. Defaults to 1. (Since 5.0)

multifd-zstd-level
Set the compression level to be used in live migration, the compression level is an integer between 0 and 20, where 0 means no compression, 1 means the best compression speed, and 20 means best compression ratio which will consume more CPU. Defaults to 1. (Since 5.0)

block-bitmap-mapping
Maps block nodes and bitmaps on them to aliases for the purpose of dirty bitmap migration. Such aliases may for example be the corresponding names on the opposite site. The mapping must be one-to-one, but not necessarily complete: On the source, unmapped bitmaps and all bitmaps on unmapped nodes will be ignored. On the destination, encountering an unmapped alias in the incoming migration stream will result in a report, and all further bitmap migration data will then be discarded. Note that the destination does not know about bitmaps it does not receive, so there is no limitation or requirement regarding the number of bitmaps received, or how they are named, or on which nodes they are placed. By default (when this parameter has never been set), bitmap names are mapped to themselves. Nodes are mapped to their block device name if there is one, and to their node name otherwise. (Since 5.2)

x-vcpu-dirty-limit-period
Periodic time (in milliseconds) of dirty limit during live migration. Should be in the range 1 to 1000ms. Defaults to 1000ms. (Since 8.1)

vcpu-dirty-limit
Dirtyrate limit (MB/s) during live migration. Defaults to 1. (Since 8.1)

mode
Migration mode. See description in `MigMode`. Default is 'normal'. (Since 8.2)

zero-page-detection
Whether and how to detect zero pages. See description in `ZeroPageDetection`. Default is 'multifd'. (since 9.0)

Features

unstable
Members `x-checkpoint-delay` and `x-vcpu-dirty-limit-period` are experimental.

Since

2.4

`MigrateSetParameters` (Object)

Members

announce-initial: int (optional)
Initial delay (in milliseconds) before sending the first announce (Since 4.0)

announce-max: int (optional)
Maximum delay (in milliseconds) between packets in the announcement (Since 4.0)

announce-rounds: int (optional)
Number of self-announce packets sent after migration (Since 4.0)

announce-step: int (optional)
Increase in delay (in milliseconds) between subsequent packets in the announcement (Since 4.0)

throttle-trigger-threshold: int (optional)
The ratio of bytes_dirty_period and bytes_xfer_period to trigger throttling. It is expressed as percentage. The default value is 50. (Since 5.0)

cpu-throttle-initial: int (optional)
Initial percentage of time guest cpus are throttled when migration auto-converge is activated. The default value is 20. (Since 2.7)

cpu-throttle-increment: int (optional)
throttle percentage increase each time auto-converge detects that migration is not making progress. The default value is 10. (Since 2.7)

cpu-throttle-tailslow: boolean (optional)
Make CPU throttling slower at tail stage At the tail stage of throttling, the Guest is very sensitive to CPU percentage while the `cpu-throttle` -increment is excessive usually at tail stage. If this parameter is true, we will compute the ideal CPU percentage used by the Guest, which may exactly make the dirty rate match the dirty rate threshold. Then we will choose a smaller throttle increment between the one specified by `cpu-throttle-increment` and the one generated by ideal CPU percentage. Therefore, it is compatible to traditional throttling, meanwhile the throttle increment won't be excessive at tail stage. The default value is false. (Since 5.1)

tls-creds: StrOrNull (optional)
ID of the 'tls-creds' object that provides credentials for establishing a TLS connection over the migration data channel. On the outgoing side of the migration, the credentials must be for a 'client' endpoint, while for the incoming side the credentials must be for a 'server' endpoint. Setting this to a non-empty string enables TLS for all migrations. An empty string means that QEMU will use plain text mode for migration, rather than TLS. This is the default. (Since 2.7)

tls-hostname: StrOrNull (optional)
migration target's hostname for validating the server's x509 certificate identity. If empty, QEMU will use the hostname from the migration URI, if any. A non-empty value is required when using x509 based TLS credentials and the migration URI does not include a hostname, such as fd: or exec: based migration. (Since 2.7)

Note: empty value works only since 2.9.

tls-authz: StrOrNull (optional)
ID of the 'authz' object subclass that provides access control checking of the TLS x509 certificate distinguished name. This object is only resolved at time of use, so can be deleted and recreated on the fly while the migration server is active. If missing, it will default to denying access (Since 4.0)

max-bandwidth: int (optional)
maximum speed for migration, in bytes per second. (Since 2.8)

avail-switchover-bandwidth: int (optional)
to set the available bandwidth that migration can use during switchover phase. NOTE! This does not limit the bandwidth during switchover, but only for calculations when making decisions to switchover. By default, this value is zero, which means QEMU will estimate the bandwidth automatically. This can be set when the estimated value is not accurate, while the user is able to guarantee such bandwidth is available when switching over. When specified correctly, this can make the switchover decision much more accurate. (Since 8.2)

downtime-limit: int (optional)
set maximum tolerated downtime for migration. maximum downtime in milliseconds (Since 2.8)

x-checkpoint-delay: int (optional)
The delay time (in ms) between two COLO checkpoints in periodic mode. (Since 2.8)

multifd-channels: int (optional)
Number of channels used to migrate data in parallel. This is the same number that the number of sockets used for migration. The default value is 2 (since 4.0)

xbzrle-cache-size: int (optional)
cache size to be used by XBZRLE migration. It needs to be a multiple of the target page size and a power of 2 (Since 2.11)

max-postcopy-bandwidth: int (optional)
Background transfer bandwidth during postcopy. Defaults to 0 (unlimited). In bytes per second. (Since 3.0)

max-cpu-throttle: int (optional)
maximum cpu throttle percentage. Defaults to 99. (Since 3.1)

multifd-compression: MultiFDCompression (optional)
Which compression method to use. Defaults to none. (Since 5.0)

multifd-zlib-level: int (optional)
Set the compression level to be used in live migration, the compression level is an integer between 0 and 9, where 0 means no compression, 1 means the best compression speed, and 9 means best compression ratio which will consume more CPU. Defaults to 1. (Since 5.0)

multifd-zstd-level: int (optional)
Set the compression level to be used in live migration, the compression level is an integer between 0 and 20, where 0 means no compression, 1 means the best compression speed, and 20 means best compression ratio which will consume more CPU. Defaults to 1. (Since 5.0)

block-bitmap-mapping: array of BitmapMigrationNodeAlias (optional)
Maps block nodes and bitmaps on them to aliases for the purpose of dirty bitmap migration. Such aliases may for example be the corresponding names on the opposite site. The mapping must be one-to-one, but not necessarily complete: On the source, unmapped bitmaps and all bitmaps on unmapped nodes will be ignored. On the destination, encountering an unmapped alias in the incoming migration stream will result in a report, and all further bitmap migration data will then be discarded. Note that the destination does not know about bitmaps it does not receive, so there is no limitation or requirement regarding the number of bitmaps received, or how they are named, or on which nodes they are placed. By default (when this parameter has never been set), bitmap names are mapped to themselves. Nodes are mapped to their block device name if there is one, and to their node name otherwise. (Since 5.2)

x-vcpu-dirty-limit-period: int (optional)
Periodic time (in milliseconds) of dirty limit during live migration. Should be in the range 1 to 1000ms. Defaults to 1000ms. (Since 8.1)

vcpu-dirty-limit: int (optional)
Dirtyrate limit (MB/s) during live migration. Defaults to 1. (Since 8.1)

mode: MigMode (optional)
Migration mode. See description in MigMode. Default is 'normal'. (Since 8.2)

zero-page-detection: ZeroPageDetection (optional)
Whether and how to detect zero pages. See description in ZeroPageDetection. Default is 'multifd'. (since 9.0)

Features

unstable
Members `x-checkpoint-delay` and `x-vcpu-dirty-limit-period` are experimental.

Since

2.4

`migrate-set-parameters` (Command)

Set various migration parameters.

Arguments

The members of `MigrateSetParameters`

Since

2.4

Example

```
-> { "execute": "migrate-set-parameters" ,
     "arguments": { "multifd-channels": 5 } }
<- { "return": {} }
```

`MigrationParameters` (Object)

The optional members aren't actually optional.

Members

`announce-initial`: `int` (optional)
Initial delay (in milliseconds) before sending the first announce (Since 4.0)

`announce-max`: `int` (optional)
Maximum delay (in milliseconds) between packets in the announcement (Since 4.0)

`announce-rounds`: `int` (optional)
Number of self-announce packets sent after migration (Since 4.0)

`announce-step`: `int` (optional)
Increase in delay (in milliseconds) between subsequent packets in the announcement (Since 4.0)

`throttle-trigger-threshold`: `int` (optional)
The ratio of bytes_dirty_period and bytes_xfer_period to trigger throttling. It is expressed as percentage. The default value is 50. (Since 5.0)

cpu-throttle-initial: int (optional)
Initial percentage of time guest cpus are throttled when migration auto-converge is activated. (Since 2.7)

cpu-throttle-increment: int (optional)
throttle percentage increase each time auto-converge detects that migration is not making progress. (Since 2.7)

cpu-throttle-tailslow: boolean (optional)
Make CPU throttling slower at tail stage At the tail stage of throttling, the Guest is very sensitive to CPU percentage while the `cpu-throttle` -increment is excessive usually at tail stage. If this parameter is true, we will compute the ideal CPU percentage used by the Guest, which may exactly make the dirty rate match the dirty rate threshold. Then we will choose a smaller throttle increment between the one specified by `cpu-throttle-increment` and the one generated by ideal CPU percentage. Therefore, it is compatible to traditional throttling, meanwhile the throttle increment won't be excessive at tail stage. The default value is false. (Since 5.1)

tls-creds: string (optional)
ID of the 'tls-creds' object that provides credentials for establishing a TLS connection over the migration data channel. On the outgoing side of the migration, the credentials must be for a 'client' endpoint, while for the incoming side the credentials must be for a 'server' endpoint. An empty string means that QEMU will use plain text mode for migration, rather than TLS. (Since 2.7)

Note: 2.8 omits empty `tls-creds` instead.

tls-hostname: string (optional)
migration target's hostname for validating the server's x509 certificate identity. If empty, QEMU will use the hostname from the migration URI, if any. (Since 2.7)

Note: 2.8 omits empty `tls-hostname` instead.

tls-authz: string (optional)
ID of the 'authz' object subclass that provides access control checking of the TLS x509 certificate distinguished name. (Since 4.0)

max-bandwidth: int (optional)
maximum speed for migration, in bytes per second. (Since 2.8)

avail-switchover-bandwidth: int (optional)
to set the available bandwidth that migration can use during switchover phase. NOTE! This does not limit the bandwidth during switchover, but only for calculations when making decisions to switchover. By default, this value is zero, which means QEMU will estimate the bandwidth automatically. This can be set when the estimated value is not accurate, while the user is able to guarantee such bandwidth is available when switching over. When specified correctly, this can make the switchover decision much more accurate. (Since 8.2)

downtime-limit: int (optional)
set maximum tolerated downtime for migration. maximum downtime in milliseconds (Since 2.8)

x-checkpoint-delay: int (optional)
the delay time between two COLO checkpoints. (Since 2.8)

multifd-channels: int (optional)
Number of channels used to migrate data in parallel. This is the same number that the number of sockets used for migration. The default value is 2 (since 4.0)

xbzrle-cache-size: int (optional)
cache size to be used by XBZRLE migration. It needs to be a multiple of the target page size and a power of 2 (Since 2.11)

max-postcopy-bandwidth: int (optional)
Background transfer bandwidth during postcopy. Defaults to 0 (unlimited). In bytes per second. (Since 3.0)

max-cpu-throttle: int (optional)
maximum cpu throttle percentage. Defaults to 99. (Since 3.1)

multifd-compression: MultiFDCompression (optional)
Which compression method to use. Defaults to none. (Since 5.0)

multifd-zlib-level: int (optional)
Set the compression level to be used in live migration, the compression level is an integer between 0 and 9, where 0 means no compression, 1 means the best compression speed, and 9 means best compression ratio which will consume more CPU. Defaults to 1. (Since 5.0)

multifd-zstd-level: int (optional)
Set the compression level to be used in live migration, the compression level is an integer between 0 and 20, where 0 means no compression, 1 means the best compression speed, and 20 means best compression ratio which will consume more CPU. Defaults to 1. (Since 5.0)

block-bitmap-mapping: array of BitmapMigrationNodeAlias (optional)
Maps block nodes and bitmaps on them to aliases for the purpose of dirty bitmap migration. Such aliases may for example be the corresponding names on the opposite site. The mapping must be one-to-one, but not necessarily complete: On the source, unmapped bitmaps and all bitmaps on unmapped nodes will be ignored. On the destination, encountering an unmapped alias in the incoming migration stream will result in a report, and all further bitmap migration data will then be discarded. Note that the destination does not know about bitmaps it does not receive, so there is no limitation or requirement regarding the number of bitmaps received, or how they are named, or on which nodes they are placed. By default (when this parameter has never been set), bitmap names are mapped to themselves. Nodes are mapped to their block device name if there is one, and to their node name otherwise. (Since 5.2)

x-vcpu-dirty-limit-period: int (optional)
Periodic time (in milliseconds) of dirty limit during live migration. Should be in the range 1 to 1000ms. Defaults to 1000ms. (Since 8.1)

vcpu-dirty-limit: int (optional)
Dirtyrate limit (MB/s) during live migration. Defaults to 1. (Since 8.1)

mode: MigMode (optional)
Migration mode. See description in MigMode. Default is 'normal'. (Since 8.2)

zero-page-detection: ZeroPageDetection (optional)
Whether and how to detect zero pages. See description in ZeroPageDetection. Default is 'multifd'. (since 9.0)

Features

unstable
Members x-checkpoint-delay and x-vcpu-dirty-limit-period are experimental.

Since

2.4

query-migrate-parameters (Command)

Returns information about the current migration parameters

Returns

MigrationParameters

Since

2.4

Example

```
-> { "execute": "query-migrate-parameters" }
<- { "return": {
         "multifd-channels": 2,
         "cpu-throttle-increment": 10,
         "cpu-throttle-initial": 20,
         "max-bandwidth": 33554432,
         "downtime-limit": 300
      }
   }
```

migrate-start-postcopy (Command)

Followup to a migration command to switch the migration to postcopy mode. The postcopy-ram capability must be set on both source and destination before the original migration command.

Since

2.5

Example

```
-> { "execute": "migrate-start-postcopy" }
<- { "return": {} }
```

MIGRATION (Event)

Emitted when a migration event happens

Arguments

status: MigrationStatus
: `MigrationStatus` describing the current migration status.

Since

2.4

Example

```
<- {"timestamp": {"seconds": 1432121972, "microseconds": 744001},
    "event": "MIGRATION",
    "data": {"status": "completed"} }
```

MIGRATION_PASS (Event)

Emitted from the source side of a migration at the start of each pass (when it syncs the dirty bitmap)

Arguments

pass: int
: An incrementing count (starting at 1 on the first pass)

Since

2.6

Example

```
<- { "timestamp": {"seconds": 1449669631, "microseconds": 239225},
     "event": "MIGRATION_PASS", "data": {"pass": 2} }
```

COLOMessage (Enum)

The message transmission between Primary side and Secondary side.

Values

checkpoint-ready
Secondary VM (SVM) is ready for checkpointing

checkpoint-request
Primary VM (PVM) tells SVM to prepare for checkpointing

checkpoint-reply
SVM gets PVM's checkpoint request

vmstate-send
VM's state will be sent by PVM.

vmstate-size
The total size of VMstate.

vmstate-received
VM's state has been received by SVM.

vmstate-loaded
VM's state has been loaded by SVM.

Since

2.8

COLOMode (Enum)

The COLO current mode.

Values

none
COLO is disabled.

primary
COLO node in primary side.

secondary
COLO node in slave side.

Since

2.8

FailoverStatus (Enum)

An enumeration of COLO failover status

Values

none
: no failover has ever happened

require
: got failover requirement but not handled

active
: in the process of doing failover

completed
: finish the process of failover

relaunch
: restart the failover process, from 'none' -> 'completed' (Since 2.9)

Since

2.8

COLO_EXIT (Event)

Emitted when VM finishes COLO mode due to some errors happening or at the request of users.

Arguments

mode: COLOMode
: report COLO mode when COLO exited.

reason: COLOExitReason
: describes the reason for the COLO exit.

Since

3.1

Example

```
<- { "timestamp": {"seconds": 2032141960, "microseconds": 417172},
     "event": "COLO_EXIT", "data": {"mode": "primary", "reason": "request" } }
```

COLOExitReason (Enum)

The reason for a COLO exit.

Values

none
: failover has never happened. This state does not occur in the COLO_EXIT event, and is only visible in the result of query-colo-status.

request
: COLO exit is due to an external request.

error
: COLO exit is due to an internal error.

processing
: COLO is currently handling a failover (since 4.0).

Since

3.1

x-colo-lost-heartbeat (Command)

Tell qemu that heartbeat is lost, request it to do takeover procedures. If this command is sent to the PVM, the Primary side will exit COLO mode. If sent to the Secondary, the Secondary side will run failover work, then takes over server operation to become the service VM.

Features

unstable
: This command is experimental.

Since

2.8

Example

```
-> { "execute": "x-colo-lost-heartbeat" }
<- { "return": {} }
```

If

CONFIG_REPLICATION

migrate_cancel (Command)

Cancel the current executing migration process.

Notes

This command succeeds even if there is no migration process running.

Since

0.14

Example

```
-> { "execute": "migrate_cancel" }
<- { "return": {} }
```

migrate-continue (Command)

Continue migration when it's in a paused state.

Arguments

state: MigrationStatus
The state the migration is currently expected to be in

Since

2.11

Example

```
-> { "execute": "migrate-continue" , "arguments":
     { "state": "pre-switchover" } }
<- { "return": {} }
```

MigrationAddressType (Enum)

The migration stream transport mechanisms.

Values

socket
Migrate via socket.

exec
Direct the migration stream to another process.

rdma
Migrate via RDMA.

file
Direct the migration stream to a file.

Since

8.2

FileMigrationArgs (Object)

Members

filename: string
The file to receive the migration stream

offset: int
The file offset where the migration stream will start

Since

8.2

MigrationExecCommand (Object)

Members

args: array of string
command (list head) and arguments to execute.

Since

8.2

MigrationAddress (Object)

Migration endpoint configuration.

Members

transport: MigrationAddressType
The migration stream transport mechanism

The members of SocketAddress when transport is "socket"
The members of MigrationExecCommand when transport is "exec"
The members of InetSocketAddress when transport is "rdma"
The members of FileMigrationArgs when transport is "file"

Since

8.2

MigrationChannelType (Enum)

The migration channel-type request options.

Values

main
Main outbound migration channel.

Since

8.1

`MigrationChannel` (Object)

Migration stream channel parameters.

Members

`channel-type`: `MigrationChannelType`
Channel type for transferring packet information.

`addr`: `MigrationAddress`
Migration endpoint configuration on destination interface.

Since

8.1

`migrate` (Command)

Migrates the current running guest to another Virtual Machine.

Arguments

`uri`: `string` (optional)
the Uniform Resource Identifier of the destination VM

`channels`: `array of MigrationChannel` (optional)
list of migration stream channels with each stream in the list connected to a destination interface endpoint.

`detach`: `boolean` (optional)
this argument exists only for compatibility reasons and is ignored by QEMU

`resume`: `boolean` (optional)
resume one paused migration, default "off". (since 3.0)

Since

0.14

Notes

1. The 'query-migrate' command should be used to check migration's progress and final result (this information is provided by the 'status' member)
2. All boolean arguments default to false
3. The user Monitor's "detach" argument is invalid in QMP and should not be used
4. The uri argument should have the Uniform Resource Identifier of default destination VM. This connection will be bound to default network.
5. For now, number of migration streams is restricted to one, i.e. number of items in 'channels' list is just 1.
6. The 'uri' and 'channels' arguments are mutually exclusive; exactly one of the two should be present.

Example

```
-> { "execute": "migrate", "arguments": { "uri": "tcp:0:4446" } }
<- { "return": {} }

-> { "execute": "migrate",
     "arguments": {
         "channels": [ { "channel-type": "main",
                         "addr": { "transport": "socket",
                                   "type": "inet",
                                   "host": "10.12.34.9",
                                   "port": "1050" } } ] } }
<- { "return": {} }

-> { "execute": "migrate",
     "arguments": {
         "channels": [ { "channel-type": "main",
                         "addr": { "transport": "exec",
                                   "args": [ "/bin/nc", "-p", "6000",
                                             "/some/sock" ] } } ] } }
<- { "return": {} }

-> { "execute": "migrate",
     "arguments": {
         "channels": [ { "channel-type": "main",
                         "addr": { "transport": "rdma",
                                   "host": "10.12.34.9",
                                   "port": "1050" } } ] } }
<- { "return": {} }

-> { "execute": "migrate",
     "arguments": {
         "channels": [ { "channel-type": "main",
                         "addr": { "transport": "file",
                                   "filename": "/tmp/migfile",
                                   "offset": "0x1000" } } ] } }
<- { "return": {} }
```

`migrate-incoming` (Command)

Start an incoming migration, the qemu must have been started with -incoming defer

Arguments

uri: string (optional)
The Uniform Resource Identifier identifying the source or address to listen on

channels: array of MigrationChannel (optional)
list of migration stream channels with each stream in the list connected to a destination interface endpoint.

exit-on-error: boolean (optional)
Exit on incoming migration failure. Default true. When set to false, the failure triggers a MIGRATION event, and error details could be retrieved with query-migrate. (since 9.1)

Since

2.3

Notes

1. It's a bad idea to use a string for the uri, but it needs to stay compatible with -incoming and the format of the uri is already exposed above libvirt.
2. QEMU must be started with -incoming defer to allow migrate-incoming to be used.
3. The uri format is the same as for -incoming
4. For now, number of migration streams is restricted to one, i.e. number of items in 'channels' list is just 1.
5. The 'uri' and 'channels' arguments are mutually exclusive; exactly one of the two should be present.

Example

```
-> { "execute": "migrate-incoming",
     "arguments": { "uri": "tcp:0:4446" } }
<- { "return": {} }

-> { "execute": "migrate-incoming",
     "arguments": {
         "channels": [ { "channel-type": "main",
                         "addr": { "transport": "socket",
                                   "type": "inet",
                                   "host": "10.12.34.9",
                                   "port": "1050" } } ] } }
<- { "return": {} }

-> { "execute": "migrate-incoming",
     "arguments": {
         "channels": [ { "channel-type": "main",
                         "addr": { "transport": "exec",
                                   "args": [ "/bin/nc", "-p", "6000",
                                             "/some/sock" ] } } ] } }
<- { "return": {} }

-> { "execute": "migrate-incoming",
```

(continues on next page)

(continued from previous page)

```
     "arguments": {
         "channels": [ { "channel-type": "main",
                         "addr": { "transport": "rdma",
                                   "host": "10.12.34.9",
                                   "port": "1050" } } ] } }
<- { "return": {} }
```

xen-save-devices-state (Command)

Save the state of all devices to file. The RAM and the block devices of the VM are not saved by this command.

Arguments

filename: string
: the file to save the state of the devices to as binary data. See xen-save-devices-state.txt for a description of the binary format.

live: boolean (optional)
: Optional argument to ask QEMU to treat this command as part of a live migration. Default to true. (since 2.11)

Since

1.1

Example

```
-> { "execute": "xen-save-devices-state",
     "arguments": { "filename": "/tmp/save" } }
<- { "return": {} }
```

xen-set-global-dirty-log (Command)

Enable or disable the global dirty log mode.

Arguments

enable: boolean
: true to enable, false to disable.

Since

1.3

Example

```
-> { "execute": "xen-set-global-dirty-log",
     "arguments": { "enable": true } }
<- { "return": {} }
```

xen-load-devices-state (Command)

Load the state of all devices from file. The RAM and the block devices of the VM are not loaded by this command.

Arguments

filename: string
: the file to load the state of the devices from as binary data. See xen-save-devices-state.txt for a description of the binary format.

Since

2.7

Example

```
-> { "execute": "xen-load-devices-state",
     "arguments": { "filename": "/tmp/resume" } }
<- { "return": {} }
```

xen-set-replication (Command)

Enable or disable replication.

Arguments

enable: boolean
: true to enable, false to disable.

primary: boolean
: true for primary or false for secondary.

failover: boolean (optional)
: true to do failover, false to stop. Cannot be specified if 'enable' is true. Default value is false.

Example

```
-> { "execute": "xen-set-replication",
     "arguments": {"enable": true, "primary": false} }
<- { "return": {} }
```

Since

2.9

If

CONFIG_REPLICATION

ReplicationStatus (Object)

The result format for 'query-xen-replication-status'.

Members

error: boolean
: true if an error happened, false if replication is normal.

desc: string (optional)
: the human readable error description string, when `error` is 'true'.

Since

2.9

If

CONFIG_REPLICATION

query-xen-replication-status (Command)

Query replication status while the vm is running.

Returns

A `ReplicationStatus` object showing the status.

Example

```
-> { "execute": "query-xen-replication-status" }
<- { "return": { "error": false } }
```

Since

2.9

If

`CONFIG_REPLICATION`

`xen-colo-do-checkpoint` (Command)

Xen uses this command to notify replication to trigger a checkpoint.

Example

```
-> { "execute": "xen-colo-do-checkpoint" }
<- { "return": {} }
```

Since

2.9

If

`CONFIG_REPLICATION`

`COLOStatus` (Object)

The result format for 'query-colo-status'.

Members

mode: COLOMode
COLO running mode. If COLO is running, this field will return 'primary' or 'secondary'.

last-mode: COLOMode
COLO last running mode. If COLO is running, this field will return same like mode field, after failover we can use this field to get last colo mode. (since 4.0)

reason: COLOExitReason
describes the reason for the COLO exit.

Since

3.1

If

CONFIG_REPLICATION

query-colo-status (Command)

Query COLO status while the vm is running.

Returns

A COLOStatus object showing the status.

Example

```
-> { "execute": "query-colo-status" }
<- { "return": { "mode": "primary", "last-mode": "none", "reason": "request" } }
```

Since

3.1

If

CONFIG_REPLICATION

migrate-recover (Command)

Provide a recovery migration stream URI.

Arguments

uri: string
: the URI to be used for the recovery of migration stream.

Example

```
-> { "execute": "migrate-recover",
     "arguments": { "uri": "tcp:192.168.1.200:12345" } }
<- { "return": {} }
```

Since

3.0

migrate-pause (Command)

Pause a migration. Currently it only supports postcopy.

Example

```
-> { "execute": "migrate-pause" }
<- { "return": {} }
```

Since

3.0

UNPLUG_PRIMARY (Event)

Emitted from source side of a migration when migration state is WAIT_UNPLUG. Device was unplugged by guest operating system. Device resources in QEMU are kept on standby to be able to re-plug it in case of migration failure.

Arguments

device-id: string
QEMU device id of the unplugged device

Since

4.2

Example

```
<- { "event": "UNPLUG_PRIMARY",
     "data": { "device-id": "hostdev0" },
     "timestamp": { "seconds": 1265044230, "microseconds": 450486 } }
```

DirtyRateVcpu (Object)

Dirty rate of vcpu.

Members

id: int
vcpu index.

dirty-rate: int
dirty rate.

Since

6.2

DirtyRateStatus (Enum)

Dirty page rate measurement status.

Values

unstarted
measuring thread has not been started yet

measuring
measuring thread is running

measured
dirty page rate is measured and the results are available

Since

5.2

DirtyRateMeasureMode (Enum)

Method used to measure dirty page rate. Differences between available methods are explained in `calc-dirty-rate`.

Values

page-sampling
: use page sampling

dirty-ring
: use dirty ring

dirty-bitmap
: use dirty bitmap

Since

6.2

TimeUnit (Enum)

Specifies unit in which time-related value is specified.

Values

second
: value is in seconds

millisecond
: value is in milliseconds

Since

8.2

DirtyRateInfo (Object)

Information about measured dirty page rate.

Members

dirty-rate: int (optional)
an estimate of the dirty page rate of the VM in units of MiB/s. Value is present only when `status` is 'measured'.

status: DirtyRateStatus
current status of dirty page rate measurements

start-time: int
start time in units of second for calculation

calc-time: int
time period for which dirty page rate was measured, expressed and rounded down to `calc-time-unit`.

calc-time-unit: TimeUnit
time unit of `calc-time` (Since 8.2)

sample-pages: int
number of sampled pages per GiB of guest memory. Valid only in page-sampling mode (Since 6.1)

mode: DirtyRateMeasureMode
mode that was used to measure dirty page rate (Since 6.2)

vcpu-dirty-rate: array of DirtyRateVcpu (optional)
dirty rate for each vCPU if dirty-ring mode was specified (Since 6.2)

Since

5.2

`calc-dirty-rate` (Command)

Start measuring dirty page rate of the VM. Results can be retrieved with `query-dirty-rate` after measurements are completed.

Dirty page rate is the number of pages changed in a given time period expressed in MiB/s. The following methods of calculation are available:

1. In page sampling mode, a random subset of pages are selected and hashed twice: once at the beginning of measurement time period, and once again at the end. If two hashes for some page are different, the page is counted as changed. Since this method relies on sampling and hashing, calculated dirty page rate is only an estimate of its true value. Increasing `sample-pages` improves estimation quality at the cost of higher computational overhead.
2. Dirty bitmap mode captures writes to memory (for example by temporarily revoking write access to all pages) and counting page faults. Information about modified pages is collected into a bitmap, where each bit corresponds to one guest page. This mode requires that KVM accelerator property "dirty-ring-size" is *not* set.
3. Dirty ring mode is similar to dirty bitmap mode, but the information about modified pages is collected into ring buffer. This mode tracks page modification per each vCPU separately. It requires that KVM accelerator property "dirty-ring-size" is set.

Arguments

calc-time: int
time period for which dirty page rate is calculated. By default it is specified in seconds, but the unit can be set explicitly with calc-time-unit. Note that larger calc-time values will typically result in smaller dirty page rates because page dirtying is a one-time event. Once some page is counted as dirty during calc-time period, further writes to this page will not increase dirty page rate anymore.

calc-time-unit: TimeUnit (optional)
time unit in which calc-time is specified. By default it is seconds. (Since 8.2)

sample-pages: int (optional)
number of sampled pages per each GiB of guest memory. Default value is 512. For 4KiB guest pages this corresponds to sampling ratio of 0.2%. This argument is used only in page sampling mode. (Since 6.1)

mode: DirtyRateMeasureMode (optional)
mechanism for tracking dirty pages. Default value is 'page-sampling'. Others are 'dirty-bitmap' and 'dirty-ring'. (Since 6.1)

Since

5.2

Example

```
-> {"execute": "calc-dirty-rate", "arguments": {"calc-time": 1,
                                                'sample-pages': 512} }
<- { "return": {} }

Measure dirty rate using dirty bitmap for 500 milliseconds:

-> {"execute": "calc-dirty-rate", "arguments": {"calc-time": 500,
    "calc-time-unit": "millisecond", "mode": "dirty-bitmap"} }

<- { "return": {} }
```

query-dirty-rate (Command)

Query results of the most recent invocation of calc-dirty-rate.

Arguments

calc-time-unit: TimeUnit (optional)
time unit in which to report calculation time. By default it is reported in seconds. (Since 8.2)

Since

5.2

Examples

```
1. Measurement is in progress:

<- {"status": "measuring", "sample-pages": 512,
    "mode": "page-sampling", "start-time": 1693900454, "calc-time": 10,
    "calc-time-unit": "second"}

2. Measurement has been completed:

<- {"status": "measured", "sample-pages": 512, "dirty-rate": 108,
    "mode": "page-sampling", "start-time": 1693900454, "calc-time": 10,
    "calc-time-unit": "second"}
```

DirtyLimitInfo (Object)

Dirty page rate limit information of a virtual CPU.

Members

cpu-index: int
: index of a virtual CPU.

limit-rate: int
: upper limit of dirty page rate (MB/s) for a virtual CPU, 0 means unlimited.

current-rate: int
: current dirty page rate (MB/s) for a virtual CPU.

Since

7.1

set-vcpu-dirty-limit (Command)

Set the upper limit of dirty page rate for virtual CPUs.

Requires KVM with accelerator property "dirty-ring-size" set. A virtual CPU's dirty page rate is a measure of its memory load. To observe dirty page rates, use `calc-dirty-rate`.

Arguments

cpu-index: int (optional)
index of a virtual CPU, default is all.

dirty-rate: int
upper limit of dirty page rate (MB/s) for virtual CPUs.

Since

7.1

Example

```
-> {"execute": "set-vcpu-dirty-limit"}
    "arguments": { "dirty-rate": 200,
                   "cpu-index": 1 } }
<- { "return": {} }
```

cancel-vcpu-dirty-limit (Command)

Cancel the upper limit of dirty page rate for virtual CPUs.

Cancel the dirty page limit for the vCPU which has been set with set-vcpu-dirty-limit command. Note that this command requires support from dirty ring, same as the "set-vcpu-dirty-limit".

Arguments

cpu-index: int (optional)
index of a virtual CPU, default is all.

Since

7.1

Example

```
-> {"execute": "cancel-vcpu-dirty-limit"},
    "arguments": { "cpu-index": 1 } }
<- { "return": {} }
```

query-vcpu-dirty-limit (Command)

Returns information about virtual CPU dirty page rate limits, if any.

Since

7.1

Example

```
-> {"execute": "query-vcpu-dirty-limit"}
<- {"return": [
       { "limit-rate": 60, "current-rate": 3, "cpu-index": 0},
       { "limit-rate": 60, "current-rate": 3, "cpu-index": 1}]}
```

MigrationThreadInfo (Object)

Information about migrationthreads

Members

name: string
: the name of migration thread

thread-id: int
: ID of the underlying host thread

Since

7.2

query-migrationthreads (Command)

Returns information of migration threads

Returns

MigrationThreadInfo

Since

7.2

snapshot-save (Command)

Save a VM snapshot

Arguments

job-id: string
identifier for the newly created job

tag: string
name of the snapshot to create

vmstate: string
block device node name to save vmstate to

devices: array of string
list of block device node names to save a snapshot to

Applications should not assume that the snapshot save is complete when this command returns. The job commands / events must be used to determine completion and to fetch details of any errors that arise.

Note that execution of the guest CPUs may be stopped during the time it takes to save the snapshot. A future version of QEMU may ensure CPUs are executing continuously.

It is strongly recommended that `devices` contain all writable block device nodes if a consistent snapshot is required.

If `tag` already exists, an error will be reported

Example

```
-> { "execute": "snapshot-save",
     "arguments": {
        "job-id": "snapsave0",
        "tag": "my-snap",
        "vmstate": "disk0",
        "devices": ["disk0", "disk1"]
     }
   }
<- { "return": { } }
<- {"event": "JOB_STATUS_CHANGE",
    "timestamp": {"seconds": 1432121972, "microseconds": 744001},
    "data": {"status": "created", "id": "snapsave0"}}
<- {"event": "JOB_STATUS_CHANGE",
    "timestamp": {"seconds": 1432122172, "microseconds": 744001},
    "data": {"status": "running", "id": "snapsave0"}}
<- {"event": "STOP",
    "timestamp": {"seconds": 1432122372, "microseconds": 744001} }
<- {"event": "RESUME",
    "timestamp": {"seconds": 1432122572, "microseconds": 744001} }
```

(continues on next page)

(continued from previous page)

```
<- {"event": "JOB_STATUS_CHANGE",
    "timestamp": {"seconds": 1432122772, "microseconds": 744001},
    "data": {"status": "waiting", "id": "snapsave0"}}
<- {"event": "JOB_STATUS_CHANGE",
    "timestamp": {"seconds": 1432122972, "microseconds": 744001},
    "data": {"status": "pending", "id": "snapsave0"}}
<- {"event": "JOB_STATUS_CHANGE",
    "timestamp": {"seconds": 1432123172, "microseconds": 744001},
    "data": {"status": "concluded", "id": "snapsave0"}}
-> {"execute": "query-jobs"}
<- {"return": [{"current-progress": 1,
                "status": "concluded",
                "total-progress": 1,
                "type": "snapshot-save",
                "id": "snapsave0"}]}
```

Since

6.0

snapshot-load (Command)

Load a VM snapshot

Arguments

job-id: string
: identifier for the newly created job

tag: string
: name of the snapshot to load.

vmstate: string
: block device node name to load vmstate from

devices: array of string
: list of block device node names to load a snapshot from

Applications should not assume that the snapshot load is complete when this command returns. The job commands / events must be used to determine completion and to fetch details of any errors that arise.

Note that execution of the guest CPUs will be stopped during the time it takes to load the snapshot.

It is strongly recommended that `devices` contain all writable block device nodes that can have changed since the original `snapshot-save` command execution.

Example

```
-> { "execute": "snapshot-load",
     "arguments": {
        "job-id": "snapload0",
        "tag": "my-snap",
        "vmstate": "disk0",
        "devices": ["disk0", "disk1"]
     }
   }
<- { "return": { } }
<- {"event": "JOB_STATUS_CHANGE",
    "timestamp": {"seconds": 1472124172, "microseconds": 744001},
    "data": {"status": "created", "id": "snapload0"}}
<- {"event": "JOB_STATUS_CHANGE",
    "timestamp": {"seconds": 1472125172, "microseconds": 744001},
    "data": {"status": "running", "id": "snapload0"}}
<- {"event": "STOP",
    "timestamp": {"seconds": 1472125472, "microseconds": 744001} }
<- {"event": "RESUME",
    "timestamp": {"seconds": 1472125872, "microseconds": 744001} }
<- {"event": "JOB_STATUS_CHANGE",
    "timestamp": {"seconds": 1472126172, "microseconds": 744001},
    "data": {"status": "waiting", "id": "snapload0"}}
<- {"event": "JOB_STATUS_CHANGE",
    "timestamp": {"seconds": 1472127172, "microseconds": 744001},
    "data": {"status": "pending", "id": "snapload0"}}
<- {"event": "JOB_STATUS_CHANGE",
    "timestamp": {"seconds": 1472128172, "microseconds": 744001},
    "data": {"status": "concluded", "id": "snapload0"}}
-> {"execute": "query-jobs"}
<- {"return": [{"current-progress": 1,
                "status": "concluded",
                "total-progress": 1,
                "type": "snapshot-load",
                "id": "snapload0"}]}
```

Since

6.0

snapshot-delete (Command)

Delete a VM snapshot

Arguments

job-id: string
identifier for the newly created job

tag: string
name of the snapshot to delete.

devices: array of string
list of block device node names to delete a snapshot from

Applications should not assume that the snapshot delete is complete when this command returns. The job commands / events must be used to determine completion and to fetch details of any errors that arise.

Example

```
-> { "execute": "snapshot-delete",
     "arguments": {
        "job-id": "snapdelete0",
        "tag": "my-snap",
        "devices": ["disk0", "disk1"]
     }
   }
<- { "return": { } }
<- {"event": "JOB_STATUS_CHANGE",
    "timestamp": {"seconds": 1442124172, "microseconds": 744001},
    "data": {"status": "created", "id": "snapdelete0"}}
<- {"event": "JOB_STATUS_CHANGE",
    "timestamp": {"seconds": 1442125172, "microseconds": 744001},
    "data": {"status": "running", "id": "snapdelete0"}}
<- {"event": "JOB_STATUS_CHANGE",
    "timestamp": {"seconds": 1442126172, "microseconds": 744001},
    "data": {"status": "waiting", "id": "snapdelete0"}}
<- {"event": "JOB_STATUS_CHANGE",
    "timestamp": {"seconds": 1442127172, "microseconds": 744001},
    "data": {"status": "pending", "id": "snapdelete0"}}
<- {"event": "JOB_STATUS_CHANGE",
    "timestamp": {"seconds": 1442128172, "microseconds": 744001},
    "data": {"status": "concluded", "id": "snapdelete0"}}
-> {"execute": "query-jobs"}
<- {"return": [{"current-progress": 1,
                "status": "concluded",
                "total-progress": 1,
                "type": "snapshot-delete",
                "id": "snapdelete0"}]}
```

Since

6.0

5.11.19 Transactions

`Abort` (Object)

This action can be used to test transaction failure.

Since

1.6

`ActionCompletionMode` (Enum)

An enumeration of Transactional completion modes.

Values

`individual`
: Do not attempt to cancel any other Actions if any Actions fail after the Transaction request succeeds. All Actions that can complete successfully will do so without waiting on others. This is the default.

`grouped`
: If any Action fails after the Transaction succeeds, cancel all Actions. Actions do not complete until all Actions are ready to complete. May be rejected by Actions that do not support this completion mode.

Since

2.5

`TransactionActionKind` (Enum)

Values

`abort`
: Since 1.6

`block-dirty-bitmap-add`
: Since 2.5

`block-dirty-bitmap-remove`
: Since 4.2

`block-dirty-bitmap-clear`
: Since 2.5

`block-dirty-bitmap-enable`
: Since 4.0

block-dirty-bitmap-disable
Since 4.0

block-dirty-bitmap-merge
Since 4.0

blockdev-backup
Since 2.3

blockdev-snapshot
Since 2.5

blockdev-snapshot-internal-sync
Since 1.7

blockdev-snapshot-sync
since 1.1

drive-backup
Since 1.6

Features

deprecated
Member `drive-backup` is deprecated. Use member `blockdev-backup` instead.

Since

1.1

`AbortWrapper` (Object)

Members

data: Abort
Not documented

Since

1.6

`BlockDirtyBitmapAddWrapper` (Object)

Members

data: BlockDirtyBitmapAdd
Not documented

Since

2.5

BlockDirtyBitmapWrapper (Object)

Members

data: BlockDirtyBitmap
Not documented

Since

2.5

BlockDirtyBitmapMergeWrapper (Object)

Members

data: BlockDirtyBitmapMerge
Not documented

Since

4.0

BlockdevBackupWrapper (Object)

Members

data: BlockdevBackup
Not documented

Since

2.3

BlockdevSnapshotWrapper (Object)

Members

data: BlockdevSnapshot
Not documented

Since

2.5

BlockdevSnapshotInternalWrapper (Object)

Members

data: BlockdevSnapshotInternal
Not documented

Since

1.7

BlockdevSnapshotSyncWrapper (Object)

Members

data: BlockdevSnapshotSync
Not documented

Since

1.1

DriveBackupWrapper (Object)

Members

data: DriveBackup
Not documented

Since

1.6

TransactionAction (Object)

A discriminated record of operations that can be performed with `transaction`.

Members

type: TransactionActionKind
the operation to be performed

The members of AbortWrapper when type is "abort"
The members of BlockDirtyBitmapAddWrapper when type is "block-dirty-bitmap-add"
The members of BlockDirtyBitmapWrapper when type is "block-dirty-bitmap-remove"
The members of BlockDirtyBitmapWrapper when type is "block-dirty-bitmap-clear"
The members of BlockDirtyBitmapWrapper when type is "block-dirty-bitmap-enable"
The members of BlockDirtyBitmapWrapper when type is "block-dirty-bitmap-disable"
The members of BlockDirtyBitmapMergeWrapper when type is "block-dirty-bitmap-merge"
The members of BlockdevBackupWrapper when type is "blockdev-backup"
The members of BlockdevSnapshotWrapper when type is "blockdev-snapshot"
The members of BlockdevSnapshotInternalWrapper when type is "blockdev-snapshot-internal-sync"
The members of BlockdevSnapshotSyncWrapper when type is "blockdev-snapshot-sync"
The members of DriveBackupWrapper when type is "drive-backup"

Since

1.1

TransactionProperties (Object)

Optional arguments to modify the behavior of a Transaction.

Members

completion-mode: ActionCompletionMode (optional)
Controls how jobs launched asynchronously by Actions will complete or fail as a group. See ActionCompletionMode for details.

Since

2.5

transaction (Command)

Executes a number of transactionable QMP commands atomically. If any operation fails, then the entire set of actions will be abandoned and the appropriate error returned.

For external snapshots, the dictionary contains the device, the file to use for the new snapshot, and the format. The default format, if not specified, is qcow2.

Each new snapshot defaults to being created by QEMU (wiping any contents if the file already exists), but it is also possible to reuse an externally-created file. In the latter case, you should ensure that the new image file has the same contents as the current one; QEMU cannot perform any meaningful check. Typically this is achieved by using the current image file as the backing file for the new image.

On failure, the original disks pre-snapshot attempt will be used.

For internal snapshots, the dictionary contains the device and the snapshot's name. If an internal snapshot matching name already exists, the request will be rejected. Only some image formats support it, for example, qcow2, and rbd,

On failure, qemu will try delete the newly created internal snapshot in the transaction. When an I/O error occurs during deletion, the user needs to fix it later with qemu-img or other command.

Arguments

actions: array of TransactionAction
: List of `TransactionAction`; information needed for the respective operations.

properties: TransactionProperties (optional)
: structure of additional options to control the execution of the transaction. See `TransactionProperties` for additional detail.

Errors

Any errors from commands in the transaction

Note

The transaction aborts on the first failure. Therefore, there will be information on only one failed operation returned in an error condition, and subsequent actions will not have been attempted.

Since

1.1

Example

```
-> { "execute": "transaction",
     "arguments": { "actions": [
         { "type": "blockdev-snapshot-sync", "data" : { "device": "ide-hd0",
                                     "snapshot-file": "/some/place/my-image",
                                     "format": "qcow2" } },
         { "type": "blockdev-snapshot-sync", "data" : { "node-name": "myfile",
                                     "snapshot-file": "/some/place/my-image2",
                                     "snapshot-node-name": "node3432",
                                     "mode": "existing",
                                     "format": "qcow2" } },
         { "type": "blockdev-snapshot-sync", "data" : { "device": "ide-hd1",
                                     "snapshot-file": "/some/place/my-image2",
                                     "mode": "existing",
                                     "format": "qcow2" } },
         { "type": "blockdev-snapshot-internal-sync", "data" : {
                                     "device": "ide-hd2",
                                     "name": "snapshot0" } } ] } }
<- { "return": {} }
```

5.11.20 Tracing

TraceEventState (Enum)

State of a tracing event.

Values

unavailable
The event is statically disabled.

disabled
The event is dynamically disabled.

enabled
The event is dynamically enabled.

Since

2.2

TraceEventInfo (Object)

Information of a tracing event.

Members

name: string
Event name.

state: TraceEventState
Tracing state.

vcpu: boolean
Whether this is a per-vCPU event (since 2.7).

Features

deprecated
Member vcpu is deprecated, and always ignored.

Since

2.2

trace-event-get-state (Command)

Query the state of events.

Arguments

name: string
Event name pattern (case-sensitive glob).

vcpu: int (optional)
The vCPU to query (since 2.7).

Features

deprecated
Member vcpu is deprecated, and always ignored.

Returns

a list of TraceEventInfo for the matching events

Since

2.2

Example

```
-> { "execute": "trace-event-get-state",
     "arguments": { "name": "qemu_memalign" } }
<- { "return": [ { "name": "qemu_memalign", "state": "disabled", "vcpu": false } ] }
```

trace-event-set-state (Command)

Set the dynamic tracing state of events.

Arguments

name: string
Event name pattern (case-sensitive glob).

enable: boolean
Whether to enable tracing.

ignore-unavailable: boolean (optional)
Do not match unavailable events with name.

vcpu: int (optional)
The vCPU to act upon (all by default; since 2.7).

Features

deprecated
Member vcpu is deprecated, and always ignored.

Since

2.2

Example

```
-> { "execute": "trace-event-set-state",
     "arguments": { "name": "qemu_memalign", "enable": true } }
<- { "return": {} }
```

5.11.21 Compatibility policy

CompatPolicyInput (Enum)

Policy for handling "funny" input.

Values

accept
Accept silently

reject
Reject with an error

crash
abort() the process

Since

6.0

CompatPolicyOutput (Enum)

Policy for handling "funny" output.

Values

accept
Pass on unchanged

hide
Filter out

Since

6.0

CompatPolicy (Object)

Policy for handling deprecated management interfaces.

This is intended for testing users of the management interfaces.

Limitation: covers only syntactic aspects of QMP, i.e. stuff tagged with feature 'deprecated' or 'unstable'. We may want to extend it to cover semantic aspects and CLI.

Limitation: deprecated-output policy `hide` is not implemented for enumeration values. They behave the same as with policy `accept`.

Members

deprecated-input: CompatPolicyInput (optional)
how to handle deprecated input (default 'accept')

deprecated-output: CompatPolicyOutput (optional)
how to handle deprecated output (default 'accept')

unstable-input: CompatPolicyInput (optional)
how to handle unstable input (default 'accept') (since 6.2)

unstable-output: CompatPolicyOutput (optional)
how to handle unstable output (default 'accept') (since 6.2)

Since

6.0

5.11.22 QMP monitor control

`qmp_capabilities` (Command)

Enable QMP capabilities.

Arguments

`enable`: `array of QMPCapability` (optional)
: An optional list of QMPCapability values to enable. The client must not enable any capability that is not mentioned in the QMP greeting message. If the field is not provided, it means no QMP capabilities will be enabled. (since 2.12)

Example

```
-> { "execute": "qmp_capabilities",
     "arguments": { "enable": [ "oob" ] } }
<- { "return": {} }
```

Notes

This command is valid exactly when first connecting: it must be issued before any other command will be accepted, and will fail once the monitor is accepting other commands. (see qemu docs/interop/qmp-spec.rst)

The QMP client needs to explicitly enable QMP capabilities, otherwise all the QMP capabilities will be turned off by default.

Since

0.13

`QMPCapability` (Enum)

Enumeration of capabilities to be advertised during initial client connection, used for agreeing on particular QMP extension behaviors.

Values

oob
QMP ability to support out-of-band requests. (Please refer to qmp-spec.rst for more information on OOB)

Since

2.12

`VersionTriple` (Object)

A three-part version number.

Members

major: int
The major version number.

minor: int
The minor version number.

micro: int
The micro version number.

Since

2.4

`VersionInfo` (Object)

A description of QEMU's version.

Members

qemu: VersionTriple
The version of QEMU. By current convention, a micro version of 50 signifies a development branch. A micro version greater than or equal to 90 signifies a release candidate for the next minor version. A micro version of less than 50 signifies a stable release.

package: string
QEMU will always set this field to an empty string. Downstream versions of QEMU should set this to a non-empty string. The exact format depends on the downstream however it highly recommended that a unique name is used.

Since

0.14

query-version (Command)

Returns the current version of QEMU.

Returns

A `VersionInfo` object describing the current version of QEMU.

Since

0.14

Example

```
-> { "execute": "query-version" }
<- {
      "return":{
         "qemu":{
            "major":0,
            "minor":11,
            "micro":5
         },
         "package":""
      }
   }
```

CommandInfo (Object)

Information about a QMP command

Members

name: string
The command name

Since

0.14

query-commands (Command)

Return a list of supported QMP commands by this server

Returns

A list of `CommandInfo` for all supported commands

Since

0.14

Example

```
-> { "execute": "query-commands" }
<- {
     "return":[
        {
           "name":"query-balloon"
        },
        {
           "name":"system_powerdown"
        }
     ]
   }
```

Note

This example has been shortened as the real response is too long.

quit (Command)

This command will cause the QEMU process to exit gracefully. While every attempt is made to send the QMP response before terminating, this is not guaranteed. When using this interface, a premature EOF would not be unexpected.

Since

0.14

Example

```
-> { "execute": "quit" }
<- { "return": {} }
```

MonitorMode (Enum)

An enumeration of monitor modes.

Values

readline
HMP monitor (human-oriented command line interface)

control
QMP monitor (JSON-based machine interface)

Since

5.0

MonitorOptions (Object)

Options to be used for adding a new monitor.

Members

id: string (optional)
Name of the monitor

mode: MonitorMode (optional)
Selects the monitor mode (default: readline in the system emulator, control in qemu-storage-daemon)

pretty: boolean (optional)
Enables pretty printing (QMP only)

chardev: string
Name of a character device to expose the monitor on

Since

5.0

5.11.23 QMP introspection

`query-qmp-schema` (Command)

Command query-qmp-schema exposes the QMP wire ABI as an array of SchemaInfo. This lets QMP clients figure out what commands and events are available in this QEMU, and their parameters and results.

However, the SchemaInfo can't reflect all the rules and restrictions that apply to QMP. It's interface introspection (figuring out what's there), not interface specification. The specification is in the QAPI schema.

Furthermore, while we strive to keep the QMP wire format backwards-compatible across qemu versions, the introspection output is not guaranteed to have the same stability. For example, one version of qemu may list an object member as an optional non-variant, while another lists the same member only through the object's variants; or the type of a member may change from a generic string into a specific enum or from one specific type into an alternate that includes the original type alongside something else.

Returns

array of `SchemaInfo`, where each element describes an entity in the ABI: command, event, type, ...

The order of the various SchemaInfo is unspecified; however, all names are guaranteed to be unique (no name will be duplicated with different meta-types).

Note

the QAPI schema is also used to help define *internal* interfaces, by defining QAPI types. These are not part of the QMP wire ABI, and therefore not returned by this command.

Since

2.5

`SchemaMetaType` (Enum)

This is a `SchemaInfo`'s meta type, i.e. the kind of entity it describes.

Values

`builtin`
: a predefined type such as 'int' or 'bool'.

`enum`
: an enumeration type

`array`
: an array type

object
an object type (struct or union)

alternate
an alternate type

command
a QMP command

event
a QMP event

Since

2.5

SchemaInfo (Object)

Members

name: string
the entity's name, inherited from `base`. The SchemaInfo is always referenced by this name. Commands and events have the name defined in the QAPI schema. Unlike command and event names, type names are not part of the wire ABI. Consequently, type names are meaningless strings here, although they are still guaranteed unique regardless of `meta-type`.

meta-type: SchemaMetaType
the entity's meta type, inherited from `base`.

features: array of string (optional)
names of features associated with the entity, in no particular order. (since 4.1 for object types, 4.2 for commands, 5.0 for the rest)

The members of SchemaInfoBuiltin when meta-type is "builtin"
The members of SchemaInfoEnum when meta-type is "enum"
The members of SchemaInfoArray when meta-type is "array"
The members of SchemaInfoObject when meta-type is "object"
The members of SchemaInfoAlternate when meta-type is "alternate"
The members of SchemaInfoCommand when meta-type is "command"
The members of SchemaInfoEvent when meta-type is "event"

Since

2.5

SchemaInfoBuiltin (Object)

Additional SchemaInfo members for meta-type 'builtin'.

Members

json-type: JSONType
the JSON type used for this type on the wire.

Since

2.5

JSONType (Enum)

The four primitive and two structured types according to RFC 8259 section 1, plus 'int' (split off 'number'), plus the obvious top type 'value'.

Values

string
Not documented

number
Not documented

int
Not documented

boolean
Not documented

null
Not documented

object
Not documented

array
Not documented

value
Not documented

Since

2.5

SchemaInfoEnum (Object)

Additional SchemaInfo members for meta-type 'enum'.

Members

members: array of SchemaInfoEnumMember
the enum type's members, in no particular order (since 6.2).

values: array of string
the enumeration type's member names, in no particular order. Redundant with `members`. Just for backward compatibility.

Features

deprecated
Member `values` is deprecated. Use `members` instead.

Values of this type are JSON string on the wire.

Since

2.5

SchemaInfoEnumMember (Object)

An object member.

Members

name: string
the member's name, as defined in the QAPI schema.

features: array of string (optional)
names of features associated with the member, in no particular order.

Since

6.2

SchemaInfoArray (Object)

Additional SchemaInfo members for meta-type 'array'.

Members

element-type: string
the array type's element type.

Values of this type are JSON array on the wire.

Since

2.5

SchemaInfoObject (Object)

Additional SchemaInfo members for meta-type 'object'.

Members

members: array of SchemaInfoObjectMember
the object type's (non-variant) members, in no particular order.

tag: string (optional)
the name of the member serving as type tag. An element of `members` with this name must exist.

variants: array of SchemaInfoObjectVariant (optional)
variant members, i.e. additional members that depend on the type tag's value. Present exactly when `tag` is present. The variants are in no particular order, and may even differ from the order of the values of the enum type of the `tag`.

Values of this type are JSON object on the wire.

Since

2.5

SchemaInfoObjectMember (Object)

An object member.

Members

name: string
the member's name, as defined in the QAPI schema.

type: string
the name of the member's type.

default: value (optional)
default when used as command parameter. If absent, the parameter is mandatory. If present, the value must be null. The parameter is optional, and behavior when it's missing is not specified here. Future extension: if present and non-null, the parameter is optional, and defaults to this value.

features: array of string (optional)
names of features associated with the member, in no particular order. (since 5.0)

Since

2.5

SchemaInfoObjectVariant (Object)

The variant members for a value of the type tag.

Members

case: string
a value of the type tag.

type: string
the name of the object type that provides the variant members when the type tag has value `case`.

Since

2.5

SchemaInfoAlternate (Object)

Additional SchemaInfo members for meta-type 'alternate'.

Members

members: array of SchemaInfoAlternateMember
the alternate type's members, in no particular order. The members' wire encoding is distinct, see *How to use the QAPI code generator* section Alternate types.

On the wire, this can be any of the members.

Since

2.5

SchemaInfoAlternateMember (Object)

An alternate member.

Members

type: string
the name of the member's type.

Since

2.5

SchemaInfoCommand (Object)

Additional SchemaInfo members for meta-type 'command'.

Members

arg-type: string
the name of the object type that provides the command's parameters.

ret-type: string
the name of the command's result type.

allow-oob: boolean (optional)
whether the command allows out-of-band execution, defaults to false (Since: 2.12)

Since

2.5

SchemaInfoEvent (Object)

Additional SchemaInfo members for meta-type 'event'.

Members

arg-type: string
the name of the object type that provides the event's parameters.

Since

2.5

5.11.24 QEMU Object Model (QOM)

ObjectPropertyInfo (Object)

Members

name: string
the name of the property

type: string
the type of the property. This will typically come in one of four forms:

1) A primitive type such as 'u8', 'u16', 'bool', 'str', or 'double'. These types are mapped to the appropriate JSON type.

2) A child type in the form 'child<subtype>' where subtype is a qdev device type name. Child properties create the composition tree.

3) A link type in the form 'link<subtype>' where subtype is a qdev device type name. Link properties form the device model graph.

description: string (optional)
if specified, the description of the property.

default-value: value (optional)
the default value, if any (since 5.0)

Since

1.2

qom-list (Command)

This command will list any properties of a object given a path in the object model.

Arguments

path: string
: the path within the object model. See qom-get for a description of this parameter.

Returns

a list of ObjectPropertyInfo that describe the properties of the object.

Since

1.2

Example

```
-> { "execute": "qom-list",
     "arguments": { "path": "/chardevs" } }
<- { "return": [ { "name": "type", "type": "string" },
                 { "name": "parallel0", "type": "child<chardev-vc>" },
                 { "name": "serial0", "type": "child<chardev-vc>" },
                 { "name": "mon0", "type": "child<chardev-stdio>" } ] }
```

qom-get (Command)

This command will get a property from a object model path and return the value.

Arguments

path: string
: The path within the object model. There are two forms of supported paths–absolute and partial paths.

  Absolute paths are derived from the root object and can follow child<> or link<> properties. Since they can follow link<> properties, they can be arbitrarily long. Absolute paths look like absolute filenames and are prefixed with a leading slash.

  Partial paths look like relative filenames. They do not begin with a prefix. The matching rules for partial paths are subtle but designed to make specifying objects easy. At each level of the composition tree, the partial path is matched as an absolute path. The first match is not returned. At least two matches are searched for. A successful result is only returned if only one match is found. If more than one match is found, a flag is return to indicate that the match was ambiguous.

property: string
The property name to read

Returns

The property value. The type depends on the property type. child<> and link<> properties are returned as #str pathnames. All integer property types (u8, u16, etc) are returned as #int.

Since

1.2

Examples

```
1. Use absolute path

-> { "execute": "qom-get",
     "arguments": { "path": "/machine/unattached/device[0]",
                    "property": "hotplugged" } }
<- { "return": false }

2. Use partial path

-> { "execute": "qom-get",
     "arguments": { "path": "unattached/sysbus",
                    "property": "type" } }
<- { "return": "System" }
```

qom-set (Command)

This command will set a property from a object model path.

Arguments

path: string
see qom-get for a description of this parameter

property: string
the property name to set

value: value
a value who's type is appropriate for the property type. See qom-get for a description of type mapping.

Since

1.2

Example

```
-> { "execute": "qom-set",
     "arguments": { "path": "/machine",
                    "property": "graphics",
                    "value": false } }
<- { "return": {} }
```

ObjectTypeInfo (Object)

This structure describes a search result from qom-list-types

Members

name: string
the type name found in the search

abstract: boolean (optional)
the type is abstract and can't be directly instantiated. Omitted if false. (since 2.10)

parent: string (optional)
Name of parent type, if any (since 2.10)

Since

1.1

qom-list-types (Command)

This command will return a list of types given search parameters

Arguments

implements: string (optional)
if specified, only return types that implement this type name

abstract: boolean (optional)
if true, include abstract types in the results

Returns

a list of `ObjectTypeInfo` or an empty list if no results are found

Since

1.1

`qom-list-properties` (Command)

List properties associated with a QOM object.

Arguments

`typename: string`
: the type name of an object

Note

objects can create properties at runtime, for example to describe links between different devices and/or objects. These properties are not included in the output of this command.

Returns

a list of ObjectPropertyInfo describing object properties

Since

2.12

`CanHostSocketcanProperties` (Object)

Properties for can-host-socketcan objects.

Members

`if: string`
: interface name of the host system CAN bus to connect to

`canbus: string`
: object ID of the can-bus object to connect to the host interface

Since

2.12

ColoCompareProperties (Object)

Properties for colo-compare objects.

Members

primary_in: string
name of the character device backend to use for the primary input (incoming packets are redirected to `outdev`)

secondary_in: string
name of the character device backend to use for secondary input (incoming packets are only compared to the input on `primary_in` and then dropped)

outdev: string
name of the character device backend to use for output

iothread: string
name of the iothread to run in

notify_dev: string (optional)
name of the character device backend to be used to communicate with the remote colo-frame (only for Xen COLO)

compare_timeout: int (optional)
the maximum time to hold a packet from `primary_in` for comparison with an incoming packet on `secondary_in` in milliseconds (default: 3000)

expired_scan_cycle: int (optional)
the interval at which colo-compare checks whether packets from `primary` have timed out, in milliseconds (default: 3000)

max_queue_size: int (optional)
the maximum number of packets to keep in the queue for comparing with incoming packets from `secondary_in`. If the queue is full and additional packets are received, the additional packets are dropped. (default: 1024)

vnet_hdr_support: boolean (optional)
if true, vnet header support is enabled (default: false)

Since

2.8

CryptodevBackendProperties (Object)

Properties for cryptodev-backend and cryptodev-backend-builtin objects.

Members

queues: int (optional)
the number of queues for the cryptodev backend. Ignored for cryptodev-backend and must be 1 for cryptodev-backend-builtin. (default: 1)

throttle-bps: int (optional)
limit total bytes per second (Since 8.0)

throttle-ops: int (optional)
limit total operations per second (Since 8.0)

Since

2.8

CryptodevVhostUserProperties (Object)

Properties for cryptodev-vhost-user objects.

Members

chardev: string
the name of a Unix domain socket character device that connects to the vhost-user server

The members of CryptodevBackendProperties

Since

2.12

DBusVMStateProperties (Object)

Properties for dbus-vmstate objects.

Members

addr: string
the name of the DBus bus to connect to

id-list: string (optional)
a comma separated list of DBus IDs of helpers whose data should be included in the VM state on migration

Since

5.0

NetfilterInsert (Enum)

Indicates where to insert a netfilter relative to a given other filter.

Values

before
: insert before the specified filter

behind
: insert behind the specified filter

Since

5.0

NetfilterProperties (Object)

Properties for objects of classes derived from netfilter.

Members

netdev: string
: id of the network device backend to filter

queue: NetFilterDirection (optional)
: indicates which queue(s) to filter (default: all)

status: string (optional)
: indicates whether the filter is enabled ("on") or disabled ("off") (default: "on")

position: string (optional)
: specifies where the filter should be inserted in the filter list. "head" means the filter is inserted at the head of the filter list, before any existing filters. "tail" means the filter is inserted at the tail of the filter list, behind any existing filters (default). "id=<id>" means the filter is inserted before or behind the filter specified by <id>, depending on the `insert` property. (default: "tail")

insert: NetfilterInsert (optional)
: where to insert the filter relative to the filter given in `position`. Ignored if `position` is "head" or "tail". (default: behind)

Since

2.5

FilterBufferProperties (Object)

Properties for filter-buffer objects.

Members

interval: int
a non-zero interval in microseconds. All packets arriving in the given interval are delayed until the end of the interval.

The members of NetfilterProperties

Since

2.5

FilterDumpProperties (Object)

Properties for filter-dump objects.

Members

file: string
the filename where the dumped packets should be stored

maxlen: int (optional)
maximum number of bytes in a packet that are stored (default: 65536)

The members of NetfilterProperties

Since

2.5

FilterMirrorProperties (Object)

Properties for filter-mirror objects.

Members

outdev: string
the name of a character device backend to which all incoming packets are mirrored

vnet_hdr_support: boolean (optional)
if true, vnet header support is enabled (default: false)

The members of NetfilterProperties

Since

2.6

FilterRedirectorProperties (Object)

Properties for filter-redirector objects.

At least one of indev or outdev must be present. If both are present, they must not refer to the same character device backend.

Members

indev: string (optional)
the name of a character device backend from which packets are received and redirected to the filtered network device

outdev: string (optional)
the name of a character device backend to which all incoming packets are redirected

vnet_hdr_support: boolean (optional)
if true, vnet header support is enabled (default: false)

The members of NetfilterProperties

Since

2.6

FilterRewriterProperties (Object)

Properties for filter-rewriter objects.

Members

vnet_hdr_support: boolean (optional)
if true, vnet header support is enabled (default: false)

The members of NetfilterProperties

Since

2.8

InputBarrierProperties (Object)

Properties for input-barrier objects.

Members

name: string
the screen name as declared in the screens section of barrier.conf

server: string (optional)
hostname of the Barrier server (default: “localhost”)

port: string (optional)
TCP port of the Barrier server (default: “24800”)

x-origin: string (optional)
x coordinate of the leftmost pixel on the guest screen (default: “0”)

y-origin: string (optional)
y coordinate of the topmost pixel on the guest screen (default: “0”)

width: string (optional)
the width of secondary screen in pixels (default: “1920”)

height: string (optional)
the height of secondary screen in pixels (default: “1080”)

Since

4.2

InputLinuxProperties (Object)

Properties for input-linux objects.

Members

evdev: string
the path of the host evdev device to use

grab_all: boolean (optional)
if true, grab is toggled for all devices (e.g. both keyboard and mouse) instead of just one device (default: false)

repeat: boolean (optional)
enables auto-repeat events (default: false)

grab-toggle: GrabToggleKeys (optional)
the key or key combination that toggles device grab (default: ctrl-ctrl)

Since

2.6

EventLoopBaseProperties (Object)

Common properties for event loops

Members

aio-max-batch: int (optional)
maximum number of requests in a batch for the AIO engine, 0 means that the engine will use its default. (default: 0)

thread-pool-min: int (optional)
minimum number of threads reserved in the thread pool (default:0)

thread-pool-max: int (optional)
maximum number of threads the thread pool can contain (default:64)

Since

7.1

IothreadProperties (Object)

Properties for iothread objects.

Members

poll-max-ns: int (optional)
the maximum number of nanoseconds to busy wait for events. 0 means polling is disabled (default: 32768 on POSIX hosts, 0 otherwise)

poll-grow: int (optional)
the multiplier used to increase the polling time when the algorithm detects it is missing events due to not polling long enough. 0 selects a default behaviour (default: 0)

poll-shrink: int (optional)
the divisor used to decrease the polling time when the algorithm detects it is spending too long polling without encountering events. 0 selects a default behaviour (default: 0)

The members of EventLoopBaseProperties

The aio-max-batch option is available since 6.1.

Since

2.0

MainLoopProperties (Object)

Properties for the main-loop object.

Members

The members of EventLoopBaseProperties

Since

7.1

MemoryBackendProperties (Object)

Properties for objects of classes derived from memory-backend.

Members

merge: boolean (optional)
if true, mark the memory as mergeable (default depends on the machine type)

dump: boolean (optional)
if true, include the memory in core dumps (default depends on the machine type)

host-nodes: array of int (optional)
the list of NUMA host nodes to bind the memory to

policy: HostMemPolicy (optional)
the NUMA policy (default: 'default')

prealloc: boolean (optional)
if true, preallocate memory (default: false)

prealloc-threads: int (optional)
number of CPU threads to use for prealloc (default: 1)

prealloc-context: string (optional)
thread context to use for creation of preallocation threads (default: none) (since 7.2)

share: boolean (optional)
if false, the memory is private to QEMU; if true, it is shared (default: false)

reserve: boolean (optional)
if true, reserve swap space (or huge pages) if applicable (default: true) (since 6.1)

size: int
size of the memory region in bytes

x-use-canonical-path-for-ramblock-id: boolean (optional)
if true, the canonical path is used for ramblock-id. Disable this for 4.0 machine types or older to allow migration with newer QEMU versions. (default: false generally, but true for machine types <= 4.0)

Note

prealloc=true and reserve=false cannot be set at the same time. With reserve=true, the behavior depends on the operating system: for example, Linux will not reserve swap space for shared file mappings – "not applicable". In contrast, reserve=false will bail out if it cannot be configured accordingly.

Since

2.1

MemoryBackendFileProperties (Object)

Properties for memory-backend-file objects.

Members

align: int (optional)
the base address alignment when QEMU mmap(2)s mem-path. Some backend stores specified by mem-path require an alignment different than the default one used by QEMU, e.g. the device DAX /dev/dax0.0 requires 2M alignment rather than 4K. In such cases, users can specify the required alignment via this option. 0 selects a default alignment (currently the page size). (default: 0)

offset: int (optional)
the offset into the target file that the region starts at. You can use this option to back multiple regions with a single file. Must be a multiple of the page size. (default: 0) (since 8.1)

discard-data: boolean (optional)
if true, the file contents can be destroyed when QEMU exits, to avoid unnecessarily flushing data to the backing file. Note that discard-data is only an optimization, and QEMU might not discard file contents if it aborts unexpectedly or is terminated using SIGKILL. (default: false)

mem-path: string
the path to either a shared memory or huge page filesystem mount

pmem: boolean (optional) (If: CONFIG_LIBPMEM)
specifies whether the backing file specified by mem-path is in host persistent memory that can be accessed using the SNIA NVM programming model (e.g. Intel NVDIMM).

readonly: boolean (optional)
if true, the backing file is opened read-only; if false, it is opened read-write. (default: false)

rom: OnOffAuto (optional)
whether to create Read Only Memory (ROM) that cannot be modified by the VM. Any write attempts to such ROM will be denied. Most use cases want writable RAM instead of ROM. However, selected use cases, like R/O NVDIMMs, can benefit from ROM. If set to 'on', create ROM; if set to 'off', create writable RAM; if set to 'auto', the value of the readonly property is used. This property is primarily helpful when we want to have proper RAM in configurations that would traditionally create ROM before this property was introduced: VM templating, where we want to open a file readonly (readonly set to true) and mark the memory to be private for QEMU (share set to false). For this use case, we need writable RAM instead of ROM, and want to set this property to 'off'. (default: auto, since 8.2)

The members of MemoryBackendProperties

Since

2.1

MemoryBackendMemfdProperties (Object)

Properties for memory-backend-memfd objects.

The share boolean option is true by default with memfd.

Members

hugetlb: boolean (optional)
if true, the file to be created resides in the hugetlbfs filesystem (default: false)

hugetlbsize: int (optional)
the hugetlb page size on systems that support multiple hugetlb page sizes (it must be a power of 2 value supported by the system). 0 selects a default page size. This option is ignored if hugetlb is false. (default: 0)

seal: boolean (optional)
if true, create a sealed-file, which will block further resizing of the memory (default: true)

The members of MemoryBackendProperties

Since

2.12

`MemoryBackendEpcProperties` (Object)

Properties for memory-backend-epc objects.

The `share` boolean option is true by default with epc

The `merge` boolean option is false by default with epc

The `dump` boolean option is false by default with epc

Members

The members of `MemoryBackendProperties`

Since

6.2

`PrManagerHelperProperties` (Object)

Properties for pr-manager-helper objects.

Members

`path`: `string`
: the path to a Unix domain socket for connecting to the external helper

Since

2.11

`QtestProperties` (Object)

Properties for qtest objects.

Members

`chardev`: `string`
: the chardev to be used to receive qtest commands on.

`log`: `string` (optional)
: the path to a log file

Since

6.0

RemoteObjectProperties (Object)

Properties for x-remote-object objects.

Members

fd: string
file descriptor name previously passed via 'getfd' command

devid: string
the id of the device to be associated with the file descriptor

Since

6.0

VfioUserServerProperties (Object)

Properties for x-vfio-user-server objects.

Members

socket: SocketAddress
socket to be used by the libvfio-user library

device: string
the ID of the device to be emulated at the server

Since

7.1

IOMMUFDProperties (Object)

Properties for iommufd objects.

Members

fd: string (optional)
file descriptor name previously passed via 'getfd' command, which represents a pre-opened /dev/iommu. This allows the iommufd object to be shared across several subsystems (VFIO, VDPA, ...), and the file descriptor to be shared with other process, e.g. DPDK. (default: QEMU opens /dev/iommu by itself)

Since

9.0

AcpiGenericInitiatorProperties (Object)

Properties for acpi-generic-initiator objects.

Members

pci-dev: string
PCI device ID to be associated with the node

node: int
NUMA node associated with the PCI device

Since

9.0

RngProperties (Object)

Properties for objects of classes derived from rng.

Members

opened: boolean (optional)
if true, the device is opened immediately when applying this option and will probably fail when processing the next option. Don't use; only provided for compatibility. (default: false)

Features

deprecated
Member opened is deprecated. Setting true doesn't make sense, and false is already the default.

Since

1.3

RngEgdProperties (Object)

Properties for rng-egd objects.

Members

chardev: string
the name of a character device backend that provides the connection to the RNG daemon

The members of RngProperties

Since

1.3

RngRandomProperties (Object)

Properties for rng-random objects.

Members

filename: string (optional)
the filename of the device on the host to obtain entropy from (default: "/dev/urandom")

The members of RngProperties

Since

1.3

SevGuestProperties (Object)

Properties for sev-guest objects.

Members

sev-device: string (optional)
SEV device to use (default: "/dev/sev")

dh-cert-file: string (optional)
guest owners DH certificate (encoded with base64)

session-file: string (optional)
guest owners session parameters (encoded with base64)

policy: int (optional)
SEV policy value (default: 0x1)

handle: int (optional)
SEV firmware handle (default: 0)

cbitpos: int (optional)
C-bit location in page table entry (default: 0)

reduced-phys-bits: int
number of bits in physical addresses that become unavailable when SEV is enabled

kernel-hashes: boolean (optional)
if true, add hashes of kernel/initrd/cmdline to a designated guest firmware page for measured boot with -kernel (default: false) (since 6.2)

legacy-vm-type: boolean (optional)
Use legacy KVM_SEV_INIT KVM interface for creating the VM. The newer KVM_SEV_INIT2 interface syncs additional vCPU state when initializing the VMSA structures, which will result in a different guest measurement. Set this to maintain compatibility with older QEMU or kernel versions that rely on legacy KVM_SEV_INIT behavior. (default: false) (since 9.1)

Since

2.12

ThreadContextProperties (Object)

Properties for thread context objects.

Members

cpu-affinity: array of int (optional)
the list of host CPU numbers used as CPU affinity for all threads created in the thread context (default: QEMU main thread CPU affinity)

node-affinity: array of int (optional)
the list of host node numbers that will be resolved to a list of host CPU numbers used as CPU affinity. This is a shortcut for specifying the list of host CPU numbers belonging to the host nodes manually by setting `cpu-affinity`. (default: QEMU main thread affinity)

Since

7.2

ObjectType (Enum)

Values

acpi-generic-initiator
Not documented

authz-list
Not documented

authz-listfile
Not documented

authz-pam
Not documented

authz-simple
Not documented

can-bus
Not documented

can-host-socketcan (**If:** CONFIG_LINUX)
Not documented

colo-compare
Not documented

cryptodev-backend
Not documented

cryptodev-backend-builtin
Not documented

cryptodev-backend-lkcf
Not documented

cryptodev-vhost-user (**If:** CONFIG_VHOST_CRYPTO)
Not documented

dbus-vmstate
Not documented

filter-buffer
Not documented

filter-dump
Not documented

filter-mirror
Not documented

filter-redirector
Not documented

filter-replay
Not documented

filter-rewriter
Not documented

input-barrier
Not documented

input-linux (If: CONFIG_LINUX)
: Not documented

iommufd
: Not documented

iothread
: Not documented

main-loop
: Not documented

memory-backend-epc (If: CONFIG_LINUX)
: Not documented

memory-backend-file
: Not documented

memory-backend-memfd (If: CONFIG_LINUX)
: Not documented

memory-backend-ram
: Not documented

pef-guest
: Not documented

pr-manager-helper (If: CONFIG_LINUX)
: Not documented

qtest
: Not documented

rng-builtin
: Not documented

rng-egd
: Not documented

rng-random (If: CONFIG_POSIX)
: Not documented

secret
: Not documented

secret_keyring (If: CONFIG_SECRET_KEYRING)
: Not documented

sev-guest
: Not documented

thread-context
: Not documented

s390-pv-guest
: Not documented

throttle-group
: Not documented

tls-creds-anon
: Not documented

tls-creds-psk
Not documented

tls-creds-x509
Not documented

tls-cipher-suites
Not documented

x-remote-object
Not documented

x-vfio-user-server
Not documented

Features

unstable
Member `x-remote-object` is experimental.

Since

6.0

`ObjectOptions` (Object)

Describes the options of a user creatable QOM object.

Members

qom-type: ObjectType
the class name for the object to be created

id: string
the name of the new object

The members of AcpiGenericInitiatorProperties when qom-type is "acpi-generic-initiator"
The members of AuthZListProperties when qom-type is "authz-list"
The members of AuthZListFileProperties when qom-type is "authz-listfile"
The members of AuthZPAMProperties when qom-type is "authz-pam"
The members of AuthZSimpleProperties when qom-type is "authz-simple"
The members of CanHostSocketcanProperties when qom-type is "can-host-socketcan" (If: CONFIG_LINUX)
The members of ColoCompareProperties when qom-type is "colo-compare"
The members of CryptodevBackendProperties when qom-type is "cryptodev-backend"
The members of CryptodevBackendProperties when qom-type is "cryptodev-backend-builtin"
The members of CryptodevBackendProperties when qom-type is "cryptodev-backend-lkcf"
The members of CryptodevVhostUserProperties when qom-type is "cryptodev-vhost-user" (If: CONFIG_VHOST_CRYPTO)
The members of DBusVMStateProperties when qom-type is "dbus-vmstate"
The members of FilterBufferProperties when qom-type is "filter-buffer"
The members of FilterDumpProperties when qom-type is "filter-dump"
The members of FilterMirrorProperties when qom-type is "filter-mirror"
The members of FilterRedirectorProperties when qom-type is "filter-redirector"
The members of NetfilterProperties when qom-type is "filter-replay"
The members of FilterRewriterProperties when qom-type is "filter-rewriter"
The members of InputBarrierProperties when qom-type is "input-barrier"
The members of InputLinuxProperties when qom-type is "input-linux" (If: CONFIG_LINUX)
The members of IOMMUFDProperties when qom-type is "iommufd"
The members of IothreadProperties when qom-type is "iothread"
The members of MainLoopProperties when qom-type is "main-loop"
The members of MemoryBackendEpcProperties when qom-type is "memory-backend-epc" (If: CONFIG_LINUX)
The members of MemoryBackendFileProperties when qom-type is "memory-backend-file"
The members of MemoryBackendMemfdProperties when qom-type is "memory-backend-memfd" (If: CONFIG_LINUX)
The members of MemoryBackendProperties when qom-type is "memory-backend-ram"
The members of PrManagerHelperProperties when qom-type is "pr-manager-helper" (If: CONFIG_LINUX)
The members of QtestProperties when qom-type is "qtest"
The members of RngProperties when qom-type is "rng-builtin"
The members of RngEgdProperties when qom-type is "rng-egd"
The members of RngRandomProperties when qom-type is "rng-random" (If: CONFIG_POSIX)
The members of SecretProperties when qom-type is "secret"
The members of SecretKeyringProperties when qom-type is "secret_keyring" (If: CONFIG_SECRET_KEYRING)
The members of SevGuestProperties when qom-type is "sev-guest"
The members of ThreadContextProperties when qom-type is "thread-context"
The members of ThrottleGroupProperties when qom-type is "throttle-group"
The members of TlsCredsAnonProperties when qom-type is "tls-creds-anon"
The members of TlsCredsPskProperties when qom-type is "tls-creds-psk"
The members of TlsCredsX509Properties when qom-type is "tls-creds-x509"
The members of TlsCredsProperties when qom-type is "tls-cipher-suites"
The members of RemoteObjectProperties when qom-type is "x-remote-object"
The members of VfioUserServerProperties when qom-type is "x-vfio-user-server"

Since

6.0

object-add (Command)

Create a QOM object.

Arguments

The members of ObjectOptions

Errors

- Error if qom-type is not a valid class name

Since

2.0

Example

```
-> { "execute": "object-add",
     "arguments": { "qom-type": "rng-random", "id": "rng1",
                    "filename": "/dev/hwrng" } }
<- { "return": {} }
```

object-del (Command)

Remove a QOM object.

Arguments

id: string
the name of the QOM object to remove

Errors

- Error if id is not a valid id for a QOM object

Since

2.0

Example

```
-> { "execute": "object-del", "arguments": { "id": "rng1" } }
<- { "return": {} }
```

5.11.25 Device infrastructure (qdev)

device-list-properties (Command)

List properties associated with a device.

Arguments

typename: string
: the type name of a device

Returns

a list of ObjectPropertyInfo describing a devices properties

Note

objects can create properties at runtime, for example to describe links between different devices and/or objects. These properties are not included in the output of this command.

Since

1.2

device_add (Command)

Add a device.

Arguments

driver: string
the name of the new device's driver

bus: string (optional)
the device's parent bus (device tree path)

id: string (optional)
the device's ID, must be unique

Features

json-cli
If present, the "-device" command line option supports JSON syntax with a structure identical to the arguments of this command.

json-cli-hotplug
If present, the "-device" command line option supports JSON syntax without the reference counting leak that broke hot-unplug

Notes

1. Additional arguments depend on the type.
2. For detailed information about this command, please refer to the 'docs/qdev-device-use.txt' file.
3. It's possible to list device properties by running QEMU with the "-device DEVICE,help" command-line argument, where DEVICE is the device's name

Example

```
-> { "execute": "device_add",
     "arguments": { "driver": "e1000", "id": "net1",
                    "bus": "pci.0",
                    "mac": "52:54:00:12:34:56" } }
<- { "return": {} }
```

Since

0.13

device_del (Command)

Remove a device from a guest

Arguments

id: string
: the device's ID or QOM path

Errors

- If id is not a valid device, DeviceNotFound

Notes

When this command completes, the device may not be removed from the guest. Hot removal is an operation that requires guest cooperation. This command merely requests that the guest begin the hot removal process. Completion of the device removal process is signaled with a DEVICE_DELETED event. Guest reset will automatically complete removal for all devices. If a guest-side error in the hot removal process is detected, the device will not be removed and a DEVICE_UNPLUG_GUEST_ERROR event is sent. Some errors cannot be detected.

Since

0.14

Examples

```
-> { "execute": "device_del",
     "arguments": { "id": "net1" } }
<- { "return": {} }

-> { "execute": "device_del",
     "arguments": { "id": "/machine/peripheral-anon/device[0]" } }
<- { "return": {} }
```

DEVICE_DELETED (Event)

Emitted whenever the device removal completion is acknowledged by the guest. At this point, it's safe to reuse the specified device ID. Device removal can be initiated by the guest or by HMP/QMP commands.

Arguments

device: string (optional)
the device's ID if it has one

path: string
the device's QOM path

Since

1.5

Example

```
<- { "event": "DEVICE_DELETED",
     "data": { "device": "virtio-net-pci-0",
               "path": "/machine/peripheral/virtio-net-pci-0" },
     "timestamp": { "seconds": 1265044230, "microseconds": 450486 } }
```

DEVICE_UNPLUG_GUEST_ERROR (Event)

Emitted when a device hot unplug fails due to a guest reported error.

Arguments

device: string (optional)
the device's ID if it has one

path: string
the device's QOM path

Since

6.2

Example

```
<- { "event": "DEVICE_UNPLUG_GUEST_ERROR",
     "data": { "device": "core1",
               "path": "/machine/peripheral/core1" },
     "timestamp": { "seconds": 1615570772, "microseconds": 202844 } }
```

5.11.26 Machines S390 data types

CpuS390Entitlement (Enum)

An enumeration of CPU entitlements that can be assumed by a virtual S390 CPU

Values

auto
Not documented

low
Not documented

medium
Not documented

high
Not documented

Since

8.2

5.11.27 Machines

SysEmuTarget (Enum)

The comprehensive enumeration of QEMU system emulation ("softmmu") targets. Run "./configure –help" in the project root directory, and look for the *-softmmu targets near the "–target-list" option. The individual target constants are not documented here, for the time being.

Values

rx
since 5.0

avr
since 5.1

aarch64
Not documented

alpha
Not documented

arm
Not documented

cris
Not documented

hppa
Not documented

i386
Not documented

loongarch64
Not documented

m68k
Not documented

microblaze
Not documented

microblazeel
Not documented

mips
Not documented

mips64
Not documented

mips64el
Not documented

mipsel
Not documented

or1k
Not documented

ppc
Not documented

ppc64
Not documented

riscv32
Not documented

riscv64
Not documented

s390x
Not documented

sh4
Not documented

sh4eb
Not documented

sparc
Not documented

sparc64
Not documented

tricore
Not documented

x86_64
Not documented

xtensa
Not documented

xtensaeb
Not documented

Notes

The resulting QMP strings can be appended to the "qemu-system-" prefix to produce the corresponding QEMU executable name. This is true even for "qemu-system-x86_64".

Since

3.0

CpuS390State (Enum)

An enumeration of cpu states that can be assumed by a virtual S390 CPU

Values

uninitialized
Not documented

stopped
Not documented

check-stop
Not documented

operating
Not documented

load
Not documented

Since

2.12

CpuInfoS390 (Object)

Additional information about a virtual S390 CPU

Members

cpu-state: CpuS390State
the virtual CPU's state

dedicated: boolean (optional)
the virtual CPU's dedication (since 8.2)

entitlement: CpuS390Entitlement (optional)
the virtual CPU's entitlement (since 8.2)

Since

2.12

CpuInfoFast (Object)

Information about a virtual CPU

Members

cpu-index: int
index of the virtual CPU

qom-path: string
path to the CPU object in the QOM tree

thread-id: int
ID of the underlying host thread

props: CpuInstanceProperties (optional)
properties associated with a virtual CPU, e.g. the socket id

target: SysEmuTarget
the QEMU system emulation target, which determines which additional fields will be listed (since 3.0)

The members of CpuInfoS390 when target is "s390x"

Since

2.12

query-cpus-fast (Command)

Returns information about all virtual CPUs.

Returns

list of `CpuInfoFast`

Since

2.12

Example

```
-> { "execute": "query-cpus-fast" }
<- { "return": [
        {
            "thread-id": 25627,
            "props": {
                "core-id": 0,
                "thread-id": 0,
                "socket-id": 0
            },
            "qom-path": "/machine/unattached/device[0]",
            "target":"x86_64",
            "cpu-index": 0
        },
        {
            "thread-id": 25628,
            "props": {
                "core-id": 0,
                "thread-id": 0,
                "socket-id": 1
            },
            "qom-path": "/machine/unattached/device[2]",
            "target":"x86_64",
            "cpu-index": 1
        }
    ]
}
```

CompatProperty (Object)

Property default values specific to a machine type, for use by scripts/compare-machine-types.

Members

qom-type: string
name of the QOM type to which the default applies

property: string
name of its property to which the default applies

value: string
the default value (machine-specific default can overwrite the "default" default, to avoid this use -machine none)

Since

9.1

MachineInfo (Object)

Information describing a machine.

Members

name: string
the name of the machine

alias: string (optional)
an alias for the machine name

is-default: boolean (optional)
whether the machine is default

cpu-max: int
maximum number of CPUs supported by the machine type (since 1.5)

hotpluggable-cpus: boolean
cpu hotplug via -device is supported (since 2.7)

numa-mem-supported: boolean
true if '-numa node,mem' option is supported by the machine type and false otherwise (since 4.1)

deprecated: boolean
if true, the machine type is deprecated and may be removed in future versions of QEMU according to the QEMU deprecation policy (since 4.1)

default-cpu-type: string (optional)
default CPU model typename if none is requested via the -cpu argument. (since 4.2)

default-ram-id: string (optional)
the default ID of initial RAM memory backend (since 5.2)

acpi: boolean
machine type supports ACPI (since 8.0)

compat-props: array of CompatProperty (optional)
The machine type's compatibility properties. Only present when query-machines argument compat-props is true. (since 9.1)

Features

unstable
Member compat-props is experimental.

Since

1.2

query-machines (Command)

Return a list of supported machines

Arguments

compat-props: boolean (optional)
if true, also return compatibility properties. (default: false) (since 9.1)

Features

unstable
Argument compat-props is experimental.

Returns

a list of MachineInfo

Since

1.2

Example

```
-> { "execute": "query-machines", "arguments": { "compat-props": true } }
<- { "return": [
          {
             "hotpluggable-cpus": true,
             "name": "pc-q35-6.2",
             "compat-props": [
                  {
                     "qom-type": "virtio-mem",
```

(continues on next page)

(continued from previous page)

```
                    "property": "unplugged-inaccessible",
                    "value": "off"
                }
            ],
            "numa-mem-supported": false,
            "default-cpu-type": "qemu64-x86_64-cpu",
            "cpu-max": 288,
            "deprecated": false,
            "default-ram-id": "pc.ram"
        },
        ...
}
```

CurrentMachineParams (Object)

Information describing the running machine parameters.

Members

wakeup-suspend-support: boolean
: true if the machine supports wake up from suspend

Since

4.0

query-current-machine (Command)

Return information on the current virtual machine.

Returns

CurrentMachineParams

Since

4.0

TargetInfo (Object)

Information describing the QEMU target.

Members

arch: SysEmuTarget
the target architecture

Since

1.2

query-target (Command)

Return information about the target for this QEMU

Returns

TargetInfo

Since

1.2

UuidInfo (Object)

Guest UUID information (Universally Unique Identifier).

Members

UUID: string
the UUID of the guest

Since

0.14

Notes

If no UUID was specified for the guest, a null UUID is returned.

query-uuid (Command)

Query the guest UUID information.

Returns

The `UuidInfo` for the guest

Since

0.14

Example

```
-> { "execute": "query-uuid" }
<- { "return": { "UUID": "550e8400-e29b-41d4-a716-446655440000" } }
```

GuidInfo (Object)

GUID information.

Members

guid: string
: the globally unique identifier

Since

2.9

query-vm-generation-id (Command)

Show Virtual Machine Generation ID

Since

2.9

system_reset (Command)

Performs a hard reset of a guest.

Since

0.14

Example

```
-> { "execute": "system_reset" }
<- { "return": {} }
```

system_powerdown (Command)

Requests that a guest perform a powerdown operation.

Since

0.14

Notes

A guest may or may not respond to this command. This command returning does not indicate that a guest has accepted the request or that it has shut down. Many guests will respond to this command by prompting the user in some way.

Example

```
-> { "execute": "system_powerdown" }
<- { "return": {} }
```

system_wakeup (Command)

Wake up guest from suspend. If the guest has wake-up from suspend support enabled (wakeup-suspend-support flag from query-current-machine), wake-up guest from suspend if the guest is in SUSPENDED state. Return an error otherwise.

Since

1.1

Note

prior to 4.0, this command does nothing in case the guest isn't suspended.

Example

```
-> { "execute": "system_wakeup" }
<- { "return": {} }
```

LostTickPolicy (Enum)

Policy for handling lost ticks in timer devices. Ticks end up getting lost when, for example, the guest is paused.

Values

discard
: throw away the missed ticks and continue with future injection normally. The guest OS will see the timer jump ahead by a potentially quite significant amount all at once, as if the intervening chunk of time had simply not existed; needless to say, such a sudden jump can easily confuse a guest OS which is not specifically prepared to deal with it. Assuming the guest OS can deal correctly with the time jump, the time in the guest and in the host should now match.

delay
: continue to deliver ticks at the normal rate. The guest OS will not notice anything is amiss, as from its point of view time will have continued to flow normally. The time in the guest should now be behind the time in the host by exactly the amount of time during which ticks have been missed.

slew
: deliver ticks at a higher rate to catch up with the missed ticks. The guest OS will not notice anything is amiss, as from its point of view time will have continued to flow normally. Once the timer has managed to catch up with all the missing ticks, the time in the guest and in the host should match.

Since

2.0

inject-nmi (Command)

Injects a Non-Maskable Interrupt into the default CPU (x86/s390) or all CPUs (ppc64). The command fails when the guest doesn't support injecting.

Since

0.14

Note

prior to 2.1, this command was only supported for x86 and s390 VMs

Example

```
-> { "execute": "inject-nmi" }
<- { "return": {} }
```

KvmInfo (Object)

Information about support for KVM acceleration

Members

enabled: boolean
: true if KVM acceleration is active

present: boolean
: true if KVM acceleration is built into this executable

Since

0.14

query-kvm (Command)

Returns information about KVM acceleration

Returns

KvmInfo

Since

0.14

Example

```
-> { "execute": "query-kvm" }
<- { "return": { "enabled": true, "present": true } }
```

NumaOptionsType (Enum)

Values

node
: NUMA nodes configuration

dist
: NUMA distance configuration (since 2.10)

cpu
: property based CPU(s) to node mapping (Since: 2.10)

hmat-lb
: memory latency and bandwidth information (Since: 5.0)

hmat-cache
: memory side cache information (Since: 5.0)

Since

2.1

NumaOptions (Object)

A discriminated record of NUMA options. (for OptsVisitor)

Members

type: NumaOptionsType
NUMA option type

The members of NumaNodeOptions when type is "node"
The members of NumaDistOptions when type is "dist"
The members of NumaCpuOptions when type is "cpu"
The members of NumaHmatLBOptions when type is "hmat-lb"
The members of NumaHmatCacheOptions when type is "hmat-cache"

Since

2.1

NumaNodeOptions (Object)

Create a guest NUMA node. (for OptsVisitor)

Members

nodeid: int (optional)
NUMA node ID (increase by 1 from 0 if omitted)

cpus: array of int (optional)
VCPUs belonging to this node (assign VCPUS round-robin if omitted)

mem: int (optional)
memory size of this node; mutually exclusive with memdev. Equally divide total memory among nodes if both mem and memdev are omitted.

memdev: string (optional)
memory backend object. If specified for one node, it must be specified for all nodes.

initiator: int (optional)
defined in ACPI 6.3 Chapter 5.2.27.3 Table 5-145, points to the nodeid which has the memory controller responsible for this NUMA node. This field provides additional information as to the initiator node that is closest (as in directly attached) to this node, and therefore has the best performance (since 5.0)

Since

2.1

NumaDistOptions (Object)

Set the distance between 2 NUMA nodes.

Members

src: int
source NUMA node.

dst: int
destination NUMA node.

val: int
NUMA distance from source node to destination node. When a node is unreachable from another node, set the distance between them to 255.

Since

2.10

CXLFixedMemoryWindowOptions (Object)

Create a CXL Fixed Memory Window

Members

size: int
Size of the Fixed Memory Window in bytes. Must be a multiple of 256MiB.

interleave-granularity: int (optional)
Number of contiguous bytes for which accesses will go to a given interleave target. Accepted values [256, 512, 1k, 2k, 4k, 8k, 16k]

targets: array of string
Target root bridge IDs from -device ...,id=<ID> for each root bridge.

Since

7.1

CXLFMWProperties (Object)

List of CXL Fixed Memory Windows.

Members

cxl-fmw: array of CXLFixedMemoryWindowOptions
List of CXLFixedMemoryWindowOptions

Since

7.1

X86CPURegister32 (Enum)

A X86 32-bit register

Values

EAX
Not documented

EBX
Not documented

ECX
Not documented

EDX
Not documented

ESP
Not documented

EBP
Not documented

ESI
Not documented

EDI
Not documented

Since

1.5

X86CPUFeatureWordInfo (Object)

Information about a X86 CPU feature word

Members

cpuid-input-eax: int
Input EAX value for CPUID instruction for that feature word

cpuid-input-ecx: int (optional)
Input ECX value for CPUID instruction for that feature word

cpuid-register: X86CPURegister32
Output register containing the feature bits

features: int
value of output register, containing the feature bits

Since

1.5

DummyForceArrays (Object)

Not used by QMP; hack to let us use X86CPUFeatureWordInfoList internally

Members

unused: array of X86CPUFeatureWordInfo
Not documented

Since

2.5

NumaCpuOptions (Object)

Option "-numa cpu" overrides default cpu to node mapping. It accepts the same set of cpu properties as returned by query-hotpluggable-cpus[].props, where node-id could be used to override default node mapping.

Members

The members of CpuInstanceProperties

Since

2.10

HmatLBMemoryHierarchy (Enum)

The memory hierarchy in the System Locality Latency and Bandwidth Information Structure of HMAT (Heterogeneous Memory Attribute Table)

For more information about HmatLBMemoryHierarchy, see chapter 5.2.27.4: Table 5-146: Field "Flags" of ACPI 6.3 spec.

Values

memory
: the structure represents the memory performance

first-level
: first level of memory side cache

second-level
: second level of memory side cache

third-level
: third level of memory side cache

Since

5.0

HmatLBDataType (Enum)

Data type in the System Locality Latency and Bandwidth Information Structure of HMAT (Heterogeneous Memory Attribute Table)

For more information about HmatLBDataType, see chapter 5.2.27.4: Table 5-146: Field "Data Type" of ACPI 6.3 spec.

Values

access-latency
: access latency (nanoseconds)

read-latency
: read latency (nanoseconds)

write-latency
: write latency (nanoseconds)

access-bandwidth
: access bandwidth (Bytes per second)

read-bandwidth
: read bandwidth (Bytes per second)

write-bandwidth
write bandwidth (Bytes per second)

Since

5.0

NumaHmatLBOptions (Object)

Set the system locality latency and bandwidth information between Initiator and Target proximity Domains.

For more information about NumaHmatLBOptions, see chapter 5.2.27.4: Table 5-146 of ACPI 6.3 spec.

Members

initiator: int
the Initiator Proximity Domain.

target: int
the Target Proximity Domain.

hierarchy: HmatLBMemoryHierarchy
the Memory Hierarchy. Indicates the performance of memory or side cache.

data-type: HmatLBDataType
presents the type of data, access/read/write latency or hit latency.

latency: int (optional)
the value of latency from initiator to target proximity domain, the latency unit is "ns(nanosecond)".

bandwidth: int (optional)
the value of bandwidth between initiator and target proximity domain, the bandwidth unit is "Bytes per second".

Since

5.0

HmatCacheAssociativity (Enum)

Cache associativity in the Memory Side Cache Information Structure of HMAT

For more information of HmatCacheAssociativity, see chapter 5.2.27.5: Table 5-147 of ACPI 6.3 spec.

Values

none
None (no memory side cache in this proximity domain, or cache associativity unknown)

direct
Direct Mapped

complex
Complex Cache Indexing (implementation specific)

Since

5.0

HmatCacheWritePolicy (Enum)

Cache write policy in the Memory Side Cache Information Structure of HMAT

For more information of HmatCacheWritePolicy, see chapter 5.2.27.5: Table 5-147: Field "Cache Attributes" of ACPI 6.3 spec.

Values

none
None (no memory side cache in this proximity domain, or cache write policy unknown)

write-back
Write Back (WB)

write-through
Write Through (WT)

Since

5.0

NumaHmatCacheOptions (Object)

Set the memory side cache information for a given memory domain.

For more information of NumaHmatCacheOptions, see chapter 5.2.27.5: Table 5-147: Field "Cache Attributes" of ACPI 6.3 spec.

Members

node-id: int
the memory proximity domain to which the memory belongs.

size: int
the size of memory side cache in bytes.

level: int
the cache level described in this structure.

associativity: HmatCacheAssociativity
the cache associativity, none/direct-mapped/complex(complex cache indexing).

policy: HmatCacheWritePolicy
the write policy, none/write-back/write-through.

line: int
the cache Line size in bytes.

Since

5.0

memsave (Command)

Save a portion of guest memory to a file.

Arguments

val: int
the virtual address of the guest to start from

size: int
the size of memory region to save

filename: string
the file to save the memory to as binary data

cpu-index: int (optional)
the index of the virtual CPU to use for translating the virtual address (defaults to CPU 0)

Since

0.14

Notes

Errors were not reliably returned until 1.1

Example

```
-> { "execute": "memsave",
     "arguments": { "val": 10,
                    "size": 100,
                    "filename": "/tmp/virtual-mem-dump" } }
<- { "return": {} }
```

pmemsave (Command)

Save a portion of guest physical memory to a file.

Arguments

val: int
: the physical address of the guest to start from

size: int
: the size of memory region to save

filename: string
: the file to save the memory to as binary data

Since

0.14

Notes

Errors were not reliably returned until 1.1

Example

```
-> { "execute": "pmemsave",
     "arguments": { "val": 10,
                    "size": 100,
                    "filename": "/tmp/physical-mem-dump" } }
<- { "return": {} }
```

Memdev (Object)

Information about memory backend

Members

id: string (optional)
backend's ID if backend has 'id' property (since 2.9)

size: int
memory backend size

merge: boolean
whether memory merge support is enabled

dump: boolean
whether memory backend's memory is included in a core dump

prealloc: boolean
whether memory was preallocated

share: boolean
whether memory is private to QEMU or shared (since 6.1)

reserve: boolean (optional)
whether swap space (or huge pages) was reserved if applicable. This corresponds to the user configuration and not the actual behavior implemented in the OS to perform the reservation. For example, Linux will never reserve swap space for shared file mappings. (since 6.1)

host-nodes: array of int
host nodes for its memory policy

policy: HostMemPolicy
memory policy of memory backend

Since

2.1

query-memdev (Command)

Returns information for all memory backends.

Returns

a list of Memdev.

Since

2.1

Example

```
-> { "execute": "query-memdev" }
<- { "return": [
       {
         "id": "mem1",
         "size": 536870912,
         "merge": false,
         "dump": true,
         "prealloc": false,
         "share": false,
         "host-nodes": [0, 1],
         "policy": "bind"
       },
       {
         "size": 536870912,
         "merge": false,
         "dump": true,
         "prealloc": true,
         "share": false,
         "host-nodes": [2, 3],
         "policy": "preferred"
       }
     ]
   }
```

`CpuInstanceProperties` (Object)

List of properties to be used for hotplugging a CPU instance, it should be passed by management with device_add command when a CPU is being hotplugged.

Which members are optional and which mandatory depends on the architecture and board.

For s390x see *CPU topology on s390x*.

The ids other than the node-id specify the position of the CPU within the CPU topology (as defined by the machine property "smp", thus see also type `SMPConfiguration`)

Members

`node-id`: `int` (optional)
: NUMA node ID the CPU belongs to

`drawer-id`: `int` (optional)
: drawer number within CPU topology the CPU belongs to (since 8.2)

`book-id`: `int` (optional)
: book number within parent container the CPU belongs to (since 8.2)

socket-id: int (optional)
socket number within parent container the CPU belongs to

die-id: int (optional)
die number within the parent container the CPU belongs to (since 4.1)

cluster-id: int (optional)
cluster number within the parent container the CPU belongs to (since 7.1)

module-id: int (optional)
module number within the parent container the CPU belongs to (since 9.1)

core-id: int (optional)
core number within the parent container the CPU belongs to

thread-id: int (optional)
thread number within the core the CPU belongs to

Note

management should be prepared to pass through additional properties with device_add.

Since

2.7

HotpluggableCPU (Object)

Members

type: string
CPU object type for usage with device_add command

props: CpuInstanceProperties
list of properties to be used for hotplugging CPU

vcpus-count: int
number of logical VCPU threads HotpluggableCPU provides

qom-path: string (optional)
link to existing CPU object if CPU is present or omitted if CPU is not present.

Since

2.7

query-hotpluggable-cpus (Command)

Returns

a list of HotpluggableCPU objects.

Since

2.7

Examples

```
For pseries machine type started with -smp 2,cores=2,maxcpus=4
-cpu POWER8:

-> { "execute": "query-hotpluggable-cpus" }
<- {"return": [
     { "props": { "core-id": 8 }, "type": "POWER8-spapr-cpu-core",
       "vcpus-count": 1 },
     { "props": { "core-id": 0 }, "type": "POWER8-spapr-cpu-core",
       "vcpus-count": 1, "qom-path": "/machine/unattached/device[0]"}
   ]}'

For pc machine type started with -smp 1,maxcpus=2:

-> { "execute": "query-hotpluggable-cpus" }
<- {"return": [
     {
        "type": "qemu64-x86_64-cpu", "vcpus-count": 1,
        "props": {"core-id": 0, "socket-id": 1, "thread-id": 0}
     },
     {
        "qom-path": "/machine/unattached/device[0]",
        "type": "qemu64-x86_64-cpu", "vcpus-count": 1,
        "props": {"core-id": 0, "socket-id": 0, "thread-id": 0}
     }
   ]}

For s390x-virtio-ccw machine type started with -smp 1,maxcpus=2
-cpu qemu (Since: 2.11):

-> { "execute": "query-hotpluggable-cpus" }
<- {"return": [
     {
        "type": "qemu-s390x-cpu", "vcpus-count": 1,
        "props": { "core-id": 1 }
     },
     {
        "qom-path": "/machine/unattached/device[0]",
        "type": "qemu-s390x-cpu", "vcpus-count": 1,
```

(continues on next page)

(continued from previous page)

```
        "props": { "core-id": 0 }
    }
  ]}
```

set-numa-node (Command)

Runtime equivalent of '-numa' CLI option, available at preconfigure stage to configure numa mapping before initializing machine.

Arguments

The members of NumaOptions

Since

3.0

balloon (Command)

Request the balloon driver to change its balloon size.

Arguments

value: int
: the target logical size of the VM in bytes. We can deduce the size of the balloon using this formula:

    logical_vm_size = vm_ram_size - balloon_size

  From it we have: balloon_size = vm_ram_size - value

Errors

- If the balloon driver is enabled but not functional because the KVM kernel module cannot support it, KVMMissingCap
- If no balloon device is present, DeviceNotActive

Notes

This command just issues a request to the guest. When it returns, the balloon size may not have changed. A guest can change the balloon size independent of this command.

Since

0.14

Example

```
-> { "execute": "balloon", "arguments": { "value": 536870912 } }
<- { "return": {} }

With a 2.5GiB guest this command inflated the ballon to 3GiB.
```

BalloonInfo (Object)

Information about the guest balloon device.

Members

actual: int
: the logical size of the VM in bytes Formula used: logical_vm_size = vm_ram_size - balloon_size

Since

0.14

query-balloon (Command)

Return information about the balloon device.

Returns

BalloonInfo

Errors

- If the balloon driver is enabled but not functional because the KVM kernel module cannot support it, KVMMissingCap
- If no balloon device is present, DeviceNotActive

Since

0.14

Example

```
-> { "execute": "query-balloon" }
<- { "return": {
         "actual": 1073741824
      }
   }
```

BALLOON_CHANGE (Event)

Emitted when the guest changes the actual BALLOON level. This value is equivalent to the `actual` field return by the 'query-balloon' command

Arguments

actual: int
: the logical size of the VM in bytes Formula used: logical_vm_size = vm_ram_size - balloon_size

Note

this event is rate-limited.

Since

1.2

Example

```
<- { "event": "BALLOON_CHANGE",
     "data": { "actual": 944766976 },
     "timestamp": { "seconds": 1267020223, "microseconds": 435656 } }
```

HvBalloonInfo (Object)

hv-balloon guest-provided memory status information.

Members

committed: int
the amount of memory in use inside the guest plus the amount of the memory unusable inside the guest (ballooned out, offline, etc.)

available: int
the amount of the memory inside the guest available for new allocations (“free”)

Since

8.2

query-hv-balloon-status-report (Command)

Returns the hv-balloon driver data contained in the last received “STATUS” message from the guest.

Returns

HvBalloonInfo

Errors

- If no hv-balloon device is present, guest memory status reporting is not enabled or no guest memory status report received yet, GenericError

Since

8.2

Example

```
-> { "execute": "query-hv-balloon-status-report" }
<- { "return": {
         "committed": 816640000,
         "available": 3333054464
     }
   }
```

HV_BALLOON_STATUS_REPORT (Event)

Emitted when the hv-balloon driver receives a "STATUS" message from the guest.

Note

this event is rate-limited.

Since

8.2

Example

```
<- { "event": "HV_BALLOON_STATUS_REPORT",
     "data": { "committed": 816640000, "available": 3333054464 },
     "timestamp": { "seconds": 1600295492, "microseconds": 661044 } }
```

MemoryInfo (Object)

Actual memory information in bytes.

Members

base-memory: int
size of "base" memory specified with command line option -m.

plugged-memory: int (optional)
size of memory that can be hot-unplugged. This field is omitted if target doesn't support memory hotplug (i.e. CONFIG_MEM_DEVICE not defined at build time).

Since

2.11

query-memory-size-summary (Command)

Return the amount of initially allocated and present hotpluggable (if enabled) memory in bytes.

Example

```
-> { "execute": "query-memory-size-summary" }
<- { "return": { "base-memory": 4294967296, "plugged-memory": 0 } }
```

Since

2.11

PCDIMMDeviceInfo (Object)

PCDIMMDevice state information

Members

id: string (optional)
device's ID

addr: int
physical address, where device is mapped

size: int
size of memory that the device provides

slot: int
slot number at which device is plugged in

node: int
NUMA node number where device is plugged in

memdev: string
memory backend linked with device

hotplugged: boolean
true if device was hotplugged

hotpluggable: boolean
true if device if could be added/removed while machine is running

Since

2.1

VirtioPMEMDeviceInfo (Object)

VirtioPMEM state information

Members

id: string (optional)
device's ID

memaddr: int
physical address in memory, where device is mapped

size: int
size of memory that the device provides

memdev: string
memory backend linked with device

Since

4.1

VirtioMEMDeviceInfo (Object)

VirtioMEMDevice state information

Members

id: string (optional)
device's ID

memaddr: int
physical address in memory, where device is mapped

requested-size: int
the user requested size of the device

size: int
the (current) size of memory that the device provides

max-size: int
the maximum size of memory that the device can provide

block-size: int
the block size of memory that the device provides

node: int
NUMA node number where device is assigned to

memdev: string
memory backend linked with the region

Since

5.1

SgxEPCDeviceInfo (Object)

Sgx EPC state information

Members

id: string (optional)
device's ID

memaddr: int
physical address in memory, where device is mapped

size: int
size of memory that the device provides

memdev: string
memory backend linked with device

node: int
the numa node (Since: 7.0)

Since

6.2

HvBalloonDeviceInfo (Object)

hv-balloon provided memory state information

Members

id: string (optional)
device's ID

memaddr: int (optional)
physical address in memory, where device is mapped

max-size: int
the maximum size of memory that the device can provide

memdev: string (optional)
memory backend linked with device

Since

8.2

MemoryDeviceInfoKind (Enum)

Values

nvdimm
since 2.12

virtio-pmem
since 4.1

virtio-mem
since 5.1

sgx-epc
since 6.2.

hv-balloon
since 8.2.

dimm
Not documented

Since

2.1

PCDIMMDeviceInfoWrapper (Object)

Members

data: PCDIMMDeviceInfo
PCDIMMDevice state information

Since

2.1

VirtioPMEMDeviceInfoWrapper (Object)

Members

data: VirtioPMEMDeviceInfo
VirtioPMEM state information

Since

2.1

VirtioMEMDeviceInfoWrapper (Object)

Members

data: VirtioMEMDeviceInfo
VirtioMEMDevice state information

Since

2.1

SgxEPCDeviceInfoWrapper (Object)

Members

data: SgxEPCDeviceInfo
Sgx EPC state information

Since

6.2

HvBalloonDeviceInfoWrapper (Object)

Members

data: HvBalloonDeviceInfo
hv-balloon provided memory state information

Since

8.2

MemoryDeviceInfo (Object)

Union containing information about a memory device

Members

type: MemoryDeviceInfoKind
memory device type

The members of PCDIMMDeviceInfoWrapper when type is "dimm"
The members of PCDIMMDeviceInfoWrapper when type is "nvdimm"
The members of VirtioPMEMDeviceInfoWrapper when type is "virtio-pmem"
The members of VirtioMEMDeviceInfoWrapper when type is "virtio-mem"
The members of SgxEPCDeviceInfoWrapper when type is "sgx-epc"
The members of HvBalloonDeviceInfoWrapper when type is "hv-balloon"

Since

2.1

SgxEPC (Object)

Sgx EPC cmdline information

Members

memdev: string
memory backend linked with device

node: int
the numa node (Since: 7.0)

Since

6.2

SgxEPCProperties (Object)

SGX properties of machine types.

Members

sgx-epc: array of SgxEPC
list of ids of memory-backend-epc objects.

Since

6.2

query-memory-devices (Command)

Lists available memory devices and their state

Since

2.1

Example

```
-> { "execute": "query-memory-devices" }
<- { "return": [ { "data":
                      { "addr": 5368709120,
                        "hotpluggable": true,
                        "hotplugged": true,
                        "id": "d1",
                        "memdev": "/objects/memX",
                        "node": 0,
                        "size": 1073741824,
                        "slot": 0},
                    "type": "dimm"
                  } ] }
```

MEMORY_DEVICE_SIZE_CHANGE (Event)

Emitted when the size of a memory device changes. Only emitted for memory devices that can actually change the size (e.g., virtio-mem due to guest action).

Arguments

id: string (optional)
: device's ID

size: int
: the new size of memory that the device provides

qom-path: string
: path to the device object in the QOM tree (since 6.2)

Note

this event is rate-limited.

Since

5.1

Example

```
<- { "event": "MEMORY_DEVICE_SIZE_CHANGE",
     "data": { "id": "vm0", "size": 1073741824,
               "qom-path": "/machine/unattached/device[2]" },
     "timestamp": { "seconds": 1588168529, "microseconds": 201316 } }
```

MEM_UNPLUG_ERROR (Event)

Emitted when memory hot unplug error occurs.

Arguments

device: string
device name

msg: string
Informative message

Features

deprecated
This event is deprecated. Use DEVICE_UNPLUG_GUEST_ERROR instead.

Since

2.4

Example

```
<- { "event": "MEM_UNPLUG_ERROR",
     "data": { "device": "dimm1",
               "msg": "acpi: device unplug for unsupported device"
     },
     "timestamp": { "seconds": 1265044230, "microseconds": 450486 } }
```

`BootConfiguration` (Object)

Schema for virtual machine boot configuration.

Members

`order`: `string` (optional)
Boot order (a=floppy, c=hard disk, d=CD-ROM, n=network)

`once`: `string` (optional)
Boot order to apply on first boot

`menu`: `boolean` (optional)
Whether to show a boot menu

`splash`: `string` (optional)
The name of the file to be passed to the firmware as logo picture, if `menu` is true.

`splash-time`: `int` (optional)
How long to show the logo picture, in milliseconds

`reboot-timeout`: `int` (optional)
Timeout before guest reboots after boot fails

`strict`: `boolean` (optional)
Whether to attempt booting from devices not included in the boot order

Since

7.1

`SMPConfiguration` (Object)

Schema for CPU topology configuration. A missing value lets QEMU figure out a suitable value based on the ones that are provided.

The members other than `cpus` and `maxcpus` define a topology of containers.

The ordering from highest/coarsest to lowest/finest is: `drawers`, `books`, `sockets`, `dies`, `clusters`, `cores`, `threads`.

Different architectures support different subsets of topology containers.

For example, s390x does not have clusters and dies, and the socket is the parent container of cores.

Members

`cpus`: `int` (optional)
number of virtual CPUs in the virtual machine

`maxcpus`: `int` (optional)
maximum number of hotpluggable virtual CPUs in the virtual machine

`drawers`: `int` (optional)
number of drawers in the CPU topology (since 8.2)

`books`: `int` (optional)
number of books in the CPU topology (since 8.2)

sockets: int (optional)
number of sockets per parent container

dies: int (optional)
number of dies per parent container

clusters: int (optional)
number of clusters per parent container (since 7.0)

modules: int (optional)
number of modules per parent container (since 9.1)

cores: int (optional)
number of cores per parent container

threads: int (optional)
number of threads per core

Since

6.1

x-query-irq (Command)

Query interrupt statistics

Features

unstable
This command is meant for debugging.

Returns

interrupt statistics

Since

6.2

x-query-jit (Command)

Query TCG compiler statistics

Features

unstable
: This command is meant for debugging.

Returns

TCG compiler statistics

Since

6.2

If

`CONFIG_TCG`

x-query-numa (Command)

Query NUMA topology information

Features

unstable
: This command is meant for debugging.

Returns

topology information

Since

6.2

x-query-opcount (Command)

Query TCG opcode counters

Features

unstable
: This command is meant for debugging.

Returns

TCG opcode counters

Since

6.2

If

`CONFIG_TCG`

x-query-ramblock (Command)

Query system ramblock information

Features

unstable
: This command is meant for debugging.

Returns

system ramblock information

Since

6.2

x-query-roms (Command)

Query information on the registered ROMS

Features

unstable
This command is meant for debugging.

Returns

registered ROMs

Since

6.2

x-query-usb (Command)

Query information on the USB devices

Features

unstable
This command is meant for debugging.

Returns

USB device information

Since

6.2

SmbiosEntryPointType (Enum)

Values

32
SMBIOS version 2.1 (32-bit) Entry Point

64
SMBIOS version 3.0 (64-bit) Entry Point

auto
Either 2.x or 3.x SMBIOS version, 2.x if configuration can be described by it and 3.x otherwise (since: 9.0)

Since

7.0

`MemorySizeConfiguration` (Object)

Schema for memory size configuration.

Members

`size`: `int` (optional)
: memory size in bytes

`max-size`: `int` (optional)
: maximum hotpluggable memory size in bytes

`slots`: `int` (optional)
: number of available memory slots for hotplug

Since

7.1

`dumpdtb` (Command)

Save the FDT in dtb format.

Arguments

`filename`: `string`
: name of the dtb file to be created

Since

7.2

Example

```
-> { "execute": "dumpdtb" }
     "arguments": { "filename": "fdt.dtb" } }
<- { "return": {} }
```

If

CONFIG_FDT

CpuModelInfo (Object)

Virtual CPU model.

A CPU model consists of the name of a CPU definition, to which delta changes are applied (e.g. features added/removed). Most magic values that an architecture might require should be hidden behind the name. However, if required, architectures can expose relevant properties.

Members

name: string
the name of the CPU definition the model is based on

props: value (optional)
a dictionary of QOM properties to be applied

deprecated-props: array of string (optional)
a list of properties that are flagged as deprecated by the CPU vendor. These props are a subset of the full model's definition list of properties. (since 9.1)

Since

2.8

CpuModelExpansionType (Enum)

An enumeration of CPU model expansion types.

Values

static
Expand to a static CPU model, a combination of a static base model name and property delta changes. As the static base model will never change, the expanded CPU model will be the same, independent of QEMU version, machine type, machine options, and accelerator options. Therefore, the resulting model can be used by tooling without having to specify a compatibility machine - e.g. when displaying the "host" model. The `static` CPU models are migration-safe.

full
Expand all properties. The produced model is not guaranteed to be migration-safe, but allows tooling to get an insight and work with model details.

Note

When a non-migration-safe CPU model is expanded in static mode, some features enabled by the CPU model may be omitted, because they can't be implemented by a static CPU model definition (e.g. cache info passthrough and PMU passthrough in x86). If you need an accurate representation of the features enabled by a non-migration-safe CPU model, use `full`. If you need a static representation that will keep ABI compatibility even when changing QEMU version or machine-type, use `static` (but keep in mind that some features may be omitted).

Since

2.8

`CpuModelCompareResult` (Enum)

An enumeration of CPU model comparison results. The result is usually calculated using e.g. CPU features or CPU generations.

Values

`incompatible`
: If model A is incompatible to model B, model A is not guaranteed to run where model B runs and the other way around.

`identical`
: If model A is identical to model B, model A is guaranteed to run where model B runs and the other way around.

`superset`
: If model A is a superset of model B, model B is guaranteed to run where model A runs. There are no guarantees about the other way.

`subset`
: If model A is a subset of model B, model A is guaranteed to run where model B runs. There are no guarantees about the other way.

Since

2.8

`CpuModelBaselineInfo` (Object)

The result of a CPU model baseline.

Members

model: CpuModelInfo
the baselined CpuModelInfo.

Since

2.8

If

TARGET_S390X

CpuModelCompareInfo (Object)

The result of a CPU model comparison.

Members

result: CpuModelCompareResult
The result of the compare operation.

responsible-properties: array of string
List of properties that led to the comparison result not being identical.

`responsible-properties` is a list of QOM property names that led to both CPUs not being detected as identical. For identical models, this list is empty. If a QOM property is read-only, that means there's no known way to make the CPU models identical. If the special property name "type" is included, the models are by definition not identical and cannot be made identical.

Since

2.8

If

TARGET_S390X

query-cpu-model-comparison (Command)

Compares two CPU models, `modela` and `modelb`, returning how they compare in a specific configuration. The results indicates how both models compare regarding runnability. This result can be used by tooling to make decisions if a certain CPU model will run in a certain configuration or if a compatible CPU model has to be created by baselining.

Usually, a CPU model is compared against the maximum possible CPU model of a certain configuration (e.g. the "host" model for KVM). If that CPU model is identical or a subset, it will run in that configuration.

The result returned by this command may be affected by:

- QEMU version: CPU models may look different depending on the QEMU version. (Except for CPU models reported as "static" in query-cpu-definitions.)
- machine-type: CPU model may look different depending on the machine-type. (Except for CPU models reported as "static" in query-cpu-definitions.)
- machine options (including accelerator): in some architectures, CPU models may look different depending on machine and accelerator options. (Except for CPU models reported as "static" in query-cpu-definitions.)
- "-cpu" arguments and global properties: arguments to the -cpu option and global properties may affect expansion of CPU models. Using query-cpu-model-expansion while using these is not advised.

Some architectures may not support comparing CPU models. s390x supports comparing CPU models.

Arguments

modela: CpuModelInfo
: description of the first CPU model to compare, referred to as "model A" in CpuModelCompareResult

modelb: CpuModelInfo
: description of the second CPU model to compare, referred to as "model B" in CpuModelCompareResult

Returns

a CpuModelCompareInfo describing how both CPU models compare

Errors

- if comparing CPU models is not supported
- if a model cannot be used
- if a model contains an unknown cpu definition name, unknown properties or properties with wrong types.

Note

this command isn't specific to s390x, but is only implemented on this architecture currently.

Since

2.8

If

TARGET_S390X

query-cpu-model-baseline (Command)

Baseline two CPU models, `modela` and `modelb`, creating a compatible third model. The created model will always be a static, migration-safe CPU model (see "static" CPU model expansion for details).

This interface can be used by tooling to create a compatible CPU model out two CPU models. The created CPU model will be identical to or a subset of both CPU models when comparing them. Therefore, the created CPU model is guaranteed to run where the given CPU models run.

The result returned by this command may be affected by:

- QEMU version: CPU models may look different depending on the QEMU version. (Except for CPU models reported as "static" in query-cpu-definitions.)
- machine-type: CPU model may look different depending on the machine-type. (Except for CPU models reported as "static" in query-cpu-definitions.)
- machine options (including accelerator): in some architectures, CPU models may look different depending on machine and accelerator options. (Except for CPU models reported as "static" in query-cpu-definitions.)
- "-cpu" arguments and global properties: arguments to the -cpu option and global properties may affect expansion of CPU models. Using query-cpu-model-expansion while using these is not advised.

Some architectures may not support baselining CPU models. s390x supports baselining CPU models.

Arguments

modela: CpuModelInfo
: description of the first CPU model to baseline

modelb: CpuModelInfo
: description of the second CPU model to baseline

Returns

a CpuModelBaselineInfo describing the baselined CPU model

Errors

- if baselining CPU models is not supported
- if a model cannot be used
- if a model contains an unknown cpu definition name, unknown properties or properties with wrong types.

Note

this command isn't specific to s390x, but is only implemented on this architecture currently.

Since

2.8

If

`TARGET_S390X`

CpuModelExpansionInfo (Object)

The result of a cpu model expansion.

Members

model: CpuModelInfo
the expanded CpuModelInfo.

Since

2.8

If

`TARGET_S390X or TARGET_I386 or TARGET_ARM or TARGET_LOONGARCH64 or TARGET_RISCV`

query-cpu-model-expansion (Command)

Expands a given CPU model, `model`, (or a combination of CPU model + additional options) to different granularities, specified by `type`, allowing tooling to get an understanding what a specific CPU model looks like in QEMU under a certain configuration.

This interface can be used to query the "host" CPU model.

The data returned by this command may be affected by:

- QEMU version: CPU models may look different depending on the QEMU version. (Except for CPU models reported as "static" in query-cpu-definitions.)
- machine-type: CPU model may look different depending on the machine-type. (Except for CPU models reported as "static" in query-cpu-definitions.)
- machine options (including accelerator): in some architectures, CPU models may look different depending on machine and accelerator options. (Except for CPU models reported as "static" in query-cpu-definitions.)
- "-cpu" arguments and global properties: arguments to the -cpu option and global properties may affect expansion of CPU models. Using query-cpu-model-expansion while using these is not advised.

Some architectures may not support all expansion types. s390x supports "full" and "static". Arm only supports "full".

Arguments

model: CpuModelInfo
description of the CPU model to expand

type: CpuModelExpansionType
expansion type, specifying how to expand the CPU model

Returns

a CpuModelExpansionInfo describing the expanded CPU model

Errors

- if expanding CPU models is not supported
- if the model cannot be expanded
- if the model contains an unknown CPU definition name, unknown properties or properties with a wrong type
- if an expansion type is not supported

Since

2.8

If

```
TARGET_S390X or TARGET_I386 or TARGET_ARM or TARGET_LOONGARCH64 or TARGET_RISCV
```

CpuDefinitionInfo (Object)

Virtual CPU definition.

Members

name: string
the name of the CPU definition

migration-safe: boolean (optional)
whether a CPU definition can be safely used for migration in combination with a QEMU compatibility machine when migrating between different QEMU versions and between hosts with different sets of (hardware or software) capabilities. If not provided, information is not available and callers should not assume the CPU definition to be migration-safe. (since 2.8)

static: boolean
whether a CPU definition is static and will not change depending on QEMU version, machine type, machine options and accelerator options. A static model is always migration-safe. (since 2.8)

unavailable-features: array of string (optional)
List of properties that prevent the CPU model from running in the current host. (since 2.8)

typename: string
Type name that can be used as argument to `device-list-properties`, to introspect properties configurable using -cpu or -global. (since 2.9)

alias-of: string (optional)
Name of CPU model this model is an alias for. The target of the CPU model alias may change depending on the machine type. Management software is supposed to translate CPU model aliases in the VM configuration, because aliases may stop being migration-safe in the future (since 4.1)

deprecated: boolean
If true, this CPU model is deprecated and may be removed in in some future version of QEMU according to the QEMU deprecation policy. (since 5.2)

`unavailable-features` is a list of QOM property names that represent CPU model attributes that prevent the CPU from running. If the QOM property is read-only, that means there's no known way to make the CPU model run in the current host. Implementations that choose not to provide specific information return the property name "type". If the property is read-write, it means that it MAY be possible to run the CPU model in the current host if that property is changed. Management software can use it as hints to suggest or choose an alternative for the user, or just to generate meaningful error messages explaining why the CPU model can't be used. If `unavailable-features` is an empty list, the CPU model is runnable using the current host and machine-type. If `unavailable-features` is not present, runnability information for the CPU is not available.

Since

1.2

If

```
TARGET_PPC or TARGET_ARM or TARGET_I386 or TARGET_S390X or TARGET_MIPS or
TARGET_LOONGARCH64 or TARGET_RISCV
```

`query-cpu-definitions` (Command)

Return a list of supported virtual CPU definitions

Returns

a list of CpuDefinitionInfo

Since

1.2

If

```
TARGET_PPC or TARGET_ARM or TARGET_I386 or TARGET_S390X or TARGET_MIPS or
TARGET_LOONGARCH64 or TARGET_RISCV
```

CpuS390Polarization (Enum)

An enumeration of CPU polarization that can be assumed by a virtual S390 CPU

Values

horizontal
Not documented

vertical
Not documented

Since

8.2

If

```
TARGET_S390X
```

set-cpu-topology (Command)

Modify the topology by moving the CPU inside the topology tree, or by changing a modifier attribute of a CPU. Absent values will not be modified.

Arguments

core-id: int
the vCPU ID to be moved

socket-id: int (optional)
destination socket to move the vCPU to

book-id: int (optional)
destination book to move the vCPU to

drawer-id: int (optional)
destination drawer to move the vCPU to

entitlement: CpuS390Entitlement (optional)
entitlement to set

dedicated: boolean (optional)
whether the provisioning of real to virtual CPU is dedicated

Features

unstable
This command is experimental.

Since

8.2

If

`TARGET_S390X and CONFIG_KVM`

CPU_POLARIZATION_CHANGE (Event)

Emitted when the guest asks to change the polarization.

The guest can tell the host (via the PTF instruction) whether the CPUs should be provisioned using horizontal or vertical polarization.

On horizontal polarization the host is expected to provision all vCPUs equally.

On vertical polarization the host can provision each vCPU differently. The guest will get information on the details of the provisioning the next time it uses the STSI(15) instruction.

Arguments

polarization: CpuS390Polarization
polarization specified by the guest

Features

unstable
This event is experimental.

Since

8.2

Example

```
<- { "event": "CPU_POLARIZATION_CHANGE",
     "data": { "polarization": "horizontal" },
     "timestamp": { "seconds": 1401385907, "microseconds": 422329 } }
```

If

TARGET_S390X and CONFIG_KVM

CpuPolarizationInfo (Object)

The result of a CPU polarization query.

Members

polarization: CpuS390Polarization
the CPU polarization

Since

8.2

If

TARGET_S390X and CONFIG_KVM

query-s390x-cpu-polarization (Command)

Features

unstable
This command is experimental.

Returns

the machine's CPU polarization

Since

8.2

If

TARGET_S390X and CONFIG_KVM

5.11.28 Record/replay

ReplayMode (Enum)

Mode of the replay subsystem.

Values

none
: normal execution mode. Replay or record are not enabled.

record
: record mode. All non-deterministic data is written into the replay log.

play
: replay mode. Non-deterministic data required for system execution is read from the log.

Since

2.5

ReplayInfo (Object)

Record/replay information.

Members

mode: ReplayMode
: current mode.

filename: string (optional)
: name of the record/replay log file. It is present only in record or replay modes, when the log is recorded or replayed.

icount: int
: current number of executed instructions.

Since

5.2

query-replay (Command)

Retrieve the record/replay information. It includes current instruction count which may be used for replay-break and replay-seek commands.

Returns

record/replay information.

Since

5.2

Example

```
-> { "execute": "query-replay" }
<- { "return": { "mode": "play", "filename": "log.rr", "icount": 220414 } }
```

replay-break (Command)

Set replay breakpoint at instruction count icount. Execution stops when the specified instruction is reached. There can be at most one breakpoint. When breakpoint is set, any prior one is removed. The breakpoint may be set only in replay mode and only "in the future", i.e. at instruction counts greater than the current one. The current instruction count can be observed with query-replay.

Arguments

icount: int
: instruction count to stop at

Since

5.2

Example

```
-> { "execute": "replay-break", "arguments": { "icount": 220414 } }
<- { "return": {} }
```

replay-delete-break (Command)

Remove replay breakpoint which was set with `replay-break`. The command is ignored when there are no replay breakpoints.

Since

5.2

Example

```
-> { "execute": "replay-delete-break" }
<- { "return": {} }
```

replay-seek (Command)

Automatically proceed to the instruction count `icount`, when replaying the execution. The command automatically loads nearest snapshot and replays the execution to find the desired instruction. When there is no preceding snapshot or the execution is not replayed, then the command fails. Instruction count can be obtained with the `query-replay` command.

Arguments

icount: int
target instruction count

Since

5.2

Example

```
-> { "execute": "replay-seek", "arguments": { "icount": 220414 } }
<- { "return": {} }
```

5.11.29 Yank feature

YankInstanceType (Enum)

An enumeration of yank instance types. See `YankInstance` for more information.

Values

block-node
Not documented

chardev
Not documented

migration
Not documented

Since

6.0

YankInstanceBlockNode (Object)

Specifies which block graph node to yank. See YankInstance for more information.

Members

node-name: string
the name of the block graph node

Since

6.0

YankInstanceChardev (Object)

Specifies which character device to yank. See YankInstance for more information.

Members

id: string
the chardev's ID

Since

6.0

YankInstance (Object)

A yank instance can be yanked with the yank qmp command to recover from a hanging QEMU.

Members

type: YankInstanceType
yank instance type

The members of YankInstanceBlockNode when type is "block-node"
The members of YankInstanceChardev when type is "chardev"

Currently implemented yank instances:

- nbd block device: Yanking it will shut down the connection to the nbd server without attempting to reconnect.
- socket chardev: Yanking it will shut down the connected socket.
- migration: Yanking it will shut down all migration connections. Unlike migrate_cancel, it will not notify the migration process, so migration will go into failed state, instead of cancelled state. yank should be used to recover from hangs.

Since

6.0

yank (Command)

Try to recover from hanging QEMU by yanking the specified instances. See YankInstance for more information.

Arguments

instances: array of YankInstance
the instances to be yanked

Errors

- If any of the YankInstances doesn't exist, DeviceNotFound

Example

```
-> { "execute": "yank",
     "arguments": {
         "instances": [
              { "type": "block-node",
                "node-name": "nbd0" }
         ] } }
<- { "return": {} }
```

Since

6.0

query-yank (Command)

Query yank instances. See YankInstance for more information.

Returns

list of YankInstance

Example

```
-> { "execute": "query-yank" }
<- { "return": [
          { "type": "block-node",
            "node-name": "nbd0" }
     ] }
```

Since

6.0

5.11.30 Miscellanea

add_client (Command)

Allow client connections for VNC, Spice and socket based character devices to be passed in to QEMU via SCM_RIGHTS.

If the FD associated with fdname is not a socket, the command will fail and the FD will be closed.

Arguments

protocol: string
: protocol name. Valid names are "vnc", "spice", "dbus-display" or the name of a character device (e.g. from -chardev id=XXXX)

fdname: string
: file descriptor name previously passed via 'getfd' command

skipauth: boolean (optional)
: whether to skip authentication. Only applies to "vnc" and "spice" protocols

tls: boolean (optional)
: whether to perform TLS. Only applies to the "spice" protocol

Since

0.14

Example

```
-> { "execute": "add_client", "arguments": { "protocol": "vnc",
                                              "fdname": "myclient" } }
<- { "return": {} }
```

NameInfo (Object)

Guest name information.

Members

name: string (optional)
The name of the guest

Since

0.14

query-name (Command)

Return the name information of a guest.

Returns

NameInfo of the guest

Since

0.14

Example

```
-> { "execute": "query-name" }
<- { "return": { "name": "qemu-name" } }
```

IOThreadInfo (Object)

Information about an iothread

Members

id: string
the identifier of the iothread

thread-id: int
ID of the underlying host thread

poll-max-ns: int
maximum polling time in ns, 0 means polling is disabled (since 2.9)

poll-grow: int
how many ns will be added to polling time, 0 means that it's not configured (since 2.9)

poll-shrink: int
how many ns will be removed from polling time, 0 means that it's not configured (since 2.9)

aio-max-batch: int
maximum number of requests in a batch for the AIO engine, 0 means that the engine will use its default (since 6.1)

Since

2.0

query-iothreads (Command)

Returns a list of information about each iothread.

Note

this list excludes the QEMU main loop thread, which is not declared using the -object iothread command-line option. It is always the main thread of the process.

Returns

a list of IOThreadInfo for each iothread

Since

2.0

Example

```
-> { "execute": "query-iothreads" }
<- { "return": [
         {
            "id":"iothread0",
            "thread-id":3134
         },
         {
            "id":"iothread1",
            "thread-id":3135
         }
      ]
   }
```

stop (Command)

Stop guest VM execution.

Since

0.14

Notes

This function will succeed even if the guest is already in the stopped state. In "inmigrate" state, it will ensure that the guest remains paused once migration finishes, as if the -S option was passed on the command line.

In the "suspended" state, it will completely stop the VM and cause a transition to the "paused" state. (Since 9.0)

Example

```
-> { "execute": "stop" }
<- { "return": {} }
```

cont (Command)

Resume guest VM execution.

Since

0.14

Notes

This command will succeed if the guest is currently running. It will also succeed if the guest is in the "inmigrate" state; in this case, the effect of the command is to make sure the guest starts once migration finishes, removing the effect of the -S command line option if it was passed.

If the VM was previously suspended, and not been reset or woken, this command will transition back to the "suspended" state. (Since 9.0)

Example

```
-> { "execute": "cont" }
<- { "return": {} }
```

x-exit-preconfig (Command)

Exit from "preconfig" state

This command makes QEMU exit the preconfig state and proceed with VM initialization using configuration data provided on the command line and via the QMP monitor during the preconfig state. The command is only available during the preconfig state (i.e. when the –preconfig command line option was in use).

Features

unstable
: This command is experimental.

Since

3.0

Example

```
-> { "execute": "x-exit-preconfig" }
<- { "return": {} }
```

human-monitor-command (Command)

Execute a command on the human monitor and return the output.

Arguments

command-line: string
: the command to execute in the human monitor

cpu-index: int (optional)
: The CPU to use for commands that require an implicit CPU

Features

savevm-monitor-nodes
: If present, HMP command savevm only snapshots monitor-owned nodes if they have no parents. This allows the use of 'savevm' with -blockdev. (since 4.2)

Returns

the output of the command as a string

Since

0.14

Notes

This command only exists as a stop-gap. Its use is highly discouraged. The semantics of this command are not guaranteed: this means that command names, arguments and responses can change or be removed at ANY time. Applications that rely on long term stability guarantees should NOT use this command.

Known limitations:

- This command is stateless, this means that commands that depend on state information (such as getfd) might not work
- Commands that prompt the user for data don't currently work

Example

```
-> { "execute": "human-monitor-command",
     "arguments": { "command-line": "info kvm" } }
<- { "return": "kvm support: enabled\r\n" }
```

getfd (Command)

Receive a file descriptor via SCM rights and assign it a name

Arguments

fdname: string
file descriptor name

Since

0.14

Notes

If fdname already exists, the file descriptor assigned to it will be closed and replaced by the received file descriptor.

The 'closefd' command can be used to explicitly close the file descriptor when it is no longer needed.

Example

```
-> { "execute": "getfd", "arguments": { "fdname": "fd1" } }
<- { "return": {} }
```

If

CONFIG_POSIX

get-win32-socket (Command)

Add a socket that was duplicated to QEMU process with WSADuplicateSocketW() via WSASocket() & WSAPROTOCOL_INFOW structure and assign it a name (the SOCKET is associated with a CRT file descriptor)

Arguments

info: string
the WSAPROTOCOL_INFOW structure (encoded in base64)

fdname: string
file descriptor name

Since

8.0

Notes

If `fdname` already exists, the file descriptor assigned to it will be closed and replaced by the received file descriptor.

The 'closefd' command can be used to explicitly close the file descriptor when it is no longer needed.

Example

```
-> { "execute": "get-win32-socket", "arguments": { "info": "abcd123..", fdname":
↪"skclient" } }
<- { "return": {} }
```

If

`CONFIG_WIN32`

`closefd` (Command)

Close a file descriptor previously passed via SCM rights

Arguments

fdname: string
file descriptor name

Since

0.14

Example

```
-> { "execute": "closefd", "arguments": { "fdname": "fd1" } }
<- { "return": {} }
```

`AddfdInfo` (Object)

Information about a file descriptor that was added to an fd set.

Members

`fdset-id`: `int`
The ID of the fd set that `fd` was added to.

`fd`: `int`
The file descriptor that was received via SCM rights and added to the fd set.

Since

1.2

`add-fd` (Command)

Add a file descriptor, that was passed via SCM rights, to an fd set.

Arguments

`fdset-id`: `int` (optional)
The ID of the fd set to add the file descriptor to.

`opaque`: `string` (optional)
A free-form string that can be used to describe the fd.

Returns

`AddfdInfo`

Errors

- If file descriptor was not received, GenericError
- If `fdset-id` is a negative value, GenericError

Notes

The list of fd sets is shared by all monitor connections.

If `fdset-id` is not specified, a new fd set will be created.

Since

1.2

Example

```
-> { "execute": "add-fd", "arguments": { "fdset-id": 1 } }
<- { "return": { "fdset-id": 1, "fd": 3 } }
```

`remove-fd` (Command)

Remove a file descriptor from an fd set.

Arguments

`fdset-id`: `int`
: The ID of the fd set that the file descriptor belongs to.

`fd`: `int` (optional)
: The file descriptor that is to be removed.

Errors

- If `fdset-id` or `fd` is not found, GenericError

Since

1.2

Notes

The list of fd sets is shared by all monitor connections.

If `fd` is not specified, all file descriptors in `fdset-id` will be removed.

Example

```
-> { "execute": "remove-fd", "arguments": { "fdset-id": 1, "fd": 3 } }
<- { "return": {} }
```

FdsetFdInfo (Object)

Information about a file descriptor that belongs to an fd set.

Members

fd: int
The file descriptor value.

opaque: string (optional)
A free-form string that can be used to describe the fd.

Since

1.2

FdsetInfo (Object)

Information about an fd set.

Members

fdset-id: int
The ID of the fd set.

fds: array of FdsetFdInfo
A list of file descriptors that belong to this fd set.

Since

1.2

query-fdsets (Command)

Return information describing all fd sets.

Returns

A list of `FdsetInfo`

Since

1.2

Note

The list of fd sets is shared by all monitor connections.

Example

```
-> { "execute": "query-fdsets" }
<- { "return": [
       {
         "fds": [
           {
             "fd": 30,
             "opaque": "rdonly:/path/to/file"
           },
           {
             "fd": 24,
             "opaque": "rdwr:/path/to/file"
           }
         ],
         "fdset-id": 1
       },
       {
         "fds": [
           {
             "fd": 28
           },
           {
             "fd": 29
           }
         ],
         "fdset-id": 0
       }
     ]
   }
```

CommandLineParameterType (Enum)

Possible types for an option parameter.

Values

string
accepts a character string

boolean
accepts "on" or "off"

number
accepts a number

size
accepts a number followed by an optional suffix (K)ilo, (M)ega, (G)iga, (T)era

Since

1.5

CommandLineParameterInfo (Object)

Details about a single parameter of a command line option.

Members

name: string
parameter name

type: CommandLineParameterType
parameter CommandLineParameterType

help: string (optional)
human readable text string, not suitable for parsing.

default: string (optional)
default value string (since 2.1)

Since

1.5

CommandLineOptionInfo (Object)

Details about a command line option, including its list of parameter details

Members

option: string
option name

parameters: array of CommandLineParameterInfo
an array of CommandLineParameterInfo

Since

1.5

query-command-line-options (Command)

Query command line option schema.

Arguments

option: string (optional)
option name

Returns

list of CommandLineOptionInfo for all options (or for the given option).

Errors

- if the given option doesn't exist

Since

1.5

Example

```
-> { "execute": "query-command-line-options",
     "arguments": { "option": "option-rom" } }
<- { "return": [
        {
            "parameters": [
                {
                    "name": "romfile",
```

(continues on next page)

(continued from previous page)

```
                    "type": "string"
                },
                {
                    "name": "bootindex",
                    "type": "number"
                }
            ],
            "option": "option-rom"
        }
    ]
}
```

RTC_CHANGE (Event)

Emitted when the guest changes the RTC time.

Arguments

offset: int
: offset in seconds between base RTC clock (as specified by -rtc base), and new RTC clock value

qom-path: string
: path to the RTC object in the QOM tree

Note

This event is rate-limited. It is not guaranteed that the RTC in the system implements this event, or even that the system has an RTC at all.

Since

0.13

Example

```
<- { "event": "RTC_CHANGE",
     "data": { "offset": 78 },
     "timestamp": { "seconds": 1267020223, "microseconds": 435656 } }
```

VFU_CLIENT_HANGUP (Event)

Emitted when the client of a TYPE_VFIO_USER_SERVER closes the communication channel

Arguments

vfu-id: string
ID of the TYPE_VFIO_USER_SERVER object. It is the last component of `vfu-qom-path` referenced below

vfu-qom-path: string
path to the TYPE_VFIO_USER_SERVER object in the QOM tree

dev-id: string
ID of attached PCI device

dev-qom-path: string
path to attached PCI device in the QOM tree

Since

7.1

Example

```
<- { "event": "VFU_CLIENT_HANGUP",
     "data": { "vfu-id": "vfu1",
               "vfu-qom-path": "/objects/vfu1",
               "dev-id": "sas1",
               "dev-qom-path": "/machine/peripheral/sas1" },
     "timestamp": { "seconds": 1265044230, "microseconds": 450486 } }
```

rtc-reset-reinjection (Command)

This command will reset the RTC interrupt reinjection backlog. Can be used if another mechanism to synchronize guest time is in effect, for example QEMU guest agent's guest-set-time command.

Since

2.1

Example

```
-> { "execute": "rtc-reset-reinjection" }
<- { "return": {} }
```

If

TARGET_I386

SevState (Enum)

An enumeration of SEV state information used during `query-sev`.

Values

uninit
The guest is uninitialized.

launch-update
The guest is currently being launched; plaintext data and register state is being imported.

launch-secret
The guest is currently being launched; ciphertext data is being imported.

running
The guest is fully launched or migrated in.

send-update
The guest is currently being migrated out to another machine.

receive-update
The guest is currently being migrated from another machine.

Since

2.12

If

TARGET_I386

SevInfo (Object)

Information about Secure Encrypted Virtualization (SEV) support

Members

enabled: boolean
: true if SEV is active

api-major: int
: SEV API major version

api-minor: int
: SEV API minor version

build-id: int
: SEV FW build id

policy: int
: SEV policy value

state: SevState
: SEV guest state

handle: int
: SEV firmware handle

Since

2.12

If

TARGET_I386

query-sev (Command)

Returns information about SEV

Returns

SevInfo

Since

2.12

Example

```
-> { "execute": "query-sev" }
<- { "return": { "enabled": true, "api-major" : 0, "api-minor" : 0,
                 "build-id" : 0, "policy" : 0, "state" : "running",
                 "handle" : 1 } }
```

If

TARGET_I386

SevLaunchMeasureInfo (Object)

SEV Guest Launch measurement information

Members

data: string
: the measurement value encoded in base64

Since

2.12

If

TARGET_I386

query-sev-launch-measure (Command)

Query the SEV guest launch information.

Returns

The SevLaunchMeasureInfo for the guest

Since

2.12

Example

```
-> { "execute": "query-sev-launch-measure" }
<- { "return": { "data": "4l8LXeNlSPUDlXPJG5966/8%YZ" } }
```

If

TARGET_I386

SevCapability (Object)

The struct describes capability for a Secure Encrypted Virtualization feature.

Members

pdh: string
Platform Diffie-Hellman key (base64 encoded)

cert-chain: string
PDH certificate chain (base64 encoded)

cpu0-id: string
Unique ID of CPU0 (base64 encoded) (since 7.1)

cbitpos: int
C-bit location in page table entry

reduced-phys-bits: int
Number of physical Address bit reduction when SEV is enabled

Since

2.12

If

TARGET_I386

query-sev-capabilities (Command)

This command is used to get the SEV capabilities, and is supported on AMD X86 platforms only.

Returns

SevCapability objects.

Since

2.12

Example

```
-> { "execute": "query-sev-capabilities" }
<- { "return": { "pdh": "8CCDD8DDD", "cert-chain": "888CCCDDDEE",
                 "cpu0-id": "2lvmGwo+...61iEinw==",
                 "cbitpos": 47, "reduced-phys-bits": 1}}
```

If

TARGET_I386

sev-inject-launch-secret (Command)

This command injects a secret blob into memory of SEV guest.

Arguments

packet-header: string
: the launch secret packet header encoded in base64

secret: string
: the launch secret data to be injected encoded in base64

gpa: int (optional)
: the guest physical address where secret will be injected.

Since

6.0

If

TARGET_I386

SevAttestationReport (Object)

The struct describes attestation report for a Secure Encrypted Virtualization feature.

Members

data: string
guest attestation report (base64 encoded)

Since

6.1

If

TARGET_I386

query-sev-attestation-report (Command)

This command is used to get the SEV attestation report, and is supported on AMD X86 platforms only.

Arguments

mnonce: string
a random 16 bytes value encoded in base64 (it will be included in report)

Returns

SevAttestationReport objects.

Since

6.1

Example

```
-> { "execute" : "query-sev-attestation-report",
                 "arguments": { "mnonce": "aaaaaaa" } }
<- { "return" : { "data": "aaaaaaaabbbddddd"} }
```

If

TARGET_I386

dump-skeys (Command)

Dump guest's storage keys

Arguments

filename: string
the path to the file to dump to

Since

2.5

Example

```
-> { "execute": "dump-skeys",
     "arguments": { "filename": "/tmp/skeys" } }
<- { "return": {} }
```

If

TARGET_S390X

GICCapability (Object)

The struct describes capability for a specific GIC (Generic Interrupt Controller) version. These bits are not only decided by QEMU/KVM software version, but also decided by the hardware that the program is running upon.

Members

version: int
version of GIC to be described. Currently, only 2 and 3 are supported.

emulated: boolean
whether current QEMU/hardware supports emulated GIC device in user space.

kernel: boolean
whether current QEMU/hardware supports hardware accelerated GIC device in kernel.

Since

2.6

If

TARGET_ARM

query-gic-capabilities (Command)

This command is ARM-only. It will return a list of GICCapability objects that describe its capability bits.

Returns

a list of GICCapability objects.

Since

2.6

Example

```
-> { "execute": "query-gic-capabilities" }
<- { "return": [{ "version": 2, "emulated": true, "kernel": false },
                { "version": 3, "emulated": false, "kernel": true } ] }
```

If

TARGET_ARM

SGXEPCSection (Object)

Information about intel SGX EPC section info

Members

node: int
the numa node

size: int
the size of EPC section

Since

7.0

SGXInfo (Object)

Information about intel Safe Guard eXtension (SGX) support

Members

sgx: boolean
true if SGX is supported

sgx1: boolean
true if SGX1 is supported

sgx2: boolean
true if SGX2 is supported

flc: boolean
true if FLC is supported

sections: array of SGXEPCSection
The EPC sections info for guest (Since: 7.0)

Since

6.2

If

TARGET_I386

query-sgx (Command)

Returns information about SGX

Returns

SGXInfo

Since

6.2

Example

```
-> { "execute": "query-sgx" }
<- { "return": { "sgx": true, "sgx1" : true, "sgx2" : true,
                 "flc": true,
                 "sections": [{"node": 0, "size": 67108864},
                 {"node": 1, "size": 29360128}]} }
```

If

TARGET_I386

query-sgx-capabilities (Command)

Returns information from host SGX capabilities

Returns

SGXInfo

Since

6.2

Example

```
-> { "execute": "query-sgx-capabilities" }
<- { "return": { "sgx": true, "sgx1" : true, "sgx2" : true,
                 "flc": true,
                 "section" : [{"node": 0, "size": 67108864},
                 {"node": 1, "size": 29360128}]} }
```

If

TARGET_I386

EvtchnPortType (Enum)

An enumeration of Xen event channel port types.

Values

closed
The port is unused.

unbound
The port is allocated and ready to be bound.

interdomain
The port is connected as an interdomain interrupt.

pirq
The port is bound to a physical IRQ (PIRQ).

virq
The port is bound to a virtual IRQ (VIRQ).

ipi
The post is an inter-processor interrupt (IPI).

Since

8.0

If

TARGET_I386

EvtchnInfo (Object)

Information about a Xen event channel port

Members

port: int
the port number

vcpu: int
target vCPU for this port

type: EvtchnPortType
the port type

remote-domain: string
remote domain for interdomain ports

target: int
remote port ID, or virq/pirq number

pending: boolean
port is currently active pending delivery

masked: boolean
port is masked

Since

8.0

If

TARGET_I386

xen-event-list (Command)

Query the Xen event channels opened by the guest.

Returns

list of open event channel ports.

Since

8.0

Example

```
-> { "execute": "xen-event-list" }
<- { "return": [
        {
            "pending": false,
            "port": 1,
            "vcpu": 1,
            "remote-domain": "qemu",
            "masked": false,
            "type": "interdomain",
            "target": 1
        },
        {
            "pending": false,
            "port": 2,
            "vcpu": 0,
```

(continues on next page)

(continued from previous page)

```
            "remote-domain": "",
            "masked": false,
            "type": "virq",
            "target": 0
        }
    ]
}
```

If

TARGET_I386

xen-event-inject (Command)

Inject a Xen event channel port (interrupt) to the guest.

Arguments

port: int
The port number

Since

8.0

Example

```
-> { "execute": "xen-event-inject", "arguments": { "port": 1 } }
<- { "return": { } }
```

If

TARGET_I386

5.11.31 Audio

AudiodevPerDirectionOptions (Object)

General audio backend options that are used for both playback and recording.

Members

mixing-engine: boolean (optional)
use QEMU's mixing engine to mix all streams inside QEMU and convert audio formats when not supported by the backend. When set to off, fixed-settings must be also off (default on, since 4.2)

fixed-settings: boolean (optional)
use fixed settings for host input/output. When off, frequency, channels and format must not be specified (default true)

frequency: int (optional)
frequency to use when using fixed settings (default 44100)

channels: int (optional)
number of channels when using fixed settings (default 2)

voices: int (optional)
number of voices to use (default 1)

format: AudioFormat (optional)
sample format to use when using fixed settings (default s16)

buffer-length: int (optional)
the buffer length in microseconds

Since

4.0

AudiodevGenericOptions (Object)

Generic driver-specific options.

Members

in: AudiodevPerDirectionOptions (optional)
options of the capture stream

out: AudiodevPerDirectionOptions (optional)
options of the playback stream

Since

4.0

AudiodevAlsaPerDirectionOptions (Object)

Options of the ALSA backend that are used for both playback and recording.

Members

dev: string (optional)
the name of the ALSA device to use (default 'default')

period-length: int (optional)
the period length in microseconds

try-poll: boolean (optional)
attempt to use poll mode, falling back to non-polling access on failure (default true)

The members of AudiodevPerDirectionOptions

Since

4.0

AudiodevAlsaOptions (Object)

Options of the ALSA audio backend.

Members

in: AudiodevAlsaPerDirectionOptions (optional)
options of the capture stream

out: AudiodevAlsaPerDirectionOptions (optional)
options of the playback stream

threshold: int (optional)
set the threshold (in microseconds) when playback starts

Since

4.0

AudiodevSndioOptions (Object)

Options of the sndio audio backend.

Members

in: AudiodevPerDirectionOptions (optional)
options of the capture stream

out: AudiodevPerDirectionOptions (optional)
options of the playback stream

dev: string (optional)
the name of the sndio device to use (default 'default')

latency: int (optional)
play buffer size (in microseconds)

Since

7.2

AudiodevCoreaudioPerDirectionOptions (Object)

Options of the Core Audio backend that are used for both playback and recording.

Members

buffer-count: int (optional)
number of buffers

The members of AudiodevPerDirectionOptions

Since

4.0

AudiodevCoreaudioOptions (Object)

Options of the coreaudio audio backend.

Members

in: AudiodevCoreaudioPerDirectionOptions (optional)
options of the capture stream

out: AudiodevCoreaudioPerDirectionOptions (optional)
options of the playback stream

Since

4.0

AudiodevDsoundOptions (Object)

Options of the DirectSound audio backend.

Members

in: AudiodevPerDirectionOptions (optional)
options of the capture stream

out: AudiodevPerDirectionOptions (optional)
options of the playback stream

latency: int (optional)
add extra latency to playback in microseconds (default 10000)

Since

4.0

AudiodevJackPerDirectionOptions (Object)

Options of the JACK backend that are used for both playback and recording.

Members

server-name: string (optional)
select from among several possible concurrent server instances (default: environment variable $JACK_DEFAULT_SERVER if set, else "default")

client-name: string (optional)
the client name to use. The server will modify this name to create a unique variant, if needed unless `exact-name` is true (default: the guest's name)

connect-ports: string (optional)
if set, a regular expression of JACK client port name(s) to monitor for and automatically connect to

start-server: boolean (optional)
start a jack server process if one is not already present (default: false)

exact-name: boolean (optional)
use the exact name requested otherwise JACK automatically generates a unique one, if needed (default: false)

The members of AudiodevPerDirectionOptions

Since

5.1

AudiodevJackOptions (Object)

Options of the JACK audio backend.

Members

in: AudiodevJackPerDirectionOptions (optional)
options of the capture stream

out: AudiodevJackPerDirectionOptions (optional)
options of the playback stream

Since

5.1

AudiodevOssPerDirectionOptions (Object)

Options of the OSS backend that are used for both playback and recording.

Members

dev: string (optional)
file name of the OSS device (default '/dev/dsp')

buffer-count: int (optional)
number of buffers

try-poll: boolean (optional)
attempt to use poll mode, falling back to non-polling access on failure (default true)

The members of AudiodevPerDirectionOptions

Since

4.0

AudiodevOssOptions (Object)

Options of the OSS audio backend.

Members

in: AudiodevOssPerDirectionOptions (optional)
options of the capture stream

out: AudiodevOssPerDirectionOptions (optional)
options of the playback stream

try-mmap: boolean (optional)
try using memory-mapped access, falling back to non-memory-mapped access on failure (default true)

exclusive: boolean (optional)
open device in exclusive mode (vmix won't work) (default false)

dsp-policy: int (optional)
set the timing policy of the device (between 0 and 10, where smaller number means smaller latency but higher CPU usage) or -1 to use fragment mode (option ignored on some platforms) (default 5)

Since

4.0

AudiodevPaPerDirectionOptions (Object)

Options of the Pulseaudio backend that are used for both playback and recording.

Members

name: string (optional)
name of the sink/source to use

stream-name: string (optional)
name of the PulseAudio stream created by qemu. Can be used to identify the stream in PulseAudio when you create multiple PulseAudio devices or run multiple qemu instances (default: audiodev's id, since 4.2)

latency: int (optional)
latency you want PulseAudio to achieve in microseconds (default 15000)

The members of AudiodevPerDirectionOptions

Since

4.0

AudiodevPaOptions (Object)

Options of the PulseAudio audio backend.

Members

in: AudiodevPaPerDirectionOptions (optional)
options of the capture stream

out: AudiodevPaPerDirectionOptions (optional)
options of the playback stream

server: string (optional)
PulseAudio server address (default: let PulseAudio choose)

Since

4.0

AudiodevPipewirePerDirectionOptions (Object)

Options of the PipeWire backend that are used for both playback and recording.

Members

name: string (optional)
name of the sink/source to use

stream-name: string (optional)
name of the PipeWire stream created by qemu. Can be used to identify the stream in PipeWire when you create multiple PipeWire devices or run multiple qemu instances (default: audiodev's id)

latency: int (optional)
latency you want PipeWire to achieve in microseconds (default 46000)

The members of AudiodevPerDirectionOptions

Since

8.1

AudiodevPipewireOptions (Object)

Options of the PipeWire audio backend.

Members

in: AudiodevPipewirePerDirectionOptions (optional)
options of the capture stream

out: AudiodevPipewirePerDirectionOptions (optional)
options of the playback stream

Since

8.1

AudiodevSdlPerDirectionOptions (Object)

Options of the SDL audio backend that are used for both playback and recording.

Members

buffer-count: int (optional)
number of buffers (default 4)

The members of AudiodevPerDirectionOptions

Since

6.0

AudiodevSdlOptions (Object)

Options of the SDL audio backend.

Members

in: AudiodevSdlPerDirectionOptions (optional)
options of the recording stream

out: AudiodevSdlPerDirectionOptions (optional)
options of the playback stream

Since

6.0

AudiodevWavOptions (Object)

Options of the wav audio backend.

Members

in: AudiodevPerDirectionOptions (optional)
: options of the capture stream

out: AudiodevPerDirectionOptions (optional)
: options of the playback stream

path: string (optional)
: name of the wav file to record (default 'qemu.wav')

Since

4.0

AudioFormat (Enum)

An enumeration of possible audio formats.

Values

u8
: unsigned 8 bit integer

s8
: signed 8 bit integer

u16
: unsigned 16 bit integer

s16
: signed 16 bit integer

u32
: unsigned 32 bit integer

s32
: signed 32 bit integer

f32
: single precision floating-point (since 5.0)

Since

4.0

AudiodevDriver (Enum)

An enumeration of possible audio backend drivers.

Values

jack (**If: CONFIG_AUDIO_JACK**)
: JACK audio backend (since 5.1)

none
: Not documented

alsa (**If: CONFIG_AUDIO_ALSA**)
: Not documented

coreaudio (**If: CONFIG_AUDIO_COREAUDIO**)
: Not documented

dbus (**If: CONFIG_DBUS_DISPLAY**)
: Not documented

dsound (**If: CONFIG_AUDIO_DSOUND**)
: Not documented

oss (**If: CONFIG_AUDIO_OSS**)
: Not documented

pa (**If: CONFIG_AUDIO_PA**)
: Not documented

pipewire (**If: CONFIG_AUDIO_PIPEWIRE**)
: Not documented

sdl (**If: CONFIG_AUDIO_SDL**)
: Not documented

sndio (**If: CONFIG_AUDIO_SNDIO**)
: Not documented

spice (**If: CONFIG_SPICE**)
: Not documented

wav
: Not documented

Since

4.0

Audiodev (Object)

Options of an audio backend.

Members

id: string
identifier of the backend

driver: AudiodevDriver
the backend driver to use

timer-period: int (optional)
timer period (in microseconds, 0: use lowest possible)

The members of AudiodevGenericOptions when driver is "none"
The members of AudiodevAlsaOptions when driver is "alsa" (If: CONFIG_AUDIO_ALSA)
The members of AudiodevCoreaudioOptions when driver is "coreaudio" (If: CONFIG_AUDIO_COREAUDIO)
The members of AudiodevGenericOptions when driver is "dbus" (If: CONFIG_DBUS_DISPLAY)
The members of AudiodevDsoundOptions when driver is "dsound" (If: CONFIG_AUDIO_DSOUND)
The members of AudiodevJackOptions when driver is "jack" (If: CONFIG_AUDIO_JACK)
The members of AudiodevOssOptions when driver is "oss" (If: CONFIG_AUDIO_OSS)
The members of AudiodevPaOptions when driver is "pa" (If: CONFIG_AUDIO_PA)
The members of AudiodevPipewireOptions when driver is "pipewire" (If: CONFIG_AUDIO_PIPEWIRE)
The members of AudiodevSdlOptions when driver is "sdl" (If: CONFIG_AUDIO_SDL)
The members of AudiodevSndioOptions when driver is "sndio" (If: CONFIG_AUDIO_SNDIO)
The members of AudiodevGenericOptions when driver is "spice" (If: CONFIG_SPICE)
The members of AudiodevWavOptions when driver is "wav"

Since

4.0

query-audiodevs (Command)

Returns information about audiodev configuration

Returns

array of Audiodev

Since

8.0

5.11.32 ACPI

`AcpiTableOptions` (Object)

Specify an ACPI table on the command line to load.

At most one of `file` and `data` can be specified. The list of files specified by any one of them is loaded and concatenated in order. If both are omitted, `data` is implied.

Other fields / optargs can be used to override fields of the generic ACPI table header; refer to the ACPI specification 5.0, section 5.2.6 System Description Table Header. If a header field is not overridden, then the corresponding value from the concatenated blob is used (in case of `file`), or it is filled in with a hard-coded value (in case of `data`).

String fields are copied into the matching ACPI member from lowest address upwards, and silently truncated / NUL-padded to length.

Members

`sig`: `string` (optional)
table signature / identifier (4 bytes)

`rev`: `int` (optional)
table revision number (dependent on signature, 1 byte)

`oem_id`: `string` (optional)
OEM identifier (6 bytes)

`oem_table_id`: `string` (optional)
OEM table identifier (8 bytes)

`oem_rev`: `int` (optional)
OEM-supplied revision number (4 bytes)

`asl_compiler_id`: `string` (optional)
identifier of the utility that created the table (4 bytes)

`asl_compiler_rev`: `int` (optional)
revision number of the utility that created the table (4 bytes)

`file`: `string` (optional)
colon (:) separated list of pathnames to load and concatenate as table data. The resultant binary blob is expected to have an ACPI table header. At least one file is required. This field excludes `data`.

`data`: `string` (optional)
colon (:) separated list of pathnames to load and concatenate as table data. The resultant binary blob must not have an ACPI table header. At least one file is required. This field excludes `file`.

Since

1.5

ACPISlotType (Enum)

Values

DIMM
memory slot

CPU
logical CPU slot (since 2.7)

ACPIOSTInfo (Object)

OSPM Status Indication for a device For description of possible values of `source` and `status` fields see "_OST (OSPM Status Indication)" chapter of ACPI5.0 spec.

Members

device: string (optional)
device ID associated with slot

slot: string
slot ID, unique per slot of a given `slot-type`

slot-type: ACPISlotType
type of the slot

source: int
an integer containing the source event

status: int
an integer containing the status code

Since

2.1

query-acpi-ospm-status (Command)

Return a list of ACPIOSTInfo for devices that support status reporting via ACPI _OST method.

Since

2.1

Example

```
-> { "execute": "query-acpi-ospm-status" }
<- { "return": [ { "device": "d1", "slot": "0", "slot-type": "DIMM", "source": 1, "status
↪": 0},
                 { "slot": "1", "slot-type": "DIMM", "source": 0, "status": 0},
                 { "slot": "2", "slot-type": "DIMM", "source": 0, "status": 0},
                 { "slot": "3", "slot-type": "DIMM", "source": 0, "status": 0}
   ]}
```

ACPI_DEVICE_OST (Event)

Emitted when guest executes ACPI _OST method.

Arguments

info: ACPIOSTInfo
OSPM Status Indication

Since

2.1

Example

```
<- { "event": "ACPI_DEVICE_OST",
     "data": { "info": { "device": "d1", "slot": "0",
                         "slot-type": "DIMM", "source": 1, "status": 0 } },
     "timestamp": { "seconds": 1265044230, "microseconds": 450486 } }
```

5.11.33 PCI

PciMemoryRange (Object)

A PCI device memory region

Members

base: int
the starting address (guest physical)

limit: int
the ending address (guest physical)

Since

0.14

PciMemoryRegion (Object)

Information about a PCI device I/O region.

Members

bar: int
the index of the Base Address Register for this region

type: string

- 'io' if the region is a PIO region
- 'memory' if the region is a MMIO region

size: int
memory size

prefetch: boolean (optional)
if `type` is 'memory', true if the memory is prefetchable

mem_type_64: boolean (optional)
if `type` is 'memory', true if the BAR is 64-bit

address: int
Not documented

Since

0.14

PciBusInfo (Object)

Information about a bus of a PCI Bridge device

Members

number: int
primary bus interface number. This should be the number of the bus the device resides on.

secondary: int
secondary bus interface number. This is the number of the main bus for the bridge

subordinate: int
This is the highest number bus that resides below the bridge.

io_range: PciMemoryRange
The PIO range for all devices on this bridge

memory_range: PciMemoryRange
The MMIO range for all devices on this bridge

prefetchable_range: PciMemoryRange
The range of prefetchable MMIO for all devices on this bridge

Since

2.4

PciBridgeInfo (Object)

Information about a PCI Bridge device

Members

bus: PciBusInfo
information about the bus the device resides on

devices: array of PciDeviceInfo (optional)
a list of PciDeviceInfo for each device on this bridge

Since

0.14

PciDeviceClass (Object)

Information about the Class of a PCI device

Members

desc: string (optional)
a string description of the device's class

class: int
the class code of the device

Since

2.4

PciDeviceId (Object)

Information about the Id of a PCI device

Members

device: int
the PCI device id

vendor: int
the PCI vendor id

subsystem: int (optional)
the PCI subsystem id (since 3.1)

subsystem-vendor: int (optional)
the PCI subsystem vendor id (since 3.1)

Since

2.4

PciDeviceInfo (Object)

Information about a PCI device

Members

bus: int
the bus number of the device

slot: int
the slot the device is located in

function: int
the function of the slot used by the device

class_info: PciDeviceClass
the class of the device

id: PciDeviceId
the PCI device id

irq: int (optional)
if an IRQ is assigned to the device, the IRQ number

irq_pin: int
the IRQ pin, zero means no IRQ (since 5.1)

qdev_id: string
the device name of the PCI device

pci_bridge: PciBridgeInfo (optional)
if the device is a PCI bridge, the bridge information

regions: array of PciMemoryRegion
a list of the PCI I/O regions associated with the device

Notes

the contents of `class_info`.desc are not stable and should only be treated as informational.

Since

0.14

PciInfo (Object)

Information about a PCI bus

Members

bus: int
the bus index

devices: array of PciDeviceInfo
a list of devices on this bus

Since

0.14

query-pci (Command)

Return information about the PCI bus topology of the guest.

Returns

a list of `PciInfo` for each PCI bus. Each bus is represented by a json-object, which has a key with a json-array of all PCI devices attached to it. Each device is represented by a json-object.

Since

0.14

Example

```
-> { "execute": "query-pci" }
<- { "return": [
         {
            "bus": 0,
            "devices": [
               {
                  "bus": 0,
                  "qdev_id": "",
                  "slot": 0,
                  "class_info": {
                     "class": 1536,
                     "desc": "Host bridge"
                  },
                  "id": {
                     "device": 32902,
                     "vendor": 4663
                  },
                  "function": 0,
                  "regions": [
                  ]
               },
               {
                  "bus": 0,
                  "qdev_id": "",
                  "slot": 1,
                  "class_info": {
                     "class": 1537,
                     "desc": "ISA bridge"
                  },
                  "id": {
                     "device": 32902,
                     "vendor": 28672
                  },
                  "function": 0,
                  "regions": [
                  ]
               },
               {
                  "bus": 0,
                  "qdev_id": "",
```

(continues on next page)

(continued from previous page)


```
                    "slot": 1,
                    "class_info": {
                       "class": 257,
                       "desc": "IDE controller"
                    },
                    "id": {
                       "device": 32902,
                       "vendor": 28688
                    },
                    "function": 1,
                    "regions": [
                       {
                          "bar": 4,
                          "size": 16,
                          "address": 49152,
                          "type": "io"
                       }
                    ]
                 },
                 {
                    "bus": 0,
                    "qdev_id": "",
                    "slot": 2,
                    "class_info": {
                       "class": 768,
                       "desc": "VGA controller"
                    },
                    "id": {
                       "device": 4115,
                       "vendor": 184
                    },
                    "function": 0,
                    "regions": [
                       {
                          "prefetch": true,
                          "mem_type_64": false,
                          "bar": 0,
                          "size": 33554432,
                          "address": 4026531840,
                          "type": "memory"
                       },
                       {
                          "prefetch": false,
                          "mem_type_64": false,
                          "bar": 1,
                          "size": 4096,
                          "address": 4060086272,
                          "type": "memory"
                       },
                       {
                          "prefetch": false,
                          "mem_type_64": false,
```


(continues on next page)

(continued from previous page)

```
                        "bar": 6,
                        "size": 65536,
                        "address": -1,
                        "type": "memory"
                     }
                  ]
               },
               {
                  "bus": 0,
                  "qdev_id": "",
                  "irq": 11,
                  "slot": 4,
                  "class_info": {
                     "class": 1280,
                     "desc": "RAM controller"
                  },
                  "id": {
                     "device": 6900,
                     "vendor": 4098
                  },
                  "function": 0,
                  "regions": [
                     {
                        "bar": 0,
                        "size": 32,
                        "address": 49280,
                        "type": "io"
                     }
                  ]
               }
            ]
         }
      ]
   }
```

Note

This example has been shortened as the real response is too long.

5.11.34 Statistics

StatsType (Enum)

Enumeration of statistics types

Values

cumulative
stat is cumulative; value can only increase.

instant
stat is instantaneous; value can increase or decrease.

peak
stat is the peak value; value can only increase.

linear-histogram
stat is a linear histogram.

log2-histogram
stat is a logarithmic histogram, with one bucket for each power of two.

Since

7.1

StatsUnit (Enum)

Enumeration of unit of measurement for statistics

Values

bytes
stat reported in bytes.

seconds
stat reported in seconds.

cycles
stat reported in clock cycles.

boolean
stat is a boolean value.

Since

7.1

StatsProvider (Enum)

Enumeration of statistics providers.

Values

kvm
since 7.1

cryptodev
since 8.0

Since

7.1

StatsTarget (Enum)

The kinds of objects on which one can request statistics.

Values

vm
statistics that apply to the entire virtual machine or the entire QEMU process.

vcpu
statistics that apply to a single virtual CPU.

cryptodev
statistics that apply to a crypto device (since 8.0)

Since

7.1

StatsRequest (Object)

Indicates a set of statistics that should be returned by query-stats.

Members

provider: StatsProvider
provider for which to return statistics.

names: array of string (optional)
statistics to be returned (all if omitted).

Since

7.1

StatsVCPUFilter (Object)

Members

vcpus: array of string (optional)
list of QOM paths for the desired vCPU objects.

Since

7.1

StatsFilter (Object)

The arguments to the query-stats command; specifies a target for which to request statistics and optionally the required subset of information for that target.

Members

target: StatsTarget
the kind of objects to query. Note that each possible target may enable additional filtering options

providers: array of StatsRequest (optional)
which providers to request statistics from, and optionally which named values to return within each provider

The members of StatsVCPUFilter when target is "vcpu"

Since

7.1

StatsValue (Alternate)

Members

scalar: int
single unsigned 64-bit integers.

boolean: boolean
single boolean value.

list: array of int
list of unsigned 64-bit integers (used for histograms).

Since

7.1

Stats (Object)

Members

name: string
name of stat.

value: StatsValue
stat value.

Since

7.1

StatsResult (Object)

Members

provider: StatsProvider
provider for this set of statistics.

qom-path: string (optional)
Path to the object for which the statistics are returned, if the object is exposed in the QOM tree

stats: array of Stats
list of statistics.

Since

7.1

query-stats (Command)

Return runtime-collected statistics for objects such as the VM or its vCPUs.

The arguments are a StatsFilter and specify the provider and objects to return statistics about.

Arguments

The members of `StatsFilter`

Returns

a list of StatsResult, one for each provider and object (e.g., for each vCPU).

Since

7.1

`StatsSchemaValue` (Object)

Schema for a single statistic.

Members

`name`: `string`
: name of the statistic; each element of the schema is uniquely identified by a target, a provider (both available in `StatsSchema`) and the name.

`type`: `StatsType`
: kind of statistic.

`unit`: `StatsUnit` (optional)
: basic unit of measure for the statistic; if missing, the statistic is a simple number or counter.

`base`: `int` (optional)
: base for the multiple of `unit` in which the statistic is measured. Only present if `exponent` is non-zero; `base` and `exponent` together form a SI prefix (e.g., _nano-_ for `base=10` and `exponent=-9`) or IEC binary prefix (e.g. _kibi-_ for `base=2` and `exponent=10`)

`exponent`: `int`
: exponent for the multiple of `unit` in which the statistic is expressed, or 0 for the basic unit

`bucket-size`: `int` (optional)
: Present when `type` is "linear-histogram", contains the width of each bucket of the histogram.

Since

7.1

StatsSchema (Object)

Schema for all available statistics for a provider and target.

Members

provider: StatsProvider
provider for this set of statistics.

target: StatsTarget
the kind of object that can be queried through the provider.

stats: array of StatsSchemaValue
list of statistics.

Since

7.1

query-stats-schemas (Command)

Return the schema for all available runtime-collected statistics.

Arguments

provider: StatsProvider (optional)
a provider to restrict the query to.

Note

runtime-collected statistics and their names fall outside QEMU's usual deprecation policies. QEMU will try to keep the set of available data stable, together with their names, but will not guarantee stability at all costs; the same is true of providers that source statistics externally, e.g. from Linux. For example, if the same value is being tracked with different names on different architectures or by different providers, one of them might be renamed. A statistic might go away if an algorithm is changed or some code is removed; changing a default might cause previously useful statistics to always report 0. Such changes, however, are expected to be rare.

Since

7.1

5.11.35 Virtio devices

`VirtioInfo` (Object)

Basic information about a given VirtIODevice

Members

`path`: `string`
The VirtIODevice's canonical QOM path

`name`: `string`
Name of the VirtIODevice

Since

7.2

`x-query-virtio` (Command)

Returns a list of all realized VirtIODevices

Features

`unstable`
This command is meant for debugging.

Returns

List of gathered VirtIODevices

Since

7.2

Example

```
-> { "execute": "x-query-virtio" }
<- { "return": [
          {
              "name": "virtio-input",
              "path": "/machine/peripheral-anon/device[4]/virtio-backend"
          },
          {
              "name": "virtio-crypto",
              "path": "/machine/peripheral/crypto0/virtio-backend"
```

(continues on next page)

(continued from previous page)

```
        },
        {
            "name": "virtio-scsi",
            "path": "/machine/peripheral-anon/device[2]/virtio-backend"
        },
        {
            "name": "virtio-net",
            "path": "/machine/peripheral-anon/device[1]/virtio-backend"
        },
        {
            "name": "virtio-serial",
            "path": "/machine/peripheral-anon/device[0]/virtio-backend"
        }
    ]
}
```

VhostStatus (Object)

Information about a vhost device. This information will only be displayed if the vhost device is active.

Members

n-mem-sections: int
vhost_dev n_mem_sections

n-tmp-sections: int
vhost_dev n_tmp_sections

nvqs: int
vhost_dev nvqs (number of virtqueues being used)

vq-index: int
vhost_dev vq_index

features: VirtioDeviceFeatures
vhost_dev features

acked-features: VirtioDeviceFeatures
vhost_dev acked_features

backend-features: VirtioDeviceFeatures
vhost_dev backend_features

protocol-features: VhostDeviceProtocols
vhost_dev protocol_features

max-queues: int
vhost_dev max_queues

backend-cap: int
vhost_dev backend_cap

log-enabled: boolean
vhost_dev log_enabled flag

log-size: int
vhost_dev log_size

Since

7.2

VirtioStatus (Object)

Full status of the virtio device with most VirtIODevice members. Also includes the full status of the corresponding vhost device if the vhost device is active.

Members

name: string
VirtIODevice name

device-id: int
VirtIODevice ID

vhost-started: boolean
VirtIODevice vhost_started flag

guest-features: VirtioDeviceFeatures
VirtIODevice guest_features

host-features: VirtioDeviceFeatures
VirtIODevice host_features

backend-features: VirtioDeviceFeatures
VirtIODevice backend_features

device-endian: string
VirtIODevice device_endian

num-vqs: int
VirtIODevice virtqueue count. This is the number of active virtqueues being used by the VirtIODevice.

status: VirtioDeviceStatus
VirtIODevice configuration status (VirtioDeviceStatus)

isr: int
VirtIODevice ISR

queue-sel: int
VirtIODevice queue_sel

vm-running: boolean
VirtIODevice vm_running flag

broken: boolean
VirtIODevice broken flag

disabled: boolean
VirtIODevice disabled flag

use-started: boolean
VirtIODevice use_started flag

started: boolean
VirtIODevice started flag

start-on-kick: boolean
VirtIODevice start_on_kick flag

disable-legacy-check: boolean
VirtIODevice disabled_legacy_check flag

bus-name: string
VirtIODevice bus_name

use-guest-notifier-mask: boolean
VirtIODevice use_guest_notifier_mask flag

vhost-dev: VhostStatus (optional)
Corresponding vhost device info for a given VirtIODevice. Present if the given VirtIODevice has an active vhost device.

Since

7.2

x-query-virtio-status (Command)

Poll for a comprehensive status of a given virtio device

Arguments

path: string
Canonical QOM path of the VirtIODevice

Features

unstable
This command is meant for debugging.

Returns

VirtioStatus of the virtio device

Since

7.2

Examples

```
1. Poll for the status of virtio-crypto (no vhost-crypto active)

-> { "execute": "x-query-virtio-status",
     "arguments": { "path": "/machine/peripheral/crypto0/virtio-backend" }
   }
<- { "return": {
          "device-endian": "little",
          "bus-name": "",
          "disable-legacy-check": false,
          "name": "virtio-crypto",
          "started": true,
          "device-id": 20,
          "backend-features": {
              "transports": [],
              "dev-features": []
          },
          "start-on-kick": false,
          "isr": 1,
          "broken": false,
          "status": {
              "statuses": [
                  "VIRTIO_CONFIG_S_ACKNOWLEDGE: Valid virtio device found",
                  "VIRTIO_CONFIG_S_DRIVER: Guest OS compatible with device",
                  "VIRTIO_CONFIG_S_FEATURES_OK: Feature negotiation complete",
                  "VIRTIO_CONFIG_S_DRIVER_OK: Driver setup and ready"
              ]
          },
          "num-vqs": 2,
          "guest-features": {
              "dev-features": [],
              "transports": [
                  "VIRTIO_RING_F_EVENT_IDX: Used & avail. event fields enabled",
                  "VIRTIO_RING_F_INDIRECT_DESC: Indirect descriptors supported",
                  "VIRTIO_F_VERSION_1: Device compliant for v1 spec (legacy)"
              ]
          },
          "host-features": {
              "unknown-dev-features": 1073741824,
              "dev-features": [],
              "transports": [
                  "VIRTIO_RING_F_EVENT_IDX: Used & avail. event fields enabled",
                  "VIRTIO_RING_F_INDIRECT_DESC: Indirect descriptors supported",
                  "VIRTIO_F_VERSION_1: Device compliant for v1 spec (legacy)",
                  "VIRTIO_F_ANY_LAYOUT: Device accepts arbitrary desc. layouts",
                  "VIRTIO_F_NOTIFY_ON_EMPTY: Notify when device runs out of avail. descs.␣
↪on VQ"
              ]
          },
          "use-guest-notifier-mask": true,
          "vm-running": true,
```

(continues on next page)

(continued from previous page)

```
         "queue-sel": 1,
         "disabled": false,
         "vhost-started": false,
         "use-started": true
      }
   }

2. Poll for the status of virtio-net (vhost-net is active)

-> { "execute": "x-query-virtio-status",
     "arguments": { "path": "/machine/peripheral-anon/device[1]/virtio-backend" }
   }
<- { "return": {
         "device-endian": "little",
         "bus-name": "",
         "disabled-legacy-check": false,
         "name": "virtio-net",
         "started": true,
         "device-id": 1,
         "vhost-dev": {
             "n-tmp-sections": 4,
             "n-mem-sections": 4,
             "max-queues": 1,
             "backend-cap": 2,
             "log-size": 0,
             "backend-features": {
                 "dev-features": [],
                 "transports": []
             },
             "nvqs": 2,
             "protocol-features": {
                 "protocols": []
             },
             "vq-index": 0,
             "log-enabled": false,
             "acked-features": {
                 "dev-features": [
                     "VIRTIO_NET_F_MRG_RXBUF: Driver can merge receive buffers"
                 ],
                 "transports": [
                     "VIRTIO_RING_F_EVENT_IDX: Used & avail. event fields enabled",
                     "VIRTIO_RING_F_INDIRECT_DESC: Indirect descriptors supported",
                     "VIRTIO_F_VERSION_1: Device compliant for v1 spec (legacy)"
                 ]
             },
             "features": {
                 "dev-features": [
                     "VHOST_F_LOG_ALL: Logging write descriptors supported",
                     "VIRTIO_NET_F_MRG_RXBUF: Driver can merge receive buffers"
                 ],
                 "transports": [
                     "VIRTIO_RING_F_EVENT_IDX: Used & avail. event fields enabled",
```

(continues on next page)

(continued from previous page)

```
                    "VIRTIO_RING_F_INDIRECT_DESC: Indirect descriptors supported",
                    "VIRTIO_F_IOMMU_PLATFORM: Device can be used on IOMMU platform",
                    "VIRTIO_F_VERSION_1: Device compliant for v1 spec (legacy)",
                    "VIRTIO_F_ANY_LAYOUT: Device accepts arbitrary desc. layouts",
                    "VIRTIO_F_NOTIFY_ON_EMPTY: Notify when device runs out of avail.␣
↪descs. on VQ"
                ]
            }
        },
        "backend-features": {
            "dev-features": [
                "VHOST_USER_F_PROTOCOL_FEATURES: Vhost-user protocol features␣
↪negotiation supported",
                "VIRTIO_NET_F_GSO: Handling GSO-type packets supported",
                "VIRTIO_NET_F_CTRL_MAC_ADDR: MAC address set through control channel",
                "VIRTIO_NET_F_GUEST_ANNOUNCE: Driver sending gratuitous packets␣
↪supported",
                "VIRTIO_NET_F_CTRL_RX_EXTRA: Extra RX mode control supported",
                "VIRTIO_NET_F_CTRL_VLAN: Control channel VLAN filtering supported",
                "VIRTIO_NET_F_CTRL_RX: Control channel RX mode supported",
                "VIRTIO_NET_F_CTRL_VQ: Control channel available",
                "VIRTIO_NET_F_STATUS: Configuration status field available",
                "VIRTIO_NET_F_MRG_RXBUF: Driver can merge receive buffers",
                "VIRTIO_NET_F_HOST_UFO: Device can receive UFO",
                "VIRTIO_NET_F_HOST_ECN: Device can receive TSO with ECN",
                "VIRTIO_NET_F_HOST_TSO6: Device can receive TSOv6",
                "VIRTIO_NET_F_HOST_TSO4: Device can receive TSOv4",
                "VIRTIO_NET_F_GUEST_UFO: Driver can receive UFO",
                "VIRTIO_NET_F_GUEST_ECN: Driver can receive TSO with ECN",
                "VIRTIO_NET_F_GUEST_TSO6: Driver can receive TSOv6",
                "VIRTIO_NET_F_GUEST_TSO4: Driver can receive TSOv4",
                "VIRTIO_NET_F_MAC: Device has given MAC address",
                "VIRTIO_NET_F_CTRL_GUEST_OFFLOADS: Control channel offloading reconfig.␣
↪supported",
                "VIRTIO_NET_F_GUEST_CSUM: Driver handling packets with partial checksum␣
↪supported",
                "VIRTIO_NET_F_CSUM: Device handling packets with partial checksum␣
↪supported"
            ],
            "transports": [
                "VIRTIO_RING_F_EVENT_IDX: Used & avail. event fields enabled",
                "VIRTIO_RING_F_INDIRECT_DESC: Indirect descriptors supported",
                "VIRTIO_F_VERSION_1: Device compliant for v1 spec (legacy)",
                "VIRTIO_F_ANY_LAYOUT: Device accepts arbitrary desc. layouts",
                "VIRTIO_F_NOTIFY_ON_EMPTY: Notify when device runs out of avail. descs.␣
↪on VQ"
            ]
        },
        "start-on-kick": false,
        "isr": 1,
        "broken": false,
        "status": {
```

(continues on next page)

(continued from previous page)

```
            "statuses": [
                "VIRTIO_CONFIG_S_ACKNOWLEDGE: Valid virtio device found",
                "VIRTIO_CONFIG_S_DRIVER: Guest OS compatible with device",
                "VIRTIO_CONFIG_S_FEATURES_OK: Feature negotiation complete",
                "VIRTIO_CONFIG_S_DRIVER_OK: Driver setup and ready"
            ]
        },
        "num-vqs": 3,
        "guest-features": {
            "dev-features": [
                "VIRTIO_NET_F_CTRL_MAC_ADDR: MAC address set through control channel",
                "VIRTIO_NET_F_GUEST_ANNOUNCE: Driver sending gratuitous packets␣
↪supported",
                "VIRTIO_NET_F_CTRL_VLAN: Control channel VLAN filtering supported",
                "VIRTIO_NET_F_CTRL_RX: Control channel RX mode supported",
                "VIRTIO_NET_F_CTRL_VQ: Control channel available",
                "VIRTIO_NET_F_STATUS: Configuration status field available",
                "VIRTIO_NET_F_MRG_RXBUF: Driver can merge receive buffers",
                "VIRTIO_NET_F_HOST_UFO: Device can receive UFO",
                "VIRTIO_NET_F_HOST_ECN: Device can receive TSO with ECN",
                "VIRTIO_NET_F_HOST_TSO6: Device can receive TSOv6",
                "VIRTIO_NET_F_HOST_TSO4: Device can receive TSOv4",
                "VIRTIO_NET_F_GUEST_UFO: Driver can receive UFO",
                "VIRTIO_NET_F_GUEST_ECN: Driver can receive TSO with ECN",
                "VIRTIO_NET_F_GUEST_TSO6: Driver can receive TSOv6",
                "VIRTIO_NET_F_GUEST_TSO4: Driver can receive TSOv4",
                "VIRTIO_NET_F_MAC: Device has given MAC address",
                "VIRTIO_NET_F_CTRL_GUEST_OFFLOADS: Control channel offloading reconfig.␣
↪supported",
                "VIRTIO_NET_F_GUEST_CSUM: Driver handling packets with partial checksum␣
↪supported",
                "VIRTIO_NET_F_CSUM: Device handling packets with partial checksum␣
↪supported"
            ],
            "transports": [
                "VIRTIO_RING_F_EVENT_IDX: Used & avail. event fields enabled",
                "VIRTIO_RING_F_INDIRECT_DESC: Indirect descriptors supported",
                "VIRTIO_F_VERSION_1: Device compliant for v1 spec (legacy)"
           ]
        },
        "host-features": {
            "dev-features": [
                "VHOST_USER_F_PROTOCOL_FEATURES: Vhost-user protocol features␣
↪negotiation supported",
                "VIRTIO_NET_F_GSO: Handling GSO-type packets supported",
                "VIRTIO_NET_F_CTRL_MAC_ADDR: MAC address set through control channel",
                "VIRTIO_NET_F_GUEST_ANNOUNCE: Driver sending gratuitous packets␣
↪supported",
                "VIRTIO_NET_F_CTRL_RX_EXTRA: Extra RX mode control supported",
                "VIRTIO_NET_F_CTRL_VLAN: Control channel VLAN filtering supported",
                "VIRTIO_NET_F_CTRL_RX: Control channel RX mode supported",
                "VIRTIO_NET_F_CTRL_VQ: Control channel available",
```

(continues on next page)

(continued from previous page)

```
                "VIRTIO_NET_F_STATUS: Configuration status field available",
                "VIRTIO_NET_F_MRG_RXBUF: Driver can merge receive buffers",
                "VIRTIO_NET_F_HOST_UFO: Device can receive UFO",
                "VIRTIO_NET_F_HOST_ECN: Device can receive TSO with ECN",
                "VIRTIO_NET_F_HOST_TSO6: Device can receive TSOv6",
                "VIRTIO_NET_F_HOST_TSO4: Device can receive TSOv4",
                "VIRTIO_NET_F_GUEST_UFO: Driver can receive UFO",
                "VIRTIO_NET_F_GUEST_ECN: Driver can receive TSO with ECN",
                "VIRTIO_NET_F_GUEST_TSO6: Driver can receive TSOv6",
                "VIRTIO_NET_F_GUEST_TSO4: Driver can receive TSOv4",
                "VIRTIO_NET_F_MAC: Device has given MAC address",
                "VIRTIO_NET_F_CTRL_GUEST_OFFLOADS: Control channel offloading reconfig.␣
↪supported",
                "VIRTIO_NET_F_GUEST_CSUM: Driver handling packets with partial checksum␣
↪supported",
                "VIRTIO_NET_F_CSUM: Device handling packets with partial checksum␣
↪supported"
            ],
            "transports": [
                "VIRTIO_RING_F_EVENT_IDX: Used & avail. event fields enabled",
                "VIRTIO_RING_F_INDIRECT_DESC: Indirect descriptors supported",
                "VIRTIO_F_VERSION_1: Device compliant for v1 spec (legacy)",
                "VIRTIO_F_ANY_LAYOUT: Device accepts arbitrary desc. layouts",
                "VIRTIO_F_NOTIFY_ON_EMPTY: Notify when device runs out of avail. descs.␣
↪on VQ"
            ]
        },
        "use-guest-notifier-mask": true,
        "vm-running": true,
        "queue-sel": 2,
        "disabled": false,
        "vhost-started": true,
        "use-started": true
    }
  }
```

`VirtioDeviceStatus` (Object)

A structure defined to list the configuration statuses of a virtio device

Members

`statuses`: `array of string`
List of decoded configuration statuses of the virtio device

`unknown-statuses`: `int` (optional)
Virtio device statuses bitmap that have not been decoded

Since

7.2

VhostDeviceProtocols (Object)

A structure defined to list the vhost user protocol features of a Vhost User device

Members

protocols: array of string
List of decoded vhost user protocol features of a vhost user device

unknown-protocols: int (optional)
Vhost user device protocol features bitmap that have not been decoded

Since

7.2

VirtioDeviceFeatures (Object)

The common fields that apply to most Virtio devices. Some devices may not have their own device-specific features (e.g. virtio-rng).

Members

transports: array of string
List of transport features of the virtio device

dev-features: array of string (optional)
List of device-specific features (if the device has unique features)

unknown-dev-features: int (optional)
Virtio device features bitmap that have not been decoded

Since

7.2

`VirtQueueStatus` (Object)

Information of a VirtIODevice VirtQueue, including most members of the VirtQueue data structure.

Members

`name`: `string`
Name of the VirtIODevice that uses this VirtQueue

`queue-index`: `int`
VirtQueue queue_index

`inuse`: `int`
VirtQueue inuse

`vring-num`: `int`
VirtQueue vring.num

`vring-num-default`: `int`
VirtQueue vring.num_default

`vring-align`: `int`
VirtQueue vring.align

`vring-desc`: `int`
VirtQueue vring.desc (descriptor area)

`vring-avail`: `int`
VirtQueue vring.avail (driver area)

`vring-used`: `int`
VirtQueue vring.used (device area)

`last-avail-idx`: `int` (optional)
VirtQueue last_avail_idx or return of vhost_dev vhost_get_vring_base (if vhost active)

`shadow-avail-idx`: `int` (optional)
VirtQueue shadow_avail_idx

`used-idx`: `int`
VirtQueue used_idx

`signalled-used`: `int`
VirtQueue signalled_used

`signalled-used-valid`: `boolean`
VirtQueue signalled_used_valid flag

Since

7.2

x-query-virtio-queue-status (Command)

Return the status of a given VirtIODevice's VirtQueue

Arguments

path: string
VirtIODevice canonical QOM path

queue: int
VirtQueue index to examine

Features

unstable
This command is meant for debugging.

Returns

VirtQueueStatus of the VirtQueue

Notes

last_avail_idx will not be displayed in the case where the selected VirtIODevice has a running vhost device and the VirtIODevice VirtQueue index (queue) does not exist for the corresponding vhost device vhost_virtqueue. Also, shadow_avail_idx will not be displayed in the case where the selected VirtIODevice has a running vhost device.

Since

7.2

Examples

```
1. Get VirtQueueStatus for virtio-vsock (vhost-vsock running)

-> { "execute": "x-query-virtio-queue-status",
     "arguments": { "path": "/machine/peripheral/vsock0/virtio-backend",
                    "queue": 1 }
   }
<- { "return": {
          "signalled-used": 0,
          "inuse": 0,
          "name": "vhost-vsock",
          "vring-align": 4096,
          "vring-desc": 5217370112,
          "signalled-used-valid": false,
          "vring-num-default": 128,
```

(continues on next page)

(continued from previous page)

```
         "vring-avail": 5217372160,
         "queue-index": 1,
         "last-avail-idx": 0,
         "vring-used": 5217372480,
         "used-idx": 0,
         "vring-num": 128
      }
   }

2. Get VirtQueueStatus for virtio-serial (no vhost)

-> { "execute": "x-query-virtio-queue-status",
     "arguments": { "path": "/machine/peripheral-anon/device[0]/virtio-backend",
                    "queue": 20 }
   }
<- { "return": {
         "signalled-used": 0,
         "inuse": 0,
         "name": "virtio-serial",
         "vring-align": 4096,
         "vring-desc": 5182074880,
         "signalled-used-valid": false,
         "vring-num-default": 128,
         "vring-avail": 5182076928,
         "queue-index": 20,
         "last-avail-idx": 0,
         "vring-used": 5182077248,
         "used-idx": 0,
         "shadow-avail-idx": 0,
         "vring-num": 128
      }
   }
```

VirtVhostQueueStatus (Object)

Information of a vhost device's vhost_virtqueue, including most members of the vhost_dev vhost_virtqueue data structure.

Members

name: string
Name of the VirtIODevice that uses this vhost_virtqueue

kick: int
vhost_virtqueue kick

call: int
vhost_virtqueue call

desc: int
vhost_virtqueue desc

avail: int
vhost_virtqueue avail

used: int
vhost_virtqueue used

num: int
vhost_virtqueue num

desc-phys: int
vhost_virtqueue desc_phys (descriptor area physical address)

desc-size: int
vhost_virtqueue desc_size

avail-phys: int
vhost_virtqueue avail_phys (driver area physical address)

avail-size: int
vhost_virtqueue avail_size

used-phys: int
vhost_virtqueue used_phys (device area physical address)

used-size: int
vhost_virtqueue used_size

Since

7.2

x-query-virtio-vhost-queue-status (Command)

Return information of a given vhost device's vhost_virtqueue

Arguments

path: string
VirtIODevice canonical QOM path

queue: int
vhost_virtqueue index to examine

Features

unstable
This command is meant for debugging.

Returns

VirtVhostQueueStatus of the vhost_virtqueue

Since

7.2

Examples

```
1. Get vhost_virtqueue status for vhost-crypto

-> { "execute": "x-query-virtio-vhost-queue-status",
     "arguments": { "path": "/machine/peripheral/crypto0/virtio-backend",
                    "queue": 0 }
   }
<- { "return": {
          "avail-phys": 5216124928,
          "name": "virtio-crypto",
          "used-phys": 5216127040,
          "avail-size": 2054,
          "desc-size": 16384,
          "used-size": 8198,
          "desc": 140141447430144,
          "num": 1024,
          "call": 0,
          "avail": 140141447446528,
          "desc-phys": 5216108544,
          "used": 140141447448640,
          "kick": 0
     }
   }

2. Get vhost_virtqueue status for vhost-vsock

-> { "execute": "x-query-virtio-vhost-queue-status",
     "arguments": { "path": "/machine/peripheral/vsock0/virtio-backend",
                    "queue": 0 }
   }
<- { "return": {
          "avail-phys": 5182261248,
          "name": "vhost-vsock",
          "used-phys": 5182261568,
          "avail-size": 262,
          "desc-size": 2048,
          "used-size": 1030,
          "desc": 140141413580800,
          "num": 128,
          "call": 0,
          "avail": 140141413582848,
          "desc-phys": 5182259200,
```

(continues on next page)

(continued from previous page)

```
        "used": 140141413583168,
        "kick": 0
    }
  }
```

VirtioRingDesc (Object)

Information regarding the vring descriptor area

Members

addr: int
Guest physical address of the descriptor area

len: int
Length of the descriptor area

flags: array of string
List of descriptor flags

Since

7.2

VirtioRingAvail (Object)

Information regarding the avail vring (a.k.a. driver area)

Members

flags: int
VRingAvail flags

idx: int
VRingAvail index

ring: int
VRingAvail ring[] entry at provided index

Since

7.2

VirtioRingUsed (Object)

Information regarding the used vring (a.k.a. device area)

Members

flags: int
VRingUsed flags

idx: int
VRingUsed index

Since

7.2

VirtioQueueElement (Object)

Information regarding a VirtQueue's VirtQueueElement including descriptor, driver, and device areas

Members

name: string
Name of the VirtIODevice that uses this VirtQueue

index: int
Index of the element in the queue

descs: array of VirtioRingDesc
List of descriptors (VirtioRingDesc)

avail: VirtioRingAvail
VRingAvail info

used: VirtioRingUsed
VRingUsed info

Since

7.2

x-query-virtio-queue-element (Command)

Return the information about a VirtQueue's VirtQueueElement

Arguments

path: string
VirtIODevice canonical QOM path

queue: int
VirtQueue index to examine

index: int (optional)
Index of the element in the queue (default: head of the queue)

Features

unstable
This command is meant for debugging.

Returns

VirtioQueueElement information

Since

7.2

Examples

```
1. Introspect on virtio-net's VirtQueue 0 at index 5

-> { "execute": "x-query-virtio-queue-element",
     "arguments": { "path": "/machine/peripheral-anon/device[1]/virtio-backend",
                    "queue": 0,
                    "index": 5 }
   }
<- { "return": {
         "index": 5,
         "name": "virtio-net",
         "descs": [
             {
                 "flags": ["write"],
                 "len": 1536,
                 "addr": 5257305600
             }
         ],
         "avail": {
             "idx": 256,
             "flags": 0,
             "ring": 5
         },
         "used": {
             "idx": 13,
```

(continues on next page)

(continued from previous page)

```
            "flags": 0
        }
    }
}

2. Introspect on virtio-crypto's VirtQueue 1 at head

-> { "execute": "x-query-virtio-queue-element",
     "arguments": { "path": "/machine/peripheral/crypto0/virtio-backend",
                    "queue": 1 }
}
<- { "return": {
        "index": 0,
        "name": "virtio-crypto",
        "descs": [
            {
                "flags": [],
                "len": 0,
                "addr": 8080268923184214134
            }
        ],
        "avail": {
            "idx": 280,
            "flags": 0,
            "ring": 0
        },
        "used": {
            "idx": 280,
            "flags": 0
        }
    }
}

3. Introspect on virtio-scsi's VirtQueue 2 at head

-> { "execute": "x-query-virtio-queue-element",
     "arguments": { "path": "/machine/peripheral-anon/device[2]/virtio-backend",
                    "queue": 2 }
}
<- { "return": {
        "index": 19,
        "name": "virtio-scsi",
        "descs": [
            {
                "flags": ["used", "indirect", "write"],
                "len": 4099327944,
                "addr": 12055409292258155293
            }
        ],
        "avail": {
            "idx": 1147,
            "flags": 0,
```

(continues on next page)

(continued from previous page)

```
            "ring": 19
        },
        "used": {
            "idx": 280,
            "flags": 0
        }
    }
}
```

IOThreadVirtQueueMapping (Object)

Describes the subset of virtqueues assigned to an IOThread.

Members

iothread: string
the id of IOThread object

vqs: array of int (optional)
an optional array of virtqueue indices that will be handled by this IOThread. When absent, virtqueues are assigned round-robin across all IOThreadVirtQueueMappings provided. Either all IOThreadVirtQueueMappings must have `vqs` or none of them must have it.

Since

9.0

DummyVirtioForceArrays (Object)

Not used by QMP; hack to let us use IOThreadVirtQueueMappingList internally

Members

unused-iothread-vq-mapping: array of IOThreadVirtQueueMapping
Not documented

Since

9.0

GranuleMode (Enum)

Values

4k
granule page size of 4KiB

8k
granule page size of 8KiB

16k
granule page size of 16KiB

64k
granule page size of 64KiB

host
granule matches the host page size

Since

9.0

5.11.36 Cryptography devices

QCryptodevBackendAlgType (Enum)

The supported algorithm types of a crypto device.

Values

sym
symmetric encryption

asym
asymmetric Encryption

Since

8.0

QCryptodevBackendServiceType (Enum)

The supported service types of a crypto device.

Values

cipher
Not documented

hash
Not documented

mac
Not documented

aead
Not documented

akcipher
Not documented

Since

8.0

QCryptodevBackendType (Enum)

The crypto device backend type

Values

builtin
the QEMU builtin support

vhost-user
vhost-user

lkcf
Linux kernel cryptographic framework

Since

8.0

QCryptodevBackendClient (Object)

Information about a queue of crypto device.

Members

queue: int
the queue index of the crypto device

type: QCryptodevBackendType
the type of the crypto device

Since

8.0

QCryptodevInfo (Object)

Information about a crypto device.

Members

id: string
the id of the crypto device

service: array of QCryptodevBackendServiceType
supported service types of a crypto device

client: array of QCryptodevBackendClient
the additional information of the crypto device

Since

8.0

query-cryptodev (Command)

Returns information about current crypto devices.

Returns

a list of `QCryptodevInfo`

Since

8.0

5.11.37 CXL devices

CxlEventLog (Enum)

CXL has a number of separate event logs for different types of events. Each such event log is handled and signaled independently.

Values

informational
Information Event Log

warning
Warning Event Log

failure
Failure Event Log

fatal
Fatal Event Log

Since

8.1

cxl-inject-general-media-event (Command)

Inject an event record for a General Media Event (CXL r3.0 8.2.9.2.1.1). This event type is reported via one of the event logs specified via the log parameter.

Arguments

path: string
CXL type 3 device canonical QOM path

log: CxlEventLog
event log to add the event to

flags: int
Event Record Flags. See CXL r3.0 Table 8-42 Common Event Record Format, Event Record Flags for subfield definitions.

dpa: int
Device Physical Address (relative to `path` device). Note lower bits include some flags. See CXL r3.0 Table 8-43 General Media Event Record, Physical Address.

descriptor: int
Memory Event Descriptor with additional memory event information. See CXL r3.0 Table 8-43 General Media Event Record, Memory Event Descriptor for bit definitions.

type: int
Type of memory event that occurred. See CXL r3.0 Table 8-43 General Media Event Record, Memory Event Type for possible values.

transaction-type: int
Type of first transaction that caused the event to occur. See CXL r3.0 Table 8-43 General Media Event Record, Transaction Type for possible values.

channel: int (optional)
The channel of the memory event location. A channel is an interface that can be independently accessed for a transaction.

rank: int (optional)
The rank of the memory event location. A rank is a set of memory devices on a channel that together execute a transaction.

device: int (optional)
Bitmask that represents all devices in the rank associated with the memory event location.

component-id: string (optional)
Device specific component identifier for the event. May describe a field replaceable sub-component of the device.

Since

8.1

cxl-inject-dram-event (Command)

Inject an event record for a DRAM Event (CXL r3.0 8.2.9.2.1.2). This event type is reported via one of the event logs specified via the log parameter.

Arguments

path: string
CXL type 3 device canonical QOM path

log: CxlEventLog
Event log to add the event to

flags: int
Event Record Flags. See CXL r3.0 Table 8-42 Common Event Record Format, Event Record Flags for subfield definitions.

dpa: int
Device Physical Address (relative to `path` device). Note lower bits include some flags. See CXL r3.0 Table 8-44 DRAM Event Record, Physical Address.

descriptor: int
Memory Event Descriptor with additional memory event information. See CXL r3.0 Table 8-44 DRAM Event Record, Memory Event Descriptor for bit definitions.

type: int
Type of memory event that occurred. See CXL r3.0 Table 8-44 DRAM Event Record, Memory Event Type for possible values.

transaction-type: int
Type of first transaction that caused the event to occur. See CXL r3.0 Table 8-44 DRAM Event Record, Transaction Type for possible values.

channel: int (optional)
The channel of the memory event location. A channel is an interface that can be independently accessed for a transaction.

rank: int (optional)
The rank of the memory event location. A rank is a set of memory devices on a channel that together execute a transaction.

nibble-mask: int (optional)
Identifies one or more nibbles that the error affects

bank-group: int (optional)
Bank group of the memory event location, incorporating a number of Banks.

bank: int (optional)
Bank of the memory event location. A single bank is accessed per read or write of the memory.

row: int (optional)
Row address within the DRAM.

column: int (optional)
Column address within the DRAM.

correction-mask: array of int (optional)
Bits within each nibble. Used in order of bits set in the nibble-mask. Up to 4 nibbles may be covered.

Since

8.1

cxl-inject-memory-module-event (Command)

Inject an event record for a Memory Module Event (CXL r3.0 8.2.9.2.1.3). This event includes a copy of the Device Health info at the time of the event.

Arguments

path: string
CXL type 3 device canonical QOM path

log: CxlEventLog
Event Log to add the event to

flags: int
Event Record Flags. See CXL r3.0 Table 8-42 Common Event Record Format, Event Record Flags for subfield definitions.

type: int
Device Event Type. See CXL r3.0 Table 8-45 Memory Module Event Record for bit definitions for bit definiions.

health-status: int
Overall health summary bitmap. See CXL r3.0 Table 8-100 Get Health Info Output Payload, Health Status for bit definitions.

media-status: int
Overall media health summary. See CXL r3.0 Table 8-100 Get Health Info Output Payload, Media Status for bit definitions.

additional-status: int
See CXL r3.0 Table 8-100 Get Health Info Output Payload, Additional Status for subfield definitions.

life-used: int
Percentage (0-100) of factory expected life span.

temperature: int
Device temperature in degrees Celsius.

dirty-shutdown-count: int
Number of times the device has been unable to determine whether data loss may have occurred.

corrected-volatile-error-count: int
Total number of correctable errors in volatile memory.

corrected-persistent-error-count: int
Total number of correctable errors in persistent memory

Since

8.1

cxl-inject-poison (Command)

Poison records indicate that a CXL memory device knows that a particular memory region may be corrupted. This may be because of locally detected errors (e.g. ECC failure) or poisoned writes received from other components in the system. This injection mechanism enables testing of the OS handling of poison records which may be queried via the CXL mailbox.

Arguments

path: string
CXL type 3 device canonical QOM path

start: int
Start address; must be 64 byte aligned.

length: int
Length of poison to inject; must be a multiple of 64 bytes.

Since

8.1

CxlUncorErrorType (Enum)

Type of uncorrectable CXL error to inject. These errors are reported via an AER uncorrectable internal error with additional information logged at the CXL device.

Values

cache-data-parity
Data error such as data parity or data ECC error CXL.cache

cache-address-parity
Address parity or other errors associated with the address field on CXL.cache

cache-be-parity
Byte enable parity or other byte enable errors on CXL.cache

cache-data-ecc
ECC error on CXL.cache

mem-data-parity
Data error such as data parity or data ECC error on CXL.mem

mem-address-parity
Address parity or other errors associated with the address field on CXL.mem

mem-be-parity
Byte enable parity or other byte enable errors on CXL.mem.

mem-data-ecc
Data ECC error on CXL.mem.

reinit-threshold
REINIT threshold hit.

rsvd-encoding
Received unrecognized encoding.

poison-received
Received poison from the peer.

receiver-overflow
Buffer overflows (first 3 bits of header log indicate which)

internal
Component specific error

cxl-ide-tx
Integrity and data encryption tx error.

cxl-ide-rx
Integrity and data encryption rx error.

Since

8.0

CXLUncorErrorRecord (Object)

Record of a single error including header log.

Members

type: CxlUncorErrorType
Type of error

header: array of int
16 DWORD of header.

Since

8.0

cxl-inject-uncorrectable-errors (Command)

Command to allow injection of multiple errors in one go. This allows testing of multiple header log handling in the OS.

Arguments

path: string
CXL Type 3 device canonical QOM path

errors: array of CXLUncorErrorRecord
Errors to inject

Since

8.0

CxlCorErrorType (Enum)

Type of CXL correctable error to inject

Values

cache-data-ecc
Data ECC error on CXL.cache

mem-data-ecc
Data ECC error on CXL.mem

crc-threshold
Component specific and applicable to 68 byte Flit mode only.

cache-poison-received
Received poison from a peer on CXL.cache.

mem-poison-received
Received poison from a peer on CXL.mem

physical
Received error indication from the physical layer.

retry-threshold
Not documented

Since

8.0

cxl-inject-correctable-error (Command)

Command to inject a single correctable error. Multiple error injection of this error type is not interesting as there is no associated header log. These errors are reported via AER as a correctable internal error, with additional detail available from the CXL device.

Arguments

path: string
CXL Type 3 device canonical QOM path

type: CxlCorErrorType
Type of error.

Since

8.0

5.12 QEMU Storage Daemon QMP Reference Manual

Contents

* qom-get (Command)
* qom-set (Command)
* ObjectTypeInfo (Object)
* qom-list-types (Command)
* qom-list-properties (Command)
* CanHostSocketcanProperties (Object)
* ColoCompareProperties (Object)
* CryptodevBackendProperties (Object)
* CryptodevVhostUserProperties (Object)
* DBusVMStateProperties (Object)
* NetfilterInsert (Enum)
* NetfilterProperties (Object)
* FilterBufferProperties (Object)
* FilterDumpProperties (Object)
* FilterMirrorProperties (Object)
* FilterRedirectorProperties (Object)
* FilterRewriterProperties (Object)
* InputBarrierProperties (Object)
* InputLinuxProperties (Object)
* EventLoopBaseProperties (Object)
* IothreadProperties (Object)
* MainLoopProperties (Object)
* MemoryBackendProperties (Object)
* MemoryBackendFileProperties (Object)
* MemoryBackendMemfdProperties (Object)
* MemoryBackendEpcProperties (Object)
* PrManagerHelperProperties (Object)
* QtestProperties (Object)
* RemoteObjectProperties (Object)
* VfioUserServerProperties (Object)
* IOMMUFDProperties (Object)
* AcpiGenericInitiatorProperties (Object)
* RngProperties (Object)
* RngEgdProperties (Object)
* RngRandomProperties (Object)

5.12.1 Common data types

IoOperationType (Enum)

An enumeration of the I/O operation types

Values

read
: read operation

write
: write operation

Since

2.1

OnOffAuto (Enum)

An enumeration of three options: on, off, and auto

Values

auto
: QEMU selects the value between on and off

on
: Enabled

off
: Disabled

Since

2.2

`OnOffSplit` (Enum)

An enumeration of three values: on, off, and split

Values

`on`
Enabled

`off`
Disabled

`split`
Mixed

Since

2.6

`StrOrNull` (Alternate)

This is a string value or the explicit lack of a string (null pointer in C). Intended for cases when 'optional absent' already has a different meaning.

Members

`s: string`
the string value

`n: null`
no string value

Since

2.10

OffAutoPCIBAR (Enum)

An enumeration of options for specifying a PCI BAR

Values

off
The specified feature is disabled

auto
The PCI BAR for the feature is automatically selected

bar0
PCI BAR0 is used for the feature

bar1
PCI BAR1 is used for the feature

bar2
PCI BAR2 is used for the feature

bar3
PCI BAR3 is used for the feature

bar4
PCI BAR4 is used for the feature

bar5
PCI BAR5 is used for the feature

Since

2.12

PCIELinkSpeed (Enum)

An enumeration of PCIe link speeds in units of GT/s

Values

2_5
2.5GT/s

5
5.0GT/s

8
8.0GT/s

16
16.0GT/s

32
32.0GT/s (since 9.0)

64
64.0GT/s (since 9.0)

Since

4.0

PCIELinkWidth (Enum)

An enumeration of PCIe link width

Values

1
x1

2
x2

4
x4

8
x8

12
x12

16
x16

32
x32

Since

4.0

HostMemPolicy (Enum)

Host memory policy types

Values

default
restore default policy, remove any nondefault policy

preferred
set the preferred host nodes for allocation

bind
a strict policy that restricts memory allocation to the host nodes specified

interleave
memory allocations are interleaved across the set of host nodes specified

Since

2.1

NetFilterDirection (Enum)

Indicates whether a netfilter is attached to a netdev's transmit queue or receive queue or both.

Values

all
the filter is attached both to the receive and the transmit queue of the netdev (default).

rx
the filter is attached to the receive queue of the netdev, where it will receive packets sent to the netdev.

tx
the filter is attached to the transmit queue of the netdev, where it will receive packets sent by the netdev.

Since

2.5

GrabToggleKeys (Enum)

Keys to toggle input-linux between host and guest.

Values

ctrl-ctrl
Not documented

alt-alt
Not documented

shift-shift
Not documented

meta-meta
Not documented

scrolllock
Not documented

ctrl-scrolllock
Not documented

Since

4.0

HumanReadableText (Object)

Members

human-readable-text: string
Formatted output intended for humans.

Since

6.2

5.12.2 Socket data types

NetworkAddressFamily (Enum)

The network address family

Values

ipv4
IPV4 family

ipv6
IPV6 family

unix
unix socket

vsock
vsock family (since 2.8)

unknown
otherwise

Since

2.1

InetSocketAddressBase (Object)

Members

host: string
host part of the address

port: string
port part of the address

InetSocketAddress (Object)

Captures a socket address or address range in the Internet namespace.

Members

numeric: boolean (optional)
true if the host/port are guaranteed to be numeric, false if name resolution should be attempted. Defaults to false. (Since 2.9)

to: int (optional)
If present, this is range of possible addresses, with port between `port` and `to`.

ipv4: boolean (optional)
whether to accept IPv4 addresses, default try both IPv4 and IPv6

ipv6: boolean (optional)
whether to accept IPv6 addresses, default try both IPv4 and IPv6

keep-alive: boolean (optional)
enable keep-alive when connecting to this socket. Not supported for passive sockets. (Since 4.2)

mptcp: boolean (optional) (If: HAVE_IPPROTO_MPTCP)
enable multi-path TCP. (Since 6.1)

The members of InetSocketAddressBase

Since

1.3

UnixSocketAddress (Object)

Captures a socket address in the local ("Unix socket") namespace.

Members

path: string
filesystem path to use

abstract: boolean (optional) (If: CONFIG_LINUX)
if true, this is a Linux abstract socket address. `path` will be prefixed by a null byte, and optionally padded with null bytes. Defaults to false. (Since 5.1)

tight: boolean (optional) (If: CONFIG_LINUX)
if false, pad an abstract socket address with enough null bytes to make it fill struct sockaddr_un member sun_path. Defaults to true. (Since 5.1)

Since

1.3

VsockSocketAddress (Object)

Captures a socket address in the vsock namespace.

Members

cid: string
unique host identifier

port: string
port

Note

string types are used to allow for possible future hostname or service resolution support.

Since

2.8

FdSocketAddress (Object)

A file descriptor name or number.

Members

str: string
decimal is for file descriptor number, otherwise it's a file descriptor name. Named file descriptors are permitted in monitor commands, in combination with the 'getfd' command. Decimal file descriptors are permitted at startup or other contexts where no monitor context is active.

Since

1.2

InetSocketAddressWrapper (Object)

Members

data: InetSocketAddress
internet domain socket address

Since

1.3

UnixSocketAddressWrapper (Object)

Members

data: UnixSocketAddress
UNIX domain socket address

Since

1.3

VsockSocketAddressWrapper (Object)

Members

data: VsockSocketAddress
VSOCK domain socket address

Since

2.8

FdSocketAddressWrapper (Object)

Members

data: FdSocketAddress
file descriptor name or number

Since

1.3

SocketAddressLegacy (Object)

Captures the address of a socket, which could also be a named file descriptor

Members

type: SocketAddressType
Transport type

The members of InetSocketAddressWrapper when type is "inet"
The members of UnixSocketAddressWrapper when type is "unix"
The members of VsockSocketAddressWrapper when type is "vsock"
The members of FdSocketAddressWrapper when type is "fd"

Note

This type is deprecated in favor of SocketAddress. The difference between SocketAddressLegacy and SocketAddress is that the latter has fewer {} on the wire.

Since

1.3

SocketAddressType (Enum)

Available SocketAddress types

Values

inet
Internet address

unix
Unix domain socket

vsock
VMCI address

fd
Socket file descriptor

Since

2.9

SocketAddress (Object)

Captures the address of a socket, which could also be a socket file descriptor

Members

type: SocketAddressType
Transport type

The members of InetSocketAddress when type is "inet"
The members of UnixSocketAddress when type is "unix"
The members of VsockSocketAddress when type is "vsock"
The members of FdSocketAddress when type is "fd"

Since

2.9

5.12.3 Cryptography

QCryptoTLSCredsEndpoint (Enum)

The type of network endpoint that will be using the credentials. Most types of credential require different setup / structures depending on whether they will be used in a server versus a client.

Values

client
the network endpoint is acting as the client

server
the network endpoint is acting as the server

Since

2.5

QCryptoSecretFormat (Enum)

The data format that the secret is provided in

Values

raw
raw bytes. When encoded in JSON only valid UTF-8 sequences can be used

base64
arbitrary base64 encoded binary data

Since

2.6

QCryptoHashAlgorithm (Enum)

The supported algorithms for computing content digests

Values

md5
MD5. Should not be used in any new code, legacy compat only

sha1
SHA-1. Should not be used in any new code, legacy compat only

sha224
SHA-224. (since 2.7)

sha256
SHA-256. Current recommended strong hash.

sha384
SHA-384. (since 2.7)

sha512
SHA-512. (since 2.7)

ripemd160
RIPEMD-160. (since 2.7)

Since

2.6

QCryptoCipherAlgorithm (Enum)

The supported algorithms for content encryption ciphers

Values

aes-128
AES with 128 bit / 16 byte keys

aes-192
AES with 192 bit / 24 byte keys

aes-256
AES with 256 bit / 32 byte keys

des
DES with 56 bit / 8 byte keys. Do not use except in VNC. (since 6.1)

3des
3DES(EDE) with 192 bit / 24 byte keys (since 2.9)

cast5-128
Cast5 with 128 bit / 16 byte keys

serpent-128
Serpent with 128 bit / 16 byte keys

serpent-192
Serpent with 192 bit / 24 byte keys

serpent-256
Serpent with 256 bit / 32 byte keys

twofish-128
Twofish with 128 bit / 16 byte keys

twofish-192
Twofish with 192 bit / 24 byte keys

twofish-256
Twofish with 256 bit / 32 byte keys

sm4
SM4 with 128 bit / 16 byte keys (since 9.0)

Since

2.6

QCryptoCipherMode (Enum)

The supported modes for content encryption ciphers

Values

ecb
Electronic Code Book

cbc
Cipher Block Chaining

xts
XEX with tweaked code book and ciphertext stealing

ctr
Counter (Since 2.8)

Since

2.6

QCryptoIVGenAlgorithm (Enum)

The supported algorithms for generating initialization vectors for full disk encryption. The 'plain' generator should not be used for disks with sector numbers larger than 2^32, except where compatibility with pre-existing Linux dm-crypt volumes is required.

Values

plain
64-bit sector number truncated to 32-bits

plain64
64-bit sector number

essiv
64-bit sector number encrypted with a hash of the encryption key

Since

2.6

QCryptoBlockFormat (Enum)

The supported full disk encryption formats

Values

qcow
: QCow/QCow2 built-in AES-CBC encryption. Use only for liberating data from old images.

luks
: LUKS encryption format. Recommended for new images

Since

2.6

QCryptoBlockOptionsBase (Object)

The common options that apply to all full disk encryption formats

Members

format: QCryptoBlockFormat
: the encryption format

Since

2.6

QCryptoBlockOptionsQCow (Object)

The options that apply to QCow/QCow2 AES-CBC encryption format

Members

key-secret: string (optional)
: the ID of a QCryptoSecret object providing the decryption key. Mandatory except when probing image for metadata only.

Since

2.6

QCryptoBlockOptionsLUKS (Object)

The options that apply to LUKS encryption format

Members

key-secret: string (optional)
the ID of a QCryptoSecret object providing the decryption key. Mandatory except when probing image for metadata only.

Since

2.6

QCryptoBlockCreateOptionsLUKS (Object)

The options that apply to LUKS encryption format initialization

Members

cipher-alg: QCryptoCipherAlgorithm (optional)
the cipher algorithm for data encryption Currently defaults to 'aes-256'.

cipher-mode: QCryptoCipherMode (optional)
the cipher mode for data encryption Currently defaults to 'xts'

ivgen-alg: QCryptoIVGenAlgorithm (optional)
the initialization vector generator Currently defaults to 'plain64'

ivgen-hash-alg: QCryptoHashAlgorithm (optional)
the initialization vector generator hash Currently defaults to 'sha256'

hash-alg: QCryptoHashAlgorithm (optional)
the master key hash algorithm Currently defaults to 'sha256'

iter-time: int (optional)
number of milliseconds to spend in PBKDF passphrase processing. Currently defaults to 2000. (since 2.8)

detached-header: boolean (optional)
create a detached LUKS header. (since 9.0)

The members of QCryptoBlockOptionsLUKS

Since

2.6

QCryptoBlockOpenOptions (Object)

The options that are available for all encryption formats when opening an existing volume

Members

The members of QCryptoBlockOptionsBase
The members of QCryptoBlockOptionsQCow when format is "qcow"
The members of QCryptoBlockOptionsLUKS when format is "luks"

Since

2.6

QCryptoBlockCreateOptions (Object)

The options that are available for all encryption formats when initializing a new volume

Members

The members of QCryptoBlockOptionsBase
The members of QCryptoBlockOptionsQCow when format is "qcow"
The members of QCryptoBlockCreateOptionsLUKS when format is "luks"

Since

2.6

QCryptoBlockInfoBase (Object)

The common information that applies to all full disk encryption formats

Members

format: QCryptoBlockFormat
: the encryption format

Since

2.7

QCryptoBlockInfoLUKSSlot (Object)

Information about the LUKS block encryption key slot options

Members

active: boolean
whether the key slot is currently in use

key-offset: int
offset to the key material in bytes

iters: int (optional)
number of PBKDF2 iterations for key material

stripes: int (optional)
number of stripes for splitting key material

Since

2.7

QCryptoBlockInfoLUKS (Object)

Information about the LUKS block encryption options

Members

cipher-alg: QCryptoCipherAlgorithm
the cipher algorithm for data encryption

cipher-mode: QCryptoCipherMode
the cipher mode for data encryption

ivgen-alg: QCryptoIVGenAlgorithm
the initialization vector generator

ivgen-hash-alg: QCryptoHashAlgorithm (optional)
the initialization vector generator hash

hash-alg: QCryptoHashAlgorithm
the master key hash algorithm

detached-header: boolean
whether the LUKS header is detached (Since 9.0)

payload-offset: int
offset to the payload data in bytes

master-key-iters: int
number of PBKDF2 iterations for key material

uuid: string
unique identifier for the volume

slots: array of QCryptoBlockInfoLUKSSlot
information about each key slot

Since

2.7

QCryptoBlockInfo (Object)

Information about the block encryption options

Members

The members of QCryptoBlockInfoBase
The members of QCryptoBlockInfoLUKS when format is "luks"

Since

2.7

QCryptoBlockLUKSKeyslotState (Enum)

Defines state of keyslots that are affected by the update

Values

active
The slots contain the given password and marked as active

inactive
The slots are erased (contain garbage) and marked as inactive

Since

5.1

QCryptoBlockAmendOptionsLUKS (Object)

This struct defines the update parameters that activate/de-activate set of keyslots

Members

state: QCryptoBlockLUKSKeyslotState
the desired state of the keyslots

new-secret: string (optional)
The ID of a QCryptoSecret object providing the password to be written into added active keyslots

old-secret: string (optional)
Optional (for deactivation only) If given will deactivate all keyslots that match password located in QCryptoSecret with this ID

iter-time: int (optional)
Optional (for activation only) Number of milliseconds to spend in PBKDF passphrase processing for the newly activated keyslot. Currently defaults to 2000.

keyslot: int (optional)
Optional. ID of the keyslot to activate/deactivate. For keyslot activation, keyslot should not be active already (this is unsafe to update an active keyslot), but possible if 'force' parameter is given. If keyslot is not given, first free keyslot will be written.

For keyslot deactivation, this parameter specifies the exact keyslot to deactivate

secret: string (optional)
Optional. The ID of a QCryptoSecret object providing the password to use to retrieve current master key. Defaults to the same secret that was used to open the image

Since

5.1

QCryptoBlockAmendOptions (Object)

The options that are available for all encryption formats when amending encryption settings

Members

The members of QCryptoBlockOptionsBase
The members of QCryptoBlockAmendOptionsLUKS when format is "luks"

Since

5.1

SecretCommonProperties (Object)

Properties for objects of classes derived from secret-common.

Members

loaded: boolean (optional)
: if true, the secret is loaded immediately when applying this option and will probably fail when processing the next option. Don't use; only provided for compatibility. (default: false)

format: QCryptoSecretFormat (optional)
: the data format that the secret is provided in (default: raw)

keyid: string (optional)
: the name of another secret that should be used to decrypt the provided data. If not present, the data is assumed to be unencrypted.

iv: string (optional)
: the random initialization vector used for encryption of this particular secret. Should be a base64 encrypted string of the 16-byte IV. Mandatory if `keyid` is given. Ignored if `keyid` is absent.

Features

deprecated
: Member `loaded` is deprecated. Setting true doesn't make sense, and false is already the default.

Since

2.6

SecretProperties (Object)

Properties for secret objects.

Either `data` or `file` must be provided, but not both.

Members

data: string (optional)
: the associated with the secret from

file: string (optional)
: the filename to load the data associated with the secret from

The members of SecretCommonProperties

Since

2.6

SecretKeyringProperties (Object)

Properties for secret_keyring objects.

Members

serial: int
serial number that identifies a key to get from the kernel

The members of SecretCommonProperties

Since

5.1

TlsCredsProperties (Object)

Properties for objects of classes derived from tls-creds.

Members

verify-peer: boolean (optional)
if true the peer credentials will be verified once the handshake is completed. This is a no-op for anonymous credentials. (default: true)

dir: string (optional)
the path of the directory that contains the credential files

endpoint: QCryptoTLSCredsEndpoint (optional)
whether the QEMU network backend that uses the credentials will be acting as a client or as a server (default: client)

priority: string (optional)
a gnutls priority string as described at https://gnutls.org/manual/html_node/Priority-Strings.html

Since

2.5

TlsCredsAnonProperties (Object)

Properties for tls-creds-anon objects.

Members

loaded: boolean (optional)
if true, the credentials are loaded immediately when applying this option and will ignore options that are processed later. Don't use; only provided for compatibility. (default: false)

The members of TlsCredsProperties

Features

deprecated
Member loaded is deprecated. Setting true doesn't make sense, and false is already the default.

Since

2.5

TlsCredsPskProperties (Object)

Properties for tls-creds-psk objects.

Members

loaded: boolean (optional)
if true, the credentials are loaded immediately when applying this option and will ignore options that are processed later. Don't use; only provided for compatibility. (default: false)

username: string (optional)
the username which will be sent to the server. For clients only. If absent, "qemu" is sent and the property will read back as an empty string.

The members of TlsCredsProperties

Features

deprecated
Member loaded is deprecated. Setting true doesn't make sense, and false is already the default.

Since

3.0

TlsCredsX509Properties (Object)

Properties for tls-creds-x509 objects.

Members

loaded: boolean (optional)
: if true, the credentials are loaded immediately when applying this option and will ignore options that are processed later. Don't use; only provided for compatibility. (default: false)

sanity-check: boolean (optional)
: if true, perform some sanity checks before using the credentials (default: true)

passwordid: string (optional)
: For the server-key.pem and client-key.pem files which contain sensitive private keys, it is possible to use an encrypted version by providing the `passwordid` parameter. This provides the ID of a previously created secret object containing the password for decryption.

The members of TlsCredsProperties

Features

deprecated
: Member `loaded` is deprecated. Setting true doesn't make sense, and false is already the default.

Since

2.5

QCryptoAkCipherAlgorithm (Enum)

The supported algorithms for asymmetric encryption ciphers

Values

rsa
: RSA algorithm

Since

7.1

QCryptoAkCipherKeyType (Enum)

The type of asymmetric keys.

Values

public
Not documented

private
Not documented

Since

7.1

QCryptoRSAPaddingAlgorithm (Enum)

The padding algorithm for RSA.

Values

raw
no padding used

pkcs1
pkcs1#v1.5

Since

7.1

QCryptoAkCipherOptionsRSA (Object)

Specific parameters for RSA algorithm.

Members

hash-alg: **QCryptoHashAlgorithm**
QCryptoHashAlgorithm

padding-alg: **QCryptoRSAPaddingAlgorithm**
QCryptoRSAPaddingAlgorithm

Since

7.1

QCryptoAkCipherOptions (Object)

The options that are available for all asymmetric key algorithms when creating a new QCryptoAkCipher.

Members

alg: **QCryptoAkCipherAlgorithm**
encryption cipher algorithm

The members of QCryptoAkCipherOptionsRSA when alg is "rsa"

Since

7.1

5.12.4 Background jobs

JobType (Enum)

Type of a background job.

Values

commit
block commit job type, see "block-commit"

stream
block stream job type, see "block-stream"

mirror
drive mirror job type, see "drive-mirror"

backup
drive backup job type, see "drive-backup"

create
image creation job type, see "blockdev-create" (since 3.0)

amend
image options amend job type, see "x-blockdev-amend" (since 5.1)

snapshot-load
snapshot load job type, see "snapshot-load" (since 6.0)

snapshot-save
snapshot save job type, see "snapshot-save" (since 6.0)

snapshot-delete
snapshot delete job type, see "snapshot-delete" (since 6.0)

Since

1.7

JobStatus (Enum)

Indicates the present state of a given job in its lifetime.

Values

undefined
Erroneous, default state. Should not ever be visible.

created
The job has been created, but not yet started.

running
The job is currently running.

paused
The job is running, but paused. The pause may be requested by either the QMP user or by internal processes.

ready
The job is running, but is ready for the user to signal completion. This is used for long-running jobs like mirror that are designed to run indefinitely.

standby
The job is ready, but paused. This is nearly identical to `paused`. The job may return to `ready` or otherwise be canceled.

waiting
The job is waiting for other jobs in the transaction to converge to the waiting state. This status will likely not be visible for the last job in a transaction.

pending
The job has finished its work, but has finalization steps that it needs to make prior to completing. These changes will require manual intervention via `job-finalize` if auto-finalize was set to false. These pending changes may still fail.

aborting
The job is in the process of being aborted, and will finish with an error. The job will afterwards report that it is `concluded`. This status may not be visible to the management process.

concluded
The job has finished all work. If auto-dismiss was set to false, the job will remain in the query list until it is dismissed via `job-dismiss`.

null
The job is in the process of being dismantled. This state should not ever be visible externally.

Since

2.12

`JobVerb` (Enum)

Represents command verbs that can be applied to a job.

Values

cancel
see `job-cancel`

pause
see `job-pause`

resume
see `job-resume`

set-speed
see `block-job-set-speed`

complete
see `job-complete`

dismiss
see `job-dismiss`

finalize
see `job-finalize`

change
see `block-job-change` (since 8.2)

Since

2.12

JOB_STATUS_CHANGE (Event)

Emitted when a job transitions to a different status.

Arguments

id: string
The job identifier

status: JobStatus
The new job status

Since

3.0

job-pause (Command)

Pause an active job.

This command returns immediately after marking the active job for pausing. Pausing an already paused job is an error.

The job will pause as soon as possible, which means transitioning into the PAUSED state if it was RUNNING, or into STANDBY if it was READY. The corresponding JOB_STATUS_CHANGE event will be emitted.

Cancelling a paused job automatically resumes it.

Arguments

id: string
The job identifier.

Since

3.0

job-resume (Command)

Resume a paused job.

This command returns immediately after resuming a paused job. Resuming an already running job is an error.

Arguments

id: string
The job identifier.

Since

3.0

job-cancel (Command)

Instruct an active background job to cancel at the next opportunity. This command returns immediately after marking the active job for cancellation.

The job will cancel as soon as possible and then emit a JOB_STATUS_CHANGE event. Usually, the status will change to ABORTING, but it is possible that a job successfully completes (e.g. because it was almost done and there was no opportunity to cancel earlier than completing the job) and transitions to PENDING instead.

Arguments

id: string
The job identifier.

Since

3.0

job-complete (Command)

Manually trigger completion of an active job in the READY state.

Arguments

id: string
The job identifier.

Since

3.0

job-dismiss (Command)

Deletes a job that is in the CONCLUDED state. This command only needs to be run explicitly for jobs that don't have automatic dismiss enabled.

This command will refuse to operate on any job that has not yet reached its terminal state, JOB_STATUS_CONCLUDED. For jobs that make use of JOB_READY event, job-cancel or job-complete will still need to be used as appropriate.

Arguments

id: string
The job identifier.

Since

3.0

job-finalize (Command)

Instructs all jobs in a transaction (or a single job if it is not part of any transaction) to finalize any graph changes and do any necessary cleanup. This command requires that all involved jobs are in the PENDING state.

For jobs in a transaction, instructing one job to finalize will force ALL jobs in the transaction to finalize, so it is only necessary to instruct a single member job to finalize.

Arguments

id: string
The identifier of any job in the transaction, or of a job that is not part of any transaction.

Since

3.0

JobInfo (Object)

Information about a job.

Members

id: string
The job identifier

type: JobType
The kind of job that is being performed

status: JobStatus
Current job state/status

current-progress: int
Progress made until now. The unit is arbitrary and the value can only meaningfully be used for the ratio of `current-progress` to `total-progress`. The value is monotonically increasing.

total-progress: int
Estimated `current-progress` value at the completion of the job. This value can arbitrarily change while the job is running, in both directions.

error: string (optional)
If this field is present, the job failed; if it is still missing in the CONCLUDED state, this indicates successful completion.

The value is a human-readable error message to describe the reason for the job failure. It should not be parsed by applications.

Since

3.0

`query-jobs` (Command)

Return information about jobs.

Returns

a list with a `JobInfo` for each active job

Since

3.0

5.12.5 Block devices

Block core (VM unrelated)

`SnapshotInfo` (Object)

Members

id: string
unique snapshot id

name: string
user chosen name

vm-state-size: int
size of the VM state

date-sec: int
UTC date of the snapshot in seconds

date-nsec: int
fractional part in nano seconds to be used with date-sec

vm-clock-sec: int
VM clock relative to boot in seconds

vm-clock-nsec: int
fractional part in nano seconds to be used with vm-clock-sec

icount: int (optional)
Current instruction count. Appears when execution record/replay is enabled. Used for "time-traveling" to match the moment in the recorded execution with the snapshots. This counter may be obtained through `query-replay` command (since 5.2)

Since

1.3

ImageInfoSpecificQCow2EncryptionBase (Object)

Members

format: BlockdevQcow2EncryptionFormat
The encryption format

Since

2.10

ImageInfoSpecificQCow2Encryption (Object)

Members

The members of ImageInfoSpecificQCow2EncryptionBase
The members of QCryptoBlockInfoLUKS when format is "luks"

Since

2.10

ImageInfoSpecificQCow2 (Object)

Members

compat: string
compatibility level

data-file: string (optional)
the filename of the external data file that is stored in the image and used as a default for opening the image (since: 4.0)

data-file-raw: boolean (optional)
True if the external data file must stay valid as a standalone (read-only) raw image without looking at qcow2 metadata (since: 4.0)

extended-l2: boolean (optional)
true if the image has extended L2 entries; only valid for compat >= 1.1 (since 5.2)

lazy-refcounts: boolean (optional)
on or off; only valid for compat >= 1.1

corrupt: boolean (optional)
true if the image has been marked corrupt; only valid for compat >= 1.1 (since 2.2)

refcount-bits: int
width of a refcount entry in bits (since 2.3)

encrypt: ImageInfoSpecificQCow2Encryption (optional)
details about encryption parameters; only set if image is encrypted (since 2.10)

bitmaps: array of Qcow2BitmapInfo (optional)
A list of qcow2 bitmap details (since 4.0)

compression-type: Qcow2CompressionType
the image cluster compression method (since 5.1)

Since

1.7

ImageInfoSpecificVmdk (Object)

Members

create-type: string
The create type of VMDK image

cid: int
Content id of image

parent-cid: int
Parent VMDK image's cid

extents: array of VmdkExtentInfo
List of extent files

Since

1.7

VmdkExtentInfo (Object)

Information about a VMDK extent file

Members

filename: string
Name of the extent file

format: string
Extent type (e.g. FLAT or SPARSE)

virtual-size: int
Number of bytes covered by this extent

cluster-size: int (optional)
Cluster size in bytes (for non-flat extents)

compressed: boolean (optional)
Whether this extent contains compressed data

Since

8.0

ImageInfoSpecificRbd (Object)

Members

encryption-format: RbdImageEncryptionFormat (optional)
Image encryption format

Since

6.1

ImageInfoSpecificFile (Object)

Members

extent-size-hint: int (optional)
Extent size hint (if available)

Since

8.0

ImageInfoSpecificKind (Enum)

Values

luks
Since 2.7

rbd
Since 6.1

file
Since 8.0

qcow2
Not documented

vmdk
Not documented

Since

1.7

ImageInfoSpecificQCow2Wrapper (Object)

Members

data: ImageInfoSpecificQCow2
image information specific to QCOW2

Since

1.7

ImageInfoSpecificVmdkWrapper (Object)

Members

data: ImageInfoSpecificVmdk
image information specific to VMDK

Since

6.1

ImageInfoSpecificLUKSWrapper (Object)

Members

data: QCryptoBlockInfoLUKS
image information specific to LUKS

Since

2.7

ImageInfoSpecificRbdWrapper (Object)

Members

data: ImageInfoSpecificRbd
image information specific to RBD

Since

6.1

ImageInfoSpecificFileWrapper (Object)

Members

data: ImageInfoSpecificFile
image information specific to files

Since

8.0

ImageInfoSpecific (Object)

A discriminated record of image format specific information structures.

Members

type: ImageInfoSpecificKind
block driver name

The members of ImageInfoSpecificQCow2Wrapper when type is "qcow2"
The members of ImageInfoSpecificVmdkWrapper when type is "vmdk"
The members of ImageInfoSpecificLUKSWrapper when type is "luks"
The members of ImageInfoSpecificRbdWrapper when type is "rbd"
The members of ImageInfoSpecificFileWrapper when type is "file"

Since

1.7

BlockNodeInfo (Object)

Information about a QEMU image file

Members

filename: string
name of the image file

format: string
format of the image file

virtual-size: int
maximum capacity in bytes of the image

actual-size: int (optional)
actual size on disk in bytes of the image

dirty-flag: boolean (optional)
true if image is not cleanly closed

cluster-size: int (optional)
size of a cluster in bytes

encrypted: boolean (optional)
true if the image is encrypted

compressed: boolean (optional)
true if the image is compressed (Since 1.7)

backing-filename: string (optional)
name of the backing file

full-backing-filename: string (optional)
full path of the backing file

backing-filename-format: string (optional)
the format of the backing file

snapshots: array of SnapshotInfo (optional)
list of VM snapshots

format-specific: ImageInfoSpecific (optional)
structure supplying additional format-specific information (since 1.7)

Since

8.0

ImageInfo (Object)

Information about a QEMU image file, and potentially its backing image

Members

backing-image: ImageInfo (optional)
info of the backing image

The members of BlockNodeInfo

Since

1.3

BlockChildInfo (Object)

Information about all nodes in the block graph starting at some node, annotated with information about that node in relation to its parent.

Members

name: string
Child name of the root node in the BlockGraphInfo struct, in its role as the child of some undescribed parent node

info: BlockGraphInfo
Block graph information starting at this node

Since

8.0

BlockGraphInfo (Object)

Information about all nodes in a block (sub)graph in the form of BlockNodeInfo data. The base BlockNodeInfo struct contains the information for the (sub)graph's root node.

Members

children: array of BlockChildInfo
Array of links to this node's child nodes' information

The members of BlockNodeInfo

Since

8.0

ImageCheck (Object)

Information about a QEMU image file check

Members

filename: string
name of the image file checked

format: string
format of the image file checked

check-errors: int
number of unexpected errors occurred during check

image-end-offset: int (optional)
offset (in bytes) where the image ends, this field is present if the driver for the image format supports it

corruptions: int (optional)
number of corruptions found during the check if any

leaks: int (optional)
number of leaks found during the check if any

corruptions-fixed: int (optional)
number of corruptions fixed during the check if any

leaks-fixed: int (optional)
number of leaks fixed during the check if any

total-clusters: int (optional)
total number of clusters, this field is present if the driver for the image format supports it

allocated-clusters: int (optional)
total number of allocated clusters, this field is present if the driver for the image format supports it

fragmented-clusters: int (optional)
total number of fragmented clusters, this field is present if the driver for the image format supports it

compressed-clusters: int (optional)
total number of compressed clusters, this field is present if the driver for the image format supports it

Since

1.4

MapEntry (Object)

Mapping information from a virtual block range to a host file range

Members

start: int
virtual (guest) offset of the first byte described by this entry

length: int
the number of bytes of the mapped virtual range

data: boolean
reading the image will actually read data from a file (in particular, if `offset` is present this means that the sectors are not simply preallocated, but contain actual data in raw format)

zero: boolean
whether the virtual blocks read as zeroes

compressed: boolean
true if the data is stored compressed (since 8.2)

depth: int
number of layers (0 = top image, 1 = top image's backing file, ..., n - 1 = bottom image (where n is the number of images in the chain)) before reaching one for which the range is allocated

present: boolean
true if this layer provides the data, false if adding a backing layer could impact this region (since 6.1)

offset: int (optional)
if present, the image file stores the data for this range in raw format at the given (host) offset

filename: string (optional)
filename that is referred to by `offset`

Since

2.6

BlockdevCacheInfo (Object)

Cache mode information for a block device

Members

writeback: boolean
: true if writeback mode is enabled

direct: boolean
: true if the host page cache is bypassed (O_DIRECT)

no-flush: boolean
: true if flush requests are ignored for the device

Since

2.3

BlockDeviceInfo (Object)

Information about the backing device for a block device.

Members

file: string
: the filename of the backing device

node-name: string (optional)
: the name of the block driver node (Since 2.0)

ro: boolean
: true if the backing device was open read-only

drv: string
: the name of the block format used to open the backing device. As of 0.14 this can be: 'blkdebug', 'bochs', 'cloop', 'cow', 'dmg', 'file', 'file', 'ftp', 'ftps', 'host_cdrom', 'host_device', 'http', 'https', 'luks', 'nbd', 'parallels', 'qcow', 'qcow2', 'raw', 'vdi', 'vmdk', 'vpc', 'vvfat' 2.2: 'archipelago' added, 'cow' dropped 2.3: 'host_floppy' deprecated 2.5: 'host_floppy' dropped 2.6: 'luks' added 2.8: 'replication' added, 'tftp' dropped 2.9: 'archipelago' dropped

backing_file: string (optional)
: the name of the backing file (for copy-on-write)

backing_file_depth: int
: number of files in the backing file chain (since: 1.2)

encrypted: boolean
: true if the backing device is encrypted

detect_zeroes: BlockdevDetectZeroesOptions
detect and optimize zero writes (Since 2.1)

bps: int
total throughput limit in bytes per second is specified

bps_rd: int
read throughput limit in bytes per second is specified

bps_wr: int
write throughput limit in bytes per second is specified

iops: int
total I/O operations per second is specified

iops_rd: int
read I/O operations per second is specified

iops_wr: int
write I/O operations per second is specified

image: ImageInfo
the info of image used (since: 1.6)

bps_max: int (optional)
total throughput limit during bursts, in bytes (Since 1.7)

bps_rd_max: int (optional)
read throughput limit during bursts, in bytes (Since 1.7)

bps_wr_max: int (optional)
write throughput limit during bursts, in bytes (Since 1.7)

iops_max: int (optional)
total I/O operations per second during bursts, in bytes (Since 1.7)

iops_rd_max: int (optional)
read I/O operations per second during bursts, in bytes (Since 1.7)

iops_wr_max: int (optional)
write I/O operations per second during bursts, in bytes (Since 1.7)

bps_max_length: int (optional)
maximum length of the bps_max burst period, in seconds. (Since 2.6)

bps_rd_max_length: int (optional)
maximum length of the bps_rd_max burst period, in seconds. (Since 2.6)

bps_wr_max_length: int (optional)
maximum length of the bps_wr_max burst period, in seconds. (Since 2.6)

iops_max_length: int (optional)
maximum length of the iops burst period, in seconds. (Since 2.6)

iops_rd_max_length: int (optional)
maximum length of the iops_rd_max burst period, in seconds. (Since 2.6)

iops_wr_max_length: int (optional)
maximum length of the iops_wr_max burst period, in seconds. (Since 2.6)

iops_size: int (optional)
an I/O size in bytes (Since 1.7)

group: string (optional)
throttle group name (Since 2.4)

cache: BlockdevCacheInfo
the cache mode used for the block device (since: 2.3)

write_threshold: int
configured write threshold for the device. 0 if disabled. (Since 2.3)

dirty-bitmaps: array of BlockDirtyInfo (optional)
dirty bitmaps information (only present if node has one or more dirty bitmaps) (Since 4.2)

Since

0.14

BlockDeviceIoStatus (Enum)

An enumeration of block device I/O status.

Values

ok
The last I/O operation has succeeded

failed
The last I/O operation has failed

nospace
The last I/O operation has failed due to a no-space condition

Since

1.0

BlockDirtyInfo (Object)

Block dirty bitmap information.

Members

name: string (optional)
the name of the dirty bitmap (Since 2.4)

count: int
number of dirty bytes according to the dirty bitmap

granularity: int
granularity of the dirty bitmap in bytes (since 1.4)

recording: boolean
true if the bitmap is recording new writes from the guest. (since 4.0)

busy: boolean
true if the bitmap is in-use by some operation (NBD or jobs) and cannot be modified via QMP or used by another operation. (since 4.0)

persistent: boolean
true if the bitmap was stored on disk, is scheduled to be stored on disk, or both. (since 4.0)

inconsistent: boolean (optional)
true if this is a persistent bitmap that was improperly stored. Implies `persistent` to be true; `recording` and `busy` to be false. This bitmap cannot be used. To remove it, use `block-dirty-bitmap-remove`. (Since 4.0)

Since

1.3

Qcow2BitmapInfoFlags (Enum)

An enumeration of flags that a bitmap can report to the user.

Values

in-use
This flag is set by any process actively modifying the qcow2 file, and cleared when the updated bitmap is flushed to the qcow2 image. The presence of this flag in an offline image means that the bitmap was not saved correctly after its last usage, and may contain inconsistent data.

auto
The bitmap must reflect all changes of the virtual disk by any application that would write to this qcow2 file.

Since

4.0

Qcow2BitmapInfo (Object)

Qcow2 bitmap information.

Members

name: string
the name of the bitmap

granularity: int
granularity of the bitmap in bytes

flags: array of Qcow2BitmapInfoFlags
flags of the bitmap

Since

4.0

BlockLatencyHistogramInfo (Object)

Block latency histogram.

Members

boundaries: array of int
: list of interval boundary values in nanoseconds, all greater than zero and in ascending order. For example, the list [10, 50, 100] produces the following histogram intervals: [0, 10), [10, 50), [50, 100), [100, +inf).

bins: array of int
: list of io request counts corresponding to histogram intervals, one more element than `boundaries` has. For the example above, `bins` may be something like [3, 1, 5, 2], and corresponding histogram looks like:

```
5|           *
4|           *
3| *         *
2| *         *    *
1| *    *    *    *
 +------------------
     10   50   100
```

Since

4.0

BlockInfo (Object)

Block device information. This structure describes a virtual device and the backing device associated with it.

Members

device: string
: The device name associated with the virtual device.

qdev: string (optional)
: The qdev ID, or if no ID is assigned, the QOM path of the block device. (since 2.10)

type: string
: This field is returned only for compatibility reasons, it should not be used (always returns 'unknown')

removable: boolean
: True if the device supports removable media.

locked: boolean
: True if the guest has locked this device from having its media removed

tray_open: boolean (optional)
True if the device's tray is open (only present if it has a tray)

io-status: BlockDeviceIoStatus (optional)
`BlockDeviceIoStatus`. Only present if the device supports it and the VM is configured to stop on errors (supported device models: virtio-blk, IDE, SCSI except scsi-generic)

inserted: BlockDeviceInfo (optional)
`BlockDeviceInfo` describing the device if media is present

Since

0.14

BlockMeasureInfo (Object)

Image file size calculation information. This structure describes the size requirements for creating a new image file.

The size requirements depend on the new image file format. File size always equals virtual disk size for the 'raw' format, even for sparse POSIX files. Compact formats such as 'qcow2' represent unallocated and zero regions efficiently so file size may be smaller than virtual disk size.

The values are upper bounds that are guaranteed to fit the new image file. Subsequent modification, such as internal snapshot or further bitmap creation, may require additional space and is not covered here.

Members

required: int
Size required for a new image file, in bytes, when copying just allocated guest-visible contents.

fully-allocated: int
Image file size, in bytes, once data has been written to all sectors, when copying just guest-visible contents.

bitmaps: int (optional)
Additional size required if all the top-level bitmap metadata in the source image were to be copied to the destination, present only when source and destination both support persistent bitmaps. (since 5.1)

Since

2.10

query-block (Command)

Get a list of BlockInfo for all virtual block devices.

Returns

a list of `BlockInfo` describing each virtual block device. Filter nodes that were created implicitly are skipped over.

Since

0.14

Example

```
-> { "execute": "query-block" }
<- {
      "return":[
         {
            "io-status": "ok",
            "device":"ide0-hd0",
            "locked":false,
            "removable":false,
            "inserted":{
               "ro":false,
               "drv":"qcow2",
               "encrypted":false,
               "file":"disks/test.qcow2",
               "backing_file_depth":1,
               "bps":1000000,
               "bps_rd":0,
               "bps_wr":0,
               "iops":1000000,
               "iops_rd":0,
               "iops_wr":0,
               "bps_max": 8000000,
               "bps_rd_max": 0,
               "bps_wr_max": 0,
               "iops_max": 0,
               "iops_rd_max": 0,
               "iops_wr_max": 0,
               "iops_size": 0,
               "detect_zeroes": "on",
               "write_threshold": 0,
               "image":{
                  "filename":"disks/test.qcow2",
                  "format":"qcow2",
                  "virtual-size":2048000,
                  "backing_file":"base.qcow2",
                  "full-backing-filename":"disks/base.qcow2",
                  "backing-filename-format":"qcow2",
                  "snapshots":[
                     {
                        "id": "1",
                        "name": "snapshot1",
                        "vm-state-size": 0,
```

(continues on next page)

(continued from previous page)


```
                        "date-sec": 10000200,
                        "date-nsec": 12,
                        "vm-clock-sec": 206,
                        "vm-clock-nsec": 30
                     }
                  ],
                  "backing-image":{
                      "filename":"disks/base.qcow2",
                      "format":"qcow2",
                      "virtual-size":2048000
                  }
               }
            },
            "qdev": "ide_disk",
            "type":"unknown"
         },
         {
            "io-status": "ok",
            "device":"ide1-cd0",
            "locked":false,
            "removable":true,
            "qdev": "/machine/unattached/device[23]",
            "tray_open": false,
            "type":"unknown"
         },
         {
            "device":"floppy0",
            "locked":false,
            "removable":true,
            "qdev": "/machine/unattached/device[20]",
            "type":"unknown"
         },
         {
            "device":"sd0",
            "locked":false,
            "removable":true,
            "type":"unknown"
         }
      ]
   }
```

BlockDeviceTimedStats (Object)

Statistics of a block device during a given interval of time.

Members

interval_length: int
Interval used for calculating the statistics, in seconds.

min_rd_latency_ns: int
Minimum latency of read operations in the defined interval, in nanoseconds.

min_wr_latency_ns: int
Minimum latency of write operations in the defined interval, in nanoseconds.

min_zone_append_latency_ns: int
Minimum latency of zone append operations in the defined interval, in nanoseconds (since 8.1)

min_flush_latency_ns: int
Minimum latency of flush operations in the defined interval, in nanoseconds.

max_rd_latency_ns: int
Maximum latency of read operations in the defined interval, in nanoseconds.

max_wr_latency_ns: int
Maximum latency of write operations in the defined interval, in nanoseconds.

max_zone_append_latency_ns: int
Maximum latency of zone append operations in the defined interval, in nanoseconds (since 8.1)

max_flush_latency_ns: int
Maximum latency of flush operations in the defined interval, in nanoseconds.

avg_rd_latency_ns: int
Average latency of read operations in the defined interval, in nanoseconds.

avg_wr_latency_ns: int
Average latency of write operations in the defined interval, in nanoseconds.

avg_zone_append_latency_ns: int
Average latency of zone append operations in the defined interval, in nanoseconds (since 8.1)

avg_flush_latency_ns: int
Average latency of flush operations in the defined interval, in nanoseconds.

avg_rd_queue_depth: number
Average number of pending read operations in the defined interval.

avg_wr_queue_depth: number
Average number of pending write operations in the defined interval.

avg_zone_append_queue_depth: number
Average number of pending zone append operations in the defined interval (since 8.1).

Since

2.5

BlockDeviceStats (Object)

Statistics of a virtual block device or a block backing device.

Members

rd_bytes: int
The number of bytes read by the device.

wr_bytes: int
The number of bytes written by the device.

zone_append_bytes: int
The number of bytes appended by the zoned devices (since 8.1)

unmap_bytes: int
The number of bytes unmapped by the device (Since 4.2)

rd_operations: int
The number of read operations performed by the device.

wr_operations: int
The number of write operations performed by the device.

zone_append_operations: int
The number of zone append operations performed by the zoned devices (since 8.1)

flush_operations: int
The number of cache flush operations performed by the device (since 0.15)

unmap_operations: int
The number of unmap operations performed by the device (Since 4.2)

rd_total_time_ns: int
Total time spent on reads in nanoseconds (since 0.15).

wr_total_time_ns: int
Total time spent on writes in nanoseconds (since 0.15).

zone_append_total_time_ns: int
Total time spent on zone append writes in nanoseconds (since 8.1)

flush_total_time_ns: int
Total time spent on cache flushes in nanoseconds (since 0.15).

unmap_total_time_ns: int
Total time spent on unmap operations in nanoseconds (Since 4.2)

wr_highest_offset: int
The offset after the greatest byte written to the device. The intended use of this information is for growable sparse files (like qcow2) that are used on top of a physical device.

rd_merged: int
Number of read requests that have been merged into another request (Since 2.3).

wr_merged: int
Number of write requests that have been merged into another request (Since 2.3).

zone_append_merged: int
Number of zone append requests that have been merged into another request (since 8.1)

unmap_merged: int
Number of unmap requests that have been merged into another request (Since 4.2)

idle_time_ns: int (optional)
Time since the last I/O operation, in nanoseconds. If the field is absent it means that there haven't been any operations yet (Since 2.5).

failed_rd_operations: int
The number of failed read operations performed by the device (Since 2.5)

failed_wr_operations: int
The number of failed write operations performed by the device (Since 2.5)

failed_zone_append_operations: int
The number of failed zone append write operations performed by the zoned devices (since 8.1)

failed_flush_operations: int
The number of failed flush operations performed by the device (Since 2.5)

failed_unmap_operations: int
The number of failed unmap operations performed by the device (Since 4.2)

invalid_rd_operations: int
The number of invalid read operations performed by the device (Since 2.5)

invalid_wr_operations: int
The number of invalid write operations performed by the device (Since 2.5)

invalid_zone_append_operations: int
The number of invalid zone append operations performed by the zoned device (since 8.1)

invalid_flush_operations: int
The number of invalid flush operations performed by the device (Since 2.5)

invalid_unmap_operations: int
The number of invalid unmap operations performed by the device (Since 4.2)

account_invalid: boolean
Whether invalid operations are included in the last access statistics (Since 2.5)

account_failed: boolean
Whether failed operations are included in the latency and last access statistics (Since 2.5)

timed_stats: array of BlockDeviceTimedStats
Statistics specific to the set of previously defined intervals of time (Since 2.5)

rd_latency_histogram: BlockLatencyHistogramInfo (optional)
`BlockLatencyHistogramInfo`. (Since 4.0)

wr_latency_histogram: BlockLatencyHistogramInfo (optional)
`BlockLatencyHistogramInfo`. (Since 4.0)

zone_append_latency_histogram: BlockLatencyHistogramInfo (optional)
`BlockLatencyHistogramInfo`. (since 8.1)

flush_latency_histogram: BlockLatencyHistogramInfo (optional)
`BlockLatencyHistogramInfo`. (Since 4.0)

Since

0.14

BlockStatsSpecificFile (Object)

File driver statistics

Members

discard-nb-ok: int
The number of successful discard operations performed by the driver.

discard-nb-failed: int
The number of failed discard operations performed by the driver.

discard-bytes-ok: int
The number of bytes discarded by the driver.

Since

4.2

BlockStatsSpecificNvme (Object)

NVMe driver statistics

Members

completion-errors: int
The number of completion errors.

aligned-accesses: int
The number of aligned accesses performed by the driver.

unaligned-accesses: int
The number of unaligned accesses performed by the driver.

Since

5.2

BlockStatsSpecific (Object)

Block driver specific statistics

Members

driver: BlockdevDriver
block driver name

The members of BlockStatsSpecificFile when driver is "file"
The members of BlockStatsSpecificFile when driver is "host_device" (If: HAVE_HOST_BLOCK_DEVICE)
The members of BlockStatsSpecificNvme when driver is "nvme"

Since

4.2

BlockStats (Object)

Statistics of a virtual block device or a block backing device.

Members

device: string (optional)
If the stats are for a virtual block device, the name corresponding to the virtual block device.

node-name: string (optional)
The node name of the device. (Since 2.3)

qdev: string (optional)
The qdev ID, or if no ID is assigned, the QOM path of the block device. (since 3.0)

stats: BlockDeviceStats
A BlockDeviceStats for the device.

driver-specific: BlockStatsSpecific (optional)
Optional driver-specific stats. (Since 4.2)

parent: BlockStats (optional)
This describes the file block device if it has one. Contains recursively the statistics of the underlying protocol (e.g. the host file for a qcow2 image). If there is no underlying protocol, this field is omitted

backing: BlockStats (optional)
This describes the backing block device if it has one. (Since 2.0)

Since

0.14

query-blockstats (Command)

Query the BlockStats for all virtual block devices.

Arguments

query-nodes: boolean (optional)
: If true, the command will query all the block nodes that have a node name, in a list which will include "parent" information, but not "backing". If false or omitted, the behavior is as before - query all the device backends, recursively including their "parent" and "backing". Filter nodes that were created implicitly are skipped over in this mode. (Since 2.3)

Returns

A list of BlockStats for each virtual block devices.

Since

0.14

Example

```
-> { "execute": "query-blockstats" }
<- {
      "return":[
         {
            "device":"ide0-hd0",
            "parent":{
               "stats":{
                  "wr_highest_offset":3686448128,
                  "wr_bytes":9786368,
                  "wr_operations":751,
                  "rd_bytes":122567168,
                  "rd_operations":36772
                  "wr_total_times_ns":313253456
                  "rd_total_times_ns":3465673657
                  "flush_total_times_ns":49653
                  "flush_operations":61,
                  "rd_merged":0,
                  "wr_merged":0,
                  "idle_time_ns":2953431879,
                  "account_invalid":true,
                  "account_failed":false
               }
```

(continues on next page)

(continued from previous page)

```
            },
            "stats":{
               "wr_highest_offset":2821110784,
               "wr_bytes":9786368,
               "wr_operations":692,
               "rd_bytes":122739200,
               "rd_operations":36604
               "flush_operations":51,
               "wr_total_times_ns":313253456
               "rd_total_times_ns":3465673657
               "flush_total_times_ns":49653,
               "rd_merged":0,
               "wr_merged":0,
               "idle_time_ns":2953431879,
               "account_invalid":true,
               "account_failed":false
            },
            "qdev": "/machine/unattached/device[23]"
         },
         {
            "device":"ide1-cd0",
            "stats":{
               "wr_highest_offset":0,
               "wr_bytes":0,
               "wr_operations":0,
               "rd_bytes":0,
               "rd_operations":0
               "flush_operations":0,
               "wr_total_times_ns":0
               "rd_total_times_ns":0
               "flush_total_times_ns":0,
               "rd_merged":0,
               "wr_merged":0,
               "account_invalid":false,
               "account_failed":false
            },
            "qdev": "/machine/unattached/device[24]"
         },
         {
            "device":"floppy0",
            "stats":{
               "wr_highest_offset":0,
               "wr_bytes":0,
               "wr_operations":0,
               "rd_bytes":0,
               "rd_operations":0
               "flush_operations":0,
               "wr_total_times_ns":0
               "rd_total_times_ns":0
               "flush_total_times_ns":0,
               "rd_merged":0,
               "wr_merged":0,
```

(continues on next page)

(continued from previous page)

```
                "account_invalid":false,
                "account_failed":false
            },
            "qdev": "/machine/unattached/device[16]"
        },
        {
            "device":"sd0",
            "stats":{
                "wr_highest_offset":0,
                "wr_bytes":0,
                "wr_operations":0,
                "rd_bytes":0,
                "rd_operations":0
                "flush_operations":0,
                "wr_total_times_ns":0
                "rd_total_times_ns":0
                "flush_total_times_ns":0,
                "rd_merged":0,
                "wr_merged":0,
                "account_invalid":false,
                "account_failed":false
            }
        }
    ]
}
```

BlockdevOnError (Enum)

An enumeration of possible behaviors for errors on I/O operations. The exact meaning depends on whether the I/O was initiated by a guest or by a block job

Values

report
: for guest operations, report the error to the guest; for jobs, cancel the job

ignore
: ignore the error, only report a QMP event (BLOCK_IO_ERROR or BLOCK_JOB_ERROR). The backup, mirror and commit block jobs retry the failing request later and may still complete successfully. The stream block job continues to stream and will complete with an error.

enospc
: same as `stop` on ENOSPC, same as `report` otherwise.

stop
: for guest operations, stop the virtual machine; for jobs, pause the job

auto
: inherit the error handling policy of the backend (since: 2.7)

Since

1.3

MirrorSyncMode (Enum)

An enumeration of possible behaviors for the initial synchronization phase of storage mirroring.

Values

top
copies data in the topmost image to the destination

full
copies data from all images to the destination

none
only copy data written from now on

incremental
only copy data described by the dirty bitmap. (since: 2.4)

bitmap
only copy data described by the dirty bitmap. (since: 4.2) Behavior on completion is determined by the BitmapSyncMode.

Since

1.3

BitmapSyncMode (Enum)

An enumeration of possible behaviors for the synchronization of a bitmap when used for data copy operations.

Values

on-success
The bitmap is only synced when the operation is successful. This is the behavior always used for 'INCREMENTAL' backups.

never
The bitmap is never synchronized with the operation, and is treated solely as a read-only manifest of blocks to copy.

always
The bitmap is always synchronized with the operation, regardless of whether or not the operation was successful.

Since

4.2

MirrorCopyMode (Enum)

An enumeration whose values tell the mirror block job when to trigger writes to the target.

Values

background
: copy data in background only.

write-blocking
: when data is written to the source, write it (synchronously) to the target as well. In addition, data is copied in background just like in `background` mode.

Since

3.0

BlockJobInfoMirror (Object)

Information specific to mirror block jobs.

Members

actively-synced: boolean
: Whether the source is actively synced to the target, i.e. same data and new writes are done synchronously to both.

Since

8.2

BlockJobInfo (Object)

Information about a long-running block device operation.

Members

type: JobType
the job type ('stream' for image streaming)

device: string
The job identifier. Originally the device name but other values are allowed since QEMU 2.7

len: int
Estimated `offset` value at the completion of the job. This value can arbitrarily change while the job is running, in both directions.

offset: int
Progress made until now. The unit is arbitrary and the value can only meaningfully be used for the ratio of `offset` to `len`. The value is monotonically increasing.

busy: boolean
false if the job is known to be in a quiescent state, with no pending I/O. (Since 1.3)

paused: boolean
whether the job is paused or, if `busy` is true, will pause itself as soon as possible. (Since 1.3)

speed: int
the rate limit, bytes per second

io-status: BlockDeviceIoStatus
the status of the job (since 1.3)

ready: boolean
true if the job may be completed (since 2.2)

status: JobStatus
Current job state/status (since 2.12)

auto-finalize: boolean
Job will finalize itself when PENDING, moving to the CONCLUDED state. (since 2.12)

auto-dismiss: boolean
Job will dismiss itself when CONCLUDED, moving to the NULL state and disappearing from the query list. (since 2.12)

error: string (optional)
Error information if the job did not complete successfully. Not set if the job completed successfully. (since 2.12.1)

The members of `BlockJobInfoMirror` when `type` is `"mirror"`

Since

1.1

query-block-jobs (Command)

Return information about long-running block device operations.

Returns

a list of BlockJobInfo for each active block job

Since

1.1

block_resize (Command)

Resize a block image while a guest is running.

Either device or node-name must be set but not both.

Arguments

device: string (optional)
: the name of the device to get the image resized

node-name: string (optional)
: graph node name to get the image resized (Since 2.0)

size: int
: new image size in bytes

Errors

- If device is not a valid block device, DeviceNotFound

Since

0.14

Example

```
-> { "execute": "block_resize",
     "arguments": { "device": "scratch", "size": 1073741824 } }
<- { "return": {} }
```

NewImageMode (Enum)

An enumeration that tells QEMU how to set the backing file path in a new image file.

Values

existing
QEMU should look for an existing image file.

absolute-paths
QEMU should create a new image with absolute paths for the backing file. If there is no backing file available, the new image will not be backed either.

Since

1.1

BlockdevSnapshotSync (Object)

Either `device` or `node-name` must be set but not both.

Members

device: string (optional)
the name of the device to take a snapshot of.

node-name: string (optional)
graph node name to generate the snapshot from (Since 2.0)

snapshot-file: string
the target of the new overlay image. If the file exists, or if it is a device, the overlay will be created in the existing file/device. Otherwise, a new file will be created.

snapshot-node-name: string (optional)
the graph node name of the new image (Since 2.0)

format: string (optional)
the format of the overlay image, default is 'qcow2'.

mode: NewImageMode (optional)
whether and how QEMU should create a new image, default is 'absolute-paths'.

BlockdevSnapshot (Object)

Members

node: string
device or node name that will have a snapshot taken.

overlay: string
reference to the existing block device that will become the overlay of `node`, as part of taking the snapshot. It must not have a current backing file (this can be achieved by passing "backing": null to blockdev-add).

Since

2.5

BackupPerf (Object)

Optional parameters for backup. These parameters don't affect functionality, but may significantly affect performance.

Members

use-copy-range: boolean (optional)
Use copy offloading. Default false.

max-workers: int (optional)
Maximum number of parallel requests for the sustained background copying process. Doesn't influence copy-before-write operations. Default 64.

max-chunk: int (optional)
Maximum request length for the sustained background copying process. Doesn't influence copy-before-write operations. 0 means unlimited. If max-chunk is non-zero then it should not be less than job cluster size which is calculated as maximum of target image cluster size and 64k. Default 0.

Since

6.0

BackupCommon (Object)

Members

job-id: string (optional)
identifier for the newly-created block job. If omitted, the device name will be used. (Since 2.7)

device: string
the device name or node-name of a root node which should be copied.

sync: MirrorSyncMode
what parts of the disk image should be copied to the destination (all the disk, only the sectors allocated in the topmost image, from a dirty bitmap, or only new I/O).

speed: int (optional)
the maximum speed, in bytes per second. The default is 0, for unlimited.

bitmap: string (optional)
The name of a dirty bitmap to use. Must be present if sync is "bitmap" or "incremental". Can be present if sync is "full" or "top". Must not be present otherwise. (Since 2.4 (drive-backup), 3.1 (blockdev-backup))

bitmap-mode: BitmapSyncMode (optional)
Specifies the type of data the bitmap should contain after the operation concludes. Must be present if a bitmap was provided, Must NOT be present otherwise. (Since 4.2)

compress: boolean (optional)
true to compress data, if the target format supports it. (default: false) (since 2.8)

on-source-error: BlockdevOnError (optional)
the action to take on an error on the source, default 'report'. 'stop' and 'enospc' can only be used if the block device supports io-status (see BlockInfo).

on-target-error: BlockdevOnError (optional)
the action to take on an error on the target, default 'report' (no limitations, since this applies to a different block device than `device`).

auto-finalize: boolean (optional)
When false, this job will wait in a PENDING state after it has finished its work, waiting for `block-job-finalize` before making any block graph changes. When true, this job will automatically perform its abort or commit actions. Defaults to true. (Since 2.12)

auto-dismiss: boolean (optional)
When false, this job will wait in a CONCLUDED state after it has completely ceased all work, and awaits `block-job-dismiss`. When true, this job will automatically disappear from the query list without user intervention. Defaults to true. (Since 2.12)

filter-node-name: string (optional)
the node name that should be assigned to the filter driver that the backup job inserts into the graph above node specified by `drive`. If this option is not given, a node name is autogenerated. (Since: 4.2)

x-perf: BackupPerf (optional)
Performance options. (Since 6.0)

Features

unstable
Member `x-perf` is experimental.

Note

`on-source-error` and `on-target-error` only affect background I/O. If an error occurs during a guest write request, the device's rerror/werror actions will be used.

Since

4.2

DriveBackup (Object)

Members

target: string
the target of the new image. If the file exists, or if it is a device, the existing file/device will be used as the new destination. If it does not exist, a new file will be created.

format: string (optional)
the format of the new destination, default is to probe if `mode` is 'existing', else the format of the source

mode: NewImageMode (optional)
whether and how QEMU should create a new image, default is 'absolute-paths'.

The members of BackupCommon

Since

1.6

BlockdevBackup (Object)

Members

target: string
: the device name or node-name of the backup target node.

The members of BackupCommon

Since

2.3

blockdev-snapshot-sync (Command)

Takes a synchronous snapshot of a block device.

For the arguments, see the documentation of BlockdevSnapshotSync.

Errors

- If `device` is not a valid block device, DeviceNotFound

Since

0.14

Example

```
-> { "execute": "blockdev-snapshot-sync",
     "arguments": { "device": "ide-hd0",
                    "snapshot-file":
                    "/some/place/my-image",
                    "format": "qcow2" } }
<- { "return": {} }
```

`blockdev-snapshot` (Command)

Takes a snapshot of a block device.

Take a snapshot, by installing 'node' as the backing image of 'overlay'. Additionally, if 'node' is associated with a block device, the block device changes to using 'overlay' as its new active image.

For the arguments, see the documentation of BlockdevSnapshot.

Features

`allow-write-only-overlay`
: If present, the check whether this operation is safe was relaxed so that it can be used to change backing file of a destination of a blockdev-mirror. (since 5.0)

Since

2.5

Example

```
-> { "execute": "blockdev-add",
     "arguments": { "driver": "qcow2",
                    "node-name": "node1534",
                    "file": { "driver": "file",
                              "filename": "hd1.qcow2" },
                    "backing": null } }

<- { "return": {} }

-> { "execute": "blockdev-snapshot",
     "arguments": { "node": "ide-hd0",
                    "overlay": "node1534" } }
<- { "return": {} }
```

`change-backing-file` (Command)

Change the backing file in the image file metadata. This does not cause QEMU to reopen the image file to reparse the backing filename (it may, however, perform a reopen to change permissions from r/o -> r/w -> r/o, if needed). The new backing file string is written into the image file metadata, and the QEMU internal strings are updated.

Arguments

image-node-name: string
The name of the block driver state node of the image to modify. The "device" argument is used to verify "image-node-name" is in the chain described by "device".

device: string
The device name or node-name of the root node that owns image-node-name.

backing-file: string
The string to write as the backing file. This string is not validated, so care should be taken when specifying the string or the image chain may not be able to be reopened again.

Errors

- If "device" does not exist or cannot be determined, DeviceNotFound

Since

2.1

block-commit (Command)

Live commit of data from overlay image nodes into backing nodes - i.e., writes data between 'top' and 'base' into 'base'.

If top == base, that is an error. If top has no overlays on top of it, or if it is in use by a writer, the job will not be completed by itself. The user needs to complete the job with the block-job-complete command after getting the ready event. (Since 2.0)

If the base image is smaller than top, then the base image will be resized to be the same size as top. If top is smaller than the base image, the base will not be truncated. If you want the base image size to match the size of the smaller top, you can safely truncate it yourself once the commit operation successfully completes.

Arguments

job-id: string (optional)
identifier for the newly-created block job. If omitted, the device name will be used. (Since 2.7)

device: string
the device name or node-name of a root node

base-node: string (optional)
The node name of the backing image to write data into. If not specified, this is the deepest backing image. (since: 3.1)

base: string (optional)
Same as base-node, except that it is a file name rather than a node name. This must be the exact filename string that was used to open the node; other strings, even if addressing the same file, are not accepted

top-node: string (optional)
The node name of the backing image within the image chain which contains the topmost data to be committed down. If not specified, this is the active layer. (since: 3.1)

top: string (optional)
Same as top-node, except that it is a file name rather than a node name. This must be the exact filename string that was used to open the node; other strings, even if addressing the same file, are not accepted

backing-file: string (optional)
The backing file string to write into the overlay image of 'top'. If 'top' does not have an overlay image, or if 'top' is in use by a writer, specifying a backing file string is an error.

This filename is not validated. If a pathname string is such that it cannot be resolved by QEMU, that means that subsequent QMP or HMP commands must use node-names for the image in question, as filename lookup methods will fail.

If not specified, QEMU will automatically determine the backing file string to use, or error out if there is no obvious choice. Care should be taken when specifying the string, to specify a valid filename or protocol. (Since 2.1)

backing-mask-protocol: boolean (optional)
If true, replace any protocol mentioned in the 'backing file format' with 'raw', rather than storing the protocol name as the backing format. Can be used even when no image header will be updated (default false; since 9.0).

speed: int (optional)
the maximum speed, in bytes per second

on-error: BlockdevOnError (optional)
the action to take on an error. 'ignore' means that the request should be retried. (default: report; Since: 5.0)

filter-node-name: string (optional)
the node name that should be assigned to the filter driver that the commit job inserts into the graph above top. If this option is not given, a node name is autogenerated. (Since: 2.9)

auto-finalize: boolean (optional)
When false, this job will wait in a PENDING state after it has finished its work, waiting for block-job-finalize before making any block graph changes. When true, this job will automatically perform its abort or commit actions. Defaults to true. (Since 3.1)

auto-dismiss: boolean (optional)
When false, this job will wait in a CONCLUDED state after it has completely ceased all work, and awaits block-job-dismiss. When true, this job will automatically disappear from the query list without user intervention. Defaults to true. (Since 3.1)

Features

deprecated
Members base and top are deprecated. Use base-node and top-node instead.

Errors

- If device does not exist, DeviceNotFound
- Any other error returns a GenericError.

Since

1.3

Example

```
-> { "execute": "block-commit",
     "arguments": { "device": "virtio0",
                    "top": "/tmp/snap1.qcow2" } }
<- { "return": {} }
```

drive-backup (Command)

Start a point-in-time copy of a block device to a new destination. The status of ongoing drive-backup operations can be checked with query-block-jobs where the BlockJobInfo.type field has the value 'backup'. The operation can be stopped before it has completed using the block-job-cancel command.

Arguments

The members of DriveBackup

Features

deprecated
: This command is deprecated. Use `blockdev-backup` instead.

Errors

- If `device` is not a valid block device, GenericError

Since

1.6

Example

```
-> { "execute": "drive-backup",
     "arguments": { "device": "drive0",
                    "sync": "full",
                    "target": "backup.img" } }
<- { "return": {} }
```

blockdev-backup (Command)

Start a point-in-time copy of a block device to a new destination. The status of ongoing blockdev-backup operations can be checked with query-block-jobs where the BlockJobInfo.type field has the value 'backup'. The operation can be stopped before it has completed using the block-job-cancel command.

Arguments

The members of BlockdevBackup

Errors

- If device is not a valid block device, DeviceNotFound

Since

2.3

Example

```
-> { "execute": "blockdev-backup",
     "arguments": { "device": "src-id",
                    "sync": "full",
                    "target": "tgt-id" } }
<- { "return": {} }
```

query-named-block-nodes (Command)

Get the named block driver list

Arguments

flat: boolean (optional)
Omit the nested data about backing image ("backing-image" key) if true. Default is false (Since 5.0)

Returns

the list of BlockDeviceInfo

Since

2.0

Example

```
-> { "execute": "query-named-block-nodes" }
<- { "return": [ { "ro":false,
                   "drv":"qcow2",
                   "encrypted":false,
                   "file":"disks/test.qcow2",
                   "node-name": "my-node",
                   "backing_file_depth":1,
                   "detect_zeroes":"off",
                   "bps":1000000,
                   "bps_rd":0,
                   "bps_wr":0,
                   "iops":1000000,
                   "iops_rd":0,
                   "iops_wr":0,
                   "bps_max": 8000000,
                   "bps_rd_max": 0,
                   "bps_wr_max": 0,
                   "iops_max": 0,
                   "iops_rd_max": 0,
                   "iops_wr_max": 0,
                   "iops_size": 0,
                   "write_threshold": 0,
                   "image":{
                      "filename":"disks/test.qcow2",
                      "format":"qcow2",
                      "virtual-size":2048000,
                      "backing_file":"base.qcow2",
                      "full-backing-filename":"disks/base.qcow2",
                      "backing-filename-format":"qcow2",
                      "snapshots":[
                         {
                            "id": "1",
                            "name": "snapshot1",
                            "vm-state-size": 0,
                            "date-sec": 10000200,
                            "date-nsec": 12,
                            "vm-clock-sec": 206,
                            "vm-clock-nsec": 30
                         }
                      ],
                      "backing-image":{
                          "filename":"disks/base.qcow2",
                          "format":"qcow2",
                          "virtual-size":2048000
                      }
                   } } ] }
```

XDbgBlockGraphNodeType (Enum)

Values

block-backend
corresponds to BlockBackend

block-job
corresponds to BlockJob

block-driver
corresponds to BlockDriverState

Since

4.0

XDbgBlockGraphNode (Object)

Members

id: int
Block graph node identifier. This `id` is generated only for x-debug-query-block-graph and does not relate to any other identifiers in Qemu.

type: XDbgBlockGraphNodeType
Type of graph node. Can be one of block-backend, block-job or block-driver-state.

name: string
Human readable name of the node. Corresponds to node-name for block-driver-state nodes; is not guaranteed to be unique in the whole graph (with block-jobs and block-backends).

Since

4.0

BlockPermission (Enum)

Enum of base block permissions.

Values

consistent-read
A user that has the "permission" of consistent reads is guaranteed that their view of the contents of the block device is complete and self-consistent, representing the contents of a disk at a specific point. For most block devices (including their backing files) this is true, but the property cannot be maintained in a few situations like for intermediate nodes of a commit block job.

write
This permission is required to change the visible disk contents.

write-unchanged
: This permission (which is weaker than BLK_PERM_WRITE) is both enough and required for writes to the block node when the caller promises that the visible disk content doesn't change. As the BLK_PERM_WRITE permission is strictly stronger, either is sufficient to perform an unchanging write.

resize
: This permission is required to change the size of a block node.

Since

4.0

XDbgBlockGraphEdge (Object)

Block Graph edge description for x-debug-query-block-graph.

Members

parent: int
: parent id

child: int
: child id

name: string
: name of the relation (examples are 'file' and 'backing')

perm: array of BlockPermission
: granted permissions for the parent operating on the child

shared-perm: array of BlockPermission
: permissions that can still be granted to other users of the child while it is still attached to this parent

Since

4.0

XDbgBlockGraph (Object)

Block Graph - list of nodes and list of edges.

Members

nodes: array of XDbgBlockGraphNode
: Not documented

edges: array of XDbgBlockGraphEdge
: Not documented

Since

4.0

x-debug-query-block-graph (Command)

Get the block graph.

Features

unstable
: This command is meant for debugging.

Since

4.0

drive-mirror (Command)

Start mirroring a block device's writes to a new destination. target specifies the target of the new image. If the file exists, or if it is a device, it will be used as the new destination for writes. If it does not exist, a new file will be created. `format` specifies the format of the mirror image, default is to probe if mode='existing', else the format of the source.

Arguments

The members of `DriveMirror`

Errors

- If `device` is not a valid block device, GenericError

Since

1.3

Example

```
-> { "execute": "drive-mirror",
     "arguments": { "device": "ide-hd0",
                    "target": "/some/place/my-image",
                    "sync": "full",
                    "format": "qcow2" } }
<- { "return": {} }
```

`DriveMirror` (Object)

A set of parameters describing drive mirror setup.

Members

`job-id`: `string` (optional)
identifier for the newly-created block job. If omitted, the device name will be used. (Since 2.7)

`device`: `string`
the device name or node-name of a root node whose writes should be mirrored.

`target`: `string`
the target of the new image. If the file exists, or if it is a device, the existing file/device will be used as the new destination. If it does not exist, a new file will be created.

`format`: `string` (optional)
the format of the new destination, default is to probe if `mode` is 'existing', else the format of the source

`node-name`: `string` (optional)
the new block driver state node name in the graph (Since 2.1)

`replaces`: `string` (optional)
with sync=full graph node name to be replaced by the new image when a whole image copy is done. This can be used to repair broken Quorum files. By default, `device` is replaced, although implicitly created filters on it are kept. (Since 2.1)

`mode`: `NewImageMode` (optional)
whether and how QEMU should create a new image, default is 'absolute-paths'.

`speed`: `int` (optional)
the maximum speed, in bytes per second

`sync`: `MirrorSyncMode`
what parts of the disk image should be copied to the destination (all the disk, only the sectors allocated in the topmost image, or only new I/O).

`granularity`: `int` (optional)
granularity of the dirty bitmap, default is 64K if the image format doesn't have clusters, 4K if the clusters are smaller than that, else the cluster size. Must be a power of 2 between 512 and 64M (since 1.4).

`buf-size`: `int` (optional)
maximum amount of data in flight from source to target (since 1.4).

`on-source-error`: `BlockdevOnError` (optional)
the action to take on an error on the source, default 'report'. 'stop' and 'enospc' can only be used if the block device supports io-status (see BlockInfo).

`on-target-error`: `BlockdevOnError` (optional)
the action to take on an error on the target, default 'report' (no limitations, since this applies to a different block device than `device`).

`unmap`: `boolean` (optional)
Whether to try to unmap target sectors where source has only zero. If true, and target unallocated sectors will read as zero, target image sectors will be unmapped; otherwise, zeroes will be written. Both will result in identical contents. Default is true. (Since 2.4)

`copy-mode`: `MirrorCopyMode` (optional)
when to copy data to the destination; defaults to 'background' (Since: 3.0)

auto-finalize: boolean (optional)
When false, this job will wait in a PENDING state after it has finished its work, waiting for `block-job-finalize` before making any block graph changes. When true, this job will automatically perform its abort or commit actions. Defaults to true. (Since 3.1)

auto-dismiss: boolean (optional)
When false, this job will wait in a CONCLUDED state after it has completely ceased all work, and awaits `block-job-dismiss`. When true, this job will automatically disappear from the query list without user intervention. Defaults to true. (Since 3.1)

Since

1.3

BlockDirtyBitmap (Object)

Members

node: string
name of device/node which the bitmap is tracking

name: string
name of the dirty bitmap

Since

2.4

BlockDirtyBitmapAdd (Object)

Members

node: string
name of device/node which the bitmap is tracking

name: string
name of the dirty bitmap (must be less than 1024 bytes)

granularity: int (optional)
the bitmap granularity, default is 64k for block-dirty-bitmap-add

persistent: boolean (optional)
the bitmap is persistent, i.e. it will be saved to the corresponding block device image file on its close. For now only Qcow2 disks support persistent bitmaps. Default is false for block-dirty-bitmap-add. (Since: 2.10)

disabled: boolean (optional)
the bitmap is created in the disabled state, which means that it will not track drive changes. The bitmap may be enabled with block-dirty-bitmap-enable. Default is false. (Since: 4.0)

Since

2.4

BlockDirtyBitmapOrStr (Alternate)

Members

local: string
name of the bitmap, attached to the same node as target bitmap.

external: BlockDirtyBitmap
bitmap with specified node

Since

4.1

BlockDirtyBitmapMerge (Object)

Members

node: string
name of device/node which the `target` bitmap is tracking

target: string
name of the destination dirty bitmap

bitmaps: array of BlockDirtyBitmapOrStr
name(s) of the source dirty bitmap(s) at `node` and/or fully specified BlockDirtyBitmap elements. The latter are supported since 4.1.

Since

4.0

block-dirty-bitmap-add (Command)

Create a dirty bitmap with a name on the node, and start tracking the writes.

Errors

- If node is not a valid block device or node, DeviceNotFound
- If name is already taken, GenericError with an explanation

Since

2.4

Example

```
-> { "execute": "block-dirty-bitmap-add",
     "arguments": { "node": "drive0", "name": "bitmap0" } }
<- { "return": {} }
```

block-dirty-bitmap-remove (Command)

Stop write tracking and remove the dirty bitmap that was created with block-dirty-bitmap-add. If the bitmap is persistent, remove it from its storage too.

Errors

- If node is not a valid block device or node, DeviceNotFound
- If name is not found, GenericError with an explanation
- if name is frozen by an operation, GenericError

Since

2.4

Example

```
-> { "execute": "block-dirty-bitmap-remove",
     "arguments": { "node": "drive0", "name": "bitmap0" } }
<- { "return": {} }
```

block-dirty-bitmap-clear (Command)

Clear (reset) a dirty bitmap on the device, so that an incremental backup from this point in time forward will only backup clusters modified after this clear operation.

Errors

- If node is not a valid block device, DeviceNotFound
- If name is not found, GenericError with an explanation

Since

2.4

Example

```
-> { "execute": "block-dirty-bitmap-clear",
     "arguments": { "node": "drive0", "name": "bitmap0" } }
<- { "return": {} }
```

block-dirty-bitmap-enable (Command)

Enables a dirty bitmap so that it will begin tracking disk changes.

Errors

- If node is not a valid block device, DeviceNotFound
- If name is not found, GenericError with an explanation

Since

4.0

Example

```
-> { "execute": "block-dirty-bitmap-enable",
     "arguments": { "node": "drive0", "name": "bitmap0" } }
<- { "return": {} }
```

block-dirty-bitmap-disable (Command)

Disables a dirty bitmap so that it will stop tracking disk changes.

Errors

- If `node` is not a valid block device, DeviceNotFound
- If `name` is not found, GenericError with an explanation

Since

4.0

Example

```
-> { "execute": "block-dirty-bitmap-disable",
     "arguments": { "node": "drive0", "name": "bitmap0" } }
<- { "return": {} }
```

block-dirty-bitmap-merge (Command)

Merge dirty bitmaps listed in `bitmaps` to the `target` dirty bitmap. Dirty bitmaps in `bitmaps` will be unchanged, except if it also appears as the `target` bitmap. Any bits already set in `target` will still be set after the merge, i.e., this operation does not clear the target. On error, `target` is unchanged.

The resulting bitmap will count as dirty any clusters that were dirty in any of the source bitmaps. This can be used to achieve backup checkpoints, or in simpler usages, to copy bitmaps.

Errors

- If `node` is not a valid block device, DeviceNotFound
- If any bitmap in `bitmaps` or `target` is not found, GenericError
- If any of the bitmaps have different sizes or granularities, GenericError

Since

4.0

Example

```
-> { "execute": "block-dirty-bitmap-merge",
     "arguments": { "node": "drive0", "target": "bitmap0",
                    "bitmaps": ["bitmap1"] } }
<- { "return": {} }
```

BlockDirtyBitmapSha256 (Object)

SHA256 hash of dirty bitmap data

Members

sha256: string
ASCII representation of SHA256 bitmap hash

Since

2.10

x-debug-block-dirty-bitmap-sha256 (Command)

Get bitmap SHA256.

Features

unstable
This command is meant for debugging.

Returns

BlockDirtyBitmapSha256

Errors

- If node is not a valid block device, DeviceNotFound
- If name is not found or if hashing has failed, GenericError with an explanation

Since

2.10

blockdev-mirror (Command)

Start mirroring a block device's writes to a new destination.

Arguments

job-id: string (optional)
identifier for the newly-created block job. If omitted, the device name will be used. (Since 2.7)

device: string
The device name or node-name of a root node whose writes should be mirrored.

target: string
the id or node-name of the block device to mirror to. This mustn't be attached to guest.

replaces: string (optional)
with sync=full graph node name to be replaced by the new image when a whole image copy is done. This can be used to repair broken Quorum files. By default, `device` is replaced, although implicitly created filters on it are kept.

speed: int (optional)
the maximum speed, in bytes per second

sync: MirrorSyncMode
what parts of the disk image should be copied to the destination (all the disk, only the sectors allocated in the topmost image, or only new I/O).

granularity: int (optional)
granularity of the dirty bitmap, default is 64K if the image format doesn't have clusters, 4K if the clusters are smaller than that, else the cluster size. Must be a power of 2 between 512 and 64M

buf-size: int (optional)
maximum amount of data in flight from source to target

on-source-error: BlockdevOnError (optional)
the action to take on an error on the source, default 'report'. 'stop' and 'enospc' can only be used if the block device supports io-status (see BlockInfo).

on-target-error: BlockdevOnError (optional)
the action to take on an error on the target, default 'report' (no limitations, since this applies to a different block device than `device`).

filter-node-name: string (optional)
the node name that should be assigned to the filter driver that the mirror job inserts into the graph above `device`. If this option is not given, a node name is autogenerated. (Since: 2.9)

copy-mode: MirrorCopyMode (optional)
when to copy data to the destination; defaults to 'background' (Since: 3.0)

auto-finalize: boolean (optional)
When false, this job will wait in a PENDING state after it has finished its work, waiting for `block-job-finalize` before making any block graph changes. When true, this job will automatically perform its abort or commit actions. Defaults to true. (Since 3.1)

auto-dismiss: boolean (optional)
When false, this job will wait in a CONCLUDED state after it has completely ceased all work, and awaits `block-job-dismiss`. When true, this job will automatically disappear from the query list without user intervention. Defaults to true. (Since 3.1)

Since

2.6

Example

```
-> { "execute": "blockdev-mirror",
     "arguments": { "device": "ide-hd0",
                    "target": "target0",
                    "sync": "full" } }
<- { "return": {} }
```

`BlockIOThrottle` (Object)

A set of parameters describing block throttling.

Members

device: string (optional)
Block device name

id: string (optional)
The name or QOM path of the guest device (since: 2.8)

bps: int
total throughput limit in bytes per second

bps_rd: int
read throughput limit in bytes per second

bps_wr: int
write throughput limit in bytes per second

iops: int
total I/O operations per second

iops_rd: int
read I/O operations per second

iops_wr: int
write I/O operations per second

bps_max: int (optional)
total throughput limit during bursts, in bytes (Since 1.7)

bps_rd_max: int (optional)
read throughput limit during bursts, in bytes (Since 1.7)

bps_wr_max: int (optional)
write throughput limit during bursts, in bytes (Since 1.7)

iops_max: int (optional)
total I/O operations per second during bursts, in bytes (Since 1.7)

iops_rd_max: int (optional)
read I/O operations per second during bursts, in bytes (Since 1.7)

iops_wr_max: int (optional)
write I/O operations per second during bursts, in bytes (Since 1.7)

bps_max_length: int (optional)
maximum length of the bps_max burst period, in seconds. It must only be set if bps_max is set as well. Defaults to 1. (Since 2.6)

bps_rd_max_length: int (optional)
maximum length of the bps_rd_max burst period, in seconds. It must only be set if bps_rd_max is set as well. Defaults to 1. (Since 2.6)

bps_wr_max_length: int (optional)
maximum length of the bps_wr_max burst period, in seconds. It must only be set if bps_wr_max is set as well. Defaults to 1. (Since 2.6)

iops_max_length: int (optional)
maximum length of the iops burst period, in seconds. It must only be set if iops_max is set as well. Defaults to 1. (Since 2.6)

iops_rd_max_length: int (optional)
maximum length of the iops_rd_max burst period, in seconds. It must only be set if iops_rd_max is set as well. Defaults to 1. (Since 2.6)

iops_wr_max_length: int (optional)
maximum length of the iops_wr_max burst period, in seconds. It must only be set if iops_wr_max is set as well. Defaults to 1. (Since 2.6)

iops_size: int (optional)
an I/O size in bytes (Since 1.7)

group: string (optional)
throttle group name (Since 2.4)

Features

deprecated
Member device is deprecated. Use id instead.

Since

1.1

`ThrottleLimits` (Object)

Limit parameters for throttling. Since some limit combinations are illegal, limits should always be set in one transaction. All fields are optional. When setting limits, if a field is missing the current value is not changed.

Members

`iops-total`: `int` (optional)
limit total I/O operations per second

`iops-total-max`: `int` (optional)
I/O operations burst

`iops-total-max-length`: `int` (optional)
length of the iops-total-max burst period, in seconds It must only be set if `iops-total-max` is set as well.

`iops-read`: `int` (optional)
limit read operations per second

`iops-read-max`: `int` (optional)
I/O operations read burst

`iops-read-max-length`: `int` (optional)
length of the iops-read-max burst period, in seconds It must only be set if `iops-read-max` is set as well.

`iops-write`: `int` (optional)
limit write operations per second

`iops-write-max`: `int` (optional)
I/O operations write burst

`iops-write-max-length`: `int` (optional)
length of the iops-write-max burst period, in seconds It must only be set if `iops-write-max` is set as well.

`bps-total`: `int` (optional)
limit total bytes per second

`bps-total-max`: `int` (optional)
total bytes burst

`bps-total-max-length`: `int` (optional)
length of the bps-total-max burst period, in seconds. It must only be set if `bps-total-max` is set as well.

`bps-read`: `int` (optional)
limit read bytes per second

`bps-read-max`: `int` (optional)
total bytes read burst

`bps-read-max-length`: `int` (optional)
length of the bps-read-max burst period, in seconds It must only be set if `bps-read-max` is set as well.

`bps-write`: `int` (optional)
limit write bytes per second

`bps-write-max`: `int` (optional)
total bytes write burst

`bps-write-max-length`: `int` (optional)
length of the bps-write-max burst period, in seconds It must only be set if `bps-write-max` is set as well.

iops-size: int (optional)
when limiting by iops max size of an I/O in bytes

Since

2.11

ThrottleGroupProperties (Object)

Properties for throttle-group objects.

Members

limits: ThrottleLimits (optional)
limits to apply for this throttle group

x-iops-total: int (optional)
Not documented

x-iops-total-max: int (optional)
Not documented

x-iops-total-max-length: int (optional)
Not documented

x-iops-read: int (optional)
Not documented

x-iops-read-max: int (optional)
Not documented

x-iops-read-max-length: int (optional)
Not documented

x-iops-write: int (optional)
Not documented

x-iops-write-max: int (optional)
Not documented

x-iops-write-max-length: int (optional)
Not documented

x-bps-total: int (optional)
Not documented

x-bps-total-max: int (optional)
Not documented

x-bps-total-max-length: int (optional)
Not documented

x-bps-read: int (optional)
Not documented

x-bps-read-max: int (optional)
Not documented

x-bps-read-max-length: int (optional)
Not documented

x-bps-write: int (optional)
Not documented

x-bps-write-max: int (optional)
Not documented

x-bps-write-max-length: int (optional)
Not documented

x-iops-size: int (optional)
Not documented

Features

unstable
All members starting with x- are aliases for the same key without x- in the `limits` object. This is not a stable interface and may be removed or changed incompatibly in the future. Use `limits` for a supported stable interface.

Since

2.11

block-stream (Command)

Copy data from a backing file into a block device.

The block streaming operation is performed in the background until the entire backing file has been copied. This command returns immediately once streaming has started. The status of ongoing block streaming operations can be checked with query-block-jobs. The operation can be stopped before it has completed using the block-job-cancel command.

The node that receives the data is called the top image, can be located in any part of the chain (but always above the base image; see below) and can be specified using its device or node name. Earlier qemu versions only allowed 'device' to name the top level node; presence of the 'base-node' parameter during introspection can be used as a witness of the enhanced semantics of 'device'.

If a base file is specified then sectors are not copied from that base file and its backing chain. This can be used to stream a subset of the backing file chain instead of flattening the entire image. When streaming completes the image file will have the base file as its backing file, unless that node was changed while the job was running. In that case, base's parent's backing (or filtered, whichever exists) child (i.e., base at the beginning of the job) will be the new backing file.

On successful completion the image file is updated to drop the backing file and the BLOCK_JOB_COMPLETED event is emitted.

In case `device` is a filter node, block-stream modifies the first non-filter overlay node below it to point to the new backing node instead of modifying `device` itself.

Arguments

job-id: string (optional)
identifier for the newly-created block job. If omitted, the device name will be used. (Since 2.7)

device: string
the device or node name of the top image

base: string (optional)
the common backing file name. It cannot be set if base-node or bottom is also set.

base-node: string (optional)
the node name of the backing file. It cannot be set if base or bottom is also set. (Since 2.8)

bottom: string (optional)
the last node in the chain that should be streamed into top. It cannot be set if base or base-node is also set. It cannot be filter node. (Since 6.0)

backing-file: string (optional)
The backing file string to write into the top image. This filename is not validated.

If a pathname string is such that it cannot be resolved by QEMU, that means that subsequent QMP or HMP commands must use node-names for the image in question, as filename lookup methods will fail.

If not specified, QEMU will automatically determine the backing file string to use, or error out if there is no obvious choice. Care should be taken when specifying the string, to specify a valid filename or protocol. (Since 2.1)

backing-mask-protocol: boolean (optional)
If true, replace any protocol mentioned in the 'backing file format' with 'raw', rather than storing the protocol name as the backing format. Can be used even when no image header will be updated (default false; since 9.0).

speed: int (optional)
the maximum speed, in bytes per second

on-error: BlockdevOnError (optional)
the action to take on an error (default report). 'stop' and 'enospc' can only be used if the block device supports io-status (see BlockInfo). (Since 1.3)

filter-node-name: string (optional)
the node name that should be assigned to the filter driver that the stream job inserts into the graph above device. If this option is not given, a node name is autogenerated. (Since: 6.0)

auto-finalize: boolean (optional)
When false, this job will wait in a PENDING state after it has finished its work, waiting for block-job-finalize before making any block graph changes. When true, this job will automatically perform its abort or commit actions. Defaults to true. (Since 3.1)

auto-dismiss: boolean (optional)
When false, this job will wait in a CONCLUDED state after it has completely ceased all work, and awaits block-job-dismiss. When true, this job will automatically disappear from the query list without user intervention. Defaults to true. (Since 3.1)

Errors

- If `device` does not exist, DeviceNotFound.

Since

1.1

Example

```
-> { "execute": "block-stream",
     "arguments": { "device": "virtio0",
                    "base": "/tmp/master.qcow2" } }
<- { "return": {} }
```

`block-job-set-speed` (Command)

Set maximum speed for a background block operation.

This command can only be issued when there is an active block job.

Throttling can be disabled by setting the speed to 0.

Arguments

`device`: `string`
: The job identifier. This used to be a device name (hence the name of the parameter), but since QEMU 2.7 it can have other values.

`speed`: `int`
: the maximum speed, in bytes per second, or 0 for unlimited. Defaults to 0.

Errors

- If no background operation is active on this device, DeviceNotActive

Since

1.1

block-job-cancel (Command)

Stop an active background block operation.

This command returns immediately after marking the active background block operation for cancellation. It is an error to call this command if no operation is in progress.

The operation will cancel as soon as possible and then emit the BLOCK_JOB_CANCELLED event. Before that happens the job is still visible when enumerated using query-block-jobs.

Note that if you issue 'block-job-cancel' after 'drive-mirror' has indicated (via the event BLOCK_JOB_READY) that the source and destination are synchronized, then the event triggered by this command changes to BLOCK_JOB_COMPLETED, to indicate that the mirroring has ended and the destination now has a point-in-time copy tied to the time of the cancellation.

For streaming, the image file retains its backing file unless the streaming operation happens to complete just as it is being cancelled. A new streaming operation can be started at a later time to finish copying all data from the backing file.

Arguments

device: string
: The job identifier. This used to be a device name (hence the name of the parameter), but since QEMU 2.7 it can have other values.

force: boolean (optional)
: If true, and the job has already emitted the event BLOCK_JOB_READY, abandon the job immediately (even if it is paused) instead of waiting for the destination to complete its final synchronization (since 1.3)

Errors

- If no background operation is active on this device, DeviceNotActive

Since

1.1

block-job-pause (Command)

Pause an active background block operation.

This command returns immediately after marking the active background block operation for pausing. It is an error to call this command if no operation is in progress or if the job is already paused.

The operation will pause as soon as possible. No event is emitted when the operation is actually paused. Cancelling a paused job automatically resumes it.

Arguments

device: string
The job identifier. This used to be a device name (hence the name of the parameter), but since QEMU 2.7 it can have other values.

Errors

- If no background operation is active on this device, DeviceNotActive

Since

1.3

block-job-resume (Command)

Resume an active background block operation.

This command returns immediately after resuming a paused background block operation. It is an error to call this command if no operation is in progress or if the job is not paused.

This command also clears the error status of the job.

Arguments

device: string
The job identifier. This used to be a device name (hence the name of the parameter), but since QEMU 2.7 it can have other values.

Errors

- If no background operation is active on this device, DeviceNotActive

Since

1.3

block-job-complete (Command)

Manually trigger completion of an active background block operation. This is supported for drive mirroring, where it also switches the device to write to the target path only. The ability to complete is signaled with a BLOCK_JOB_READY event.

This command completes an active background block operation synchronously. The ordering of this command's return with the BLOCK_JOB_COMPLETED event is not defined. Note that if an I/O error occurs during the processing of this command: 1) the command itself will fail; 2) the error will be processed according to the rerror/werror arguments that were specified when starting the operation.

A cancelled or paused job cannot be completed.

Arguments

device: string
The job identifier. This used to be a device name (hence the name of the parameter), but since QEMU 2.7 it can have other values.

Errors

- If no background operation is active on this device, DeviceNotActive

Since

1.3

block-job-dismiss (Command)

For jobs that have already concluded, remove them from the block-job-query list. This command only needs to be run for jobs which were started with QEMU 2.12+ job lifetime management semantics.

This command will refuse to operate on any job that has not yet reached its terminal state, JOB_STATUS_CONCLUDED. For jobs that make use of the BLOCK_JOB_READY event, block-job-cancel or block-job-complete will still need to be used as appropriate.

Arguments

id: string
The job identifier.

Since

2.12

block-job-finalize (Command)

Once a job that has manual=true reaches the pending state, it can be instructed to finalize any graph changes and do any necessary cleanup via this command. For jobs in a transaction, instructing one job to finalize will force ALL jobs in the transaction to finalize, so it is only necessary to instruct a single member job to finalize.

Arguments

id: string
The job identifier.

Since

2.12

BlockJobChangeOptionsMirror (Object)

Members

copy-mode: MirrorCopyMode
Switch to this copy mode. Currently, only the switch from 'background' to 'write-blocking' is implemented.

Since

8.2

BlockJobChangeOptions (Object)

Block job options that can be changed after job creation.

Members

id: string
The job identifier

type: JobType
The job type

The members of BlockJobChangeOptionsMirror when type is "mirror"

Since

8.2

block-job-change (Command)

Change the block job's options.

Arguments

The members of `BlockJobChangeOptions`

Since

8.2

`BlockdevDiscardOptions` (Enum)

Determines how to handle discard requests.

Values

`ignore`
: Ignore the request

`unmap`
: Forward as an unmap request

Since

2.9

`BlockdevDetectZeroesOptions` (Enum)

Describes the operation mode for the automatic conversion of plain zero writes by the OS to driver specific optimized zero write commands.

Values

`off`
: Disabled (default)

`on`
: Enabled

`unmap`
: Enabled and even try to unmap blocks if possible. This requires also that `BlockdevDiscardOptions` is set to unmap for this device.

Since

2.1

BlockdevAioOptions (Enum)

Selects the AIO backend to handle I/O requests

Values

threads
Use qemu's thread pool

native
Use native AIO backend (only Linux and Windows)

io_uring (If: CONFIG_LINUX_IO_URING)
Use linux io_uring (since 5.0)

Since

2.9

BlockdevCacheOptions (Object)

Includes cache-related options for block devices

Members

direct: boolean (optional)
enables use of O_DIRECT (bypass the host page cache; default: false)

no-flush: boolean (optional)
ignore any flush requests for the device (default: false)

Since

2.9

BlockdevDriver (Enum)

Drivers that are supported in block device operations.

Values

throttle
Since 2.11

nvme
Since 2.12

copy-on-read
Since 3.0

blklogwrites
Since 3.0

blkreplay
Since 4.2

compress
Since 5.0

copy-before-write
Since 6.2

snapshot-access
Since 7.0

blkdebug
Not documented

blkverify
Not documented

bochs
Not documented

cloop
Not documented

dmg
Not documented

file
Not documented

ftp
Not documented

ftps
Not documented

gluster
Not documented

host_cdrom (**If: HAVE_HOST_BLOCK_DEVICE**)
Not documented

host_device (**If: HAVE_HOST_BLOCK_DEVICE**)
Not documented

http
Not documented

https
Not documented

io_uring (**If:** CONFIG_BLKIO)
Not documented

iscsi
Not documented

luks
Not documented

nbd
Not documented

nfs
Not documented

null-aio
Not documented

null-co
Not documented

nvme-io_uring (**If:** CONFIG_BLKIO)
Not documented

parallels
Not documented

preallocate
Not documented

qcow
Not documented

qcow2
Not documented

qed
Not documented

quorum
Not documented

raw
Not documented

rbd
Not documented

replication (**If:** CONFIG_REPLICATION)
Not documented

ssh
Not documented

vdi
Not documented

vhdx
Not documented

virtio-blk-vfio-pci (**If:** CONFIG_BLKIO)
Not documented

virtio-blk-vhost-user (**If:** CONFIG_BLKIO)
Not documented

virtio-blk-vhost-vdpa (**If:** CONFIG_BLKIO)
Not documented

vmdk
Not documented

vpc
Not documented

vvfat
Not documented

Since

2.9

BlockdevOptionsFile (Object)

Driver specific block device options for the file backend.

Members

filename: string
path to the image file

pr-manager: string (optional)
the id for the object that will handle persistent reservations for this device (default: none, forward the commands via SG_IO; since 2.11)

aio: BlockdevAioOptions (optional)
AIO backend (default: threads) (since: 2.8)

aio-max-batch: int (optional)
maximum number of requests to batch together into a single submission in the AIO backend. The smallest value between this and the aio-max-batch value of the IOThread object is chosen. 0 means that the AIO backend will handle it automatically. (default: 0, since 6.2)

locking: OnOffAuto (optional)
whether to enable file locking. If set to 'auto', only enable when Open File Descriptor (OFD) locking API is available (default: auto, since 2.10)

drop-cache: boolean (optional) (If: CONFIG_LINUX)
invalidate page cache during live migration. This prevents stale data on the migration destination with cache.direct=off. Currently only supported on Linux hosts. (default: on, since: 4.0)

x-check-cache-dropped: boolean (optional)
whether to check that page cache was dropped on live migration. May cause noticeable delays if the image file is large, do not use in production. (default: off) (since: 3.0)

Features

dynamic-auto-read-only
If present, enabled auto-read-only means that the driver will open the image read-only at first, dynamically reopen the image file read-write when the first writer is attached to the node and reopen read-only when the last writer is detached. This allows giving QEMU write permissions only on demand when an operation actually needs write access.

unstable
Member x-check-cache-dropped is meant for debugging.

Since

2.9

BlockdevOptionsNull (Object)

Driver specific block device options for the null backend.

Members

size: int (optional)
size of the device in bytes.

latency-ns: int (optional)
emulated latency (in nanoseconds) in processing requests. Default to zero which completes requests immediately. (Since 2.4)

read-zeroes: boolean (optional)
if true, reads from the device produce zeroes; if false, the buffer is left unchanged. (default: false; since: 4.1)

Since

2.9

BlockdevOptionsNVMe (Object)

Driver specific block device options for the NVMe backend.

Members

device: string
PCI controller address of the NVMe device in format hhhh:bb:ss.f (host:bus:slot.function)

namespace: int
namespace number of the device, starting from 1.

Note that the PCI `device` must have been unbound from any host kernel driver before instructing QEMU to add the blockdev.

Since

2.12

BlockdevOptionsVVFAT (Object)

Driver specific block device options for the vvfat protocol.

Members

dir: string
directory to be exported as FAT image

fat-type: int (optional)
FAT type: 12, 16 or 32

floppy: boolean (optional)
whether to export a floppy image (true) or partitioned hard disk (false; default)

label: string (optional)
set the volume label, limited to 11 bytes. FAT16 and FAT32 traditionally have some restrictions on labels, which are ignored by most operating systems. Defaults to "QEMU VVFAT". (since 2.4)

rw: boolean (optional)
whether to allow write operations (default: false)

Since

2.9

BlockdevOptionsGenericFormat (Object)

Driver specific block device options for image format that have no option besides their data source.

Members

file: BlockdevRef
reference to or definition of the data source block device

Since

2.9

BlockdevOptionsLUKS (Object)

Driver specific block device options for LUKS.

Members

key-secret: string (optional)
the ID of a QCryptoSecret object providing the decryption key (since 2.6). Mandatory except when doing a metadata-only probe of the image.

header: BlockdevRef (optional)
block device holding a detached LUKS header. (since 9.0)

The members of BlockdevOptionsGenericFormat

Since

2.9

BlockdevOptionsGenericCOWFormat (Object)

Driver specific block device options for image format that have no option besides their data source and an optional backing file.

Members

backing: BlockdevRefOrNull (optional)
reference to or definition of the backing file block device, null disables the backing file entirely. Defaults to the backing file stored the image file.

The members of BlockdevOptionsGenericFormat

Since

2.9

Qcow2OverlapCheckMode (Enum)

General overlap check modes.

Values

none
Do not perform any checks

constant
Perform only checks which can be done in constant time and without reading anything from disk

cached
Perform only checks which can be done without reading anything from disk

all
Perform all available overlap checks

Since

2.9

Qcow2OverlapCheckFlags (Object)

Structure of flags for each metadata structure. Setting a field to 'true' makes QEMU guard that Qcow2 format structure against unintended overwriting. See Qcow2 format specification for detailed information on these structures. The default value is chosen according to the template given.

Members

template: Qcow2OverlapCheckMode (optional)
Specifies a template mode which can be adjusted using the other flags, defaults to 'cached'

main-header: boolean (optional)
Qcow2 format header

active-l1: boolean (optional)
Qcow2 active L1 table

active-l2: boolean (optional)
Qcow2 active L2 table

refcount-table: boolean (optional)
Qcow2 refcount table

refcount-block: boolean (optional)
Qcow2 refcount blocks

snapshot-table: boolean (optional)
Qcow2 snapshot table

inactive-l1: boolean (optional)
Qcow2 inactive L1 tables

inactive-l2: boolean (optional)
Qcow2 inactive L2 tables

bitmap-directory: boolean (optional)
Qcow2 bitmap directory (since 3.0)

Since

2.9

Qcow2OverlapChecks (Alternate)

Specifies which metadata structures should be guarded against unintended overwriting.

Members

flags: Qcow2OverlapCheckFlags
set of flags for separate specification of each metadata structure type

mode: Qcow2OverlapCheckMode
named mode which chooses a specific set of flags

Since

2.9

BlockdevQcowEncryptionFormat (Enum)

Values

aes
AES-CBC with plain64 initialization vectors

Since

2.10

BlockdevQcowEncryption (Object)

Members

format: BlockdevQcowEncryptionFormat
encryption format

The members of QCryptoBlockOptionsQCow when format is "aes"

Since

2.10

BlockdevOptionsQcow (Object)

Driver specific block device options for qcow.

Members

encrypt: BlockdevQcowEncryption (optional)
Image decryption options. Mandatory for encrypted images, except when doing a metadata-only probe of the image.

The members of BlockdevOptionsGenericCOWFormat

Since

2.10

BlockdevQcow2EncryptionFormat (Enum)

Values

aes
AES-CBC with plain64 initialization vectors

luks
Not documented

Since

2.10

BlockdevQcow2Encryption (Object)

Members

format: BlockdevQcow2EncryptionFormat
encryption format

The members of QCryptoBlockOptionsQCow when format is "aes"
The members of QCryptoBlockOptionsLUKS when format is "luks"

Since

2.10

BlockdevOptionsPreallocate (Object)

Filter driver intended to be inserted between format and protocol node and do preallocation in protocol node on write.

Members

prealloc-align: int (optional)
on preallocation, align file length to this number, default 1048576 (1M)

prealloc-size: int (optional)
how much to preallocate, default 134217728 (128M)

The members of BlockdevOptionsGenericFormat

Since

6.0

BlockdevOptionsQcow2 (Object)

Driver specific block device options for qcow2.

Members

lazy-refcounts: boolean (optional)
whether to enable the lazy refcounts feature (default is taken from the image file)

pass-discard-request: boolean (optional)
whether discard requests to the qcow2 device should be forwarded to the data source

pass-discard-snapshot: boolean (optional)
whether discard requests for the data source should be issued when a snapshot operation (e.g. deleting a snapshot) frees clusters in the qcow2 file

pass-discard-other: boolean (optional)
whether discard requests for the data source should be issued on other occasions where a cluster gets freed

discard-no-unref: boolean (optional)
when enabled, data clusters will remain preallocated when they are no longer used, e.g. because they are discarded or converted to zero clusters. As usual, whether the old data is discarded or kept on the protocol level (i.e. in the image file) depends on the setting of the pass-discard-request option. Keeping the clusters preallocated prevents qcow2 fragmentation that would otherwise be caused by freeing and re-allocating them later. Besides potential performance degradation, such fragmentation can lead to increased allocation of clusters past the end of the image file, resulting in image files whose file length can grow much larger than their guest disk size would suggest. If image file length is of concern (e.g. when storing qcow2 images directly on block devices), you should consider enabling this option. (since 8.1)

overlap-check: Qcow2OverlapChecks (optional)
which overlap checks to perform for writes to the image, defaults to 'cached' (since 2.2)

cache-size: int (optional)
the maximum total size of the L2 table and refcount block caches in bytes (since 2.2)

l2-cache-size: int (optional)
the maximum size of the L2 table cache in bytes (since 2.2)

l2-cache-entry-size: int (optional)
the size of each entry in the L2 cache in bytes. It must be a power of two between 512 and the cluster size. The default value is the cluster size (since 2.12)

refcount-cache-size: int (optional)
the maximum size of the refcount block cache in bytes (since 2.2)

cache-clean-interval: int (optional)
clean unused entries in the L2 and refcount caches. The interval is in seconds. The default value is 600 on supporting platforms, and 0 on other platforms. 0 disables this feature. (since 2.5)

encrypt: BlockdevQcow2Encryption (optional)
Image decryption options. Mandatory for encrypted images, except when doing a metadata-only probe of the image. (since 2.10)

data-file: BlockdevRef (optional)
reference to or definition of the external data file. This may only be specified for images that require an external data file. If it is not specified for such an image, the data file name is loaded from the image file. (since 4.0)

The members of BlockdevOptionsGenericCOWFormat

Since

2.9

SshHostKeyCheckMode (Enum)

Values

none
Don't check the host key at all

hash
Compare the host key with a given hash

known_hosts
Check the host key against the known_hosts file

Since

2.12

SshHostKeyCheckHashType (Enum)

Values

md5
The given hash is an md5 hash

sha1
The given hash is an sha1 hash

sha256
The given hash is an sha256 hash

Since

2.12

SshHostKeyHash (Object)

Members

type: SshHostKeyCheckHashType
The hash algorithm used for the hash

hash: string
The expected hash value

Since

2.12

SshHostKeyCheck (Object)

Members

mode: SshHostKeyCheckMode
How to check the host key

The members of SshHostKeyHash when mode is "hash"

Since

2.12

BlockdevOptionsSsh (Object)

Members

server: InetSocketAddress
host address

path: string
path to the image on the host

user: string (optional)
user as which to connect, defaults to current local user name

host-key-check: SshHostKeyCheck (optional)
Defines how and what to check the host key against (default: known_hosts)

Since

2.9

BlkdebugEvent (Enum)

Trigger events supported by blkdebug.

Values

l1_shrink_write_table
write zeros to the l1 table to shrink image. (since 2.11)

l1_shrink_free_l2_clusters
discard the l2 tables. (since 2.11)

cor_write
a write due to copy-on-read (since 2.11)

cluster_alloc_space
an allocation of file space for a cluster (since 4.1)

none
triggers once at creation of the blkdebug node (since 4.1)

l1_update
Not documented

l1_grow_alloc_table
Not documented

l1_grow_write_table
Not documented

l1_grow_activate_table
: Not documented

l2_load
: Not documented

l2_update
: Not documented

l2_update_compressed
: Not documented

l2_alloc_cow_read
: Not documented

l2_alloc_write
: Not documented

read_aio
: Not documented

read_backing_aio
: Not documented

read_compressed
: Not documented

write_aio
: Not documented

write_compressed
: Not documented

vmstate_load
: Not documented

vmstate_save
: Not documented

cow_read
: Not documented

cow_write
: Not documented

reftable_load
: Not documented

reftable_grow
: Not documented

reftable_update
: Not documented

refblock_load
: Not documented

refblock_update
: Not documented

refblock_update_part
: Not documented

refblock_alloc
: Not documented

refblock_alloc_hookup
: Not documented

refblock_alloc_write
: Not documented

refblock_alloc_write_blocks
: Not documented

refblock_alloc_write_table
: Not documented

refblock_alloc_switch_table
: Not documented

cluster_alloc
: Not documented

cluster_alloc_bytes
: Not documented

cluster_free
: Not documented

flush_to_os
: Not documented

flush_to_disk
: Not documented

pwritev_rmw_head
: Not documented

pwritev_rmw_after_head
: Not documented

pwritev_rmw_tail
: Not documented

pwritev_rmw_after_tail
: Not documented

pwritev
: Not documented

pwritev_zero
: Not documented

pwritev_done
: Not documented

empty_image_prepare
: Not documented

Since

2.9

BlkdebugIOType (Enum)

Kinds of I/O that blkdebug can inject errors in.

Values

read
: .bdrv_co_preadv()

write
: .bdrv_co_pwritev()

write-zeroes
: .bdrv_co_pwrite_zeroes()

discard
: .bdrv_co_pdiscard()

flush
: .bdrv_co_flush_to_disk()

block-status
: .bdrv_co_block_status()

Since

4.1

BlkdebugInjectErrorOptions (Object)

Describes a single error injection for blkdebug.

Members

event: BlkdebugEvent
: trigger event

state: int (optional)
: the state identifier blkdebug needs to be in to actually trigger the event; defaults to "any"

iotype: BlkdebugIOType (optional)
: the type of I/O operations on which this error should be injected; defaults to "all read, write, write-zeroes, discard, and flush operations" (since: 4.1)

errno: int (optional)
: error identifier (errno) to be returned; defaults to EIO

sector: int (optional)
: specifies the sector index which has to be affected in order to actually trigger the event; defaults to "any sector"

once: boolean (optional)
disables further events after this one has been triggered; defaults to false

immediately: boolean (optional)
fail immediately; defaults to false

Since

2.9

BlkdebugSetStateOptions (Object)

Describes a single state-change event for blkdebug.

Members

event: BlkdebugEvent
trigger event

state: int (optional)
the current state identifier blkdebug needs to be in; defaults to "any"

new_state: int
the state identifier blkdebug is supposed to assume if this event is triggered

Since

2.9

BlockdevOptionsBlkdebug (Object)

Driver specific block device options for blkdebug.

Members

image: BlockdevRef
underlying raw block device (or image file)

config: string (optional)
filename of the configuration file

align: int (optional)
required alignment for requests in bytes, must be positive power of 2, or 0 for default

max-transfer: int (optional)
maximum size for I/O transfers in bytes, must be positive multiple of `align` and of the underlying file's request alignment (but need not be a power of 2), or 0 for default (since 2.10)

opt-write-zero: int (optional)
preferred alignment for write zero requests in bytes, must be positive multiple of `align` and of the underlying file's request alignment (but need not be a power of 2), or 0 for default (since 2.10)

max-write-zero: int (optional)
maximum size for write zero requests in bytes, must be positive multiple of `align`, of `opt-write-zero`, and of the underlying file's request alignment (but need not be a power of 2), or 0 for default (since 2.10)

opt-discard: int (optional)
preferred alignment for discard requests in bytes, must be positive multiple of `align` and of the underlying file's request alignment (but need not be a power of 2), or 0 for default (since 2.10)

max-discard: int (optional)
maximum size for discard requests in bytes, must be positive multiple of `align`, of `opt-discard`, and of the underlying file's request alignment (but need not be a power of 2), or 0 for default (since 2.10)

inject-error: array of BlkdebugInjectErrorOptions (optional)
array of error injection descriptions

set-state: array of BlkdebugSetStateOptions (optional)
array of state-change descriptions

take-child-perms: array of BlockPermission (optional)
Permissions to take on `image` in addition to what is necessary anyway (which depends on how the blkdebug node is used). Defaults to none. (since 5.0)

unshare-child-perms: array of BlockPermission (optional)
Permissions not to share on `image` in addition to what cannot be shared anyway (which depends on how the blkdebug node is used). Defaults to none. (since 5.0)

Since

2.9

BlockdevOptionsBlklogwrites (Object)

Driver specific block device options for blklogwrites.

Members

file: BlockdevRef
block device

log: BlockdevRef
block device used to log writes to `file`

log-sector-size: int (optional)
sector size used in logging writes to `file`, determines granularity of offsets and sizes of writes (default: 512)

log-append: boolean (optional)
append to an existing log (default: false)

log-super-update-interval: int (optional)
interval of write requests after which the log super block is updated to disk (default: 4096)

Since

3.0

BlockdevOptionsBlkverify (Object)

Driver specific block device options for blkverify.

Members

test: BlockdevRef
block device to be tested

raw: BlockdevRef
raw image used for verification

Since

2.9

BlockdevOptionsBlkreplay (Object)

Driver specific block device options for blkreplay.

Members

image: BlockdevRef
disk image which should be controlled with blkreplay

Since

4.2

QuorumReadPattern (Enum)

An enumeration of quorum read patterns.

Values

quorum
read all the children and do a quorum vote on reads

fifo
read only from the first child that has not failed

Since

2.9

BlockdevOptionsQuorum (Object)

Driver specific block device options for Quorum

Members

blkverify: boolean (optional)
true if the driver must print content mismatch set to false by default

children: array of BlockdevRef
the children block devices to use

vote-threshold: int
the vote limit under which a read will fail

rewrite-corrupted: boolean (optional)
rewrite corrupted data when quorum is reached (Since 2.1)

read-pattern: QuorumReadPattern (optional)
choose read pattern and set to quorum by default (Since 2.2)

Since

2.9

BlockdevOptionsGluster (Object)

Driver specific block device options for Gluster

Members

volume: string
name of gluster volume where VM image resides

path: string
absolute path to image file in gluster volume

server: array of SocketAddress
gluster servers description

debug: int (optional)
libgfapi log level (default '4' which is Error) (Since 2.8)

logfile: string (optional)
libgfapi log file (default /dev/stderr) (Since 2.8)

Since

2.9

BlockdevOptionsIoUring (Object)

Driver specific block device options for the io_uring backend.

Members

filename: string
path to the image file

Since

7.2

If

CONFIG_BLKIO

BlockdevOptionsNvmeIoUring (Object)

Driver specific block device options for the nvme-io_uring backend.

Members

path: string
path to the NVMe namespace's character device (e.g. /dev/ng0n1).

Since

7.2

If

CONFIG_BLKIO

BlockdevOptionsVirtioBlkVfioPci (Object)

Driver specific block device options for the virtio-blk-vfio-pci backend.

Members

path: string
path to the PCI device's sysfs directory (e.g. /sys/bus/pci/devices/0000:00:01.0).

Since

7.2

If

CONFIG_BLKIO

BlockdevOptionsVirtioBlkVhostUser (Object)

Driver specific block device options for the virtio-blk-vhost-user backend.

Members

path: string
path to the vhost-user UNIX domain socket.

Since

7.2

If

CONFIG_BLKIO

BlockdevOptionsVirtioBlkVhostVdpa (Object)

Driver specific block device options for the virtio-blk-vhost-vdpa backend.

Members

path: string
path to the vhost-vdpa character device.

Features

fdset
Member path supports the special "/dev/fdset/N" path (since 8.1)

Since

7.2

If

CONFIG_BLKIO

IscsiTransport (Enum)

An enumeration of libiscsi transport types

Values

tcp
Not documented

iser
Not documented

Since

2.9

IscsiHeaderDigest (Enum)

An enumeration of header digests supported by libiscsi

Values

crc32c
Not documented

none
Not documented

crc32c-none
Not documented

none-crc32c
Not documented

Since

2.9

BlockdevOptionsIscsi (Object)

Driver specific block device options for iscsi

Members

transport: IscsiTransport
The iscsi transport type

portal: string
The address of the iscsi portal

target: string
The target iqn name

lun: int (optional)
LUN to connect to. Defaults to 0.

user: string (optional)
User name to log in with. If omitted, no CHAP authentication is performed.

password-secret: string (optional)
The ID of a QCryptoSecret object providing the password for the login. This option is required if `user` is specified.

initiator-name: string (optional)
The iqn name we want to identify to the target as. If this option is not specified, an initiator name is generated automatically.

header-digest: IscsiHeaderDigest (optional)
The desired header digest. Defaults to none-crc32c.

timeout: int (optional)
Timeout in seconds after which a request will timeout. 0 means no timeout and is the default.

Since

2.9

RbdAuthMode (Enum)

Values

cephx
Not documented

none
Not documented

Since

3.0

RbdImageEncryptionFormat (Enum)

Values

luks-any
Used for opening either luks or luks2 (Since 8.0)

luks
Not documented

luks2
Not documented

Since

6.1

RbdEncryptionOptionsLUKSBase (Object)

Members

key-secret: string
ID of a QCryptoSecret object providing a passphrase for unlocking the encryption

Since

6.1

RbdEncryptionCreateOptionsLUKSBase (Object)

Members

cipher-alg: QCryptoCipherAlgorithm (optional)
The encryption algorithm

The members of RbdEncryptionOptionsLUKSBase

Since

6.1

RbdEncryptionOptionsLUKS (Object)

Members

The members of RbdEncryptionOptionsLUKSBase

Since

6.1

RbdEncryptionOptionsLUKS2 (Object)

Members

The members of RbdEncryptionOptionsLUKSBase

Since

6.1

RbdEncryptionOptionsLUKSAny (Object)

Members

The members of RbdEncryptionOptionsLUKSBase

Since

8.0

RbdEncryptionCreateOptionsLUKS (Object)

Members

The members of RbdEncryptionCreateOptionsLUKSBase

Since

6.1

RbdEncryptionCreateOptionsLUKS2 (Object)

Members

The members of RbdEncryptionCreateOptionsLUKSBase

Since

6.1

RbdEncryptionOptions (Object)

Members

format: RbdImageEncryptionFormat
Encryption format.

parent: RbdEncryptionOptions (optional)
Parent image encryption options (for cloned images). Can be left unspecified if this cloned image is encrypted using the same format and secret as its parent image (i.e. not explicitly formatted) or if its parent image is not encrypted. (Since 8.0)

The members of RbdEncryptionOptionsLUKS when format is "luks"
The members of RbdEncryptionOptionsLUKS2 when format is "luks2"
The members of RbdEncryptionOptionsLUKSAny when format is "luks-any"

Since

6.1

RbdEncryptionCreateOptions (Object)

Members

format: RbdImageEncryptionFormat
Encryption format.

The members of RbdEncryptionCreateOptionsLUKS when format is "luks"
The members of RbdEncryptionCreateOptionsLUKS2 when format is "luks2"

Since

6.1

BlockdevOptionsRbd (Object)

Members

pool: string
Ceph pool name.

namespace: string (optional)
Rados namespace name in the Ceph pool. (Since 5.0)

image: string
Image name in the Ceph pool.

conf: string (optional)
path to Ceph configuration file. Values in the configuration file will be overridden by options specified via QAPI.

snapshot: string (optional)
Ceph snapshot name.

encrypt: RbdEncryptionOptions (optional)
Image encryption options. (Since 6.1)

user: string (optional)
Ceph id name.

auth-client-required: array of RbdAuthMode (optional)
Acceptable authentication modes. This maps to Ceph configuration option "auth_client_required". (Since 3.0)

key-secret: string (optional)
ID of a QCryptoSecret object providing a key for cephx authentication. This maps to Ceph configuration option "key". (Since 3.0)

server: array of InetSocketAddressBase (optional)
Monitor host address and port. This maps to the "mon_host" Ceph option.

Since

2.9

ReplicationMode (Enum)

An enumeration of replication modes.

Values

primary
Primary mode, the vm's state will be sent to secondary QEMU.

secondary
Secondary mode, receive the vm's state from primary QEMU.

Since

2.9

If

CONFIG_REPLICATION

BlockdevOptionsReplication (Object)

Driver specific block device options for replication

Members

mode: ReplicationMode
the replication mode

top-id: string (optional)
In secondary mode, node name or device ID of the root node who owns the replication node chain. Must not be given in primary mode.

The members of BlockdevOptionsGenericFormat

Since

2.9

If

`CONFIG_REPLICATION`

`NFSTransport` (Enum)

An enumeration of NFS transport types

Values

`inet`
: TCP transport

Since

2.9

`NFSServer` (Object)

Captures the address of the socket

Members

`type: NFSTransport`
: transport type used for NFS (only TCP supported)

`host: string`
: host address for NFS server

Since

2.9

`BlockdevOptionsNfs` (Object)

Driver specific block device option for NFS

Members

server: NFSServer
host address

path: string
path of the image on the host

user: int (optional)
UID value to use when talking to the server (defaults to 65534 on Windows and getuid() on unix)

group: int (optional)
GID value to use when talking to the server (defaults to 65534 on Windows and getgid() in unix)

tcp-syn-count: int (optional)
number of SYNs during the session establishment (defaults to libnfs default)

readahead-size: int (optional)
set the readahead size in bytes (defaults to libnfs default)

page-cache-size: int (optional)
set the pagecache size in bytes (defaults to libnfs default)

debug: int (optional)
set the NFS debug level (max 2) (defaults to libnfs default)

Since

2.9

BlockdevOptionsCurlBase (Object)

Driver specific block device options shared by all protocols supported by the curl backend.

Members

url: string
URL of the image file

readahead: int (optional)
Size of the read-ahead cache; must be a multiple of 512 (defaults to 256 kB)

timeout: int (optional)
Timeout for connections, in seconds (defaults to 5)

username: string (optional)
Username for authentication (defaults to none)

password-secret: string (optional)
ID of a QCryptoSecret object providing a password for authentication (defaults to no password)

proxy-username: string (optional)
Username for proxy authentication (defaults to none)

proxy-password-secret: string (optional)
ID of a QCryptoSecret object providing a password for proxy authentication (defaults to no password)

Since

2.9

BlockdevOptionsCurlHttp (Object)

Driver specific block device options for HTTP connections over the curl backend. URLs must start with "http://".

Members

cookie: string (optional)
List of cookies to set; format is "name1=content1; name2=content2;" as explained by CURLOPT_COOKIE(3). Defaults to no cookies.

cookie-secret: string (optional)
ID of a QCryptoSecret object providing the cookie data in a secure way. See cookie for the format. (since 2.10)

The members of BlockdevOptionsCurlBase

Since

2.9

BlockdevOptionsCurlHttps (Object)

Driver specific block device options for HTTPS connections over the curl backend. URLs must start with "https://".

Members

cookie: string (optional)
List of cookies to set; format is "name1=content1; name2=content2;" as explained by CURLOPT_COOKIE(3). Defaults to no cookies.

sslverify: boolean (optional)
Whether to verify the SSL certificate's validity (defaults to true)

cookie-secret: string (optional)
ID of a QCryptoSecret object providing the cookie data in a secure way. See cookie for the format. (since 2.10)

The members of BlockdevOptionsCurlBase

Since

2.9

BlockdevOptionsCurlFtp (Object)

Driver specific block device options for FTP connections over the curl backend. URLs must start with "ftp://".

Members

The members of BlockdevOptionsCurlBase

Since

2.9

BlockdevOptionsCurlFtps (Object)

Driver specific block device options for FTPS connections over the curl backend. URLs must start with "ftps://".

Members

sslverify: boolean (optional)
Whether to verify the SSL certificate's validity (defaults to true)

The members of BlockdevOptionsCurlBase

Since

2.9

BlockdevOptionsNbd (Object)

Driver specific block device options for NBD.

Members

server: SocketAddress
NBD server address

export: string (optional)
export name

tls-creds: string (optional)
TLS credentials ID

tls-hostname: string (optional)
TLS hostname override for certificate validation (Since 7.0)

x-dirty-bitmap: string (optional)
A metadata context name such as “qemu:dirty-bitmap:NAME” or “qemu:allocation-depth” to query in place of the traditional “base:allocation” block status (see NBD_OPT_LIST_META_CONTEXT in the NBD protocol; and yes, naming this option x-context would have made more sense) (since 3.0)

reconnect-delay: int (optional)
On an unexpected disconnect, the nbd client tries to connect again until succeeding or encountering a serious error. During the first `reconnect-delay` seconds, all requests are paused and will be rerun on a successful reconnect. After that time, any delayed requests and all future requests before a successful reconnect will immediately fail. Default 0 (Since 4.2)

open-timeout: int (optional)
In seconds. If zero, the nbd driver tries the connection only once, and fails to open if the connection fails. If non-zero, the nbd driver will repeat connection attempts until successful or until `open-timeout` seconds have elapsed. Default 0 (Since 7.0)

Features

unstable
Member `x-dirty-bitmap` is experimental.

Since

2.9

BlockdevOptionsRaw (Object)

Driver specific block device options for the raw driver.

Members

offset: int (optional)
position where the block device starts

size: int (optional)
the assumed size of the device

The members of BlockdevOptionsGenericFormat

Since

2.9

BlockdevOptionsThrottle (Object)

Driver specific block device options for the throttle driver

Members

throttle-group: string
the name of the throttle-group object to use. It must already exist.

file: BlockdevRef
reference to or definition of the data source block device

Since

2.11

BlockdevOptionsCor (Object)

Driver specific block device options for the copy-on-read driver.

Members

bottom: string (optional)
The name of a non-filter node (allocation-bearing layer) that limits the COR operations in the backing chain (inclusive), so that no data below this node will be copied by this filter. If option is absent, the limit is not applied, so that data from all backing layers may be copied.

The members of BlockdevOptionsGenericFormat

Since

6.0

OnCbwError (Enum)

An enumeration of possible behaviors for copy-before-write operation failures.

Values

break-guest-write
report the error to the guest. This way, the guest will not be able to overwrite areas that cannot be backed up, so the backup process remains valid.

break-snapshot
continue guest write. Doing so will make the provided snapshot state invalid and any backup or export process based on it will finally fail.

Since

7.1

BlockdevOptionsCbw (Object)

Driver specific block device options for the copy-before-write driver, which does so called copy-before-write operations: when data is written to the filter, the filter first reads corresponding blocks from its file child and copies them to `target` child. After successfully copying, the write request is propagated to file child. If copying fails, the original write request is failed too and no data is written to file child.

Members

target: BlockdevRef
The target for copy-before-write operations.

bitmap: BlockDirtyBitmap (optional)
If specified, copy-before-write filter will do copy-before-write operations only for dirty regions of the bitmap. Bitmap size must be equal to length of file and target child of the filter. Note also, that bitmap is used only to initialize internal bitmap of the process, so further modifications (or removing) of specified bitmap doesn't influence the filter. (Since 7.0)

on-cbw-error: OnCbwError (optional)
Behavior on failure of copy-before-write operation. Default is `break-guest-write`. (Since 7.1)

cbw-timeout: int (optional)
Zero means no limit. Non-zero sets the timeout in seconds for copy-before-write operation. When a timeout occurs, the respective copy-before-write operation will fail, and the `on-cbw-error` parameter will decide how this failure is handled. Default 0. (Since 7.1)

The members of BlockdevOptionsGenericFormat

Since

6.2

BlockdevOptions (Object)

Options for creating a block device. Many options are available for all block devices, independent of the block driver:

Members

driver: BlockdevDriver
block driver name

node-name: string (optional)
the node name of the new node (Since 2.0). This option is required on the top level of blockdev-add. Valid node names start with an alphabetic character and may contain only alphanumeric characters, '-', '.' and '_'. Their maximum length is 31 characters.

discard: BlockdevDiscardOptions (optional)
discard-related options (default: ignore)

cache: BlockdevCacheOptions (optional)
cache-related options

read-only: boolean (optional)
whether the block device should be read-only (default: false). Note that some block drivers support only read-only access, either generally or in certain configurations. In this case, the default value does not work and the option must be specified explicitly.

auto-read-only: boolean (optional)
if true and `read-only` is false, QEMU may automatically decide not to open the image read-write as requested, but fall back to read-only instead (and switch between the modes later), e.g. depending on whether the image file is writable or whether a writing user is attached to the node (default: false, since 3.1)

detect-zeroes: BlockdevDetectZeroesOptions (optional)
detect and optimize zero writes (Since 2.1) (default: off)

force-share: boolean (optional)
force share all permission on added nodes. Requires read-only=true. (Since 2.10)

The members of BlockdevOptionsBlkdebug when driver is "blkdebug"
The members of BlockdevOptionsBlklogwrites when driver is "blklogwrites"
The members of BlockdevOptionsBlkverify when driver is "blkverify"
The members of BlockdevOptionsBlkreplay when driver is "blkreplay"
The members of BlockdevOptionsGenericFormat when driver is "bochs"
The members of BlockdevOptionsGenericFormat when driver is "cloop"
The members of BlockdevOptionsGenericFormat when driver is "compress"
The members of BlockdevOptionsCbw when driver is "copy-before-write"
The members of BlockdevOptionsCor when driver is "copy-on-read"
The members of BlockdevOptionsGenericFormat when driver is "dmg"
The members of BlockdevOptionsFile when driver is "file"
The members of BlockdevOptionsCurlFtp when driver is "ftp"
The members of BlockdevOptionsCurlFtps when driver is "ftps"
The members of BlockdevOptionsGluster when driver is "gluster"
The members of BlockdevOptionsFile when driver is "host_cdrom" (If: HAVE_HOST_BLOCK_DEVICE)
The members of BlockdevOptionsFile when driver is "host_device" (If: HAVE_HOST_BLOCK_DEVICE)
The members of BlockdevOptionsCurlHttp when driver is "http"
The members of BlockdevOptionsCurlHttps when driver is "https"
The members of BlockdevOptionsIoUring when driver is "io_uring" (If: CONFIG_BLKIO)
The members of BlockdevOptionsIscsi when driver is "iscsi"
The members of BlockdevOptionsLUKS when driver is "luks"
The members of BlockdevOptionsNbd when driver is "nbd"
The members of BlockdevOptionsNfs when driver is "nfs"
The members of BlockdevOptionsNull when driver is "null-aio"
The members of BlockdevOptionsNull when driver is "null-co"
The members of BlockdevOptionsNVMe when driver is "nvme"
The members of BlockdevOptionsNvmeIoUring when driver is "nvme-io_uring" (If: CONFIG_BLKIO)
The members of BlockdevOptionsGenericFormat when driver is "parallels"
The members of BlockdevOptionsPreallocate when driver is "preallocate"
The members of BlockdevOptionsQcow2 when driver is "qcow2"
The members of BlockdevOptionsQcow when driver is "qcow"
The members of BlockdevOptionsGenericCOWFormat when driver is "qed"
The members of BlockdevOptionsQuorum when driver is "quorum"
The members of BlockdevOptionsRaw when driver is "raw"
The members of BlockdevOptionsRbd when driver is "rbd"
The members of BlockdevOptionsReplication when driver is "replication" (If: CONFIG_REPLICATION)
The members of BlockdevOptionsGenericFormat when driver is "snapshot-access"
The members of BlockdevOptionsSsh when driver is "ssh"
The members of BlockdevOptionsThrottle when driver is "throttle"
The members of BlockdevOptionsGenericFormat when driver is "vdi"
The members of BlockdevOptionsGenericFormat when driver is "vhdx"
The members of BlockdevOptionsVirtioBlkVfioPci when driver is "virtio-blk-vfio-pci" (If: CONFIG_BLKIO)
The members of BlockdevOptionsVirtioBlkVhostUser when driver is "virtio-blk-vhost-user" (If: CONFIG_BLKIO)
The members of BlockdevOptionsVirtioBlkVhostVdpa when driver is "virtio-blk-vhost-vdpa" (If: CONFIG_BLKIO)
The members of BlockdevOptionsGenericCOWFormat when driver is "vmdk"
The members of BlockdevOptionsGenericFormat when driver is "vpc"
The members of BlockdevOptionsVVFAT when driver is "vvfat"

Since

2.9

BlockdevRef (Alternate)

Reference to a block device.

Members

definition: BlockdevOptions
defines a new block device inline

reference: string
references the ID of an existing block device

Since

2.9

BlockdevRefOrNull (Alternate)

Reference to a block device.

Members

definition: BlockdevOptions
defines a new block device inline

reference: string
references the ID of an existing block device. An empty string means that no block device should be referenced. Deprecated; use null instead.

null: null
No block device should be referenced (since 2.10)

Since

2.9

blockdev-add (Command)

Creates a new block device.

Arguments

The members of BlockdevOptions

Since

2.9

Examples

```
-> { "execute": "blockdev-add",
     "arguments": {
          "driver": "qcow2",
          "node-name": "test1",
          "file": {
              "driver": "file",
              "filename": "test.qcow2"
           }
      }
    }
<- { "return": {} }

-> { "execute": "blockdev-add",
     "arguments": {
          "driver": "qcow2",
          "node-name": "node0",
          "discard": "unmap",
          "cache": {
             "direct": true
           },
           "file": {
             "driver": "file",
             "filename": "/tmp/test.qcow2"
           },
           "backing": {
              "driver": "raw",
              "file": {
                 "driver": "file",
                 "filename": "/dev/fdset/4"
               }
           }
       }
     }

<- { "return": {} }
```

blockdev-reopen (Command)

Reopens one or more block devices using the given set of options. Any option not specified will be reset to its default value regardless of its previous status. If an option cannot be changed or a particular driver does not support reopening then the command will return an error. All devices in the list are reopened in one transaction, so if one of them fails then the whole transaction is cancelled.

The command receives a list of block devices to reopen. For each one of them, the top-level node-name option (from BlockdevOptions) must be specified and is used to select the block device to be reopened. Other node-name options must be either omitted or set to the current name of the appropriate node. This command won't change any node name and any attempt to do it will result in an error.

In the case of options that refer to child nodes, the behavior of this command depends on the value:

1) A set of options (BlockdevOptions): the child is reopened with the specified set of options.

2) A reference to the current child: the child is reopened using its existing set of options.

3) A reference to a different node: the current child is replaced with the specified one.

4) NULL: the current child (if any) is detached.

Options (1) and (2) are supported in all cases. Option (3) is supported for file and backing, and option (4) for backing only.

Unlike with blockdev-add, the backing option must always be present unless the node being reopened does not have a backing file and its image does not have a default backing file name as part of its metadata.

Arguments

options: array of BlockdevOptions
Not documented

Since

6.1

blockdev-del (Command)

Deletes a block device that has been added using blockdev-add. The command will fail if the node is attached to a device or is otherwise being used.

Arguments

node-name: string
Name of the graph node to delete.

Since

2.9

Example

```
-> { "execute": "blockdev-add",
     "arguments": {
          "driver": "qcow2",
          "node-name": "node0",
          "file": {
              "driver": "file",
              "filename": "test.qcow2"
          }
     }
   }
<- { "return": {} }

-> { "execute": "blockdev-del",
     "arguments": { "node-name": "node0" }
   }
<- { "return": {} }
```

BlockdevCreateOptionsFile (Object)

Driver specific image creation options for file.

Members

filename: string
Filename for the new image file

size: int
Size of the virtual disk in bytes

preallocation: PreallocMode (optional)
Preallocation mode for the new image (default: off; allowed values: off, falloc (if CONFIG_POSIX_FALLOCATE), full (if CONFIG_POSIX))

nocow: boolean (optional)
Turn off copy-on-write (valid only on btrfs; default: off)

extent-size-hint: int (optional)
Extent size hint to add to the image file; 0 for not adding an extent size hint (default: 1 MB, since 5.1)

Since

2.12

BlockdevCreateOptionsGluster (Object)

Driver specific image creation options for gluster.

Members

location: BlockdevOptionsGluster
Where to store the new image file

size: int
Size of the virtual disk in bytes

preallocation: PreallocMode (optional)
Preallocation mode for the new image (default: off; allowed values: off, falloc (if CONFIG_GLUSTERFS_FALLOCATE), full (if CONFIG_GLUSTERFS_ZEROFILL))

Since

2.12

BlockdevCreateOptionsLUKS (Object)

Driver specific image creation options for LUKS.

Members

file: BlockdevRef (optional)
Node to create the image format on, mandatory except when 'preallocation' is not requested

header: BlockdevRef (optional)
Block device holding a detached LUKS header. (since 9.0)

size: int
Size of the virtual disk in bytes

preallocation: PreallocMode (optional)
Preallocation mode for the new image (since: 4.2) (default: off; allowed values: off, metadata, falloc, full)

The members of QCryptoBlockCreateOptionsLUKS

Since

2.12

BlockdevCreateOptionsNfs (Object)

Driver specific image creation options for NFS.

Members

location: BlockdevOptionsNfs
Where to store the new image file

size: int
Size of the virtual disk in bytes

Since

2.12

BlockdevCreateOptionsParallels (Object)

Driver specific image creation options for parallels.

Members

file: BlockdevRef
Node to create the image format on

size: int
Size of the virtual disk in bytes

cluster-size: int (optional)
Cluster size in bytes (default: 1 MB)

Since

2.12

BlockdevCreateOptionsQcow (Object)

Driver specific image creation options for qcow.

Members

file: BlockdevRef
Node to create the image format on

size: int
Size of the virtual disk in bytes

backing-file: string (optional)
File name of the backing file if a backing file should be used

encrypt: QCryptoBlockCreateOptions (optional)
Encryption options if the image should be encrypted

Since

2.12

BlockdevQcow2Version (Enum)

Values

v2
The original QCOW2 format as introduced in qemu 0.10 (version 2)

v3
The extended QCOW2 format as introduced in qemu 1.1 (version 3)

Since

2.12

Qcow2CompressionType (Enum)

Compression type used in qcow2 image file

Values

zlib
zlib compression, see <http://zlib.net/>

zstd (If: CONFIG_ZSTD)
zstd compression, see <http://github.com/facebook/zstd>

Since

5.1

BlockdevCreateOptionsQcow2 (Object)

Driver specific image creation options for qcow2.

Members

file: BlockdevRef
Node to create the image format on

data-file: BlockdevRef (optional)
Node to use as an external data file in which all guest data is stored so that only metadata remains in the qcow2 file (since: 4.0)

data-file-raw: boolean (optional)
True if the external data file must stay valid as a standalone (read-only) raw image without looking at qcow2 metadata (default: false; since: 4.0)

extended-l2: boolean (optional)
True to make the image have extended L2 entries (default: false; since 5.2)

size: int
Size of the virtual disk in bytes

version: BlockdevQcow2Version (optional)
Compatibility level (default: v3)

backing-file: string (optional)
File name of the backing file if a backing file should be used

backing-fmt: BlockdevDriver (optional)
Name of the block driver to use for the backing file

encrypt: QCryptoBlockCreateOptions (optional)
Encryption options if the image should be encrypted

cluster-size: int (optional)
qcow2 cluster size in bytes (default: 65536)

preallocation: PreallocMode (optional)
Preallocation mode for the new image (default: off; allowed values: off, falloc, full, metadata)

lazy-refcounts: boolean (optional)
True if refcounts may be updated lazily (default: off)

refcount-bits: int (optional)
Width of reference counts in bits (default: 16)

compression-type: Qcow2CompressionType (optional)
The image cluster compression method (default: zlib, since 5.1)

Since

2.12

BlockdevCreateOptionsQed (Object)

Driver specific image creation options for qed.

Members

file: BlockdevRef
Node to create the image format on

size: int
Size of the virtual disk in bytes

backing-file: string (optional)
File name of the backing file if a backing file should be used

backing-fmt: BlockdevDriver (optional)
Name of the block driver to use for the backing file

cluster-size: int (optional)
Cluster size in bytes (default: 65536)

table-size: int (optional)
L1/L2 table size (in clusters)

Since

2.12

BlockdevCreateOptionsRbd (Object)

Driver specific image creation options for rbd/Ceph.

Members

location: BlockdevOptionsRbd
Where to store the new image file. This location cannot point to a snapshot.

size: int
Size of the virtual disk in bytes

cluster-size: int (optional)
RBD object size

encrypt: RbdEncryptionCreateOptions (optional)
Image encryption options. (Since 6.1)

Since

2.12

BlockdevVmdkSubformat (Enum)

Subformat options for VMDK images

Values

monolithicSparse
: Single file image with sparse cluster allocation

monolithicFlat
: Single flat data image and a descriptor file

twoGbMaxExtentSparse
: Data is split into 2GB (per virtual LBA) sparse extent files, in addition to a descriptor file

twoGbMaxExtentFlat
: Data is split into 2GB (per virtual LBA) flat extent files, in addition to a descriptor file

streamOptimized
: Single file image sparse cluster allocation, optimized for streaming over network.

Since

4.0

BlockdevVmdkAdapterType (Enum)

Adapter type info for VMDK images

Values

ide
: Not documented

buslogic
: Not documented

lsilogic
: Not documented

legacyESX
: Not documented

Since

4.0

BlockdevCreateOptionsVmdk (Object)

Driver specific image creation options for VMDK.

Members

file: BlockdevRef
Where to store the new image file. This refers to the image file for monolithcSparse and streamOptimized format, or the descriptor file for other formats.

size: int
Size of the virtual disk in bytes

extents: array of BlockdevRef (optional)
Where to store the data extents. Required for monolithcFlat, twoGbMaxExtentSparse and twoGbMaxExtentFlat formats. For monolithicFlat, only one entry is required; for twoGbMaxExtent* formats, the number of entries required is calculated as extent_number = virtual_size / 2GB. Providing more extents than will be used is an error.

subformat: BlockdevVmdkSubformat (optional)
The subformat of the VMDK image. Default: "monolithicSparse".

backing-file: string (optional)
The path of backing file. Default: no backing file is used.

adapter-type: BlockdevVmdkAdapterType (optional)
The adapter type used to fill in the descriptor. Default: ide.

hwversion: string (optional)
Hardware version. The meaningful options are "4" or "6". Default: "4".

toolsversion: string (optional)
VMware guest tools version. Default: "2147483647" (Since 6.2)

zeroed-grain: boolean (optional)
Whether to enable zeroed-grain feature for sparse subformats. Default: false.

Since

4.0

BlockdevCreateOptionsSsh (Object)

Driver specific image creation options for SSH.

Members

location: BlockdevOptionsSsh
Where to store the new image file

size: int
Size of the virtual disk in bytes

Since

2.12

BlockdevCreateOptionsVdi (Object)

Driver specific image creation options for VDI.

Members

file: BlockdevRef
Node to create the image format on

size: int
Size of the virtual disk in bytes

preallocation: PreallocMode (optional)
Preallocation mode for the new image (default: off; allowed values: off, metadata)

Since

2.12

BlockdevVhdxSubformat (Enum)

Values

dynamic
Growing image file

fixed
Preallocated fixed-size image file

Since

2.12

BlockdevCreateOptionsVhdx (Object)

Driver specific image creation options for vhdx.

Members

file: BlockdevRef
Node to create the image format on

size: int
Size of the virtual disk in bytes

log-size: int (optional)
Log size in bytes, must be a multiple of 1 MB (default: 1 MB)

block-size: int (optional)
Block size in bytes, must be a multiple of 1 MB and not larger than 256 MB (default: automatically choose a block size depending on the image size)

subformat: BlockdevVhdxSubformat (optional)
vhdx subformat (default: dynamic)

block-state-zero: boolean (optional)
Force use of payload blocks of type 'ZERO'. Non-standard, but default. Do not set to 'off' when using 'qemu-img convert' with subformat=dynamic.

Since

2.12

BlockdevVpcSubformat (Enum)

Values

dynamic
Growing image file

fixed
Preallocated fixed-size image file

Since

2.12

BlockdevCreateOptionsVpc (Object)

Driver specific image creation options for vpc (VHD).

Members

file: BlockdevRef
Node to create the image format on

size: int
Size of the virtual disk in bytes

subformat: BlockdevVpcSubformat (optional)
vhdx subformat (default: dynamic)

force-size: boolean (optional)
Force use of the exact byte size instead of rounding to the next size that can be represented in CHS geometry (default: false)

Since

2.12

BlockdevCreateOptions (Object)

Options for creating an image format on a given node.

Members

driver: BlockdevDriver
block driver to create the image format

The members of BlockdevCreateOptionsFile when driver is "file"
The members of BlockdevCreateOptionsGluster when driver is "gluster"
The members of BlockdevCreateOptionsLUKS when driver is "luks"
The members of BlockdevCreateOptionsNfs when driver is "nfs"
The members of BlockdevCreateOptionsParallels when driver is "parallels"
The members of BlockdevCreateOptionsQcow when driver is "qcow"
The members of BlockdevCreateOptionsQcow2 when driver is "qcow2"
The members of BlockdevCreateOptionsQed when driver is "qed"
The members of BlockdevCreateOptionsRbd when driver is "rbd"
The members of BlockdevCreateOptionsSsh when driver is "ssh"
The members of BlockdevCreateOptionsVdi when driver is "vdi"
The members of BlockdevCreateOptionsVhdx when driver is "vhdx"
The members of BlockdevCreateOptionsVmdk when driver is "vmdk"
The members of BlockdevCreateOptionsVpc when driver is "vpc"

Since

2.12

blockdev-create (Command)

Starts a job to create an image format on a given node. The job is automatically finalized, but a manual job-dismiss is required.

Arguments

job-id: string
Identifier for the newly created job.

options: BlockdevCreateOptions
Options for the image creation.

Since

3.0

BlockdevAmendOptionsLUKS (Object)

Driver specific image amend options for LUKS.

Members

The members of QCryptoBlockAmendOptionsLUKS

Since

5.1

BlockdevAmendOptionsQcow2 (Object)

Driver specific image amend options for qcow2. For now, only encryption options can be amended

Members

encrypt: QCryptoBlockAmendOptions (optional)
Encryption options to be amended

Since

5.1

BlockdevAmendOptions (Object)

Options for amending an image format

Members

driver: BlockdevDriver
Block driver of the node to amend.

The members of BlockdevAmendOptionsLUKS when driver is "luks"
The members of BlockdevAmendOptionsQcow2 when driver is "qcow2"

Since

5.1

x-blockdev-amend (Command)

Starts a job to amend format specific options of an existing open block device The job is automatically finalized, but a manual job-dismiss is required.

Arguments

job-id: string
Identifier for the newly created job.

node-name: string
Name of the block node to work on

options: BlockdevAmendOptions
Options (driver specific)

force: boolean (optional)
Allow unsafe operations, format specific For luks that allows erase of the last active keyslot (permanent loss of data), and replacement of an active keyslot (possible loss of data if IO error happens)

Features

unstable
This command is experimental.

Since

5.1

BlockErrorAction (Enum)

An enumeration of action that has been taken when a DISK I/O occurs

Values

ignore
error has been ignored

report
error has been reported to the device

stop
error caused VM to be stopped

Since

2.1

BLOCK_IMAGE_CORRUPTED (Event)

Emitted when a disk image is being marked corrupt. The image can be identified by its device or node name. The 'device' field is always present for compatibility reasons, but it can be empty ("") if the image does not have a device name associated.

Arguments

device: string
device name. This is always present for compatibility reasons, but it can be empty ("") if the image does not have a device name associated.

node-name: string (optional)
node name (Since: 2.4)

msg: string
informative message for human consumption, such as the kind of corruption being detected. It should not be parsed by machine as it is not guaranteed to be stable

offset: int (optional)
if the corruption resulted from an image access, this is the host's access offset into the image

size: int (optional)
if the corruption resulted from an image access, this is the access size

fatal: boolean
if set, the image is marked corrupt and therefore unusable after this event and must be repaired (Since 2.2; before, every BLOCK_IMAGE_CORRUPTED event was fatal)

Note

If action is "stop", a STOP event will eventually follow the BLOCK_IO_ERROR event.

Example

```
<- { "event": "BLOCK_IMAGE_CORRUPTED",
     "data": { "device": "", "node-name": "drive", "fatal": false,
               "msg": "L2 table offset 0x2a2a2a00 unaligned (L1 index: 0)" },
     "timestamp": { "seconds": 1648243240, "microseconds": 906060 } }
```

Since

1.7

BLOCK_IO_ERROR (Event)

Emitted when a disk I/O error occurs

Arguments

device: string
device name. This is always present for compatibility reasons, but it can be empty ("") if the image does not have a device name associated.

node-name: string (optional)
node name. Note that errors may be reported for the root node that is directly attached to a guest device rather than for the node where the error occurred. The node name is not present if the drive is empty. (Since: 2.8)

operation: IoOperationType
I/O operation

action: BlockErrorAction
action that has been taken

nospace: boolean (optional)
true if I/O error was caused due to a no-space condition. This key is only present if query-block's io-status is present, please see query-block documentation for more information (since: 2.2)

reason: string
human readable string describing the error cause. (This field is a debugging aid for humans, it should not be parsed by applications) (since: 2.2)

Note

If action is "stop", a STOP event will eventually follow the BLOCK_IO_ERROR event

Since

0.13

Example

```
<- { "event": "BLOCK_IO_ERROR",
     "data": { "device": "ide0-hd1",
               "node-name": "#block212",
               "operation": "write",
               "action": "stop",
               "reason": "No space left on device" },
     "timestamp": { "seconds": 1265044230, "microseconds": 450486 } }
```

BLOCK_JOB_COMPLETED (Event)

Emitted when a block job has completed

Arguments

type: JobType
job type

device: string
The job identifier. Originally the device name but other values are allowed since QEMU 2.7

len: int
maximum progress value

offset: int
current progress value. On success this is equal to len. On failure this is less than len

speed: int
rate limit, bytes per second

error: string (optional)
error message. Only present on failure. This field contains a human-readable error message. There are no semantics other than that streaming has failed and clients should not try to interpret the error string

Since

1.1

Example

```
<- { "event": "BLOCK_JOB_COMPLETED",
     "data": { "type": "stream", "device": "virtio-disk0",
               "len": 10737418240, "offset": 10737418240,
               "speed": 0 },
     "timestamp": { "seconds": 1267061043, "microseconds": 959568 } }
```

BLOCK_JOB_CANCELLED (Event)

Emitted when a block job has been cancelled

Arguments

type: JobType
job type

device: string
The job identifier. Originally the device name but other values are allowed since QEMU 2.7

len: int
maximum progress value

offset: int
current progress value. On success this is equal to len. On failure this is less than len

speed: int
rate limit, bytes per second

Since

1.1

Example

```
<- { "event": "BLOCK_JOB_CANCELLED",
     "data": { "type": "stream", "device": "virtio-disk0",
               "len": 10737418240, "offset": 134217728,
               "speed": 0 },
     "timestamp": { "seconds": 1267061043, "microseconds": 959568 } }
```

BLOCK_JOB_ERROR (Event)

Emitted when a block job encounters an error

Arguments

device: string
The job identifier. Originally the device name but other values are allowed since QEMU 2.7

operation: IoOperationType
I/O operation

action: BlockErrorAction
action that has been taken

Since

1.3

Example

```
<- { "event": "BLOCK_JOB_ERROR",
     "data": { "device": "ide0-hd1",
               "operation": "write",
               "action": "stop" },
     "timestamp": { "seconds": 1265044230, "microseconds": 450486 } }
```

BLOCK_JOB_READY (Event)

Emitted when a block job is ready to complete

Arguments

type: JobType
job type

device: string
The job identifier. Originally the device name but other values are allowed since QEMU 2.7

len: int
maximum progress value

offset: int
current progress value. On success this is equal to len. On failure this is less than len

speed: int
rate limit, bytes per second

Note

The "ready to complete" status is always reset by a BLOCK_JOB_ERROR event

Since

1.3

Example

```
<- { "event": "BLOCK_JOB_READY",
     "data": { "device": "drive0", "type": "mirror", "speed": 0,
               "len": 2097152, "offset": 2097152 },
     "timestamp": { "seconds": 1265044230, "microseconds": 450486 } }
```

BLOCK_JOB_PENDING (Event)

Emitted when a block job is awaiting explicit authorization to finalize graph changes via block-job-finalize. If this job is part of a transaction, it will not emit this event until the transaction has converged first.

Arguments

type: JobType
job type

id: string
The job identifier.

Since

2.12

Example

```
<- { "event": "BLOCK_JOB_PENDING",
     "data": { "type": "mirror", "id": "backup_1" },
     "timestamp": { "seconds": 1265044230, "microseconds": 450486 } }
```

PreallocMode (Enum)

Preallocation mode of QEMU image file

Values

off
no preallocation

metadata
preallocate only for metadata

falloc
like `full` preallocation but allocate disk space by posix_fallocate() rather than writing data.

full
preallocate all data by writing it to the device to ensure disk space is really available. This data may or may not be zero, depending on the image format and storage. `full` preallocation also sets up metadata correctly.

Since

2.2

BLOCK_WRITE_THRESHOLD (Event)

Emitted when writes on block device reaches or exceeds the configured write threshold. For thin-provisioned devices, this means the device should be extended to avoid pausing for disk exhaustion. The event is one shot. Once triggered, it needs to be re-registered with another block-set-write-threshold command.

Arguments

node-name: string
graph node name on which the threshold was exceeded.

amount-exceeded: int
amount of data which exceeded the threshold, in bytes.

write-threshold: int
last configured threshold, in bytes.

Since

2.3

block-set-write-threshold (Command)

Change the write threshold for a block drive. An event will be delivered if a write to this block drive crosses the configured threshold. The threshold is an offset, thus must be non-negative. Default is no write threshold. Setting the threshold to zero disables it.

This is useful to transparently resize thin-provisioned drives without the guest OS noticing.

Arguments

node-name: string
graph node name on which the threshold must be set.

write-threshold: int
configured threshold for the block device, bytes. Use 0 to disable the threshold.

Since

2.3

Example

```
-> { "execute": "block-set-write-threshold",
     "arguments": { "node-name": "mydev",
                    "write-threshold": 17179869184 } }
<- { "return": {} }
```

x-blockdev-change (Command)

Dynamically reconfigure the block driver state graph. It can be used to add, remove, insert or replace a graph node. Currently only the Quorum driver implements this feature to add or remove its child. This is useful to fix a broken quorum child.

If `node` is specified, it will be inserted under `parent`. `child` may not be specified in this case. If both `parent` and `child` are specified but `node` is not, `child` will be detached from `parent`.

Arguments

parent: string
the id or name of the parent node.

child: string (optional)
the name of a child under the given parent node.

node: string (optional)
the name of the node that will be added.

Features

unstable

This command is experimental, and its API is not stable. It does not support all kinds of operations, all kinds of children, nor all block drivers.

FIXME Removing children from a quorum node means introducing gaps in the child indices. This cannot be represented in the 'children' list of BlockdevOptionsQuorum, as returned by .bdrv_refresh_filename().

Warning: The data in a new quorum child MUST be consistent with that of the rest of the array.

Since

2.7

Examples

```
1. Add a new node to a quorum

-> { "execute": "blockdev-add",
     "arguments": {
         "driver": "raw",
         "node-name": "new_node",
         "file": { "driver": "file",
                   "filename": "test.raw" } } }
<- { "return": {} }
-> { "execute": "x-blockdev-change",
     "arguments": { "parent": "disk1",
                    "node": "new_node" } }
<- { "return": {} }

2. Delete a quorum's node

-> { "execute": "x-blockdev-change",
     "arguments": { "parent": "disk1",
                    "child": "children.1" } }
<- { "return": {} }
```

x-blockdev-set-iothread (Command)

Move `node` and its children into the `iothread`. If `iothread` is null then move `node` and its children into the main loop.

The node must not be attached to a BlockBackend.

Arguments

node-name: string
the name of the block driver node

iothread: StrOrNull
the name of the IOThread object or null for the main loop

force: boolean (optional)
true if the node and its children should be moved when a BlockBackend is already attached

Features

unstable
This command is experimental and intended for test cases that need control over IOThreads only.

Since

2.12

Examples

```
1. Move a node into an IOThread

-> { "execute": "x-blockdev-set-iothread",
     "arguments": { "node-name": "disk1",
                    "iothread": "iothread0" } }
<- { "return": {} }

2. Move a node into the main loop

-> { "execute": "x-blockdev-set-iothread",
     "arguments": { "node-name": "disk1",
                    "iothread": null } }
<- { "return": {} }
```

QuorumOpType (Enum)

An enumeration of the quorum operation types

Values

read
read operation

write
write operation

flush
flush operation

Since

2.6

QUORUM_FAILURE (Event)

Emitted by the Quorum block driver if it fails to establish a quorum

Arguments

reference: string
device name if defined else node name

sector-num: int
number of the first sector of the failed read operation

sectors-count: int
failed read operation sector count

Note

This event is rate-limited.

Since

2.0

Example

```
<- { "event": "QUORUM_FAILURE",
     "data": { "reference": "usr1", "sector-num": 345435, "sectors-count": 5 },
     "timestamp": { "seconds": 1344522075, "microseconds": 745528 } }
```

QUORUM_REPORT_BAD (Event)

Emitted to report a corruption of a Quorum file

Arguments

type: QuorumOpType
quorum operation type (Since 2.6)

error: string (optional)
error message. Only present on failure. This field contains a human-readable error message. There are no semantics other than that the block layer reported an error and clients should not try to interpret the error string.

node-name: string
the graph node name of the block driver state

sector-num: int
number of the first sector of the failed read operation

sectors-count: int
failed read operation sector count

Note

This event is rate-limited.

Since

2.0

Examples

```
1. Read operation

<- { "event": "QUORUM_REPORT_BAD",
     "data": { "node-name": "node0", "sector-num": 345435, "sectors-count": 5,
               "type": "read" },
     "timestamp": { "seconds": 1344522075, "microseconds": 745528 } }

2. Flush operation

<- { "event": "QUORUM_REPORT_BAD",
     "data": { "node-name": "node0", "sector-num": 0, "sectors-count": 2097120,
               "type": "flush", "error": "Broken pipe" },
     "timestamp": { "seconds": 1456406829, "microseconds": 291763 } }
```

BlockdevSnapshotInternal (Object)

Members

device: string
the device name or node-name of a root node to generate the snapshot from

name: string
the name of the internal snapshot to be created

Notes

In transaction, if name is empty, or any snapshot matching name exists, the operation will fail. Only some image formats support it, for example, qcow2, and rbd.

Since

1.7

blockdev-snapshot-internal-sync (Command)

Synchronously take an internal snapshot of a block device, when the format of the image used supports it. If the name is an empty string, or a snapshot with name already exists, the operation will fail.

For the arguments, see the documentation of BlockdevSnapshotInternal.

Errors

- If device is not a valid block device, GenericError
- If any snapshot matching name exists, or name is empty, GenericError
- If the format of the image used does not support it, GenericError

Since

1.7

Example

```
-> { "execute": "blockdev-snapshot-internal-sync",
     "arguments": { "device": "ide-hd0",
                    "name": "snapshot0" }
   }
<- { "return": {} }
```

blockdev-snapshot-delete-internal-sync (Command)

Synchronously delete an internal snapshot of a block device, when the format of the image used support it. The snapshot is identified by name or id or both. One of the name or id is required. Return SnapshotInfo for the successfully deleted snapshot.

Arguments

device: string
: the device name or node-name of a root node to delete the snapshot from

id: string (optional)
: optional the snapshot's ID to be deleted

name: string (optional)
: optional the snapshot's name to be deleted

Returns

SnapshotInfo

Errors

- If `device` is not a valid block device, GenericError
- If snapshot not found, GenericError
- If the format of the image used does not support it, GenericError
- If `id` and `name` are both not specified, GenericError

Since

1.7

Example

```
-> { "execute": "blockdev-snapshot-delete-internal-sync",
     "arguments": { "device": "ide-hd0",
                    "name": "snapshot0" }
   }
<- { "return": {
                    "id": "1",
                    "name": "snapshot0",
                    "vm-state-size": 0,
                    "date-sec": 1000012,
                    "date-nsec": 10,
                    "vm-clock-sec": 100,
                    "vm-clock-nsec": 20,
                    "icount": 220414
```

(continues on next page)

(continued from previous page)

```
    }
  }
```

DummyBlockCoreForceArrays (Object)

Not used by QMP; hack to let us use BlockGraphInfoList internally

Members

unused-block-graph-info: array of BlockGraphInfo
Not documented

Since

8.0

Block device exports

NbdServerOptions (Object)

Keep this type consistent with the nbd-server-start arguments. The only intended difference is using SocketAddress instead of SocketAddressLegacy.

Members

addr: SocketAddress
Address on which to listen.

tls-creds: string (optional)
ID of the TLS credentials object (since 2.6).

tls-authz: string (optional)
ID of the QAuthZ authorization object used to validate the client's x509 distinguished name. This object is is only resolved at time of use, so can be deleted and recreated on the fly while the NBD server is active. If missing, it will default to denying access (since 4.0).

max-connections: int (optional)
The maximum number of connections to allow at the same time, 0 for unlimited. Setting this to 1 also stops the server from advertising multiple client support (since 5.2; default: 0)

Since

4.2

nbd-server-start (Command)

Start an NBD server listening on the given host and port. Block devices can then be exported using nbd-server-add. The NBD server will present them as named exports; for example, another QEMU instance could refer to them as "nbd:HOST:PORT:exportname=NAME".

Keep this type consistent with the NbdServerOptions type. The only intended difference is using SocketAddressLegacy instead of SocketAddress.

Arguments

addr: SocketAddressLegacy
Address on which to listen.

tls-creds: string (optional)
ID of the TLS credentials object (since 2.6).

tls-authz: string (optional)
ID of the QAuthZ authorization object used to validate the client's x509 distinguished name. This object is is only resolved at time of use, so can be deleted and recreated on the fly while the NBD server is active. If missing, it will default to denying access (since 4.0).

max-connections: int (optional)
The maximum number of connections to allow at the same time, 0 for unlimited. Setting this to 1 also stops the server from advertising multiple client support (since 5.2; default: 0).

Errors

- if the server is already running

Since

1.3

BlockExportOptionsNbdBase (Object)

An NBD block export (common options shared between nbd-server-add and the NBD branch of block-export-add).

Members

name: string (optional)
Export name. If unspecified, the device parameter is used as the export name. (Since 2.12)

description: string (optional)
Free-form description of the export, up to 4096 bytes. (Since 5.0)

Since

5.0

BlockExportOptionsNbd (Object)

An NBD block export (distinct options used in the NBD branch of block-export-add).

Members

bitmaps: array of BlockDirtyBitmapOrStr (optional)
Also export each of the named dirty bitmaps reachable from device, so the NBD client can use NBD_OPT_SET_META_CONTEXT with the metadata context name "qemu:dirty-bitmap:BITMAP" to inspect each bitmap. Since 7.1 bitmap may be specified by node/name pair.

allocation-depth: boolean (optional)
Also export the allocation depth map for device, so the NBD client can use NBD_OPT_SET_META_CONTEXT with the metadata context name "qemu:allocation-depth" to inspect allocation details. (since 5.2)

The members of BlockExportOptionsNbdBase

Since

5.2

BlockExportOptionsVhostUserBlk (Object)

A vhost-user-blk block export.

Members

addr: SocketAddress
The vhost-user socket on which to listen. Both 'unix' and 'fd' SocketAddress types are supported. Passed fds must be UNIX domain sockets.

logical-block-size: int (optional)
Logical block size in bytes. Defaults to 512 bytes.

num-queues: int (optional)
Number of request virtqueues. Must be greater than 0. Defaults to 1.

Since

5.2

FuseExportAllowOther (Enum)

Possible allow_other modes for FUSE exports.

Values

off
: Do not pass allow_other as a mount option.

on
: Pass allow_other as a mount option.

auto
: Try mounting with allow_other first, and if that fails, retry without allow_other.

Since

6.1

BlockExportOptionsFuse (Object)

Options for exporting a block graph node on some (file) mountpoint as a raw image.

Members

mountpoint: string
: Path on which to export the block device via FUSE. This must point to an existing regular file.

growable: boolean (optional)
: Whether writes beyond the EOF should grow the block node accordingly. (default: false)

allow-other: FuseExportAllowOther (optional)
: If this is off, only qemu's user is allowed access to this export. That cannot be changed even with chmod or chown. Enabling this option will allow other users access to the export with the FUSE mount option "allow_other". Note that using allow_other as a non-root user requires user_allow_other to be enabled in the global fuse.conf configuration file. In auto mode (the default), the FUSE export driver will first attempt to mount the export with allow_other, and if that fails, try again without. (since 6.1; default: auto)

Since

6.0

If

CONFIG_FUSE

BlockExportOptionsVduseBlk (Object)

A vduse-blk block export.

Members

name: string
the name of VDUSE device (must be unique across the host).

num-queues: int (optional)
the number of virtqueues. Defaults to 1.

queue-size: int (optional)
the size of virtqueue. Defaults to 256.

logical-block-size: int (optional)
Logical block size in bytes. Range [512, PAGE_SIZE] and must be power of 2. Defaults to 512 bytes.

serial: string (optional)
the serial number of virtio block device. Defaults to empty string.

Since

7.1

NbdServerAddOptions (Object)

An NBD block export, per legacy nbd-server-add command.

Members

device: string
The device name or node name of the node to be exported

writable: boolean (optional)
Whether clients should be able to write to the device via the NBD connection (default false).

bitmap: string (optional)
Also export a single dirty bitmap reachable from `device`, so the NBD client can use NBD_OPT_SET_META_CONTEXT with the metadata context name "qemu:dirty-bitmap:BITMAP" to inspect the bitmap (since 4.0).

The members of BlockExportOptionsNbdBase

Since

5.0

nbd-server-add (Command)

Export a block node to QEMU's embedded NBD server.

The export name will be used as the id for the resulting block export.

Arguments

The members of NbdServerAddOptions

Features

deprecated
This command is deprecated. Use block-export-add instead.

Errors

- if the server is not running
- if an export with the same name already exists

Since

1.3

BlockExportRemoveMode (Enum)

Mode for removing a block export.

Values

safe
Remove export if there are no existing connections, fail otherwise.

hard
Drop all connections immediately and remove export.

Since

2.12

nbd-server-remove (Command)

Remove NBD export by name.

Arguments

name: string
Block export id.

mode: BlockExportRemoveMode (optional)
Mode of command operation. See BlockExportRemoveMode description. Default is 'safe'.

Features

deprecated
This command is deprecated. Use block-export-del instead.

Errors

- if the server is not running
- if export is not found
- if mode is 'safe' and there are existing connections

Since

2.12

nbd-server-stop (Command)

Stop QEMU's embedded NBD server, and unregister all devices previously added via nbd-server-add.

Since

1.3

BlockExportType (Enum)

An enumeration of block export types

Values

nbd
NBD export

vhost-user-blk (If: CONFIG_VHOST_USER_BLK_SERVER)
vhost-user-blk export (since 5.2)

fuse (If: CONFIG_FUSE)
FUSE export (since: 6.0)

vduse-blk (If: CONFIG_VDUSE_BLK_EXPORT)
vduse-blk export (since 7.1)

Since

4.2

BlockExportOptions (Object)

Describes a block export, i.e. how single node should be exported on an external interface.

Members

type: BlockExportType
Block export type

id: string
A unique identifier for the block export (across all export types)

node-name: string
The node name of the block node to be exported (since: 5.2)

writable: boolean (optional)
True if clients should be able to write to the export (default false)

writethrough: boolean (optional)
If true, caches are flushed after every write request to the export before completion is signalled. (since: 5.2; default: false)

iothread: string (optional)
The name of the iothread object where the export will run. The default is to use the thread currently associated with the block node. (since: 5.2)

fixed-iothread: boolean (optional)
True prevents the block node from being moved to another thread while the export is active. If true and `iothread` is given, export creation fails if the block node cannot be moved to the iothread. The default is false. (since: 5.2)

The members of BlockExportOptionsNbd when type is "nbd"
The members of BlockExportOptionsVhostUserBlk when type is "vhost-user-blk" (If: CONFIG_VHOST_USER_BLK_SERVER)
The members of BlockExportOptionsFuse when type is "fuse" (If: CONFIG_FUSE)
The members of BlockExportOptionsVduseBlk when type is "vduse-blk" (If: CONFIG_VDUSE_BLK_EXPORT)

Since

4.2

block-export-add (Command)

Creates a new block export.

Arguments

The members of BlockExportOptions

Since

5.2

block-export-del (Command)

Request to remove a block export. This drops the user's reference to the export, but the export may still stay around after this command returns until the shutdown of the export has completed.

Arguments

id: string
Block export id.

mode: BlockExportRemoveMode (optional)
Mode of command operation. See BlockExportRemoveMode description. Default is 'safe'.

Errors

- if the export is not found
- if mode is 'safe' and the export is still in use (e.g. by existing client connections)

Since

5.2

BLOCK_EXPORT_DELETED (Event)

Emitted when a block export is removed and its id can be reused.

Arguments

id: string
Block export id.

Since

5.2

BlockExportInfo (Object)

Information about a single block export.

Members

id: string
The unique identifier for the block export

type: BlockExportType
The block export type

node-name: string
The node name of the block node that is exported

shutting-down: boolean
True if the export is shutting down (e.g. after a block-export-del command, but before the shutdown has completed)

Since

5.2

query-block-exports (Command)

Returns

A list of BlockExportInfo describing all block exports

Since

5.2

5.12.6 Character devices

ChardevInfo (Object)

Information about a character device.

Members

label: string
: the label of the character device

filename: string
: the filename of the character device

frontend-open: boolean
: shows whether the frontend device attached to this backend (e.g. with the chardev=... option) is in open or closed state (since 2.1)

Notes

filename is encoded using the QEMU command line character device encoding. See the QEMU man page for details.

Since

0.14

query-chardev (Command)

Returns information about current character devices.

Returns

a list of `ChardevInfo`

Since

0.14

Example

```
-> { "execute": "query-chardev" }
<- {
      "return": [
         {
            "label": "charchannel0",
            "filename": "unix:/var/lib/libvirt/qemu/seabios.rhel6.agent,server=on",
            "frontend-open": false
         },
         {
            "label": "charmonitor",
            "filename": "unix:/var/lib/libvirt/qemu/seabios.rhel6.monitor,server=on",
            "frontend-open": true
         },
         {
            "label": "charserial0",
            "filename": "pty:/dev/pts/2",
            "frontend-open": true
         }
      ]
   }
```

`ChardevBackendInfo` (Object)

Information about a character device backend

Members

`name: string`
The backend name

Since

2.0

query-chardev-backends (Command)

Returns information about character device backends.

Returns

a list of `ChardevBackendInfo`

Since

2.0

Example

```
-> { "execute": "query-chardev-backends" }
<- {
      "return":[
         {
            "name":"udp"
         },
         {
            "name":"tcp"
         },
         {
            "name":"unix"
         },
         {
            "name":"spiceport"
         }
      ]
   }
```

DataFormat (Enum)

An enumeration of data format.

Values

utf8
Data is a UTF-8 string (RFC 3629)

base64
Data is Base64 encoded binary (RFC 3548)

Since

1.4

ringbuf-write (Command)

Write to a ring buffer character device.

Arguments

device: string
the ring buffer character device name

data: string
data to write

format: DataFormat (optional)
data encoding (default 'utf8').

- base64: data must be base64 encoded text. Its binary decoding gets written.
- utf8: data's UTF-8 encoding is written
- data itself is always Unicode regardless of format, like any other string.

Since

1.4

Example

```
-> { "execute": "ringbuf-write",
     "arguments": { "device": "foo",
                    "data": "abcdefgh",
                    "format": "utf8" } }
<- { "return": {} }
```

ringbuf-read (Command)

Read from a ring buffer character device.

Arguments

device: string
the ring buffer character device name

size: int
how many bytes to read at most

format: DataFormat (optional)
data encoding (default 'utf8').

- base64: the data read is returned in base64 encoding.
- utf8: the data read is interpreted as UTF-8. Bug: can screw up when the buffer contains invalid UTF-8 sequences, NUL characters, after the ring buffer lost data, and when reading stops because the size limit is reached.
- The return value is always Unicode regardless of format, like any other string.

Returns

data read from the device

Since

1.4

Example

```
-> { "execute": "ringbuf-read",
     "arguments": { "device": "foo",
                    "size": 1000,
                    "format": "utf8" } }
<- { "return": "abcdefgh" }
```

ChardevCommon (Object)

Configuration shared across all chardev backends

Members

logfile: string (optional)
The name of a logfile to save output

logappend: boolean (optional)
true to append instead of truncate (default to false to truncate)

Since

2.6

ChardevFile (Object)

Configuration info for file chardevs.

Members

in: string (optional)
The name of the input file

out: string
The name of the output file

append: boolean (optional)
Open the file in append mode (default false to truncate) (Since 2.6)

The members of ChardevCommon

Since

1.4

ChardevHostdev (Object)

Configuration info for device and pipe chardevs.

Members

device: string
The name of the special file for the device, i.e. /dev/ttyS0 on Unix or COM1: on Windows

The members of ChardevCommon

Since

1.4

ChardevSocket (Object)

Configuration info for (stream) socket chardevs.

Members

addr: SocketAddressLegacy
socket address to listen on (server=true) or connect to (server=false)

tls-creds: string (optional)
the ID of the TLS credentials object (since 2.6)

tls-authz: string (optional)
the ID of the QAuthZ authorization object against which the client's x509 distinguished name will be validated. This object is only resolved at time of use, so can be deleted and recreated on the fly while the chardev server is active. If missing, it will default to denying access (since 4.0)

server: boolean (optional)
create server socket (default: true)

wait: boolean (optional)
wait for incoming connection on server sockets (default: false). Silently ignored with server: false. This use is deprecated.

nodelay: boolean (optional)
set TCP_NODELAY socket option (default: false)

telnet: boolean (optional)
enable telnet protocol on server sockets (default: false)

tn3270: boolean (optional)
enable tn3270 protocol on server sockets (default: false) (Since: 2.10)

websocket: boolean (optional)
enable websocket protocol on server sockets (default: false) (Since: 3.1)

reconnect: int (optional)
For a client socket, if a socket is disconnected, then attempt a reconnect after the given number of seconds. Setting this to zero disables this function. (default: 0) (Since: 2.2)

The members of ChardevCommon

Since

1.4

ChardevUdp (Object)

Configuration info for datagram socket chardevs.

Members

remote: SocketAddressLegacy
remote address

local: SocketAddressLegacy (optional)
local address

The members of ChardevCommon

Since

1.5

ChardevMux (Object)

Configuration info for mux chardevs.

Members

chardev: string
name of the base chardev.

The members of ChardevCommon

Since

1.5

ChardevStdio (Object)

Configuration info for stdio chardevs.

Members

signal: boolean (optional)
Allow signals (such as SIGINT triggered by ^C) be delivered to qemu. Default: true.

The members of ChardevCommon

Since

1.5

ChardevSpiceChannel (Object)

Configuration info for spice vm channel chardevs.

Members

type: string
kind of channel (for example vdagent).

The members of ChardevCommon

Since

1.5

If

CONFIG_SPICE

ChardevSpicePort (Object)

Configuration info for spice port chardevs.

Members

fqdn: string
name of the channel (see docs/spice-port-fqdn.txt)

The members of ChardevCommon

Since

1.5

If

CONFIG_SPICE

ChardevDBus (Object)

Configuration info for DBus chardevs.

Members

name: string
name of the channel (following docs/spice-port-fqdn.txt)

The members of ChardevCommon

Since

7.0

If

CONFIG_DBUS_DISPLAY

ChardevVC (Object)

Configuration info for virtual console chardevs.

Members

width: int (optional)
console width, in pixels

height: int (optional)
console height, in pixels

cols: int (optional)
console width, in chars

rows: int (optional)
console height, in chars

The members of ChardevCommon

Note

the options are only effective when the VNC or SDL graphical display backend is active. They are ignored with the GTK, Spice, VNC and D-Bus display backends.

Since

1.5

ChardevRingbuf (Object)

Configuration info for ring buffer chardevs.

Members

size: int (optional)
ring buffer size, must be power of two, default is 65536

The members of ChardevCommon

Since

1.5

ChardevQemuVDAgent (Object)

Configuration info for qemu vdagent implementation.

Members

mouse: boolean (optional)
enable/disable mouse, default is enabled.

clipboard: boolean (optional)
enable/disable clipboard, default is disabled.

The members of ChardevCommon

Since

6.1

If

CONFIG_SPICE_PROTOCOL

ChardevBackendKind (Enum)

Values

pipe
Since 1.5

udp
Since 1.5

mux
Since 1.5

msmouse
Since 1.5

wctablet
Since 2.9

braille (**If:** CONFIG_BRLAPI)
Since 1.5

testdev
Since 2.2

stdio
Since 1.5

console (**If:** CONFIG_WIN32)
Since 1.5

spicevmc (**If:** CONFIG_SPICE)
Since 1.5

spiceport (**If:** CONFIG_SPICE)
Since 1.5

qemu-vdagent (**If:** CONFIG_SPICE_PROTOCOL)
Since 6.1

dbus (**If:** CONFIG_DBUS_DISPLAY)
Since 7.0

vc
v1.5

ringbuf
Since 1.6

memory
Since 1.5

file
Not documented

serial (**If:** HAVE_CHARDEV_SERIAL)
Not documented

parallel (**If:** HAVE_CHARDEV_PARALLEL)
Not documented

socket
Not documented

pty
Not documented

null
Not documented

Features

deprecated
Member `memory` is deprecated. Use `ringbuf` instead.

Since

1.4

`ChardevFileWrapper` (Object)

Members

data: ChardevFile
Configuration info for file chardevs

Since

1.4

`ChardevHostdevWrapper` (Object)

Members

data: ChardevHostdev
Configuration info for device and pipe chardevs

Since

1.4

`ChardevSocketWrapper` (Object)

Members

data: ChardevSocket
Configuration info for (stream) socket chardevs

Since

1.4

ChardevUdpWrapper (Object)

Members

data: ChardevUdp
Configuration info for datagram socket chardevs

Since

1.5

ChardevCommonWrapper (Object)

Members

data: ChardevCommon
Configuration shared across all chardev backends

Since

2.6

ChardevMuxWrapper (Object)

Members

data: ChardevMux
Configuration info for mux chardevs

Since

1.5

ChardevStdioWrapper (Object)

Members

data: ChardevStdio
Configuration info for stdio chardevs

Since

1.5

ChardevSpiceChannelWrapper (Object)

Members

data: ChardevSpiceChannel
Configuration info for spice vm channel chardevs

Since

1.5

If

CONFIG_SPICE

ChardevSpicePortWrapper (Object)

Members

data: ChardevSpicePort
Configuration info for spice port chardevs

Since

1.5

If

CONFIG_SPICE

ChardevQemuVDAgentWrapper (Object)

Members

data: ChardevQemuVDAgent
Configuration info for qemu vdagent implementation

Since

6.1

If

CONFIG_SPICE_PROTOCOL

ChardevDBusWrapper (Object)

Members

data: ChardevDBus
Configuration info for DBus chardevs

Since

7.0

If

CONFIG_DBUS_DISPLAY

ChardevVCWrapper (Object)

Members

data: ChardevVC
Configuration info for virtual console chardevs

Since

1.5

ChardevRingbufWrapper (Object)

Members

data: ChardevRingbuf
Configuration info for ring buffer chardevs

Since

1.5

ChardevBackend (Object)

Configuration info for the new chardev backend.

Members

type: ChardevBackendKind
backend type

The members of ChardevFileWrapper when type is "file"
The members of ChardevHostdevWrapper when type is "serial" (If: HAVE_CHARDEV_SERIAL)
The members of ChardevHostdevWrapper when type is "parallel" (If: HAVE_CHARDEV_PARALLEL)
The members of ChardevHostdevWrapper when type is "pipe"
The members of ChardevSocketWrapper when type is "socket"
The members of ChardevUdpWrapper when type is "udp"
The members of ChardevCommonWrapper when type is "pty"
The members of ChardevCommonWrapper when type is "null"
The members of ChardevMuxWrapper when type is "mux"
The members of ChardevCommonWrapper when type is "msmouse"
The members of ChardevCommonWrapper when type is "wctablet"
The members of ChardevCommonWrapper when type is "braille" (If: CONFIG_BRLAPI)
The members of ChardevCommonWrapper when type is "testdev"
The members of ChardevStdioWrapper when type is "stdio"
The members of ChardevCommonWrapper when type is "console" (If: CONFIG_WIN32)
The members of ChardevSpiceChannelWrapper when type is "spicevmc" (If: CONFIG_SPICE)
The members of ChardevSpicePortWrapper when type is "spiceport" (If: CONFIG_SPICE)
The members of ChardevQemuVDAgentWrapper when type is "qemu-vdagent" (If: CONFIG_SPICE_PROTOCOL)
The members of ChardevDBusWrapper when type is "dbus" (If: CONFIG_DBUS_DISPLAY)
The members of ChardevVCWrapper when type is "vc"
The members of ChardevRingbufWrapper when type is "ringbuf"
The members of ChardevRingbufWrapper when type is "memory"

Since

1.4

ChardevReturn (Object)

Return info about the chardev backend just created.

Members

pty: string (optional)
name of the slave pseudoterminal device, present if and only if a chardev of type 'pty' was created

Since

1.4

chardev-add (Command)

Add a character device backend

Arguments

id: string
the chardev's ID, must be unique

backend: ChardevBackend
backend type and parameters

Returns

ChardevReturn.

Since

1.4

Examples

```
-> { "execute" : "chardev-add",
     "arguments" : { "id" : "foo",
                     "backend" : { "type" : "null", "data" : {} } } }
<- { "return": {} }

-> { "execute" : "chardev-add",
     "arguments" : { "id" : "bar",
                     "backend" : { "type" : "file",
                                   "data" : { "out" : "/tmp/bar.log" } } } }
<- { "return": {} }

-> { "execute" : "chardev-add",
     "arguments" : { "id" : "baz",
                     "backend" : { "type" : "pty", "data" : {} } } }
<- { "return": { "pty" : "/dev/pty/42" } }
```

chardev-change (Command)

Change a character device backend

Arguments

id: string
: the chardev's ID, must exist

backend: ChardevBackend
: new backend type and parameters

Returns

ChardevReturn.

Since

2.10

Examples

```
-> { "execute" : "chardev-change",
     "arguments" : { "id" : "baz",
                     "backend" : { "type" : "pty", "data" : {} } } }
<- { "return": { "pty" : "/dev/pty/42" } }

-> {"execute" : "chardev-change",
    "arguments" : {
        "id" : "charchannel2",
        "backend" : {
            "type" : "socket",
            "data" : {
                "addr" : {
                    "type" : "unix" ,
                    "data" : {
                        "path" : "/tmp/charchannel2.socket"
                    }
                },
                "server" : true,
                "wait" : false }}}}
<- {"return": {}}
```

chardev-remove (Command)

Remove a character device backend

Arguments

id: string
: the chardev's ID, must exist and not be in use

Since

1.4

Example

```
-> { "execute": "chardev-remove", "arguments": { "id" : "foo" } }
<- { "return": {} }
```

chardev-send-break (Command)

Send a break to a character device

Arguments

id: string
: the chardev's ID, must exist

Since

2.10

Example

```
-> { "execute": "chardev-send-break", "arguments": { "id" : "foo" } }
<- { "return": {} }
```

VSERPORT_CHANGE (Event)

Emitted when the guest opens or closes a virtio-serial port.

Arguments

id: string
: device identifier of the virtio-serial port

open: boolean
: true if the guest has opened the virtio-serial port

Note

This event is rate-limited.

Since

2.1

Example

```
<- { "event": "VSERPORT_CHANGE",
     "data": { "id": "channel0", "open": true },
     "timestamp": { "seconds": 1401385907, "microseconds": 422329 } }
```

5.12.7 User authorization

QAuthZListPolicy (Enum)

The authorization policy result

Values

deny
: deny access

allow
: allow access

Since

4.0

QAuthZListFormat (Enum)

The authorization policy match format

Values

exact
: an exact string match

glob
: string with ? and * shell wildcard support

Since

4.0

QAuthZListRule (Object)

A single authorization rule.

Members

match: string
: a string or glob to match against a user identity

policy: QAuthZListPolicy
: the result to return if `match` evaluates to true

format: QAuthZListFormat (optional)
: the format of the `match` rule (default 'exact')

Since

4.0

AuthZListProperties (Object)

Properties for authz-list objects.

Members

policy: QAuthZListPolicy (optional)
Default policy to apply when no rule matches (default: deny)

rules: array of QAuthZListRule (optional)
Authorization rules based on matching user

Since

4.0

AuthZListFileProperties (Object)

Properties for authz-listfile objects.

Members

filename: string
File name to load the configuration from. The file must contain valid JSON for AuthZListProperties.

refresh: boolean (optional)
If true, inotify is used to monitor the file, automatically reloading changes. If an error occurs during reloading, all authorizations will fail until the file is next successfully loaded. (default: true if the binary was built with CONFIG_INOTIFY1, false otherwise)

Since

4.0

AuthZPAMProperties (Object)

Properties for authz-pam objects.

Members

service: string
PAM service name to use for authorization

Since

4.0

AuthZSimpleProperties (Object)

Properties for authz-simple objects.

Members

identity: string
Identifies the allowed user. Its format depends on the network service that authorization object is associated with. For authorizing based on TLS x509 certificates, the identity must be the x509 distinguished name.

Since

4.0

5.12.8 Transactions

Abort (Object)

This action can be used to test transaction failure.

Since

1.6

ActionCompletionMode (Enum)

An enumeration of Transactional completion modes.

Values

individual
Do not attempt to cancel any other Actions if any Actions fail after the Transaction request succeeds. All Actions that can complete successfully will do so without waiting on others. This is the default.

grouped
If any Action fails after the Transaction succeeds, cancel all Actions. Actions do not complete until all Actions are ready to complete. May be rejected by Actions that do not support this completion mode.

Since

2.5

TransactionActionKind (Enum)

Values

abort
Since 1.6

block-dirty-bitmap-add
Since 2.5

block-dirty-bitmap-remove
Since 4.2

block-dirty-bitmap-clear
Since 2.5

block-dirty-bitmap-enable
Since 4.0

block-dirty-bitmap-disable
Since 4.0

block-dirty-bitmap-merge
Since 4.0

blockdev-backup
Since 2.3

blockdev-snapshot
Since 2.5

blockdev-snapshot-internal-sync
Since 1.7

blockdev-snapshot-sync
since 1.1

drive-backup
Since 1.6

Features

deprecated
Member `drive-backup` is deprecated. Use member `blockdev-backup` instead.

Since

1.1

AbortWrapper (Object)

Members

data: Abort
Not documented

Since

1.6

BlockDirtyBitmapAddWrapper (Object)

Members

data: BlockDirtyBitmapAdd
Not documented

Since

2.5

BlockDirtyBitmapWrapper (Object)

Members

data: BlockDirtyBitmap
Not documented

Since

2.5

BlockDirtyBitmapMergeWrapper (Object)

Members

data: BlockDirtyBitmapMerge
Not documented

Since

4.0

BlockdevBackupWrapper (Object)

Members

data: BlockdevBackup
Not documented

Since

2.3

BlockdevSnapshotWrapper (Object)

Members

data: BlockdevSnapshot
Not documented

Since

2.5

BlockdevSnapshotInternalWrapper (Object)

Members

data: BlockdevSnapshotInternal
Not documented

Since

1.7

BlockdevSnapshotSyncWrapper (Object)

Members

data: BlockdevSnapshotSync
Not documented

Since

1.1

DriveBackupWrapper (Object)

Members

data: DriveBackup
Not documented

Since

1.6

TransactionAction (Object)

A discriminated record of operations that can be performed with `transaction`.

Members

type: TransactionActionKind
the operation to be performed

The members of AbortWrapper when type is "abort"
The members of BlockDirtyBitmapAddWrapper when type is "block-dirty-bitmap-add"
The members of BlockDirtyBitmapWrapper when type is "block-dirty-bitmap-remove"
The members of BlockDirtyBitmapWrapper when type is "block-dirty-bitmap-clear"
The members of BlockDirtyBitmapWrapper when type is "block-dirty-bitmap-enable"
The members of BlockDirtyBitmapWrapper when type is "block-dirty-bitmap-disable"
The members of BlockDirtyBitmapMergeWrapper when type is "block-dirty-bitmap-merge"
The members of BlockdevBackupWrapper when type is "blockdev-backup"
The members of BlockdevSnapshotWrapper when type is "blockdev-snapshot"
The members of BlockdevSnapshotInternalWrapper when type is "blockdev-snapshot-internal-sync"
The members of BlockdevSnapshotSyncWrapper when type is "blockdev-snapshot-sync"
The members of DriveBackupWrapper when type is "drive-backup"

Since

1.1

`TransactionProperties` (Object)

Optional arguments to modify the behavior of a Transaction.

Members

`completion-mode`: `ActionCompletionMode` (optional)
Controls how jobs launched asynchronously by Actions will complete or fail as a group. See `ActionCompletionMode` for details.

Since

2.5

`transaction` (Command)

Executes a number of transactionable QMP commands atomically. If any operation fails, then the entire set of actions will be abandoned and the appropriate error returned.

For external snapshots, the dictionary contains the device, the file to use for the new snapshot, and the format. The default format, if not specified, is qcow2.

Each new snapshot defaults to being created by QEMU (wiping any contents if the file already exists), but it is also possible to reuse an externally-created file. In the latter case, you should ensure that the new image file has the same contents as the current one; QEMU cannot perform any meaningful check. Typically this is achieved by using the current image file as the backing file for the new image.

On failure, the original disks pre-snapshot attempt will be used.

For internal snapshots, the dictionary contains the device and the snapshot's name. If an internal snapshot matching name already exists, the request will be rejected. Only some image formats support it, for example, qcow2, and rbd,

On failure, qemu will try delete the newly created internal snapshot in the transaction. When an I/O error occurs during deletion, the user needs to fix it later with qemu-img or other command.

Arguments

`actions`: `array of TransactionAction`
List of `TransactionAction`; information needed for the respective operations.

`properties`: `TransactionProperties` (optional)
structure of additional options to control the execution of the transaction. See `TransactionProperties` for additional detail.

Errors

Any errors from commands in the transaction

Note

The transaction aborts on the first failure. Therefore, there will be information on only one failed operation returned in an error condition, and subsequent actions will not have been attempted.

Since

1.1

Example

```
-> { "execute": "transaction",
     "arguments": { "actions": [
         { "type": "blockdev-snapshot-sync", "data" : { "device": "ide-hd0",
                                       "snapshot-file": "/some/place/my-image",
                                       "format": "qcow2" } },
         { "type": "blockdev-snapshot-sync", "data" : { "node-name": "myfile",
                                       "snapshot-file": "/some/place/my-image2",
                                       "snapshot-node-name": "node3432",
                                       "mode": "existing",
                                       "format": "qcow2" } },
         { "type": "blockdev-snapshot-sync", "data" : { "device": "ide-hd1",
                                       "snapshot-file": "/some/place/my-image2",
                                       "mode": "existing",
                                       "format": "qcow2" } },
         { "type": "blockdev-snapshot-internal-sync", "data" : {
                                       "device": "ide-hd2",
                                       "name": "snapshot0" } } ] } }
<- { "return": {} }
```

5.12.9 QMP monitor control

`qmp_capabilities` (Command)

Enable QMP capabilities.

Arguments

enable: array of QMPCapability (optional)
An optional list of QMPCapability values to enable. The client must not enable any capability that is not mentioned in the QMP greeting message. If the field is not provided, it means no QMP capabilities will be enabled. (since 2.12)

Example

```
-> { "execute": "qmp_capabilities",
     "arguments": { "enable": [ "oob" ] } }
<- { "return": {} }
```

Notes

This command is valid exactly when first connecting: it must be issued before any other command will be accepted, and will fail once the monitor is accepting other commands. (see qemu docs/interop/qmp-spec.rst)

The QMP client needs to explicitly enable QMP capabilities, otherwise all the QMP capabilities will be turned off by default.

Since

0.13

QMPCapability (Enum)

Enumeration of capabilities to be advertised during initial client connection, used for agreeing on particular QMP extension behaviors.

Values

oob
QMP ability to support out-of-band requests. (Please refer to qmp-spec.rst for more information on OOB)

Since

2.12

`VersionTriple` (Object)

A three-part version number.

Members

`major: int`
The major version number.

`minor: int`
The minor version number.

`micro: int`
The micro version number.

Since

2.4

`VersionInfo` (Object)

A description of QEMU's version.

Members

`qemu: VersionTriple`
The version of QEMU. By current convention, a micro version of 50 signifies a development branch. A micro version greater than or equal to 90 signifies a release candidate for the next minor version. A micro version of less than 50 signifies a stable release.

`package: string`
QEMU will always set this field to an empty string. Downstream versions of QEMU should set this to a non-empty string. The exact format depends on the downstream however it highly recommended that a unique name is used.

Since

0.14

`query-version` (Command)

Returns the current version of QEMU.

Returns

A `VersionInfo` object describing the current version of QEMU.

Since

0.14

Example

```
-> { "execute": "query-version" }
<- {
      "return":{
         "qemu":{
            "major":0,
            "minor":11,
            "micro":5
         },
         "package":""
      }
   }
```

`CommandInfo` (Object)

Information about a QMP command

Members

`name`: `string`
The command name

Since

0.14

`query-commands` (Command)

Return a list of supported QMP commands by this server

Returns

A list of `CommandInfo` for all supported commands

Since

0.14

Example

```
-> { "execute": "query-commands" }
<- {
     "return":[
        {
           "name":"query-balloon"
        },
        {
           "name":"system_powerdown"
        }
     ]
   }
```

Note

This example has been shortened as the real response is too long.

`quit` (Command)

This command will cause the QEMU process to exit gracefully. While every attempt is made to send the QMP response before terminating, this is not guaranteed. When using this interface, a premature EOF would not be unexpected.

Since

0.14

Example

```
-> { "execute": "quit" }
<- { "return": {} }
```

MonitorMode (Enum)

An enumeration of monitor modes.

Values

readline
HMP monitor (human-oriented command line interface)

control
QMP monitor (JSON-based machine interface)

Since

5.0

MonitorOptions (Object)

Options to be used for adding a new monitor.

Members

id: string (optional)
Name of the monitor

mode: MonitorMode (optional)
Selects the monitor mode (default: readline in the system emulator, control in qemu-storage-daemon)

pretty: boolean (optional)
Enables pretty printing (QMP only)

chardev: string
Name of a character device to expose the monitor on

Since

5.0

5.12.10 QMP introspection

query-qmp-schema (Command)

Command query-qmp-schema exposes the QMP wire ABI as an array of SchemaInfo. This lets QMP clients figure out what commands and events are available in this QEMU, and their parameters and results.

However, the SchemaInfo can't reflect all the rules and restrictions that apply to QMP. It's interface introspection (figuring out what's there), not interface specification. The specification is in the QAPI schema.

Furthermore, while we strive to keep the QMP wire format backwards-compatible across qemu versions, the introspection output is not guaranteed to have the same stability. For example, one version of qemu may list an object member as an optional non-variant, while another lists the same member only through the object's variants; or the type of a

member may change from a generic string into a specific enum or from one specific type into an alternate that includes the original type alongside something else.

Returns

array of `SchemaInfo`, where each element describes an entity in the ABI: command, event, type, ...

The order of the various SchemaInfo is unspecified; however, all names are guaranteed to be unique (no name will be duplicated with different meta-types).

Note

the QAPI schema is also used to help define *internal* interfaces, by defining QAPI types. These are not part of the QMP wire ABI, and therefore not returned by this command.

Since

2.5

`SchemaMetaType` (Enum)

This is a `SchemaInfo`'s meta type, i.e. the kind of entity it describes.

Values

`builtin`
: a predefined type such as 'int' or 'bool'.

`enum`
: an enumeration type

`array`
: an array type

`object`
: an object type (struct or union)

`alternate`
: an alternate type

`command`
: a QMP command

`event`
: a QMP event

Since

2.5

SchemaInfo (Object)

Members

name: string
: the entity's name, inherited from `base`. The SchemaInfo is always referenced by this name. Commands and events have the name defined in the QAPI schema. Unlike command and event names, type names are not part of the wire ABI. Consequently, type names are meaningless strings here, although they are still guaranteed unique regardless of `meta-type`.

meta-type: SchemaMetaType
: the entity's meta type, inherited from `base`.

features: array of string (optional)
: names of features associated with the entity, in no particular order. (since 4.1 for object types, 4.2 for commands, 5.0 for the rest)

The members of SchemaInfoBuiltin when meta-type is "builtin"
The members of SchemaInfoEnum when meta-type is "enum"
The members of SchemaInfoArray when meta-type is "array"
The members of SchemaInfoObject when meta-type is "object"
The members of SchemaInfoAlternate when meta-type is "alternate"
The members of SchemaInfoCommand when meta-type is "command"
The members of SchemaInfoEvent when meta-type is "event"

Since

2.5

SchemaInfoBuiltin (Object)

Additional SchemaInfo members for meta-type 'builtin'.

Members

json-type: JSONType
: the JSON type used for this type on the wire.

Since

2.5

JSONType (Enum)

The four primitive and two structured types according to RFC 8259 section 1, plus 'int' (split off 'number'), plus the obvious top type 'value'.

Values

string
Not documented

number
Not documented

int
Not documented

boolean
Not documented

null
Not documented

object
Not documented

array
Not documented

value
Not documented

Since

2.5

SchemaInfoEnum (Object)

Additional SchemaInfo members for meta-type 'enum'.

Members

members: array of SchemaInfoEnumMember
the enum type's members, in no particular order (since 6.2).

values: array of string
the enumeration type's member names, in no particular order. Redundant with `members`. Just for backward compatibility.

Features

deprecated
Member `values` is deprecated. Use `members` instead.

Values of this type are JSON string on the wire.

Since

2.5

`SchemaInfoEnumMember` (Object)

An object member.

Members

name: `string`
the member's name, as defined in the QAPI schema.

features: `array of string` (optional)
names of features associated with the member, in no particular order.

Since

6.2

`SchemaInfoArray` (Object)

Additional SchemaInfo members for meta-type 'array'.

Members

element-type: `string`
the array type's element type.

Values of this type are JSON array on the wire.

Since

2.5

SchemaInfoObject (Object)

Additional SchemaInfo members for meta-type 'object'.

Members

members: array of SchemaInfoObjectMember
the object type's (non-variant) members, in no particular order.

tag: string (optional)
the name of the member serving as type tag. An element of `members` with this name must exist.

variants: array of SchemaInfoObjectVariant (optional)
variant members, i.e. additional members that depend on the type tag's value. Present exactly when `tag` is present. The variants are in no particular order, and may even differ from the order of the values of the enum type of the `tag`.

Values of this type are JSON object on the wire.

Since

2.5

SchemaInfoObjectMember (Object)

An object member.

Members

name: string
the member's name, as defined in the QAPI schema.

type: string
the name of the member's type.

default: value (optional)
default when used as command parameter. If absent, the parameter is mandatory. If present, the value must be null. The parameter is optional, and behavior when it's missing is not specified here. Future extension: if present and non-null, the parameter is optional, and defaults to this value.

features: array of string (optional)
names of features associated with the member, in no particular order. (since 5.0)

Since

2.5

SchemaInfoObjectVariant (Object)

The variant members for a value of the type tag.

Members

case: string
: a value of the type tag.

type: string
: the name of the object type that provides the variant members when the type tag has value `case`.

Since

2.5

SchemaInfoAlternate (Object)

Additional SchemaInfo members for meta-type 'alternate'.

Members

members: array of SchemaInfoAlternateMember
: the alternate type's members, in no particular order. The members' wire encoding is distinct, see *How to use the QAPI code generator* section Alternate types.

On the wire, this can be any of the members.

Since

2.5

SchemaInfoAlternateMember (Object)

An alternate member.

Members

type: string
the name of the member's type.

Since

2.5

SchemaInfoCommand (Object)

Additional SchemaInfo members for meta-type 'command'.

Members

arg-type: string
the name of the object type that provides the command's parameters.

ret-type: string
the name of the command's result type.

allow-oob: boolean (optional)
whether the command allows out-of-band execution, defaults to false (Since: 2.12)

Since

2.5

SchemaInfoEvent (Object)

Additional SchemaInfo members for meta-type 'event'.

Members

arg-type: string
the name of the object type that provides the event's parameters.

Since

2.5

5.12.11 QEMU Object Model (QOM)

`ObjectPropertyInfo` (Object)

Members

`name: string`
the name of the property

`type: string`
the type of the property. This will typically come in one of four forms:

1) A primitive type such as 'u8', 'u16', 'bool', 'str', or 'double'. These types are mapped to the appropriate JSON type.

2) A child type in the form 'child<subtype>' where subtype is a qdev device type name. Child properties create the composition tree.

3) A link type in the form 'link<subtype>' where subtype is a qdev device type name. Link properties form the device model graph.

`description: string` (optional)
if specified, the description of the property.

`default-value: value` (optional)
the default value, if any (since 5.0)

Since

1.2

`qom-list` (Command)

This command will list any properties of a object given a path in the object model.

Arguments

`path: string`
the path within the object model. See `qom-get` for a description of this parameter.

Returns

a list of `ObjectPropertyInfo` that describe the properties of the object.

Since

1.2

Example

```
-> { "execute": "qom-list",
     "arguments": { "path": "/chardevs" } }
<- { "return": [ { "name": "type", "type": "string" },
                 { "name": "parallel0", "type": "child<chardev-vc>" },
                 { "name": "serial0", "type": "child<chardev-vc>" },
                 { "name": "mon0", "type": "child<chardev-stdio>" } ] }
```

qom-get (Command)

This command will get a property from a object model path and return the value.

Arguments

path: string
: The path within the object model. There are two forms of supported paths–absolute and partial paths.

  Absolute paths are derived from the root object and can follow child<> or link<> properties. Since they can follow link<> properties, they can be arbitrarily long. Absolute paths look like absolute filenames and are prefixed with a leading slash.

  Partial paths look like relative filenames. They do not begin with a prefix. The matching rules for partial paths are subtle but designed to make specifying objects easy. At each level of the composition tree, the partial path is matched as an absolute path. The first match is not returned. At least two matches are searched for. A successful result is only returned if only one match is found. If more than one match is found, a flag is return to indicate that the match was ambiguous.

property: string
: The property name to read

Returns

The property value. The type depends on the property type. child<> and link<> properties are returned as #str pathnames. All integer property types (u8, u16, etc) are returned as #int.

Since

1.2

Examples

```
1. Use absolute path

-> { "execute": "qom-get",
     "arguments": { "path": "/machine/unattached/device[0]",
                    "property": "hotplugged" } }
<- { "return": false }

2. Use partial path

-> { "execute": "qom-get",
     "arguments": { "path": "unattached/sysbus",
                    "property": "type" } }
<- { "return": "System" }
```

qom-set (Command)

This command will set a property from a object model path.

Arguments

path: string
: see qom-get for a description of this parameter

property: string
: the property name to set

value: value
: a value who's type is appropriate for the property type. See qom-get for a description of type mapping.

Since

1.2

Example

```
-> { "execute": "qom-set",
     "arguments": { "path": "/machine",
                    "property": "graphics",
                    "value": false } }
<- { "return": {} }
```

`ObjectTypeInfo` (Object)

This structure describes a search result from `qom-list-types`

Members

`name: string`
the type name found in the search

`abstract: boolean (optional)`
the type is abstract and can't be directly instantiated. Omitted if false. (since 2.10)

`parent: string (optional)`
Name of parent type, if any (since 2.10)

Since

1.1

`qom-list-types` (Command)

This command will return a list of types given search parameters

Arguments

`implements: string (optional)`
if specified, only return types that implement this type name

`abstract: boolean (optional)`
if true, include abstract types in the results

Returns

a list of `ObjectTypeInfo` or an empty list if no results are found

Since

1.1

`qom-list-properties` (Command)

List properties associated with a QOM object.

Arguments

typename: string
the type name of an object

Note

objects can create properties at runtime, for example to describe links between different devices and/or objects. These properties are not included in the output of this command.

Returns

a list of ObjectPropertyInfo describing object properties

Since

2.12

CanHostSocketcanProperties (Object)

Properties for can-host-socketcan objects.

Members

if: string
interface name of the host system CAN bus to connect to

canbus: string
object ID of the can-bus object to connect to the host interface

Since

2.12

ColoCompareProperties (Object)

Properties for colo-compare objects.

Members

primary_in: string
name of the character device backend to use for the primary input (incoming packets are redirected to `outdev`)

secondary_in: string
name of the character device backend to use for secondary input (incoming packets are only compared to the input on `primary_in` and then dropped)

outdev: string
name of the character device backend to use for output

iothread: string
name of the iothread to run in

notify_dev: string (optional)
name of the character device backend to be used to communicate with the remote colo-frame (only for Xen COLO)

compare_timeout: int (optional)
the maximum time to hold a packet from `primary_in` for comparison with an incoming packet on `secondary_in` in milliseconds (default: 3000)

expired_scan_cycle: int (optional)
the interval at which colo-compare checks whether packets from `primary` have timed out, in milliseconds (default: 3000)

max_queue_size: int (optional)
the maximum number of packets to keep in the queue for comparing with incoming packets from `secondary_in`. If the queue is full and additional packets are received, the additional packets are dropped. (default: 1024)

vnet_hdr_support: boolean (optional)
if true, vnet header support is enabled (default: false)

Since

2.8

`CryptodevBackendProperties` (Object)

Properties for cryptodev-backend and cryptodev-backend-builtin objects.

Members

queues: int (optional)
the number of queues for the cryptodev backend. Ignored for cryptodev-backend and must be 1 for cryptodev-backend-builtin. (default: 1)

throttle-bps: int (optional)
limit total bytes per second (Since 8.0)

throttle-ops: int (optional)
limit total operations per second (Since 8.0)

Since

2.8

CryptodevVhostUserProperties (Object)

Properties for cryptodev-vhost-user objects.

Members

chardev: string
: the name of a Unix domain socket character device that connects to the vhost-user server

The members of CryptodevBackendProperties

Since

2.12

DBusVMStateProperties (Object)

Properties for dbus-vmstate objects.

Members

addr: string
: the name of the DBus bus to connect to

id-list: string (optional)
: a comma separated list of DBus IDs of helpers whose data should be included in the VM state on migration

Since

5.0

NetfilterInsert (Enum)

Indicates where to insert a netfilter relative to a given other filter.

Values

before
insert before the specified filter

behind
insert behind the specified filter

Since

5.0

NetfilterProperties (Object)

Properties for objects of classes derived from netfilter.

Members

netdev: string
id of the network device backend to filter

queue: NetFilterDirection (optional)
indicates which queue(s) to filter (default: all)

status: string (optional)
indicates whether the filter is enabled ("on") or disabled ("off") (default: "on")

position: string (optional)
specifies where the filter should be inserted in the filter list. "head" means the filter is inserted at the head of the filter list, before any existing filters. "tail" means the filter is inserted at the tail of the filter list, behind any existing filters (default). "id=<id>" means the filter is inserted before or behind the filter specified by <id>, depending on the `insert` property. (default: "tail")

insert: NetfilterInsert (optional)
where to insert the filter relative to the filter given in `position`. Ignored if `position` is "head" or "tail". (default: behind)

Since

2.5

FilterBufferProperties (Object)

Properties for filter-buffer objects.

Members

interval: int
a non-zero interval in microseconds. All packets arriving in the given interval are delayed until the end of the interval.

The members of NetfilterProperties

Since

2.5

FilterDumpProperties (Object)

Properties for filter-dump objects.

Members

file: string
the filename where the dumped packets should be stored

maxlen: int (optional)
maximum number of bytes in a packet that are stored (default: 65536)

The members of NetfilterProperties

Since

2.5

FilterMirrorProperties (Object)

Properties for filter-mirror objects.

Members

outdev: string
the name of a character device backend to which all incoming packets are mirrored

vnet_hdr_support: boolean (optional)
if true, vnet header support is enabled (default: false)

The members of NetfilterProperties

Since

2.6

FilterRedirectorProperties (Object)

Properties for filter-redirector objects.

At least one of `indev` or `outdev` must be present. If both are present, they must not refer to the same character device backend.

Members

indev: string (optional)
the name of a character device backend from which packets are received and redirected to the filtered network device

outdev: string (optional)
the name of a character device backend to which all incoming packets are redirected

vnet_hdr_support: boolean (optional)
if true, vnet header support is enabled (default: false)

The members of NetfilterProperties

Since

2.6

FilterRewriterProperties (Object)

Properties for filter-rewriter objects.

Members

vnet_hdr_support: boolean (optional)
if true, vnet header support is enabled (default: false)

The members of NetfilterProperties

Since

2.8

`InputBarrierProperties` (Object)

Properties for input-barrier objects.

Members

`name`: `string`
the screen name as declared in the screens section of barrier.conf

`server`: `string` (optional)
hostname of the Barrier server (default: "localhost")

`port`: `string` (optional)
TCP port of the Barrier server (default: "24800")

`x-origin`: `string` (optional)
x coordinate of the leftmost pixel on the guest screen (default: "0")

`y-origin`: `string` (optional)
y coordinate of the topmost pixel on the guest screen (default: "0")

`width`: `string` (optional)
the width of secondary screen in pixels (default: "1920")

`height`: `string` (optional)
the height of secondary screen in pixels (default: "1080")

Since

4.2

`InputLinuxProperties` (Object)

Properties for input-linux objects.

Members

`evdev`: `string`
the path of the host evdev device to use

`grab_all`: `boolean` (optional)
if true, grab is toggled for all devices (e.g. both keyboard and mouse) instead of just one device (default: false)

`repeat`: `boolean` (optional)
enables auto-repeat events (default: false)

`grab-toggle`: `GrabToggleKeys` (optional)
the key or key combination that toggles device grab (default: ctrl-ctrl)

Since

2.6

EventLoopBaseProperties (Object)

Common properties for event loops

Members

aio-max-batch: int (optional)
: maximum number of requests in a batch for the AIO engine, 0 means that the engine will use its default. (default: 0)

thread-pool-min: int (optional)
: minimum number of threads reserved in the thread pool (default:0)

thread-pool-max: int (optional)
: maximum number of threads the thread pool can contain (default:64)

Since

7.1

IothreadProperties (Object)

Properties for iothread objects.

Members

poll-max-ns: int (optional)
: the maximum number of nanoseconds to busy wait for events. 0 means polling is disabled (default: 32768 on POSIX hosts, 0 otherwise)

poll-grow: int (optional)
: the multiplier used to increase the polling time when the algorithm detects it is missing events due to not polling long enough. 0 selects a default behaviour (default: 0)

poll-shrink: int (optional)
: the divisor used to decrease the polling time when the algorithm detects it is spending too long polling without encountering events. 0 selects a default behaviour (default: 0)

The members of EventLoopBaseProperties

The `aio-max-batch` option is available since 6.1.

Since

2.0

MainLoopProperties (Object)

Properties for the main-loop object.

Members

The members of EventLoopBaseProperties

Since

7.1

MemoryBackendProperties (Object)

Properties for objects of classes derived from memory-backend.

Members

merge: boolean (optional)
if true, mark the memory as mergeable (default depends on the machine type)

dump: boolean (optional)
if true, include the memory in core dumps (default depends on the machine type)

host-nodes: array of int (optional)
the list of NUMA host nodes to bind the memory to

policy: HostMemPolicy (optional)
the NUMA policy (default: 'default')

prealloc: boolean (optional)
if true, preallocate memory (default: false)

prealloc-threads: int (optional)
number of CPU threads to use for prealloc (default: 1)

prealloc-context: string (optional)
thread context to use for creation of preallocation threads (default: none) (since 7.2)

share: boolean (optional)
if false, the memory is private to QEMU; if true, it is shared (default: false)

reserve: boolean (optional)
if true, reserve swap space (or huge pages) if applicable (default: true) (since 6.1)

size: int
size of the memory region in bytes

x-use-canonical-path-for-ramblock-id: boolean (optional)
if true, the canonical path is used for ramblock-id. Disable this for 4.0 machine types or older to allow migration with newer QEMU versions. (default: false generally, but true for machine types <= 4.0)

Note

prealloc=true and reserve=false cannot be set at the same time. With reserve=true, the behavior depends on the operating system: for example, Linux will not reserve swap space for shared file mappings – "not applicable". In contrast, reserve=false will bail out if it cannot be configured accordingly.

Since

2.1

MemoryBackendFileProperties (Object)

Properties for memory-backend-file objects.

Members

align: int (optional)
the base address alignment when QEMU mmap(2)s mem-path. Some backend stores specified by mem-path require an alignment different than the default one used by QEMU, e.g. the device DAX /dev/dax0.0 requires 2M alignment rather than 4K. In such cases, users can specify the required alignment via this option. 0 selects a default alignment (currently the page size). (default: 0)

offset: int (optional)
the offset into the target file that the region starts at. You can use this option to back multiple regions with a single file. Must be a multiple of the page size. (default: 0) (since 8.1)

discard-data: boolean (optional)
if true, the file contents can be destroyed when QEMU exits, to avoid unnecessarily flushing data to the backing file. Note that discard-data is only an optimization, and QEMU might not discard file contents if it aborts unexpectedly or is terminated using SIGKILL. (default: false)

mem-path: string
the path to either a shared memory or huge page filesystem mount

pmem: boolean (optional) (If: CONFIG_LIBPMEM)
specifies whether the backing file specified by mem-path is in host persistent memory that can be accessed using the SNIA NVM programming model (e.g. Intel NVDIMM).

readonly: boolean (optional)
if true, the backing file is opened read-only; if false, it is opened read-write. (default: false)

rom: OnOffAuto (optional)
whether to create Read Only Memory (ROM) that cannot be modified by the VM. Any write attempts to such ROM will be denied. Most use cases want writable RAM instead of ROM. However, selected use cases, like R/O NVDIMMs, can benefit from ROM. If set to 'on', create ROM; if set to 'off', create writable RAM; if set to 'auto', the value of the readonly property is used. This property is primarily helpful when we want to have proper RAM in configurations that would traditionally create ROM before this property was introduced: VM templating, where we want to open a file readonly (readonly set to true) and mark the memory to be private

for QEMU (`share` set to false). For this use case, we need writable RAM instead of ROM, and want to set this property to 'off'. (default: auto, since 8.2)

The members of `MemoryBackendProperties`

Since

2.1

`MemoryBackendMemfdProperties` (Object)

Properties for memory-backend-memfd objects.

The `share` boolean option is true by default with memfd.

Members

`hugetlb`: `boolean` (optional)
if true, the file to be created resides in the hugetlbfs filesystem (default: false)

`hugetlbsize`: `int` (optional)
the hugetlb page size on systems that support multiple hugetlb page sizes (it must be a power of 2 value supported by the system). 0 selects a default page size. This option is ignored if `hugetlb` is false. (default: 0)

`seal`: `boolean` (optional)
if true, create a sealed-file, which will block further resizing of the memory (default: true)

The members of `MemoryBackendProperties`

Since

2.12

`MemoryBackendEpcProperties` (Object)

Properties for memory-backend-epc objects.

The `share` boolean option is true by default with epc

The `merge` boolean option is false by default with epc

The `dump` boolean option is false by default with epc

Members

The members of `MemoryBackendProperties`

Since

6.2

PrManagerHelperProperties (Object)

Properties for pr-manager-helper objects.

Members

path: string
the path to a Unix domain socket for connecting to the external helper

Since

2.11

QtestProperties (Object)

Properties for qtest objects.

Members

chardev: string
the chardev to be used to receive qtest commands on.

log: string (optional)
the path to a log file

Since

6.0

RemoteObjectProperties (Object)

Properties for x-remote-object objects.

Members

fd: string
file descriptor name previously passed via 'getfd' command

devid: string
the id of the device to be associated with the file descriptor

Since

6.0

VfioUserServerProperties (Object)

Properties for x-vfio-user-server objects.

Members

socket: SocketAddress
: socket to be used by the libvfio-user library

device: string
: the ID of the device to be emulated at the server

Since

7.1

IOMMUFDProperties (Object)

Properties for iommufd objects.

Members

fd: string (optional)
: file descriptor name previously passed via 'getfd' command, which represents a pre-opened /dev/iommu. This allows the iommufd object to be shared across several subsystems (VFIO, VDPA, ...), and the file descriptor to be shared with other process, e.g. DPDK. (default: QEMU opens /dev/iommu by itself)

Since

9.0

AcpiGenericInitiatorProperties (Object)

Properties for acpi-generic-initiator objects.

Members

pci-dev: string
PCI device ID to be associated with the node

node: int
NUMA node associated with the PCI device

Since

9.0

RngProperties (Object)

Properties for objects of classes derived from rng.

Members

opened: boolean (optional)
if true, the device is opened immediately when applying this option and will probably fail when processing the next option. Don't use; only provided for compatibility. (default: false)

Features

deprecated
Member opened is deprecated. Setting true doesn't make sense, and false is already the default.

Since

1.3

RngEgdProperties (Object)

Properties for rng-egd objects.

Members

chardev: string
the name of a character device backend that provides the connection to the RNG daemon

The members of RngProperties

Since

1.3

RngRandomProperties (Object)

Properties for rng-random objects.

Members

filename: string (optional)
the filename of the device on the host to obtain entropy from (default: "/dev/urandom")

The members of RngProperties

Since

1.3

SevGuestProperties (Object)

Properties for sev-guest objects.

Members

sev-device: string (optional)
SEV device to use (default: "/dev/sev")

dh-cert-file: string (optional)
guest owners DH certificate (encoded with base64)

session-file: string (optional)
guest owners session parameters (encoded with base64)

policy: int (optional)
SEV policy value (default: 0x1)

handle: int (optional)
SEV firmware handle (default: 0)

cbitpos: int (optional)
C-bit location in page table entry (default: 0)

reduced-phys-bits: int
number of bits in physical addresses that become unavailable when SEV is enabled

kernel-hashes: boolean (optional)
if true, add hashes of kernel/initrd/cmdline to a designated guest firmware page for measured boot with -kernel (default: false) (since 6.2)

legacy-vm-type: boolean (optional)
Use legacy KVM_SEV_INIT KVM interface for creating the VM. The newer KVM_SEV_INIT2 interface syncs additional vCPU state when initializing the VMSA structures, which will result in a different guest measurement.

Set this to maintain compatibility with older QEMU or kernel versions that rely on legacy KVM_SEV_INIT behavior. (default: false) (since 9.1)

Since

2.12

ThreadContextProperties (Object)

Properties for thread context objects.

Members

cpu-affinity: array of int (optional)
the list of host CPU numbers used as CPU affinity for all threads created in the thread context (default: QEMU main thread CPU affinity)

node-affinity: array of int (optional)
the list of host node numbers that will be resolved to a list of host CPU numbers used as CPU affinity. This is a shortcut for specifying the list of host CPU numbers belonging to the host nodes manually by setting `cpu-affinity`. (default: QEMU main thread affinity)

Since

7.2

ObjectType (Enum)

Values

acpi-generic-initiator
Not documented

authz-list
Not documented

authz-listfile
Not documented

authz-pam
Not documented

authz-simple
Not documented

can-bus
Not documented

can-host-socketcan (If: CONFIG_LINUX)
Not documented

colo-compare
Not documented

cryptodev-backend
: Not documented

cryptodev-backend-builtin
: Not documented

cryptodev-backend-lkcf
: Not documented

cryptodev-vhost-user (If: CONFIG_VHOST_CRYPTO)
: Not documented

dbus-vmstate
: Not documented

filter-buffer
: Not documented

filter-dump
: Not documented

filter-mirror
: Not documented

filter-redirector
: Not documented

filter-replay
: Not documented

filter-rewriter
: Not documented

input-barrier
: Not documented

input-linux (If: CONFIG_LINUX)
: Not documented

iommufd
: Not documented

iothread
: Not documented

main-loop
: Not documented

memory-backend-epc (If: CONFIG_LINUX)
: Not documented

memory-backend-file
: Not documented

memory-backend-memfd (If: CONFIG_LINUX)
: Not documented

memory-backend-ram
: Not documented

pef-guest
: Not documented

pr-manager-helper (**If:** CONFIG_LINUX)
Not documented

qtest
Not documented

rng-builtin
Not documented

rng-egd
Not documented

rng-random (**If:** CONFIG_POSIX)
Not documented

secret
Not documented

secret_keyring (**If:** CONFIG_SECRET_KEYRING)
Not documented

sev-guest
Not documented

thread-context
Not documented

s390-pv-guest
Not documented

throttle-group
Not documented

tls-creds-anon
Not documented

tls-creds-psk
Not documented

tls-creds-x509
Not documented

tls-cipher-suites
Not documented

x-remote-object
Not documented

x-vfio-user-server
Not documented

Features

unstable
Member `x-remote-object` is experimental.

Since

6.0

`ObjectOptions` (Object)

Describes the options of a user creatable QOM object.

Members

`qom-type: ObjectType`
the class name for the object to be created

`id: string`
the name of the new object

The members of AcpiGenericInitiatorProperties when qom-type is "acpi-generic-initiator"
The members of AuthZListProperties when qom-type is "authz-list"
The members of AuthZListFileProperties when qom-type is "authz-listfile"
The members of AuthZPAMProperties when qom-type is "authz-pam"
The members of AuthZSimpleProperties when qom-type is "authz-simple"
The members of CanHostSocketcanProperties when qom-type is "can-host-socketcan" (If: CONFIG_LINUX)
The members of ColoCompareProperties when qom-type is "colo-compare"
The members of CryptodevBackendProperties when qom-type is "cryptodev-backend"
The members of CryptodevBackendProperties when qom-type is "cryptodev-backend-builtin"
The members of CryptodevBackendProperties when qom-type is "cryptodev-backend-lkcf"
The members of CryptodevVhostUserProperties when qom-type is "cryptodev-vhost-user" (If: CONFIG_VHOST_CRYPTO)
The members of DBusVMStateProperties when qom-type is "dbus-vmstate"
The members of FilterBufferProperties when qom-type is "filter-buffer"
The members of FilterDumpProperties when qom-type is "filter-dump"
The members of FilterMirrorProperties when qom-type is "filter-mirror"
The members of FilterRedirectorProperties when qom-type is "filter-redirector"
The members of NetfilterProperties when qom-type is "filter-replay"
The members of FilterRewriterProperties when qom-type is "filter-rewriter"
The members of InputBarrierProperties when qom-type is "input-barrier"
The members of InputLinuxProperties when qom-type is "input-linux" (If: CONFIG_LINUX)
The members of IOMMUFDProperties when qom-type is "iommufd"
The members of IothreadProperties when qom-type is "iothread"
The members of MainLoopProperties when qom-type is "main-loop"
The members of MemoryBackendEpcProperties when qom-type is "memory-backend-epc" (If: CONFIG_LINUX)
The members of MemoryBackendFileProperties when qom-type is "memory-backend-file"
The members of MemoryBackendMemfdProperties when qom-type is "memory-backend-memfd" (If: CONFIG_LINUX)
The members of MemoryBackendProperties when qom-type is "memory-backend-ram"
The members of PrManagerHelperProperties when qom-type is "pr-manager-helper" (If: CONFIG_LINUX)
The members of QtestProperties when qom-type is "qtest"
The members of RngProperties when qom-type is "rng-builtin"
The members of RngEgdProperties when qom-type is "rng-egd"
The members of RngRandomProperties when qom-type is "rng-random" (If: CONFIG_POSIX)
The members of SecretProperties when qom-type is "secret"
The members of SecretKeyringProperties when qom-type is "secret_keyring" (If: CONFIG_SECRET_KEYRING)
The members of SevGuestProperties when qom-type is "sev-guest"
The members of ThreadContextProperties when qom-type is "thread-context"
The members of ThrottleGroupProperties when qom-type is "throttle-group"
The members of TlsCredsAnonProperties when qom-type is "tls-creds-anon"
The members of TlsCredsPskProperties when qom-type is "tls-creds-psk"
The members of TlsCredsX509Properties when qom-type is "tls-creds-x509"
The members of TlsCredsProperties when qom-type is "tls-cipher-suites"
The members of RemoteObjectProperties when qom-type is "x-remote-object"
The members of VfioUserServerProperties when qom-type is "x-vfio-user-server"

Since

6.0

object-add (Command)

Create a QOM object.

Arguments

The members of ObjectOptions

Errors

- Error if qom-type is not a valid class name

Since

2.0

Example

```
-> { "execute": "object-add",
     "arguments": { "qom-type": "rng-random", "id": "rng1",
                    "filename": "/dev/hwrng" } }
<- { "return": {} }
```

object-del (Command)

Remove a QOM object.

Arguments

id: string
: the name of the QOM object to remove

Errors

- Error if id is not a valid id for a QOM object

Since

2.0

Example

```
-> { "execute": "object-del", "arguments": { "id": "rng1" } }
<- { "return": {} }
```

5.13 Vhost-user Protocol

Table of Contents

5.13.1 Introduction

This protocol is aiming to complement the `ioctl` interface used to control the vhost implementation in the Linux kernel. It implements the control plane needed to establish virtqueue sharing with a user space process on the same host. It uses communication over a Unix domain socket to share file descriptors in the ancillary data of the message.

The protocol defines 2 sides of the communication, *front-end* and *back-end*. The *front-end* is the application that shares its virtqueues, in our case QEMU. The *back-end* is the consumer of the virtqueues.

In the current implementation QEMU is the *front-end*, and the *back-end* is the external process consuming the virtio queues, for example a software Ethernet switch running in user space, such as Snabbswitch, or a block device back-end processing read & write to a virtual disk. In order to facilitate interoperability between various back-end implementations, it is recommended to follow the *Backend program conventions*.

The *front-end* and *back-end* can be either a client (i.e. connecting) or server (listening) in the socket communication.

Support for platforms other than Linux

While vhost-user was initially developed targeting Linux, nowadays it is supported on any platform that provides the following features:

- A way for requesting shared memory represented by a file descriptor so it can be passed over a UNIX domain socket and then mapped by the other process.
- AF_UNIX sockets with SCM_RIGHTS, so QEMU and the other process can exchange messages through it, including ancillary data when needed.

- Either eventfd or pipe/pipe2. On platforms where eventfd is not available, QEMU will automatically fall back to pipe2 or, as a last resort, pipe. Each file descriptor will be used for receiving or sending events by reading or writing (respectively) an 8-byte value to the corresponding it. The 8-value itself has no meaning and should not be interpreted.

5.13.2 Message Specification

Note: All numbers are in the machine native byte order.

A vhost-user message consists of 3 header fields and a payload.

request	flags	size	payload

Header

request
32-bit type of the request

flags
32-bit bit field

- Lower 2 bits are the version (currently 0x01)
- Bit 2 is the reply flag - needs to be sent on each reply from the back-end
- Bit 3 is the need_reply flag - see *REPLY_ACK* for details.

size
32-bit size of the payload

Payload

Depending on the request type, **payload** can be:

A single 64-bit integer

u64

u64
a 64-bit unsigned integer

A vring state description

index	num

index
a 32-bit index

num
a 32-bit number

A vring descriptor index for split virtqueues

vring index	index in avail ring

vring index
32-bit index of the respective virtqueue

index in avail ring
32-bit value, of which currently only the lower 16 bits are used:

- Bits 0–15: Index of the next *Available Ring* descriptor that the back-end will process. This is a free-running index that is not wrapped by the ring size.
- Bits 16–31: Reserved (set to zero)

Vring descriptor indices for packed virtqueues

vring index	descriptor indices

vring index
32-bit index of the respective virtqueue

descriptor indices
32-bit value:

- Bits 0–14: Index of the next *Available Ring* descriptor that the back-end will process. This is a free-running index that is not wrapped by the ring size.
- Bit 15: Driver (Available) Ring Wrap Counter
- Bits 16–30: Index of the entry in the *Used Ring* where the back-end will place the next descriptor. This is a free-running index that is not wrapped by the ring size.
- Bit 31: Device (Used) Ring Wrap Counter

A vring address description

index	flags	descriptor	used	available	log

index
a 32-bit vring index

flags
a 32-bit vring flags

descriptor
a 64-bit ring address of the vring descriptor table

used
a 64-bit ring address of the vring used ring

available
a 64-bit ring address of the vring available ring

log
a 64-bit guest address for logging

Note that a ring address is an IOVA if `VIRTIO_F_IOMMU_PLATFORM` has been negotiated. Otherwise it is a user address.

Memory region description

guest address	size	user address	mmap offset

guest address
a 64-bit guest address of the region

size
a 64-bit size

user address
a 64-bit user address

mmap offset
64-bit offset where region starts in the mapped memory

When the `VHOST_USER_PROTOCOL_F_XEN_MMAP` protocol feature has been successfully negotiated, the memory region description contains two extra fields at the end.

guest address	size	user address	mmap offset	xen mmap flags	domid

xen mmap flags
32-bit bit field

- Bit 0 is set for Xen foreign memory mapping.
- Bit 1 is set for Xen grant memory mapping.
- Bit 8 is set if the memory region can not be mapped in advance, and memory areas within this region must be mapped / unmapped only when required by the back-end. The back-end shouldn't try to map the entire region at once, as the front-end may not allow it. The back-end should rather map only the required amount of memory at once and unmap it after it is used.

domid
a 32-bit Xen hypervisor specific domain id.

Single memory region description

padding	region

padding
64-bit

A region is represented by Memory region description.

Multiple Memory regions description

num regions	padding	region0	…	region7

num regions
a 32-bit number of regions

padding
32-bit

A region is represented by Memory region description.

Log description

log size	log offset

log size
size of area used for logging

log offset
offset from start of supplied file descriptor where logging starts (i.e. where guest address 0 would be logged)

An IOTLB message

iova	size	user address	permissions flags	type

iova
a 64-bit I/O virtual address programmed by the guest

size
a 64-bit size

user address
a 64-bit user address

permissions flags
an 8-bit value: - 0: No access - 1: Read access - 2: Write access - 3: Read/Write access

type
an 8-bit IOTLB message type: - 1: IOTLB miss - 2: IOTLB update - 3: IOTLB invalidate - 4: IOTLB access fail

Virtio device config space

offset	size	flags	payload

offset
a 32-bit offset of virtio device's configuration space

size
a 32-bit configuration space access size in bytes

flags
a 32-bit value: - 0: Vhost front-end messages used for writable fields - 1: Vhost front-end messages used for live migration

payload
Size bytes array holding the contents of the virtio device's configuration space

Vring area description

u64	size	offset

u64
a 64-bit integer contains vring index and flags

size
a 64-bit size of this area

offset
a 64-bit offset of this area from the start of the supplied file descriptor

Inflight description

mmap size	mmap offset	num queues	queue size

mmap size
a 64-bit size of area to track inflight I/O

mmap offset
a 64-bit offset of this area from the start of the supplied file descriptor

num queues
a 16-bit number of virtqueues

queue size
a 16-bit size of virtqueues

VhostUserShared

UUID

UUID
16 bytes UUID, whose first three components (a 32-bit value, then two 16-bit values) are stored in big endian.

Device state transfer parameters

transfer direction	migration phase

transfer direction
a 32-bit enum, describing the direction in which the state is transferred:

- 0: Save: Transfer the state from the back-end to the front-end, which happens on the source side of migration
- 1: Load: Transfer the state from the front-end to the back-end, which happens on the destination side of migration

migration phase
a 32-bit enum, describing the state in which the VM guest and devices are:

- 0: Stopped (in the period after the transfer of memory-mapped regions before switch-over to the destination): The VM guest is stopped, and the vhost-user device is suspended (see *Suspended device state*).

In the future, additional phases might be added e.g. to allow iterative migration while the device is running.

C structure

In QEMU the vhost-user message is implemented with the following struct:

```
typedef struct VhostUserMsg {
    VhostUserRequest request;
    uint32_t flags;
    uint32_t size;
    union {
        uint64_t u64;
        struct vhost_vring_state state;
        struct vhost_vring_addr addr;
        VhostUserMemory memory;
        VhostUserLog log;
        struct vhost_iotlb_msg iotlb;
        VhostUserConfig config;
        VhostUserVringArea area;
        VhostUserInflight inflight;
    };
} QEMU_PACKED VhostUserMsg;
```

5.13.3 Communication

The protocol for vhost-user is based on the existing implementation of vhost for the Linux Kernel. Most messages that can be sent via the Unix domain socket implementing vhost-user have an equivalent ioctl to the kernel implementation.

The communication consists of the *front-end* sending message requests and the *back-end* sending message replies. Most of the requests don't require replies. Here is a list of the ones that do:

- VHOST_USER_GET_FEATURES
- VHOST_USER_GET_PROTOCOL_FEATURES
- VHOST_USER_GET_VRING_BASE
- VHOST_USER_SET_LOG_BASE (if VHOST_USER_PROTOCOL_F_LOG_SHMFD)
- VHOST_USER_GET_INFLIGHT_FD (if VHOST_USER_PROTOCOL_F_INFLIGHT_SHMFD)

See also:

REPLY_ACK
The section on REPLY_ACK protocol extension.

There are several messages that the front-end sends with file descriptors passed in the ancillary data:

- VHOST_USER_ADD_MEM_REG
- VHOST_USER_SET_MEM_TABLE
- VHOST_USER_SET_LOG_BASE (if VHOST_USER_PROTOCOL_F_LOG_SHMFD)
- VHOST_USER_SET_LOG_FD
- VHOST_USER_SET_VRING_KICK
- VHOST_USER_SET_VRING_CALL
- VHOST_USER_SET_VRING_ERR
- VHOST_USER_SET_BACKEND_REQ_FD (previous name VHOST_USER_SET_SLAVE_REQ_FD)
- VHOST_USER_SET_INFLIGHT_FD (if VHOST_USER_PROTOCOL_F_INFLIGHT_SHMFD)
- VHOST_USER_SET_DEVICE_STATE_FD

If *front-end* is unable to send the full message or receives a wrong reply it will close the connection. An optional reconnection mechanism can be implemented.

If *back-end* detects some error such as incompatible features, it may also close the connection. This should only happen in exceptional circumstances.

Any protocol extensions are gated by protocol feature bits, which allows full backwards compatibility on both front-end and back-end. As older back-ends don't support negotiating protocol features, a feature bit was dedicated for this purpose:

```
#define VHOST_USER_F_PROTOCOL_FEATURES 30
```

Note that VHOST_USER_F_PROTOCOL_FEATURES is the UNUSED (30) feature bit defined in VIRTIO 1.1 6.3 Legacy Interface: Reserved Feature Bits. VIRTIO devices do not advertise this feature bit and therefore VIRTIO drivers cannot negotiate it.

This reserved feature bit was reused by the vhost-user protocol to add vhost-user protocol feature negotiation in a backwards compatible fashion. Old vhost-user front-end and back-end implementations continue to work even though they are not aware of vhost-user protocol feature negotiation.

Ring states

Rings have two independent states: started/stopped, and enabled/disabled.

- While a ring is stopped, the back-end must not process the ring at all, regardless of whether it is enabled or disabled. The enabled/disabled state should still be tracked, though, so it can come into effect once the ring is started.
- started and disabled: The back-end must process the ring without causing any side effects. For example, for a networking device, in the disabled state the back-end must not supply any new RX packets, but must process and discard any TX packets.
- started and enabled: The back-end must process the ring normally, i.e. process all requests and execute them.

Each ring is initialized in a stopped and disabled state. The back-end must start a ring upon receiving a kick (that is, detecting that file descriptor is readable) on the descriptor specified by `VHOST_USER_SET_VRING_KICK` or receiving the in-band message `VHOST_USER_VRING_KICK` if negotiated, and stop a ring upon receiving `VHOST_USER_GET_VRING_BASE`.

Rings can be enabled or disabled by `VHOST_USER_SET_VRING_ENABLE`.

In addition, upon receiving a `VHOST_USER_SET_FEATURES` message from the front-end without `VHOST_USER_F_PROTOCOL_FEATURES` set, the back-end must enable all rings immediately.

While processing the rings (whether they are enabled or not), the back-end must support changing some configuration aspects on the fly.

Suspended device state

While all vrings are stopped, the device is *suspended*. In addition to not processing any vring (because they are stopped), the device must:

- not write to any guest memory regions,
- not send any notifications to the guest,
- not send any messages to the front-end,
- still process and reply to messages from the front-end.

Multiple queue support

Many devices have a fixed number of virtqueues. In this case the front-end already knows the number of available virtqueues without communicating with the back-end.

Some devices do not have a fixed number of virtqueues. Instead the maximum number of virtqueues is chosen by the back-end. The number can depend on host resource availability or back-end implementation details. Such devices are called multiple queue devices.

Multiple queue support allows the back-end to advertise the maximum number of queues. This is treated as a protocol extension, hence the back-end has to implement protocol features first. The multiple queues feature is supported only when the protocol feature `VHOST_USER_PROTOCOL_F_MQ` (bit 0) is set.

The max number of queues the back-end supports can be queried with message `VHOST_USER_GET_QUEUE_NUM`. Front-end should stop when the number of requested queues is bigger than that.

As all queues share one connection, the front-end uses a unique index for each queue in the sent message to identify a specified queue.

The front-end enables queues by sending message VHOST_USER_SET_VRING_ENABLE. vhost-user-net has historically automatically enabled the first queue pair.

Back-ends should always implement the VHOST_USER_PROTOCOL_F_MQ protocol feature, even for devices with a fixed number of virtqueues, since it is simple to implement and offers a degree of introspection.

Front-ends must not rely on the VHOST_USER_PROTOCOL_F_MQ protocol feature for devices with a fixed number of virtqueues. Only true multiqueue devices require this protocol feature.

Migration

During live migration, the front-end may need to track the modifications the back-end makes to the memory mapped regions. The front-end should mark the dirty pages in a log. Once it complies to this logging, it may declare the VHOST_F_LOG_ALL vhost feature.

To start/stop logging of data/used ring writes, the front-end may send messages VHOST_USER_SET_FEATURES with VHOST_F_LOG_ALL and VHOST_USER_SET_VRING_ADDR with VHOST_VRING_F_LOG in ring's flags set to 1/0, respectively.

All the modifications to memory pointed by vring "descriptor" should be marked. Modifications to "used" vring should be marked if VHOST_VRING_F_LOG is part of ring's flags.

Dirty pages are of size:

```
#define VHOST_LOG_PAGE 0x1000
```

The log memory fd is provided in the ancillary data of VHOST_USER_SET_LOG_BASE message when the back-end has VHOST_USER_PROTOCOL_F_LOG_SHMFD protocol feature.

The size of the log is supplied as part of VhostUserMsg which should be large enough to cover all known guest addresses. Log starts at the supplied offset in the supplied file descriptor. The log covers from address 0 to the maximum of guest regions. In pseudo-code, to mark page at addr as dirty:

```
page = addr / VHOST_LOG_PAGE
log[page / 8] |= 1 << page % 8
```

Where addr is the guest physical address.

Use atomic operations, as the log may be concurrently manipulated.

Note that when logging modifications to the used ring (when VHOST_VRING_F_LOG is set for this ring), log_guest_addr should be used to calculate the log offset: the write to first byte of the used ring is logged at this offset from log start. Also note that this value might be outside the legal guest physical address range (i.e. does not have to be covered by the VhostUserMemory table), but the bit offset of the last byte of the ring must fall within the size supplied by VhostUserLog.

VHOST_USER_SET_LOG_FD is an optional message with an eventfd in ancillary data, it may be used to inform the front-end that the log has been modified.

Once the source has finished migration, rings will be stopped by the source (*Suspended device state*). No further update must be done before rings are restarted.

In postcopy migration the back-end is started before all the memory has been received from the source host, and care must be taken to avoid accessing pages that have yet to be received. The back-end opens a 'userfault'-fd and registers the memory with it; this fd is then passed back over to the front-end. The front-end services requests on the userfaultfd for pages that are accessed and when the page is available it performs WAKE ioctl's on the userfaultfd to wake the stalled back-end. The front-end indicates support for this via the VHOST_USER_PROTOCOL_F_PAGEFAULT feature.

Migrating back-end state

Migrating device state involves transferring the state from one back-end, called the source, to another back-end, called the destination. After migration, the destination transparently resumes operation without requiring the driver to re-initialize the device at the VIRTIO level. If the migration fails, then the source can transparently resume operation until another migration attempt is made.

Generally, the front-end is connected to a virtual machine guest (which contains the driver), which has its own state to transfer between source and destination, and therefore will have an implementation-specific mechanism to do so. The `VHOST_USER_PROTOCOL_F_DEVICE_STATE` feature provides functionality to have the front-end include the back-end's state in this transfer operation so the back-end does not need to implement its own mechanism, and so the virtual machine may have its complete state, including vhost-user devices' states, contained within a single stream of data.

To do this, the back-end state is transferred from back-end to front-end on the source side, and vice versa on the destination side. This transfer happens over a channel that is negotiated using the `VHOST_USER_SET_DEVICE_STATE_FD` message. This message has two parameters:

- Direction of transfer: On the source, the data is saved, transferring it from the back-end to the front-end. On the destination, the data is loaded, transferring it from the front-end to the back-end.
- Migration phase: Currently, the only supported phase is the period after the transfer of memory-mapped regions before switch-over to the destination, when both the source and destination devices are suspended (*Suspended device state*). In the future, additional phases might be supported to allow iterative migration while the device is running.

The nature of the channel is implementation-defined, but it must generally behave like a pipe: The writing end will write all the data it has into it, signalling the end of data by closing its end. The reading end must read all of this data (until encountering the end of file) and process it.

- When saving, the writing end is the source back-end, and the reading end is the source front-end. After reading the state data from the channel, the source front-end must transfer it to the destination front-end through an implementation-defined mechanism.
- When loading, the writing end is the destination front-end, and the reading end is the destination back-end. After reading the state data from the channel, the destination back-end must deserialize its internal state from that data and set itself up to allow the driver to seamlessly resume operation on the VIRTIO level.

Seamlessly resuming operation means that the migration must be transparent to the guest driver, which operates on the VIRTIO level. This driver will not perform any re-initialization steps, but continue to use the device as if no migration had occurred. The vhost-user front-end, however, will re-initialize the vhost state on the destination, following the usual protocol for establishing a connection to a vhost-user back-end: This includes, for example, setting up memory mappings and kick and call FDs as necessary, negotiating protocol features, or setting the initial vring base indices (to the same value as on the source side, so that operation can resume).

Both on the source and on the destination side, after the respective front-end has seen all data transferred (when the transfer FD has been closed), it sends the `VHOST_USER_CHECK_DEVICE_STATE` message to verify that data transfer was successful in the back-end, too. The back-end responds once it knows whether the transfer and processing was successful or not.

Memory access

The front-end sends a list of vhost memory regions to the back-end using the VHOST_USER_SET_MEM_TABLE message. Each region has two base addresses: a guest address and a user address.

Messages contain guest addresses and/or user addresses to reference locations within the shared memory. The mapping of these addresses works as follows.

User addresses map to the vhost memory region containing that user address.

When the VIRTIO_F_IOMMU_PLATFORM feature has not been negotiated:

- Guest addresses map to the vhost memory region containing that guest address.

When the VIRTIO_F_IOMMU_PLATFORM feature has been negotiated:

- Guest addresses are also called I/O virtual addresses (IOVAs). They are translated to user addresses via the IOTLB.
- The vhost memory region guest address is not used.

IOMMU support

When the VIRTIO_F_IOMMU_PLATFORM feature has been negotiated, the front-end sends IOTLB entries update & invalidation by sending VHOST_USER_IOTLB_MSG requests to the back-end with a `struct vhost_iotlb_msg` as payload. For update events, the `iotlb` payload has to be filled with the update message type (2), the I/O virtual address, the size, the user virtual address, and the permissions flags. Addresses and size must be within vhost memory regions set via the VHOST_USER_SET_MEM_TABLE request. For invalidation events, the `iotlb` payload has to be filled with the invalidation message type (3), the I/O virtual address and the size. On success, the back-end is expected to reply with a zero payload, non-zero otherwise.

The back-end relies on the back-end communication channel (see *Back-end communication* section below) to send IOTLB miss and access failure events, by sending VHOST_USER_BACKEND_IOTLB_MSG requests to the front-end with a `struct vhost_iotlb_msg` as payload. For miss events, the iotlb payload has to be filled with the miss message type (1), the I/O virtual address and the permissions flags. For access failure event, the iotlb payload has to be filled with the access failure message type (4), the I/O virtual address and the permissions flags. For synchronization purpose, the back-end may rely on the reply-ack feature, so the front-end may send a reply when operation is completed if the reply-ack feature is negotiated and back-ends requests a reply. For miss events, completed operation means either front-end sent an update message containing the IOTLB entry containing requested address and permission, or front-end sent nothing if the IOTLB miss message is invalid (invalid IOVA or permission).

The front-end isn't expected to take the initiative to send IOTLB update messages, as the back-end sends IOTLB miss messages for the guest virtual memory areas it needs to access.

Back-end communication

An optional communication channel is provided if the back-end declares VHOST_USER_PROTOCOL_F_BACKEND_REQ protocol feature, to allow the back-end to make requests to the front-end.

The fd is provided via VHOST_USER_SET_BACKEND_REQ_FD ancillary data.

A back-end may then send VHOST_USER_BACKEND_* messages to the front-end using this fd communication channel.

If VHOST_USER_PROTOCOL_F_BACKEND_SEND_FD protocol feature is negotiated, back-end can send file descriptors (at most 8 descriptors in each message) to front-end via ancillary data using this fd communication channel.

Inflight I/O tracking

To support reconnecting after restart or crash, back-end may need to resubmit inflight I/Os. If virtqueue is processed in order, we can easily achieve that by getting the inflight descriptors from descriptor table (split virtqueue) or descriptor ring (packed virtqueue). However, it can't work when we process descriptors out-of-order because some entries which store the information of inflight descriptors in available ring (split virtqueue) or descriptor ring (packed virtqueue) might be overridden by new entries. To solve this problem, the back-end need to allocate an extra buffer to store this information of inflight descriptors and share it with front-end for persistent. `VHOST_USER_GET_INFLIGHT_FD` and `VHOST_USER_SET_INFLIGHT_FD` are used to transfer this buffer between front-end and back-end. And the format of this buffer is described below:

queue0 region	queue1 region	...	queueN region

N is the number of available virtqueues. The back-end could get it from num queues field of `VhostUserInflight`.

For split virtqueue, queue region can be implemented as:

```
typedef struct DescStateSplit {
    /* Indicate whether this descriptor is inflight or not.
     * Only available for head-descriptor. */
    uint8_t inflight;

    /* Padding */
    uint8_t padding[5];

    /* Maintain a list for the last batch of used descriptors.
     * Only available when batching is used for submitting */
    uint16_t next;

    /* Used to preserve the order of fetching available descriptors.
     * Only available for head-descriptor. */
    uint64_t counter;
} DescStateSplit;

typedef struct QueueRegionSplit {
    /* The feature flags of this region. Now it's initialized to 0. */
    uint64_t features;

    /* The version of this region. It's 1 currently.
     * Zero value indicates an uninitialized buffer */
    uint16_t version;

    /* The size of DescStateSplit array. It's equal to the virtqueue size.
     * The back-end could get it from queue size field of VhostUserInflight. */
    uint16_t desc_num;

    /* The head of list that track the last batch of used descriptors. */
    uint16_t last_batch_head;

    /* Store the idx value of used ring */
    uint16_t used_idx;

    /* Used to track the state of each descriptor in descriptor table */
```

(continues on next page)

(continued from previous page)

```
    DescStateSplit desc[];
} QueueRegionSplit;
```

To track inflight I/O, the queue region should be processed as follows:

When receiving available buffers from the driver:

1. Get the next available head-descriptor index from available ring, `i`
2. Set `desc[i].counter` to the value of global counter
3. Increase global counter by 1
4. Set `desc[i].inflight` to 1

When supplying used buffers to the driver:

1. Get corresponding used head-descriptor index, i
2. Set `desc[i].next` to `last_batch_head`
3. Set `last_batch_head` to `i`
4. Steps 1,2,3 may be performed repeatedly if batching is possible
5. Increase the `idx` value of used ring by the size of the batch
6. Set the `inflight` field of each `DescStateSplit` entry in the batch to 0
7. Set `used_idx` to the `idx` value of used ring

When reconnecting:

1. If the value of `used_idx` does not match the `idx` value of used ring (means the inflight field of `DescStateSplit` entries in last batch may be incorrect),
    a. Subtract the value of `used_idx` from the `idx` value of used ring to get last batch size of `DescStateSplit` entries
    b. Set the `inflight` field of each `DescStateSplit` entry to 0 in last batch list which starts from `last_batch_head`
    c. Set `used_idx` to the `idx` value of used ring
2. Resubmit inflight `DescStateSplit` entries in order of their counter value

For packed virtqueue, queue region can be implemented as:

```
typedef struct DescStatePacked {
    /* Indicate whether this descriptor is inflight or not.
     * Only available for head-descriptor. */
    uint8_t inflight;

    /* Padding */
    uint8_t padding;

    /* Link to the next free entry */
    uint16_t next;

    /* Link to the last entry of descriptor list.
     * Only available for head-descriptor. */
    uint16_t last;
```

(continues on next page)

(continued from previous page)

```
    /* The length of descriptor list.
     * Only available for head-descriptor. */
    uint16_t num;

    /* Used to preserve the order of fetching available descriptors.
     * Only available for head-descriptor. */
    uint64_t counter;

    /* The buffer id */
    uint16_t id;

    /* The descriptor flags */
    uint16_t flags;

    /* The buffer length */
    uint32_t len;

    /* The buffer address */
    uint64_t addr;
} DescStatePacked;

typedef struct QueueRegionPacked {
    /* The feature flags of this region. Now it's initialized to 0. */
    uint64_t features;

    /* The version of this region. It's 1 currently.
     * Zero value indicates an uninitialized buffer */
    uint16_t version;

    /* The size of DescStatePacked array. It's equal to the virtqueue size.
     * The back-end could get it from queue size field of VhostUserInflight. */
    uint16_t desc_num;

    /* The head of free DescStatePacked entry list */
    uint16_t free_head;

    /* The old head of free DescStatePacked entry list */
    uint16_t old_free_head;

    /* The used index of descriptor ring */
    uint16_t used_idx;

    /* The old used index of descriptor ring */
    uint16_t old_used_idx;

    /* Device ring wrap counter */
    uint8_t used_wrap_counter;

    /* The old device ring wrap counter */
    uint8_t old_used_wrap_counter;
```

(continues on next page)

(continued from previous page)

```
    /* Padding */
    uint8_t padding[7];

    /* Used to track the state of each descriptor fetched from descriptor ring */
    DescStatePacked desc[];
} QueueRegionPacked;
```

To track inflight I/O, the queue region should be processed as follows:

When receiving available buffers from the driver:

1. Get the next available descriptor entry from descriptor ring, `d`
2. If `d` is head descriptor,
    a. Set `desc[old_free_head].num` to 0
    b. Set `desc[old_free_head].counter` to the value of global counter
    c. Increase global counter by 1
    d. Set `desc[old_free_head].inflight` to 1
3. If `d` is last descriptor, set `desc[old_free_head].last` to `free_head`
4. Increase `desc[old_free_head].num` by 1
5. Set `desc[free_head].addr`, `desc[free_head].len`, `desc[free_head].flags`, `desc[free_head].id` to `d.addr`, `d.len`, `d.flags`, `d.id`
6. Set `free_head` to `desc[free_head].next`
7. If `d` is last descriptor, set `old_free_head` to `free_head`

When supplying used buffers to the driver:

1. Get corresponding used head-descriptor entry from descriptor ring, `d`
2. Get corresponding `DescStatePacked` entry, `e`
3. Set `desc[e.last].next` to `free_head`
4. Set `free_head` to the index of `e`
5. Steps 1,2,3,4 may be performed repeatedly if batching is possible
6. Increase `used_idx` by the size of the batch and update `used_wrap_counter` if needed
7. Update `d.flags`
8. Set the `inflight` field of each head `DescStatePacked` entry in the batch to 0
9. Set `old_free_head`, `old_used_idx`, `old_used_wrap_counter` to `free_head`, `used_idx`, `used_wrap_counter`

When reconnecting:

1. If `used_idx` does not match `old_used_idx` (means the `inflight` field of `DescStatePacked` entries in last batch may be incorrect),
    a. Get the next descriptor ring entry through `old_used_idx`, `d`
    b. Use `old_used_wrap_counter` to calculate the available flags

   c. If `d.flags` is not equal to the calculated flags value (means back-end has submitted the buffer to guest driver before crash, so it has to commit the in-progress update), set `old_free_head`, `old_used_idx`, `old_used_wrap_counter` to `free_head`, `used_idx`, `used_wrap_counter`

2. Set `free_head`, `used_idx`, `used_wrap_counter` to `old_free_head`, `old_used_idx`, `old_used_wrap_counter` (roll back any in-progress update)

3. Set the `inflight` field of each `DescStatePacked` entry in free list to 0

4. Resubmit inflight `DescStatePacked` entries in order of their counter value

In-band notifications

In some limited situations (e.g. for simulation) it is desirable to have the kick, call and error (if used) signals done via in-band messages instead of asynchronous eventfd notifications. This can be done by negotiating the `VHOST_USER_PROTOCOL_F_INBAND_NOTIFICATIONS` protocol feature.

Note that due to the fact that too many messages on the sockets can cause the sending application(s) to block, it is not advised to use this feature unless absolutely necessary. It is also considered an error to negotiate this feature without also negotiating `VHOST_USER_PROTOCOL_F_BACKEND_REQ` and `VHOST_USER_PROTOCOL_F_REPLY_ACK`, the former is necessary for getting a message channel from the back-end to the front-end, while the latter needs to be used with the in-band notification messages to block until they are processed, both to avoid blocking later and for proper processing (at least in the simulation use case.) As it has no other way of signalling this error, the back-end should close the connection as a response to a `VHOST_USER_SET_PROTOCOL_FEATURES` message that sets the in-band notifications feature flag without the other two.

Protocol features

```
#define VHOST_USER_PROTOCOL_F_MQ                    0
#define VHOST_USER_PROTOCOL_F_LOG_SHMFD             1
#define VHOST_USER_PROTOCOL_F_RARP                  2
#define VHOST_USER_PROTOCOL_F_REPLY_ACK             3
#define VHOST_USER_PROTOCOL_F_MTU                   4
#define VHOST_USER_PROTOCOL_F_BACKEND_REQ           5
#define VHOST_USER_PROTOCOL_F_CROSS_ENDIAN          6
#define VHOST_USER_PROTOCOL_F_CRYPTO_SESSION        7
#define VHOST_USER_PROTOCOL_F_PAGEFAULT             8
#define VHOST_USER_PROTOCOL_F_CONFIG                9
#define VHOST_USER_PROTOCOL_F_BACKEND_SEND_FD      10
#define VHOST_USER_PROTOCOL_F_HOST_NOTIFIER        11
#define VHOST_USER_PROTOCOL_F_INFLIGHT_SHMFD       12
#define VHOST_USER_PROTOCOL_F_RESET_DEVICE         13
#define VHOST_USER_PROTOCOL_F_INBAND_NOTIFICATIONS 14
#define VHOST_USER_PROTOCOL_F_CONFIGURE_MEM_SLOTS  15
#define VHOST_USER_PROTOCOL_F_STATUS               16
#define VHOST_USER_PROTOCOL_F_XEN_MMAP             17
#define VHOST_USER_PROTOCOL_F_SHARED_OBJECT        18
#define VHOST_USER_PROTOCOL_F_DEVICE_STATE         19
```

Front-end message types

VHOST_USER_GET_FEATURES

id
1

equivalent ioctl
VHOST_GET_FEATURES

request payload
N/A

reply payload
u64

Get from the underlying vhost implementation the features bitmask. Feature bit VHOST_USER_F_PROTOCOL_FEATURES signals back-end support for VHOST_USER_GET_PROTOCOL_FEATURES and VHOST_USER_SET_PROTOCOL_FEATURES.

VHOST_USER_SET_FEATURES

id
2

equivalent ioctl
VHOST_SET_FEATURES

request payload
u64

reply payload
N/A

Enable features in the underlying vhost implementation using a bitmask. Feature bit VHOST_USER_F_PROTOCOL_FEATURES signals back-end support for VHOST_USER_GET_PROTOCOL_FEATURES and VHOST_USER_SET_PROTOCOL_FEATURES.

VHOST_USER_GET_PROTOCOL_FEATURES

id
15

equivalent ioctl
VHOST_GET_FEATURES

request payload
N/A

reply payload
u64

Get the protocol feature bitmask from the underlying vhost implementation. Only legal if feature bit VHOST_USER_F_PROTOCOL_FEATURES is present in VHOST_USER_GET_FEATURES. It does not need to be acknowledged by VHOST_USER_SET_FEATURES.

Note: Back-ends that report VHOST_USER_F_PROTOCOL_FEATURES must support this message even before VHOST_USER_SET_FEATURES was called.

VHOST_USER_SET_PROTOCOL_FEATURES

id
16

equivalent ioctl
VHOST_SET_FEATURES

request payload
u64

reply payload
N/A

Enable protocol features in the underlying vhost implementation.

Only legal if feature bit VHOST_USER_F_PROTOCOL_FEATURES is present in VHOST_USER_GET_FEATURES. It does not need to be acknowledged by VHOST_USER_SET_FEATURES.

Note: Back-ends that report VHOST_USER_F_PROTOCOL_FEATURES must support this message even before VHOST_USER_SET_FEATURES was called.

VHOST_USER_SET_OWNER

id
3

equivalent ioctl
VHOST_SET_OWNER

request payload
N/A

reply payload
N/A

Issued when a new connection is established. It marks the sender as the front-end that owns of the session. This can be used on the *back-end* as a "session start" flag.

VHOST_USER_RESET_OWNER

id
4

request payload
N/A

reply payload
N/A

Deprecated

This is no longer used. Used to be sent to request disabling all rings, but some back-ends interpreted it to also discard connection state (this interpretation would lead to bugs). It is recommended that back-ends either ignore this message, or use it to disable all rings.

VHOST_USER_SET_MEM_TABLE

id
5

equivalent ioctl
VHOST_SET_MEM_TABLE

request payload
multiple memory regions description

reply payload
(postcopy only) multiple memory regions description

Sets the memory map regions on the back-end so it can translate the vring addresses. In the ancillary data there is an array of file descriptors for each memory mapped region. The size and ordering of the fds matches the number and ordering of memory regions.

When VHOST_USER_POSTCOPY_LISTEN has been received, SET_MEM_TABLE replies with the bases of the memory mapped regions to the front-end. The back-end must have mmap'd the regions but not yet accessed them and should not yet generate a userfault event.

Note: NEED_REPLY_MASK is not set in this case. QEMU will then reply back to the list of mappings with an empty VHOST_USER_SET_MEM_TABLE as an acknowledgement; only upon reception of this message may the guest start accessing the memory and generating faults.

VHOST_USER_SET_LOG_BASE

id
6

equivalent ioctl
VHOST_SET_LOG_BASE

request payload
u64

reply payload
N/A

Sets logging shared memory space.

When the back-end has VHOST_USER_PROTOCOL_F_LOG_SHMFD protocol feature, the log memory fd is provided in the ancillary data of VHOST_USER_SET_LOG_BASE message, the size and offset of shared memory area provided in the message.

VHOST_USER_SET_LOG_FD

id
7

equivalent ioctl
VHOST_SET_LOG_FD

request payload
N/A

reply payload
N/A

Sets the logging file descriptor, which is passed as ancillary data.

VHOST_USER_SET_VRING_NUM

id
8

equivalent ioctl
VHOST_SET_VRING_NUM

request payload
vring state description

reply payload
N/A

Set the size of the queue.

VHOST_USER_SET_VRING_ADDR

id
9

equivalent ioctl
VHOST_SET_VRING_ADDR

request payload
vring address description

reply payload
N/A

Sets the addresses of the different aspects of the vring.

VHOST_USER_SET_VRING_BASE

id
10

equivalent ioctl
VHOST_SET_VRING_BASE

request payload
vring descriptor index/indices

reply payload
N/A

Sets the next index to use for descriptors in this vring:

- For a split virtqueue, sets only the next descriptor index to process in the *Available Ring*. The device is supposed to read the next index in the *Used Ring* from the respective vring structure in guest memory.
- For a packed virtqueue, both indices are supplied, as they are not explicitly available in memory.

Consequently, the payload type is specific to the type of virt queue (*a vring descriptor index for split virtqueues* vs. *vring descriptor indices for packed virtqueues*).

VHOST_USER_GET_VRING_BASE

id
11

equivalent ioctl
VHOST_USER_GET_VRING_BASE

request payload
vring state description

reply payload
vring descriptor index/indices

Stops the vring and returns the current descriptor index or indices:

- For a split virtqueue, returns only the 16-bit next descriptor index to process in the *Available Ring*. Note that this may differ from the available ring index in the vring structure in memory, which points to where the driver will put new available descriptors. For the *Used Ring*, the device only needs the next descriptor index at which to put new descriptors, which is the value in the vring structure in memory, so this value is not covered by this message.
- For a packed virtqueue, neither index is explicitly available to read from memory, so both indices (as maintained by the device) are returned.

Consequently, the payload type is specific to the type of virt queue (*a vring descriptor index for split virtqueues* vs. *vring descriptor indices for packed virtqueues*).

When and as long as all of a device's vrings are stopped, it is *suspended*, see *Suspended device state*.

The request payload's *num* field is currently reserved and must be set to 0.

`VHOST_USER_SET_VRING_KICK`

id
12

equivalent ioctl
`VHOST_SET_VRING_KICK`

request payload
`u64`

reply payload
N/A

Set the event file descriptor for adding buffers to the vring. It is passed in the ancillary data.

Bits (0-7) of the payload contain the vring index. Bit 8 is the invalid FD flag. This flag is set when there is no file descriptor in the ancillary data. This signals that polling should be used instead of waiting for the kick. Note that if the protocol feature `VHOST_USER_PROTOCOL_F_INBAND_NOTIFICATIONS` has been negotiated this message isn't necessary as the ring is also started on the `VHOST_USER_VRING_KICK` message, it may however still be used to set an event file descriptor (which will be preferred over the message) or to enable polling.

`VHOST_USER_SET_VRING_CALL`

id
13

equivalent ioctl
`VHOST_SET_VRING_CALL`

request payload
`u64`

reply payload
N/A

Set the event file descriptor to signal when buffers are used. It is passed in the ancillary data.

Bits (0-7) of the payload contain the vring index. Bit 8 is the invalid FD flag. This flag is set when there is no file descriptor in the ancillary data. This signals that polling will be used instead of waiting for the call. Note that if the protocol features `VHOST_USER_PROTOCOL_F_INBAND_NOTIFICATIONS` and `VHOST_USER_PROTOCOL_F_BACKEND_REQ` have been negotiated this message isn't necessary as the `VHOST_USER_BACKEND_VRING_CALL` message can be used, it may however still be used to set an event file descriptor or to enable polling.

`VHOST_USER_SET_VRING_ERR`

id
14

equivalent ioctl
VHOST_SET_VRING_ERR

request payload
u64

reply payload
N/A

Set the event file descriptor to signal when error occurs. It is passed in the ancillary data.

Bits (0-7) of the payload contain the vring index. Bit 8 is the invalid FD flag. This flag is set when there is no file descriptor in the ancillary data. Note that if the protocol features VHOST_USER_PROTOCOL_F_INBAND_NOTIFICATIONS and VHOST_USER_PROTOCOL_F_BACKEND_REQ have been negotiated this message isn't necessary as the VHOST_USER_BACKEND_VRING_ERR message can be used, it may however still be used to set an event file descriptor (which will be preferred over the message).

VHOST_USER_GET_QUEUE_NUM

id
17

equivalent ioctl
N/A

request payload
N/A

reply payload
u64

Query how many queues the back-end supports.

This request should be sent only when VHOST_USER_PROTOCOL_F_MQ is set in queried protocol features by VHOST_USER_GET_PROTOCOL_FEATURES.

VHOST_USER_SET_VRING_ENABLE

id
18

equivalent ioctl
N/A

request payload
vring state description

reply payload
N/A

Signal the back-end to enable or disable corresponding vring.

This request should be sent only when VHOST_USER_F_PROTOCOL_FEATURES has been negotiated.

VHOST_USER_SEND_RARP

id
19

equivalent ioctl
N/A

request payload
u64

reply payload
N/A

Ask vhost user back-end to broadcast a fake RARP to notify the migration is terminated for guest that does not support GUEST_ANNOUNCE.

Only legal if feature bit VHOST_USER_F_PROTOCOL_FEATURES is present in VHOST_USER_GET_FEATURES and protocol feature bit VHOST_USER_PROTOCOL_F_RARP is present in VHOST_USER_GET_PROTOCOL_FEATURES. The first 6 bytes of the payload contain the mac address of the guest to allow the vhost user back-end to construct and broadcast the fake RARP.

VHOST_USER_NET_SET_MTU

id
20

equivalent ioctl
N/A

request payload
u64

reply payload
N/A

Set host MTU value exposed to the guest.

This request should be sent only when VIRTIO_NET_F_MTU feature has been successfully negotiated, VHOST_USER_F_PROTOCOL_FEATURES is present in VHOST_USER_GET_FEATURES and protocol feature bit VHOST_USER_PROTOCOL_F_NET_MTU is present in VHOST_USER_GET_PROTOCOL_FEATURES.

If VHOST_USER_PROTOCOL_F_REPLY_ACK is negotiated, the back-end must respond with zero in case the specified MTU is valid, or non-zero otherwise.

VHOST_USER_SET_BACKEND_REQ_FD (previous name VHOST_USER_SET_SLAVE_REQ_FD)

id
21

equivalent ioctl
N/A

request payload
N/A

reply payload
N/A

Set the socket file descriptor for back-end initiated requests. It is passed in the ancillary data.

This request should be sent only when VHOST_USER_F_PROTOCOL_FEATURES has been negotiated, and protocol feature bit VHOST_USER_PROTOCOL_F_BACKEND_REQ bit is present in VHOST_USER_GET_PROTOCOL_FEATURES. If VHOST_USER_PROTOCOL_F_REPLY_ACK is negotiated, the back-end must respond with zero for success, non-zero otherwise.

VHOST_USER_IOTLB_MSG

id
22

equivalent ioctl
N/A (equivalent to VHOST_IOTLB_MSG message type)

request payload
`struct vhost_iotlb_msg`

reply payload
`u64`

Send IOTLB messages with `struct vhost_iotlb_msg` as payload.

The front-end sends such requests to update and invalidate entries in the device IOTLB. The back-end has to acknowledge the request with sending zero as `u64` payload for success, non-zero otherwise.

This request should be send only when VIRTIO_F_IOMMU_PLATFORM feature has been successfully negotiated.

VHOST_USER_SET_VRING_ENDIAN

id
23

equivalent ioctl
VHOST_SET_VRING_ENDIAN

request payload
vring state description

reply payload
N/A

Set the endianness of a VQ for legacy devices. Little-endian is indicated with state.num set to 0 and big-endian is indicated with state.num set to 1. Other values are invalid.

This request should be sent only when VHOST_USER_PROTOCOL_F_CROSS_ENDIAN has been negotiated. Back-ends that negotiated this feature should handle both endiannesses and expect this message once (per VQ) during device configuration (ie. before the front-end starts the VQ).

VHOST_USER_GET_CONFIG

id
24

equivalent ioctl
N/A

request payload
virtio device config space

reply payload
virtio device config space

When VHOST_USER_PROTOCOL_F_CONFIG is negotiated, this message is submitted by the vhost-user front-end to fetch the contents of the virtio device configuration space, vhost-user back-end's payload size MUST match the front-end's request, vhost-user back-end uses zero length of payload to indicate an error to the vhost-user front-end. The vhost-user front-end may cache the contents to avoid repeated VHOST_USER_GET_CONFIG calls.

VHOST_USER_SET_CONFIG

id
25

equivalent ioctl
N/A

request payload
virtio device config space

reply payload
N/A

When VHOST_USER_PROTOCOL_F_CONFIG is negotiated, this message is submitted by the vhost-user front-end when the Guest changes the virtio device configuration space and also can be used for live migration on the destination host. The vhost-user back-end must check the flags field, and back-ends MUST NOT accept SET_CONFIG for read-only configuration space fields unless the live migration bit is set.

VHOST_USER_CREATE_CRYPTO_SESSION

id
26

equivalent ioctl
N/A

request payload
crypto session description

reply payload
crypto session description

Create a session for crypto operation. The back-end must return the session id, 0 or positive for success, negative for failure. This request should be sent only when VHOST_USER_PROTOCOL_F_CRYPTO_SESSION feature has been successfully negotiated. It's a required feature for crypto devices.

VHOST_USER_CLOSE_CRYPTO_SESSION

id
27

equivalent ioctl
N/A

request payload
u64

reply payload
N/A

Close a session for crypto operation which was previously created by VHOST_USER_CREATE_CRYPTO_SESSION.

This request should be sent only when VHOST_USER_PROTOCOL_F_CRYPTO_SESSION feature has been successfully negotiated. It's a required feature for crypto devices.

VHOST_USER_POSTCOPY_ADVISE

id
28

request payload
N/A

reply payload
userfault fd

When VHOST_USER_PROTOCOL_F_PAGEFAULT is supported, the front-end advises back-end that a migration with postcopy enabled is underway, the back-end must open a userfaultfd for later use. Note that at this stage the migration is still in precopy mode.

VHOST_USER_POSTCOPY_LISTEN

id
29

request payload
N/A

reply payload
N/A

The front-end advises back-end that a transition to postcopy mode has happened. The back-end must ensure that shared memory is registered with userfaultfd to cause faulting of non-present pages.

This is always sent sometime after a `VHOST_USER_POSTCOPY_ADVISE`, and thus only when `VHOST_USER_PROTOCOL_F_PAGEFAULT` is supported.

`VHOST_USER_POSTCOPY_END`

id
30

request payload
N/A

reply payload
`u64`

The front-end advises that postcopy migration has now completed. The back-end must disable the userfaultfd. The reply is an acknowledgement only.

When `VHOST_USER_PROTOCOL_F_PAGEFAULT` is supported, this message is sent at the end of the migration, after `VHOST_USER_POSTCOPY_LISTEN` was previously sent.

The value returned is an error indication; 0 is success.

`VHOST_USER_GET_INFLIGHT_FD`

id
31

equivalent ioctl
N/A

request payload
inflight description

reply payload
N/A

When `VHOST_USER_PROTOCOL_F_INFLIGHT_SHMFD` protocol feature has been successfully negotiated, this message is submitted by the front-end to get a shared buffer from back-end. The shared buffer will be used to track inflight I/O by back-end. QEMU should retrieve a new one when vm reset.

`VHOST_USER_SET_INFLIGHT_FD`

id
32

equivalent ioctl
N/A

request payload
inflight description

reply payload
N/A

When VHOST_USER_PROTOCOL_F_INFLIGHT_SHMFD protocol feature has been successfully negotiated, this message is submitted by the front-end to send the shared inflight buffer back to the back-end so that the back-end could get inflight I/O after a crash or restart.

VHOST_USER_GPU_SET_SOCKET

id
33

equivalent ioctl
N/A

request payload
N/A

reply payload
N/A

Sets the GPU protocol socket file descriptor, which is passed as ancillary data. The GPU protocol is used to inform the front-end of rendering state and updates. See vhost-user-gpu.rst for details.

VHOST_USER_RESET_DEVICE

id
34

equivalent ioctl
N/A

request payload
N/A

reply payload
N/A

Ask the vhost user back-end to disable all rings and reset all internal device state to the initial state, ready to be reinitialized. The back-end retains ownership of the device throughout the reset operation.

Only valid if the VHOST_USER_PROTOCOL_F_RESET_DEVICE protocol feature is set by the back-end.

VHOST_USER_VRING_KICK

id
35

equivalent ioctl
N/A

request payload
vring state description

reply payload
N/A

When the VHOST_USER_PROTOCOL_F_INBAND_NOTIFICATIONS protocol feature has been successfully negotiated, this message may be submitted by the front-end to indicate that a buffer was added to the vring instead of signalling it using the vring's kick file descriptor or having the back-end rely on polling.

The state.num field is currently reserved and must be set to 0.

VHOST_USER_GET_MAX_MEM_SLOTS

id
36

equivalent ioctl
N/A

request payload
N/A

reply payload
u64

When the VHOST_USER_PROTOCOL_F_CONFIGURE_MEM_SLOTS protocol feature has been successfully negotiated, this message is submitted by the front-end to the back-end. The back-end should return the message with a u64 payload containing the maximum number of memory slots for QEMU to expose to the guest. The value returned by the back-end will be capped at the maximum number of ram slots which can be supported by the target platform.

VHOST_USER_ADD_MEM_REG

id
37

equivalent ioctl
N/A

request payload
N/A

reply payload
single memory region description

When the VHOST_USER_PROTOCOL_F_CONFIGURE_MEM_SLOTS protocol feature has been successfully negotiated, this message is submitted by the front-end to the back-end. The message payload contains a memory region descriptor struct, describing a region of guest memory which the back-end device must map in. When the VHOST_USER_PROTOCOL_F_CONFIGURE_MEM_SLOTS protocol feature has been successfully negotiated, along with the VHOST_USER_REM_MEM_REG message, this message is used to set and update the memory tables of the back-end device.

Exactly one file descriptor from which the memory is mapped is passed in the ancillary data.

In postcopy mode (see VHOST_USER_POSTCOPY_LISTEN), the back-end replies with the bases of the memory mapped region to the front-end. For further details on postcopy, see VHOST_USER_SET_MEM_TABLE. They apply to VHOST_USER_ADD_MEM_REG accordingly.

VHOST_USER_REM_MEM_REG

id
38

equivalent ioctl
N/A

request payload
N/A

reply payload
single memory region description

When the VHOST_USER_PROTOCOL_F_CONFIGURE_MEM_SLOTS protocol feature has been successfully negotiated, this message is submitted by the front-end to the back-end. The message payload contains a memory region descriptor struct, describing a region of guest memory which the back-end device must unmap. When the VHOST_USER_PROTOCOL_F_CONFIGURE_MEM_SLOTS protocol feature has been successfully negotiated, along with the VHOST_USER_ADD_MEM_REG message, this message is used to set and update the memory tables of the back-end device.

The memory region to be removed is identified by its guest address, user address and size. The mmap offset is ignored.

No file descriptors SHOULD be passed in the ancillary data. For compatibility with existing incorrect implementations, the back-end MAY accept messages with one file descriptor. If a file descriptor is passed, the back-end MUST close it without using it otherwise.

VHOST_USER_SET_STATUS

id
39

equivalent ioctl
VHOST_VDPA_SET_STATUS

request payload
`u64`

reply payload
N/A

When the VHOST_USER_PROTOCOL_F_STATUS protocol feature has been successfully negotiated, this message is submitted by the front-end to notify the back-end with updated device status as defined in the Virtio specification.

VHOST_USER_GET_STATUS

id
40

equivalent ioctl
VHOST_VDPA_GET_STATUS

request payload
N/A

reply payload
`u64`

When the VHOST_USER_PROTOCOL_F_STATUS protocol feature has been successfully negotiated, this message is submitted by the front-end to query the back-end for its device status as defined in the Virtio specification.

VHOST_USER_GET_SHARED_OBJECT

id
41

equivalent ioctl
N/A

request payload
`struct VhostUserShared`

reply payload
dmabuf fd

When the VHOST_USER_PROTOCOL_F_SHARED_OBJECT protocol feature has been successfully negotiated, and the UUID is found in the exporters cache, this message is submitted by the front-end to retrieve a given dma-buf fd from a given back-end, determined by the requested UUID. Back-end will reply passing the fd when the operation is successful, or no fd otherwise.

VHOST_USER_SET_DEVICE_STATE_FD

id
42

equivalent ioctl
N/A

request payload
device state transfer parameters

reply payload
`u64`

Front-end and back-end negotiate a channel over which to transfer the back-end's internal state during migration. Either side (front-end or back-end) may create the channel. The nature of this channel is not restricted or defined in this document, but whichever side creates it must create a file descriptor that is provided to the respectively other side, allowing access to the channel. This FD must behave as follows:

- For the writing end, it must allow writing the whole back-end state sequentially. Closing the file descriptor signals the end of transfer.
- For the reading end, it must allow reading the whole back-end state sequentially. The end of file signals the end of the transfer.

For example, the channel may be a pipe, in which case the two ends of the pipe fulfill these requirements respectively.

Initially, the front-end creates a channel along with such an FD. It passes the FD to the back-end as ancillary data of a `VHOST_USER_SET_DEVICE_STATE_FD` message. The back-end may create a different transfer channel, passing the respective FD back to the front-end as ancillary data of the reply. If so, the front-end must then discard its channel and use the one provided by the back-end.

Whether the back-end should decide to use its own channel is decided based on efficiency: If the channel is a pipe, both ends will most likely need to copy data into and out of it. Any channel that allows for more efficient processing on at least one end, e.g. through zero-copy, is considered more efficient and thus preferred. If the back-end can provide such a channel, it should decide to use it.

The request payload contains parameters for the subsequent data transfer, as described in the *Migrating back-end state* section.

The value returned is both an indication for success, and whether a file descriptor for a back-end-provided channel is returned: Bits 0–7 are 0 on success, and non-zero on error. Bit 8 is the invalid FD flag; this flag is set when there is no file descriptor returned. When this flag is not set, the front-end must use the returned file descriptor as its end of the transfer channel. The back-end must not both indicate an error and return a file descriptor.

Using this function requires prior negotiation of the `VHOST_USER_PROTOCOL_F_DEVICE_STATE` feature.

`VHOST_USER_CHECK_DEVICE_STATE`

id
43

equivalent ioctl
N/A

request payload
N/A

reply payload
`u64`

After transferring the back-end's internal state during migration (see the *Migrating back-end state* section), check whether the back-end was able to successfully fully process the state.

The value returned indicates success or error; 0 is success, any non-zero value is an error.

Using this function requires prior negotiation of the `VHOST_USER_PROTOCOL_F_DEVICE_STATE` feature.

Back-end message types

For this type of message, the request is sent by the back-end and the reply is sent by the front-end.

VHOST_USER_BACKEND_IOTLB_MSG (previous name VHOST_USER_SLAVE_IOTLB_MSG)

id
1

equivalent ioctl
N/A (equivalent to VHOST_IOTLB_MSG message type)

request payload
`struct vhost_iotlb_msg`

reply payload
N/A

Send IOTLB messages with `struct vhost_iotlb_msg` as payload. The back-end sends such requests to notify of an IOTLB miss, or an IOTLB access failure. If VHOST_USER_PROTOCOL_F_REPLY_ACK is negotiated, and back-end set the VHOST_USER_NEED_REPLY flag, the front-end must respond with zero when operation is successfully completed, or non-zero otherwise. This request should be send only when VIRTIO_F_IOMMU_PLATFORM feature has been successfully negotiated.

VHOST_USER_BACKEND_CONFIG_CHANGE_MSG (previous name VHOST_USER_SLAVE_CONFIG_CHANGE_MSG)

id
2

equivalent ioctl
N/A

request payload
N/A

reply payload
N/A

When VHOST_USER_PROTOCOL_F_CONFIG is negotiated, vhost-user back-end sends such messages to notify that the virtio device's configuration space has changed, for those host devices which can support such feature, host driver can send VHOST_USER_GET_CONFIG message to the back-end to get the latest content. If VHOST_USER_PROTOCOL_F_REPLY_ACK is negotiated, and the back-end sets the VHOST_USER_NEED_REPLY flag, the front-end must respond with zero when operation is successfully completed, or non-zero otherwise.

VHOST_USER_BACKEND_VRING_HOST_NOTIFIER_MSG (previous name VHOST_USER_SLAVE_VRING_HOST_NOTIFIER_MSG)

id
3

equivalent ioctl
N/A

request payload
vring area description

reply payload
N/A

Sets host notifier for a specified queue. The queue index is contained in the `u64` field of the vring area description. The host notifier is described by the file descriptor (typically it's a VFIO device fd) which is passed as ancillary data and the size (which is mmap size and should be the same as host page size) and offset (which is mmap offset) carried in the vring area description. QEMU can mmap the file descriptor based on the size and offset to get a

memory range. Registering a host notifier means mapping this memory range to the VM as the specified queue's notify MMIO region. The back-end sends this request to tell QEMU to de-register the existing notifier if any and register the new notifier if the request is sent with a file descriptor.

This request should be sent only when VHOST_USER_PROTOCOL_F_HOST_NOTIFIER protocol feature has been successfully negotiated.

VHOST_USER_BACKEND_VRING_CALL (previous name VHOST_USER_SLAVE_VRING_CALL)

id
4

equivalent ioctl
N/A

request payload
vring state description

reply payload
N/A

When the VHOST_USER_PROTOCOL_F_INBAND_NOTIFICATIONS protocol feature has been successfully negotiated, this message may be submitted by the back-end to indicate that a buffer was used from the vring instead of signalling this using the vring's call file descriptor or having the front-end relying on polling.

The state.num field is currently reserved and must be set to 0.

VHOST_USER_BACKEND_VRING_ERR (previous name VHOST_USER_SLAVE_VRING_ERR)

id
5

equivalent ioctl
N/A

request payload
vring state description

reply payload
N/A

When the VHOST_USER_PROTOCOL_F_INBAND_NOTIFICATIONS protocol feature has been successfully negotiated, this message may be submitted by the back-end to indicate that an error occurred on the specific vring, instead of signalling the error file descriptor set by the front-end via VHOST_USER_SET_VRING_ERR.

The state.num field is currently reserved and must be set to 0.

VHOST_USER_BACKEND_SHARED_OBJECT_ADD

id
6

equivalent ioctl
N/A

request payload
`struct VhostUserShared`

reply payload
N/A

When the VHOST_USER_PROTOCOL_F_SHARED_OBJECT protocol feature has been successfully negotiated, this message can be submitted by the backends to add themselves as exporters to the virtio shared lookup table.

The back-end device gets associated with a UUID in the shared table. The back-end is responsible of keeping its own table with exported dma-buf fds. When another back-end tries to import the resource associated with the UUID, it will send a message to the front-end, which will act as a proxy to the exporter back-end. If VHOST_USER_PROTOCOL_F_REPLY_ACK is negotiated, and the back-end sets the VHOST_USER_NEED_REPLY flag, the front-end must respond with zero when operation is successfully completed, or non-zero otherwise.

VHOST_USER_BACKEND_SHARED_OBJECT_REMOVE

id
7

equivalent ioctl
N/A

request payload
`struct VhostUserShared`

reply payload
N/A

When the VHOST_USER_PROTOCOL_F_SHARED_OBJECT protocol feature has been successfully negotiated, this message can be submitted by the backend to remove themselves from to the virtio-dmabuf shared table API. Only the back-end owning the entry (i.e., the one that first added it) will have permission to remove it. Otherwise, the message is ignored. The shared table will remove the back-end device associated with the UUID. If VHOST_USER_PROTOCOL_F_REPLY_ACK is negotiated, and the back-end sets the VHOST_USER_NEED_REPLY flag, the front-end must respond with zero when operation is successfully completed, or non-zero otherwise.

VHOST_USER_BACKEND_SHARED_OBJECT_LOOKUP

id
8

equivalent ioctl
N/A

request payload
`struct VhostUserShared`

reply payload
dmabuf fd and `u64`

When the VHOST_USER_PROTOCOL_F_SHARED_OBJECT protocol feature has been successfully negotiated, this message can be submitted by the backends to retrieve a given dma-buf fd from the virtio-dmabuf shared table given a UUID. Frontend will reply passing the fd and a zero when the operation is successful, or non-zero otherwise. Note that if the operation fails, no fd is sent to the backend.

VHOST_USER_PROTOCOL_F_REPLY_ACK

The original vhost-user specification only demands replies for certain commands. This differs from the vhost protocol implementation where commands are sent over an `ioctl()` call and block until the back-end has completed.

With this protocol extension negotiated, the sender (QEMU) can set the `need_reply` [Bit 3] flag to any command. This indicates that the back-end MUST respond with a Payload `VhostUserMsg` indicating success or failure. The payload should be set to zero on success or non-zero on failure, unless the message already has an explicit reply body.

The reply payload gives QEMU a deterministic indication of the result of the command. Today, QEMU is expected to terminate the main vhost-user loop upon receiving such errors. In future, qemu could be taught to be more resilient for selective requests.

For the message types that already solicit a reply from the back-end, the presence of VHOST_USER_PROTOCOL_F_REPLY_ACK or need_reply bit being set brings no behavioural change. (See the *Communication* section for details.)

5.13.4 Backend program conventions

vhost-user back-ends can provide various devices & services and may need to be configured manually depending on the use case. However, it is a good idea to follow the conventions listed here when possible. Users, QEMU or libvirt, can then rely on some common behaviour to avoid heterogeneous configuration and management of the back-end programs and facilitate interoperability.

Each back-end installed on a host system should come with at least one JSON file that conforms to the vhost-user.json schema. Each file informs the management applications about the back-end type, and binary location. In addition, it defines rules for management apps for picking the highest priority back-end when multiple match the search criteria (see `@VhostUserBackend` documentation in the schema file).

If the back-end is not capable of enabling a requested feature on the host (such as 3D acceleration with virgl), or the initialization failed, the back-end should fail to start early and exit with a status != 0. It may also print a message to stderr for further details.

The back-end program must not daemonize itself, but it may be daemonized by the management layer. It may also have a restricted access to the system.

File descriptors 0, 1 and 2 will exist, and have regular stdin/stdout/stderr usage (they may have been redirected to /dev/null by the management layer, or to a log handler).

The back-end program must end (as quickly and cleanly as possible) when the SIGTERM signal is received. Eventually, it may receive SIGKILL by the management layer after a few seconds.

The following command line options have an expected behaviour. They are mandatory, unless explicitly said differently:

--socket-path=PATH This option specify the location of the vhost-user Unix domain socket. It is incompatible with –fd.

--fd=FDNUM When this argument is given, the back-end program is started with the vhost-user socket as file descriptor FDNUM. It is incompatible with –socket-path.

--print-capabilities Output to stdout the back-end capabilities in JSON format, and then exit successfully. Other options and arguments should be ignored, and the back-end program should not perform its normal function. The capabilities can be reported dynamically depending on the host capabilities.

The JSON output is described in the `vhost-user.json` schema, by `` `@VHostUserBackendCapabilities ``. Example:

```
{
  "type": "foo",
  "features": [
    "feature-a",
    "feature-b"
  ]
}
```

vhost-user-input

Command line options:

--evdev-path=PATH Specify the linux input device.
(optional)

--no-grab Do no request exclusive access to the input device.
(optional)

vhost-user-gpu

Command line options:

--render-node=PATH Specify the GPU DRM render node.
(optional)

--virgl Enable virgl rendering support.
(optional)

vhost-user-blk

Command line options:

--blk-file=PATH Specify block device or file path.
(optional)

--read-only Enable read-only.
(optional)

5.14 Vhost-user-gpu Protocol

Table of Contents

5.14.1 Introduction

The vhost-user-gpu protocol is aiming at sharing the rendering result of a virtio-gpu, done from a vhost-user back-end process to a vhost-user front-end process (such as QEMU). It bears a resemblance to a display server protocol, if you consider QEMU as the display server and the back-end as the client, but in a very limited way. Typically, it will work by setting a scanout/display configuration, before sending flush events for the display updates. It will also update the cursor shape and position.

The protocol is sent over a UNIX domain stream socket, since it uses socket ancillary data to share opened file descriptors (DMABUF fds or shared memory). The socket is usually obtained via `VHOST_USER_GPU_SET_SOCKET`.

Requests are sent by the *back-end*, and the optional replies by the *front-end*.

5.14.2 Wire format

Unless specified differently, numbers are in the machine native byte order.

A vhost-user-gpu message (request and reply) consists of 3 header fields and a payload.

request	flags	size	payload

Header

request
`u32`, type of the request

flags
`u32`, 32-bit bit field:

- Bit 2 is the reply flag - needs to be set on each reply

size
`u32`, size of the payload

Payload types

Depending on the request type, **payload** can be:

VhostUserGpuCursorPos

scanout-id	x	y

scanout-id
`u32`, the scanout where the cursor is located

x/y
`u32`, the cursor position

VhostUserGpuCursorUpdate

pos	hot_x	hot_y	cursor

pos
a `VhostUserGpuCursorPos`, the cursor location

hot_x/hot_y
`u32`, the cursor hot location

cursor
`[u32; 64 * 64]`, 64x64 RGBA cursor data (PIXMAN_a8r8g8b8 format)

VhostUserGpuScanout

scanout-id	w	h

scanout-id
`u32`, the scanout configuration to set

w/h
`u32`, the scanout width/height size

VhostUserGpuUpdate

scanout-id	x	y	w	h	data

scanout-id
`u32`, the scanout content to update

x/y/w/h
`u32`, region of the update

data
RGB data (PIXMAN_x8r8g8b8 format)

VhostUserGpuDMABUFScanout

scanout-id	x	y	w	h	fdw	fwh	stride	flags	fourcc

scanout-id
: u32, the scanout configuration to set

x/y
: u32, the location of the scanout within the DMABUF

w/h
: u32, the scanout width/height size

fdw/fdh/stride/flags
: u32, the DMABUF width/height/stride/flags

fourcc
: i32, the DMABUF fourcc

VhostUserGpuEdidRequest

scanout-id

scanout-id
: u32, the scanout to get edid from

VhostUserGpuDMABUFScanout2

dmabuf_scanout	modifier

dmabuf_scanout
: VhostUserGpuDMABUFScanout, filled as described in the VhostUserGpuDMABUFScanout structure.

modifier
: u64, the DMABUF modifiers

C structure

In QEMU the vhost-user-gpu message is implemented with the following struct:

```
typedef struct VhostUserGpuMsg {
    uint32_t request; /* VhostUserGpuRequest */
    uint32_t flags;
    uint32_t size; /* the following payload size */
    union {
        VhostUserGpuCursorPos cursor_pos;
        VhostUserGpuCursorUpdate cursor_update;
        VhostUserGpuScanout scanout;
        VhostUserGpuUpdate update;
        VhostUserGpuDMABUFScanout dmabuf_scanout;
```

(continues on next page)

(continued from previous page)

```
        VhostUserGpuEdidRequest edid_req;
        struct virtio_gpu_resp_edid resp_edid;
        struct virtio_gpu_resp_display_info display_info;
        uint64_t u64;
    } payload;
} QEMU_PACKED VhostUserGpuMsg;
```

Protocol features

```
#define VHOST_USER_GPU_PROTOCOL_F_EDID    0
#define VHOST_USER_GPU_PROTOCOL_F_DMABUF2 1
```

New messages and communication changes are negotiated thanks to the `VHOST_USER_GPU_GET_PROTOCOL_FEATURES` and `VHOST_USER_GPU_SET_PROTOCOL_FEATURES` requests.

5.14.3 Communication

Message types

`VHOST_USER_GPU_GET_PROTOCOL_FEATURES`

id
1

request payload
N/A

reply payload
`u64`

Get the supported protocol features bitmask.

`VHOST_USER_GPU_SET_PROTOCOL_FEATURES`

id
2

request payload
`u64`

reply payload
N/A

Enable protocol features using a bitmask.

`VHOST_USER_GPU_GET_DISPLAY_INFO`

id
3

request payload
N/A

reply payload
`struct virtio_gpu_resp_display_info` (from virtio specification)

Get the preferred display configuration.

VHOST_USER_GPU_CURSOR_POS

id
4

request payload
VhostUserGpuCursorPos

reply payload
N/A

Set/show the cursor position.

VHOST_USER_GPU_CURSOR_POS_HIDE

id
5

request payload
VhostUserGpuCursorPos

reply payload
N/A

Set/hide the cursor.

VHOST_USER_GPU_CURSOR_UPDATE

id
6

request payload
VhostUserGpuCursorUpdate

reply payload
N/A

Update the cursor shape and location.

VHOST_USER_GPU_SCANOUT

id
7

request payload
VhostUserGpuScanout

reply payload
N/A

Set the scanout resolution. To disable a scanout, the dimensions width/height are set to 0.

VHOST_USER_GPU_UPDATE

id
8

request payload
VhostUserGpuUpdate

reply payload
N/A

Update the scanout content. The data payload contains the graphical bits. The display should be flushed and presented.

VHOST_USER_GPU_DMABUF_SCANOUT

id
9

request payload
`VhostUserGpuDMABUFScanout`

reply payload
N/A

Set the scanout resolution/configuration, and share a DMABUF file descriptor for the scanout content, which is passed as ancillary data. To disable a scanout, the dimensions width/height are set to 0, there is no file descriptor passed.

VHOST_USER_GPU_DMABUF_UPDATE

id
10

request payload
`VhostUserGpuUpdate`

reply payload
empty payload

The display should be flushed and presented according to updated region from `VhostUserGpuUpdate`.

Note: there is no data payload, since the scanout is shared thanks to DMABUF, that must have been set previously with `VHOST_USER_GPU_DMABUF_SCANOUT`.

VHOST_USER_GPU_GET_EDID

id
11

request payload
`struct VhostUserGpuEdidRequest`

reply payload
`struct virtio_gpu_resp_edid` (from virtio specification)

Retrieve the EDID data for a given scanout. This message requires the `VHOST_USER_GPU_PROTOCOL_F_EDID` protocol feature to be supported.

VHOST_USER_GPU_DMABUF_SCANOUT2

id
12

request payload
`VhostUserGpuDMABUFScanout2`

reply payload
N/A

Same as VHOST_USER_GPU_DMABUF_SCANOUT, but also sends the dmabuf modifiers appended to the message, which were not provided in the other message. This message requires the `VHOST_USER_GPU_PROTOCOL_F_DMABUF2` protocol feature to be supported.

5.15 Vhost-vdpa Protocol

5.15.1 Introduction

vDPA(Virtual data path acceleration) device is a device that uses a datapath which complies with the virtio specifications with vendor specific control path. vDPA devices can be both physically located on the hardware or emulated by software.

This document describes the vDPA support in qemu

Here is the kernel commit here https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4c8cf31885f69e86be0b5b9e6677a26797365e1d

TODO : More information will add later

5.16 Virtio balloon memory statistics

The virtio balloon driver supports guest memory statistics reporting. These statistics are available to QEMU users as QOM (QEMU Object Model) device properties via a polling mechanism.

Before querying the available stats, clients first have to enable polling. This is done by writing a time interval value (in seconds) to the guest-stats-polling-interval property. This value can be:

> 0
: enables polling in the specified interval. If polling is already enabled, the polling time interval is changed to the new value

0
: disables polling. Previous polled statistics are still valid and can be queried.

Once polling is enabled, the virtio-balloon device in QEMU will start polling the guest's balloon driver for new stats in the specified time interval.

To retrieve those stats, clients have to query the guest-stats property, which will return a dictionary containing:

- A key named 'stats', containing all available stats. If the guest doesn't support a particular stat, or if it couldn't be retrieved, its value will be -1. Currently, the following stats are supported:
  - stat-swap-in
  - stat-swap-out
  - stat-major-faults
  - stat-minor-faults
  - stat-free-memory
  - stat-total-memory
  - stat-available-memory
  - stat-disk-caches
  - stat-htlb-pgalloc
  - stat-htlb-pgfail
- A key named last-update, which contains the last stats update timestamp in seconds. Since this timestamp is generated by the host, a buggy guest can't influence its value. The value is 0 if the guest has not updated the stats (yet).

It's also important to note the following:

- Previously polled statistics remain available even if the polling is later disabled
- As noted above, if a guest doesn't support a particular stat its value will always be -1. However, it's also possible that a guest temporarily couldn't update one or even all stats. If this happens, just wait for the next update
- Polling can be enabled even if the guest doesn't have stats support or the balloon driver wasn't loaded in the guest. If this is the case and stats are queried, last-update will be 0.
- The polling timer is only re-armed when the guest responds to the statistics request. This means that if a (buggy) guest doesn't ever respond to the request the timer will never be re-armed, which has the same effect as disabling polling

Here are a few examples. QEMU is started with `-device virtio-balloon`, which generates `/machine/peripheral-anon/device[1]` as the QOM path for the balloon device.

Enable polling with 2 seconds interval:

```
{ "execute": "qom-set",
              "arguments": { "path": "/machine/peripheral-anon/device[1]",
              "property": "guest-stats-polling-interval", "value": 2 } }

{ "return": {} }
```

Change polling to 10 seconds:

```
{ "execute": "qom-set",
              "arguments": { "path": "/machine/peripheral-anon/device[1]",
              "property": "guest-stats-polling-interval", "value": 10 } }

{ "return": {} }
```

Get stats:

```
{ "execute": "qom-get",
              "arguments": { "path": "/machine/peripheral-anon/device[1]",
              "property": "guest-stats" } }
{
  "return": {
      "stats": {
          "stat-swap-out": 0,
          "stat-free-memory": 844943360,
          "stat-minor-faults": 219028,
          "stat-major-faults": 235,
          "stat-total-memory": 1044406272,
          "stat-swap-in": 0
      },
      "last-update": 1358529861
  }
}
```

Disable polling:

```
{ "execute": "qom-set",
              "arguments": { "path": "/machine/peripheral-anon/device[1]",
              "property": "stats-polling-interval", "value": 0 } }
```

(continues on next page)

(continued from previous page)

```
{ "return": {} }
```

5.17 VNC LED state Pseudo-encoding

5.17.1 Introduction

This document describes the Pseudo-encoding of LED state for RFB which is the protocol used in VNC as reference link below:

http://tigervnc.svn.sourceforge.net/viewvc/tigervnc/rfbproto/rfbproto.rst?content-type=text/plain

When accessing a guest by console through VNC, there might be mismatch between the lock keys notification LED on the computer running the VNC client session and the current status of the lock keys on the guest machine.

To solve this problem it attempts to add LED state Pseudo-encoding extension to VNC protocol to deal with setting LED state.

5.17.2 Pseudo-encoding

This Pseudo-encoding requested by client declares to server that it supports LED state extensions to the protocol.

The Pseudo-encoding number for LED state defined as:

Number	Name
-261	'LED state Pseudo-encoding'

5.17.3 LED state Pseudo-encoding

The LED state Pseudo-encoding describes the encoding of LED state which consists of 3 bits, from left to right each bit represents the Caps, Num, and Scroll lock key respectively. '1' indicates that the LED should be on and '0' should be off.

Some example encodings for it as following:

Code	Description
100	CapsLock is on, NumLock and ScrollLock are off
010	NumLock is on, CapsLock and ScrollLock are off
111	CapsLock, NumLock and ScrollLock are on

CHAPTER

SIX

SYSTEM EMULATION GUEST HARDWARE SPECIFICATIONS

This section of the manual contains specifications of guest hardware that is specific to QEMU.

6.1 PCI IDs for QEMU

Red Hat, Inc. donates a part of its device ID range to QEMU, to be used for virtual devices. The vendor IDs are 1af4 (formerly Qumranet ID) and 1b36.

Contact Gerd Hoffmann <kraxel@redhat.com> to get a device ID assigned for your devices.

6.1.1 1af4 vendor ID

The 1000 -> 10ff device ID range is used as follows for virtio-pci devices. Note that this allocation is separate from the virtio device IDs, which are maintained as part of the virtio specification.

1af4:1000
network device (legacy)

1af4:1001
block device (legacy)

1af4:1002
balloon device (legacy)

1af4:1003
console device (legacy)

1af4:1004
SCSI host bus adapter device (legacy)

1af4:1005
entropy generator device (legacy)

1af4:1009
9p filesystem device (legacy)

1af4:1012
vsock device (bug compatibility)

1af4:1040 to 1af4:10ef
ID range for modern virtio devices. The PCI device ID is calculated from the virtio device ID by adding the 0x1040 offset. The virtio IDs are defined in the virtio specification. The Linux kernel has a header file with defines for all virtio IDs (`linux/virtio_ids.h`); QEMU has a copy in `include/standard-headers/`.

1af4:10f0 to 1a4f:10ff
Available for experimental usage without registration. Must get official ID when the code leaves the test lab (i.e. when seeking upstream merge or shipping a distro/product) to avoid conflicts.

1af4:1100
Used as PCI Subsystem ID for existing hardware devices emulated by QEMU.

1af4:1110
ivshmem device (*Device Specification for Inter-VM shared memory device*)

All other device IDs are reserved.

6.1.2 1b36 vendor ID

The 0000 -> 00ff device ID range is used as follows for QEMU-specific PCI devices (other than virtio):

1b36:0001
PCI-PCI bridge

1b36:0002
PCI serial port (16550A) adapter (*QEMU PCI serial devices*)

1b36:0003
PCI Dual-port 16550A adapter (*QEMU PCI serial devices*)

1b36:0004
PCI Quad-port 16550A adapter (*QEMU PCI serial devices*)

1b36:0005
PCI test device (*QEMU PCI test device*)

1b36:0006
PCI Rocker Ethernet switch device

1b36:0007
PCI SD Card Host Controller Interface (SDHCI)

1b36:0008
PCIe host bridge

1b36:0009
PCI Expander Bridge (-device pxb)

1b36:000a
PCI-PCI bridge (multiseat)

1b36:000b
PCIe Expander Bridge (-device pxb-pcie)

1b36:000d
PCI xhci usb host adapter

1b36:000f
mdpy (mdev sample device), `linux/samples/vfio-mdev/mdpy.c`

1b36:0010
PCIe NVMe device (`-device nvme`)

1b36:0011
PCI PVPanic device (`-device pvpanic-pci`)

1b36:0012
PCI ACPI ERST device (`-device acpi-erst`)

1b36:0013
PCI UFS device (`-device ufs`)

All these devices are documented in *System Emulation Guest Hardware Specifications*.

The 0100 device ID is used for the QXL video card device.

6.2 QEMU PCI serial devices

QEMU implements some PCI serial devices which are simple PCI wrappers around one or more 16550 UARTs.

There is one single-port variant and two multiport-variants. Linux guests work out-of-the box with all cards. There is a Windows inf file (`docs/qemupciserial.inf`) to set up the cards in Windows guests.

6.2.1 Single-port card

Name:
`pci-serial`

PCI ID:
1b36:0002

PCI Region 0:
IO bar, 8 bytes long, with the 16550 UART mapped to it.

Interrupt:
Wired to pin A.

6.2.2 Multiport cards

Name:
`pci-serial-2x`, `pci-serial-4x`

PCI ID:
1b36:0003 (`-2x`) and 1b36:0004 (`-4x`)

PCI Region 0:
IO bar, with two or four 16550 UARTs mapped after each other. The first is at offset 0, the second at offset 8, and so on.

Interrupt:
Wired to pin A.

6.3 QEMU PCI test device

`pci-testdev` is a device used for testing low level IO.

The device implements up to three BARs: BAR0, BAR1 and BAR2. Each of BAR 0+1 can be memory or IO. Guests must detect BAR types and act accordingly.

BAR 0+1 size is up to 4K bytes each. BAR 0+1 starts with the following header:

```
typedef struct PCITestDevHdr {
    uint8_t test;        /* write-only, starts a given test number */
    uint8_t width_type;  /*
                          * read-only, type and width of access for a given test.
                          * 1,2,4 for byte,word or long write.
                          * any other value if test not supported on this BAR
                          */
    uint8_t pad0[2];
    uint32_t offset;     /* read-only, offset in this BAR for a given test */
    uint32_t data;       /* read-only, data to use for a given test */
    uint32_t count;      /* for debugging. number of writes detected. */
    uint8_t name[];      /* for debugging. 0-terminated ASCII string. */
} PCITestDevHdr;
```

All registers are little endian.

The device is expected to always implement tests 0 to N on each BAR, and to add new tests with higher numbers. In this way a guest can scan test numbers until it detects an access type that it does not support on this BAR, then stop.

BAR2 is a 64bit memory BAR, without backing storage. It is disabled by default and can be enabled using the `membar=<size>` property. This can be used to test whether guests handle PCI BARs of a specific (possibly quite large) size correctly.

6.4 POWER9 XIVE interrupt controller

The POWER9 processor comes with a new interrupt controller architecture, called XIVE as "eXternal Interrupt Virtualization Engine".

Compared to the previous architecture, the main characteristics of XIVE are to support a larger number of interrupt sources and to deliver interrupts directly to virtual processors without hypervisor assistance. This removes the context switches required for the delivery process.

6.4.1 XIVE architecture

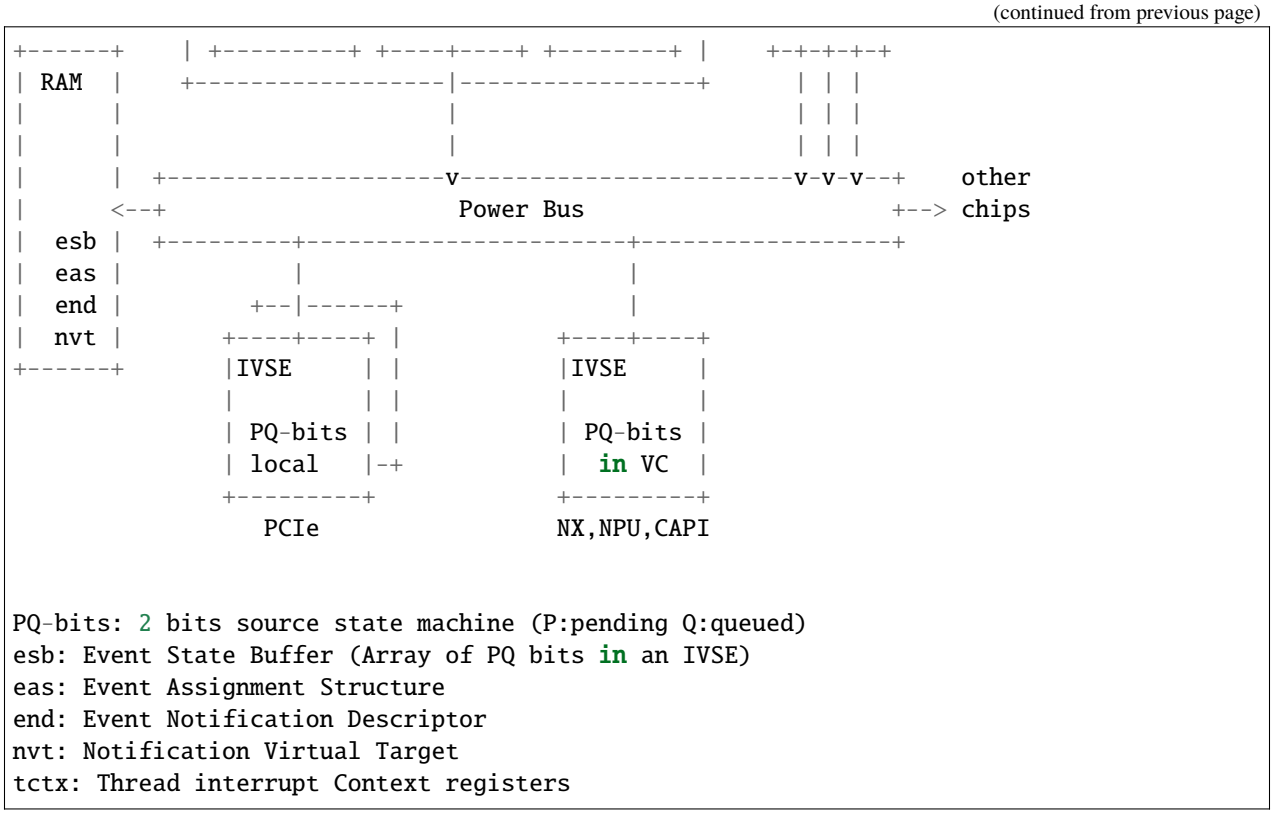
The XIVE IC is composed of three sub-engines, each taking care of a processing layer of external interrupts:

- Interrupt Virtualization Source Engine (IVSE), or Source Controller (SC). These are found in PCI PHBs, in the Processor Service Interface (PSI) host bridge Controller, but also inside the main controller for the core IPIs and other sub-chips (NX, CAP, NPU) of the chip/processor. They are configured to feed the IVRE with events.
- Interrupt Virtualization Routing Engine (IVRE) or Virtualization Controller (VC). It handles event coalescing and perform interrupt routing by matching an event source number with an Event Notification Descriptor (END).
- Interrupt Virtualization Presentation Engine (IVPE) or Presentation Controller (PC). It maintains the interrupt context state of each thread and handles the delivery of the external interrupt to the thread.

```
            XIVE Interrupt Controller
            +-------------------------------------+      IPIs
            | +---------+ +---------+ +--------+ |    +-------+
            | |IVRE     | |Common Q | |IVPE    |----> | CORES |
            | |     esb | |         | |        |----> |       |
            | |     eas | |  Bridge | |   tctx |----> |       |
            | |SC   end | |         | |    nvt | |    |       |
```

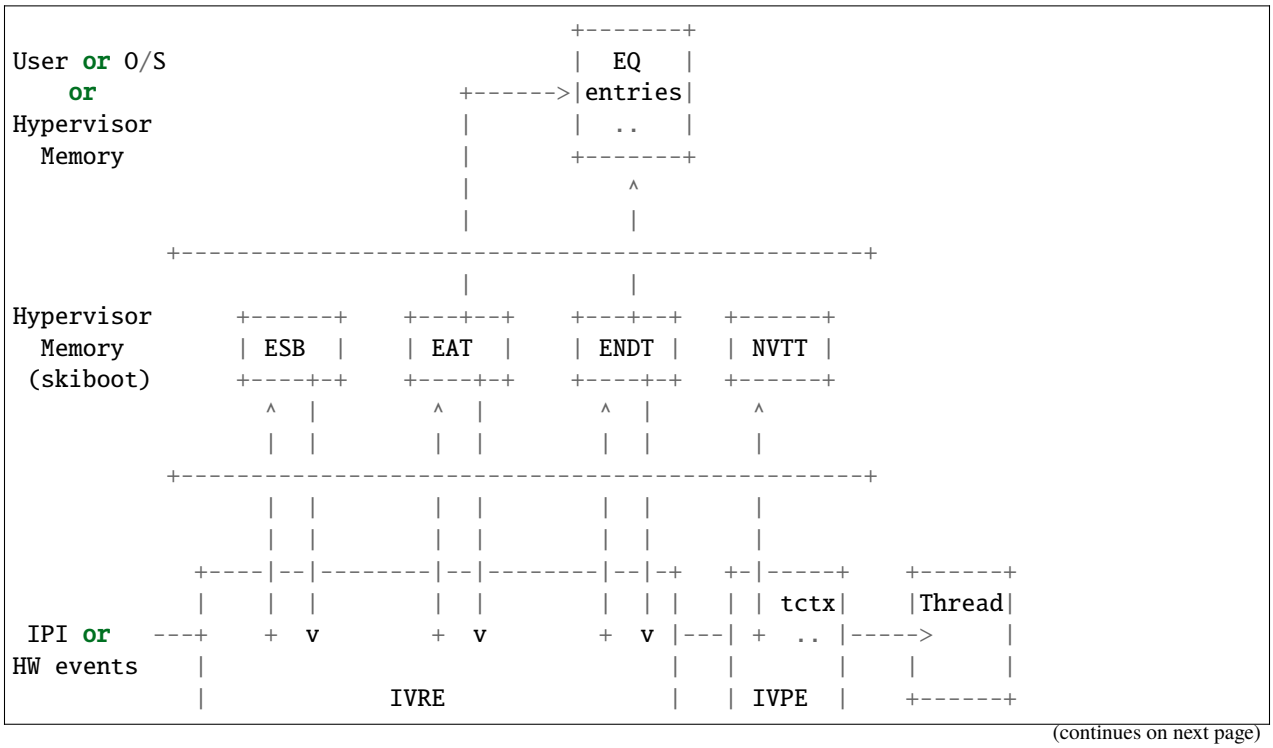
(continues on next page)

(continued from previous page)



XIVE internal tables

Each of the sub-engines uses a set of tables to redirect interrupts from event sources to CPU threads.



(continues on next page)

(continued from previous page)

```
                +---------------------------------+    +-------+
```

The IVSE have a 2-bits state machine, P for pending and Q for queued, for each source that allows events to be triggered. They are stored in an Event State Buffer (ESB) array and can be controlled by MMIOs.

If the event is let through, the IVRE looks up in the Event Assignment Structure (EAS) table for an Event Notification Descriptor (END) configured for the source. Each Event Notification Descriptor defines a notification path to a CPU and an in-memory Event Queue, in which will be enqueued an EQ data for the O/S to pull.

The IVPE determines if a Notification Virtual Target (NVT) can handle the event by scanning the thread contexts of the VCPUs dispatched on the processor HW threads. It maintains the interrupt context state of each thread in a NVT table.

XIVE thread interrupt context

The XIVE presenter can generate four different exceptions to its HW threads:

- hypervisor exception
- O/S exception
- Event-Based Branch (user level)
- msgsnd (doorbell)

Each exception has a state independent from the others called a Thread Interrupt Management context. This context is a set of registers which lets the thread handle priority management and interrupt acknowledgment among other things. The most important ones being :

- Interrupt Priority Register (PIPR)
- Interrupt Pending Buffer (IPB)
- Current Processor Priority (CPPR)
- Notification Source Register (NSR)

TIMA

The Thread Interrupt Management registers are accessible through a specific MMIO region, called the Thread Interrupt Management Area (TIMA), four aligned pages, each exposing a different view of the registers. First page (page address ending in `0b00`) gives access to the entire context and is reserved for the ring 0 view for the physical thread context. The second (page address ending in `0b01`) is for the hypervisor, ring 1 view. The third (page address ending in `0b10`) is for the operating system, ring 2 view. The fourth (page address ending in `0b11`) is for user level, ring 3 view.

Interrupt flow from an O/S perspective

After an event data has been enqueued in the O/S Event Queue, the IVPE raises the bit corresponding to the priority of the pending interrupt in the register IBP (Interrupt Pending Buffer) to indicate that an event is pending in one of the 8 priority queues. The Pending Interrupt Priority Register (PIPR) is also updated using the IPB. This register represent the priority of the most favored pending notification.

The PIPR is then compared to the Current Processor Priority Register (CPPR). If it is more favored (numerically less than), the CPU interrupt line is raised and the EO bit of the Notification Source Register (NSR) is updated to notify the presence of an exception for the O/S. The O/S acknowledges the interrupt with a special load in the Thread Interrupt Management Area.

The O/S handles the interrupt and when done, performs an EOI using a MMIO operation on the ESB management page of the associate source.

6.4.2 Overview of the QEMU models for XIVE

The XiveSource models the IVSE in general, internal and external. It handles the source ESBs and the MMIO interface to control them.

The XiveNotifier is a small helper interface interconnecting the XiveSource to the XiveRouter.

The XiveRouter is an abstract model acting as a combined IVRE and IVPE. It routes event notifications using the EAS and END tables to the IVPE sub-engine which does a CAM scan to find a CPU to deliver the exception. Storage should be provided by the inheriting classes.

XiveEnDSource is a special source object. It exposes the END ESB MMIOs of the Event Queues which are used for coalescing event notifications and for escalation. Not used on the field, only to sync the EQ cache in OPAL.

Finally, the XiveTCTX contains the interrupt state context of a thread, four sets of registers, one for each exception that can be delivered to a CPU. These contexts are scanned by the IVPE to find a matching VP when a notification is triggered. It also models the Thread Interrupt Management Area (TIMA), which exposes the thread context registers to the CPU for interrupt management.

6.5 QEMU and ACPI BIOS Generic Event Device interface

The ACPI *Generic Event Device* (GED) is a HW reduced platform specific device introduced in ACPI v6.1 that handles all platform events, including the hotplug ones. GED is modelled as a device in the namespace with a _HID defined to be ACPI0013. This document describes the interface between QEMU and the ACPI BIOS.

GED allows HW reduced platforms to handle interrupts in ACPI ASL statements. It follows a very similar approach to the _EVT method from GPIO events. All interrupts are listed in _CRS and the handler is written in _EVT method. However, the QEMU implementation uses a single interrupt for the GED device, relying on an IO memory region to communicate the type of device affected by the interrupt. This way, we can support up to 32 events with a unique interrupt.

Here is an example,

```
Device (\_SB.GED)
{
    Name (_HID, "ACPI0013")
    Name (_UID, Zero)
    Name (_CRS, ResourceTemplate ()
    {
        Interrupt (ResourceConsumer, Edge, ActiveHigh, Exclusive, ,, )
        {
            0x00000029,
        }
    })
    OperationRegion (EREG, SystemMemory, 0x09080000, 0x04)
    Field (EREG, DWordAcc, NoLock, WriteAsZeros)
    {
        ESEL,   32
    }
    Method (_EVT, 1, Serialized)
    {
```

(continues on next page)

(continued from previous page)

```
        Local0 = ESEL // ESEL = IO memory region which specifies the
                      // device type.
        If (((Local0 & One) == One))
        {
            MethodEvent1()
        }
        If ((Local0 & 0x2) == 0x2)
        {
            MethodEvent2()
        }
        ...
    }
}
```

6.5.1 GED IO interface (4 byte access)

read access:

```
[0x0-0x3] Event selector bit field (32 bit) set by QEMU.

 bits:
    0: Memory hotplug event
    1: System power down event
    2: NVDIMM hotplug event
 3-31: Reserved
```

write_access:

Nothing is expected to be written into GED IO memory

6.6 QEMU TPM Device

6.6.1 Guest-side hardware interface

TIS interface

The QEMU TPM emulation implements a TPM TIS hardware interface following the Trusted Computing Group's specification "TCG PC Client Specific TPM Interface Specification (TIS)", Specification Version 1.3, 21 March 2013. (see the TIS specification, or a later version of it).

The TIS interface makes a memory mapped IO region in the area 0xfed40000-0xfed44fff available to the guest operating system.

QEMU files related to TPM TIS interface:

- `hw/tpm/tpm_tis_common.c`
- `hw/tpm/tpm_tis_isa.c`
- `hw/tpm/tpm_tis_sysbus.c`
- `hw/tpm/tpm_tis_i2c.c`

- `hw/tpm/tpm_tis.h`

Both an ISA device and a sysbus device are available. The former is used with pc/q35 machine while the latter can be instantiated in the Arm virt machine.

An I2C device support is also provided which can be instantiated in the Arm based emulation machines. This device only supports the TPM 2 protocol.

CRB interface

QEMU also implements a TPM CRB interface following the Trusted Computing Group's specification "TCG PC Client Platform TPM Profile (PTP) Specification", Family "2.0", Level 00 Revision 01.03 v22, May 22, 2017. (see the CRB specification, or a later version of it)

The CRB interface makes a memory mapped IO region in the area 0xfed40000-0xfed40fff (1 locality) available to the guest operating system.

QEMU files related to TPM CRB interface:

- `hw/tpm/tpm_crb.c`

SPAPR interface

pSeries (ppc64) machines offer a tpm-spapr device model.

QEMU files related to the SPAPR interface:

- `hw/tpm/tpm_spapr.c`

6.6.2 fw_cfg interface

The bios/firmware may read the "`etc/tpm/config`" fw_cfg entry for configuring the guest appropriately.

The entry of 6 bytes has the following content, in little-endian:

```
#define TPM_VERSION_UNSPEC          0
#define TPM_VERSION_1_2             1
#define TPM_VERSION_2_0             2

#define TPM_PPI_VERSION_NONE        0
#define TPM_PPI_VERSION_1_30        1

struct FwCfgTPMConfig {
    uint32_t tpmppi_address;         /* PPI memory location */
    uint8_t tpm_version;             /* TPM version */
    uint8_t tpmppi_version;          /* PPI version */
};
```

6.6.3 ACPI interface

The TPM device is defined with ACPI ID "PNP0C31". QEMU builds a SSDT and passes it into the guest through the fw_cfg device. The device description contains the base address of the TIS interface 0xfed40000 and the size of the MMIO area (0x5000). In case a TPM2 is used by QEMU, a TPM2 ACPI table is also provided. The device is described to be used in polling mode rather than interrupt mode primarily because no unused IRQ could be found.

To support measurement logs to be written by the firmware, e.g. SeaBIOS, a TCPA table is implemented. This table provides a 64kb buffer where the firmware can write its log into. For TPM 2 only a more recent version of the TPM2 table provides support for measurements logs and a TCPA table does not need to be created.

The TCPA and TPM2 ACPI tables follow the Trusted Computing Group specification "TCG ACPI Specification" Family "1.2" and "2.0", Level 00 Revision 00.37. (see the ACPI specification, or a later version of it)

ACPI PPI Interface

QEMU supports the Physical Presence Interface (PPI) for TPM 1.2 and TPM 2. This interface requires ACPI and firmware support. (see the PPI specification)

PPI enables a system administrator (root) to request a modification to the TPM upon reboot. The PPI specification defines the operation requests and the actions the firmware has to take. The system administrator passes the operation request number to the firmware through an ACPI interface which writes this number to a memory location that the firmware knows. Upon reboot, the firmware finds the number and sends commands to the TPM. The firmware writes the TPM result code and the operation request number to a memory location that ACPI can read from and pass the result on to the administrator.

The PPI specification defines a set of mandatory and optional operations for the firmware to implement. The ACPI interface also allows an administrator to list the supported operations. In QEMU the ACPI code is generated by QEMU, yet the firmware needs to implement support on a per-operations basis, and different firmwares may support a different subset. Therefore, QEMU introduces the virtual memory device for PPI where the firmware can indicate which operations it supports and ACPI can enable the ones that are supported and disable all others. This interface lies in main memory and has the following layout:

Field	Length	Offset	Description
`func`	0x100	0x000	Firmware sets values for each supported operation. See defined values below.
`ppin`	0x1	0x100	SMI interrupt to use. Set by firmware. Not supported.
`ppip`	0x4	0x101	ACPI function index to pass to SMM code. Set by ACPI. Not supported.
`pprp`	0x4	0x105	Result of last executed operation. Set by firmware. See function index 5 for values.
`pprq`	0x4	0x109	Operation request number to execute. See 'Physical Presence Interface Operation Summary' tables in specs. Set by ACPI.
`pprm`	0x4	0x10d	Operation request optional parameter. Values depend on operation. Set by ACPI.
`lppr`	0x4	0x111	Last executed operation request number. Copied from pprq field by firmware.
`fret`	0x4	0x115	Result code from SMM function. Not supported.
`res1`	0x40	0x119	Reserved for future use
`next_step`	0x1	0x159	Operation to execute after reboot by firmware. Used by firmware.
`movv`	0x1	0x15a	Memory overwrite variable

The following values are supported for the `func` field. They correspond to the values used by ACPI function index 8.

Value	Description
0	Operation is not implemented.
1	Operation is only accessible through firmware.
2	Operation is blocked for OS by firmware configuration.
3	Operation is allowed and physically present user required.
4	Operation is allowed and physically present user is not required.

The location of the table is given by the fw_cfg `tpmppi_address` field. The PPI memory region size is 0x400 (`TPM_PPI_ADDR_SIZE`) to leave enough room for future updates.

QEMU files related to TPM ACPI tables:

- `hw/i386/acpi-build.c`
- `include/hw/acpi/tpm.h`

6.6.4 TPM backend devices

The TPM implementation is split into two parts, frontend and backend. The frontend part is the hardware interface, such as the TPM TIS interface described earlier, and the other part is the TPM backend interface. The backend interfaces implement the interaction with a TPM device, which may be a physical or an emulated device. The split between the front- and backend devices allows a frontend to be connected with any available backend. This enables the TIS interface to be used with the passthrough backend or the swtpm backend.

QEMU files related to TPM backends:

- `backends/tpm.c`
- `include/sysemu/tpm.h`
- `include/sysemu/tpm_backend.h`

The QEMU TPM passthrough device

In case QEMU is run on Linux as the host operating system it is possible to make the hardware TPM device available to a single QEMU guest. In this case the user must make sure that no other program is using the device, e.g., /dev/tpm0, before trying to start QEMU with it.

The passthrough driver uses the host's TPM device for sending TPM commands and receiving responses from. Besides that it accesses the TPM device's sysfs entry for support of command cancellation. Since none of the state of a hardware TPM can be migrated between hosts, virtual machine migration is disabled when the TPM passthrough driver is used.

Since the host's TPM device will already be initialized by the host's firmware, certain commands, e.g. `TPM_Startup()`, sent by the virtual firmware for device initialization, will fail. In this case the firmware should not use the TPM.

Sharing the device with the host is generally not a recommended usage scenario for a TPM device. The primary reason for this is that two operating systems can then access the device's single set of resources, such as platform configuration registers (PCRs). Applications or kernel security subsystems, such as the Linux Integrity Measurement Architecture (IMA), are not expecting to share PCRs.

QEMU files related to the TPM passthrough device:

- `backends/tpm/tpm_passthrough.c`
- `backends/tpm/tpm_util.c`
- `include/sysemu/tpm_util.h`

Command line to start QEMU with the TPM passthrough device using the host's hardware TPM `/dev/tpm0`:

```
qemu-system-x86_64 -display sdl -accel kvm \
-m 1024 -boot d -bios bios-256k.bin -boot menu=on \
-tpmdev passthrough,id=tpm0,path=/dev/tpm0 \
-device tpm-tis,tpmdev=tpm0 test.img
```

The following commands should result in similar output inside the VM with a Linux kernel that either has the TPM TIS driver built-in or available as a module (assuming a TPM 2 is passed through):

```
# dmesg | grep -i tpm
[    0.012560] ACPI: TPM2 0x000000000BFFD1900 00004C (v04 BOCHS  \
    BXPC     0000001 BXPC 00000001)

# ls -l /dev/tpm*
crw-rw----. 1 tss root  10,   224 Sep  6 12:36 /dev/tpm0
crw-rw----. 1 tss rss  253, 65536 Sep  6 12:36 /dev/tpmrm0

Starting with Linux 5.12 there are PCR entries for TPM 2 in sysfs:
# find /sys/devices/ -type f | grep pcr-sha
...
/sys/devices/LNXSYSTEM:00/LNXSYBUS:00/MSFT0101:00/tpm/tpm0/pcr-sha256/1
...
/sys/devices/LNXSYSTEM:00/LNXSYBUS:00/MSFT0101:00/tpm/tpm0/pcr-sha256/9
...
```

The QEMU TPM emulator device

The TPM emulator device uses an external TPM emulator called 'swtpm' for sending TPM commands to and receiving responses from. The swtpm program must have been started before trying to access it through the TPM emulator with QEMU.

The TPM emulator implements a command channel for transferring TPM commands and responses as well as a control channel over which control commands can be sent. (see the SWTPM protocol specification)

The control channel serves the purpose of resetting, initializing, and migrating the TPM state, among other things.

The swtpm program behaves like a hardware TPM and therefore needs to be initialized by the firmware running inside the QEMU virtual machine. One necessary step for initializing the device is to send the TPM_Startup command to it. SeaBIOS, for example, has been instrumented to initialize a TPM 1.2 or TPM 2 device using this command.

QEMU files related to the TPM emulator device:

- `backends/tpm/tpm_emulator.c`
- `backends/tpm/tpm_util.c`
- `include/sysemu/tpm_util.h`

The following commands start the swtpm with a UnixIO control channel over a socket interface. They do not need to be run as root.

```
mkdir /tmp/mytpm1
swtpm socket --tpmstate dir=/tmp/mytpm1 \
  --ctrl type=unixio,path=/tmp/mytpm1/swtpm-sock \
  --tpm2 \
  --log level=20
```

Command line to start QEMU with the TPM emulator device communicating with the swtpm (x86):

```
qemu-system-x86_64 -display sdl -accel kvm \
  -m 1024 -boot d -bios bios-256k.bin -boot menu=on \
  -chardev socket,id=chrtpm,path=/tmp/mytpm1/swtpm-sock \
  -tpmdev emulator,id=tpm0,chardev=chrtpm \
  -device tpm-tis,tpmdev=tpm0 test.img
```

In case a pSeries machine is emulated, use the following command line:

```
qemu-system-ppc64 -display sdl -machine pseries,accel=kvm \
  -m 1024 -bios slof.bin -boot menu=on \
  -nodefaults -device VGA -device pci-ohci -device usb-kbd \
  -chardev socket,id=chrtpm,path=/tmp/mytpm1/swtpm-sock \
  -tpmdev emulator,id=tpm0,chardev=chrtpm \
  -device tpm-spapr,tpmdev=tpm0 \
  -device spapr-vscsi,id=scsi0,reg=0x00002000 \
  -device virtio-blk-pci,scsi=off,bus=pci.0,addr=0x3,drive=drive-virtio-disk0,id=virtio-
↪disk0 \
  -drive file=test.img,format=raw,if=none,id=drive-virtio-disk0
```

In case an Arm virt machine is emulated, use the following command line:

```
qemu-system-aarch64 -machine virt,gic-version=3,acpi=off \
  -cpu host -m 4G \
  -nographic -accel kvm \
  -chardev socket,id=chrtpm,path=/tmp/mytpm1/swtpm-sock \
  -tpmdev emulator,id=tpm0,chardev=chrtpm \
  -device tpm-tis-device,tpmdev=tpm0 \
  -device virtio-blk-pci,drive=drv0 \
  -drive format=qcow2,file=hda.qcow2,if=none,id=drv0 \
  -drive if=pflash,format=raw,file=flash0.img,readonly=on \
  -drive if=pflash,format=raw,file=flash1.img
```

In case a ast2600-evb bmc machine is emulated and you want to use a TPM device attached to I2C bus, use the following command line:

```
qemu-system-arm -M ast2600-evb -nographic \
  -kernel arch/arm/boot/zImage \
  -dtb arch/arm/boot/dts/aspeed-ast2600-evb.dtb \
  -initrd rootfs.cpio \
  -chardev socket,id=chrtpm,path=/tmp/mytpm1/swtpm-sock \
  -tpmdev emulator,id=tpm0,chardev=chrtpm \
  -device tpm-tis-i2c,tpmdev=tpm0,bus=aspeed.i2c.bus.12,address=0x2e

For testing, use this command to load the driver to the correct address

echo tpm_tis_i2c 0x2e > /sys/bus/i2c/devices/i2c-12/new_device
```

In case SeaBIOS is used as firmware, it should show the TPM menu item after entering the menu with 'ESC'.

```
Select boot device:
1. DVD/CD [ata1-0: QEMU DVD-ROM ATAPI-4 DVD/CD]
[...]
5. Legacy option rom
```

(continues on next page)

(continued from previous page)

```
t. TPM Configuration
```

The following commands should result in similar output inside the VM with a Linux kernel that either has the TPM TIS driver built-in or available as a module:

```
# dmesg | grep -i tpm
[    0.012560] ACPI: TPM2 0x000000000BFFD1900 00004C (v04 BOCHS  \
    BXPC     0000001 BXPC 00000001)

# ls -l /dev/tpm*
crw-rw----. 1 tss root  10,   224 Sep  6 12:36 /dev/tpm0
crw-rw----. 1 tss rss  253, 65536 Sep  6 12:36 /dev/tpmrm0

Starting with Linux 5.12 there are PCR entries for TPM 2 in sysfs:
# find /sys/devices/ -type f | grep pcr-sha
...
/sys/devices/LNXSYSTEM:00/LNXSYBUS:00/MSFT0101:00/tpm/tpm0/pcr-sha256/1
...
/sys/devices/LNXSYSTEM:00/LNXSYBUS:00/MSFT0101:00/tpm/tpm0/pcr-sha256/9
...
```

6.6.5 Migration with the TPM emulator

The TPM emulator supports the following types of virtual machine migration:

- VM save / restore (migration into a file)
- Network migration
- Snapshotting (migration into storage like QoW2 or QED)

The following command sequences can be used to test VM save / restore.

In a 1st terminal start an instance of a swtpm using the following command:

```
mkdir /tmp/mytpm1
swtpm socket --tpmstate dir=/tmp/mytpm1 \
  --ctrl type=unixio,path=/tmp/mytpm1/swtpm-sock \
  --tpm2 \
  --log level=20
```

In a 2nd terminal start the VM:

```
qemu-system-x86_64 -display sdl -accel kvm \
  -m 1024 -boot d -bios bios-256k.bin -boot menu=on \
  -chardev socket,id=chrtpm,path=/tmp/mytpm1/swtpm-sock \
  -tpmdev emulator,id=tpm0,chardev=chrtpm \
  -device tpm-tis,tpmdev=tpm0 \
  -monitor stdio \
  test.img
```

Verify that the attached TPM is working as expected using applications inside the VM.

To store the state of the VM use the following command in the QEMU monitor in the 2nd terminal:

```
(qemu) migrate "exec:cat > testvm.bin"
(qemu) quit
```

At this point a file called `testvm.bin` should exists and the swtpm and QEMU processes should have ended.

To test 'VM restore' you have to start the swtpm with the same parameters as before. If previously a TPM 2 [–tpm2] was saved, –tpm2 must now be passed again on the command line.

In the 1st terminal restart the swtpm with the same command line as before:

```
swtpm socket --tpmstate dir=/tmp/mytpm1 \
  --ctrl type=unixio,path=/tmp/mytpm1/swtpm-sock \
  --log level=20 --tpm2
```

In the 2nd terminal restore the state of the VM using the additional '-incoming' option.

```
qemu-system-x86_64 -display sdl -accel kvm \
  -m 1024 -boot d -bios bios-256k.bin -boot menu=on \
  -chardev socket,id=chrtpm,path=/tmp/mytpm1/swtpm-sock \
  -tpmdev emulator,id=tpm0,chardev=chrtpm \
  -device tpm-tis,tpmdev=tpm0 \
  -incoming "exec:cat < testvm.bin" \
  test.img
```

Troubleshooting migration

There are several reasons why migration may fail. In case of problems, please ensure that the command lines adhere to the following rules and, if possible, that identical versions of QEMU and swtpm are used at all times.

VM save and restore:

- QEMU command line parameters should be identical apart from the '-incoming' option on VM restore
- swtpm command line parameters should be identical

VM migration to 'localhost':

- QEMU command line parameters should be identical apart from the '-incoming' option on the destination side
- swtpm command line parameters should point to two different directories on the source and destination swtpm (–tpmstate dir=...) (especially if different versions of libtpms were to be used on the same machine).

VM migration across the network:

- QEMU command line parameters should be identical apart from the '-incoming' option on the destination side
- swtpm command line parameters should be identical

VM Snapshotting:

- QEMU command line parameters should be identical
- swtpm command line parameters should be identical

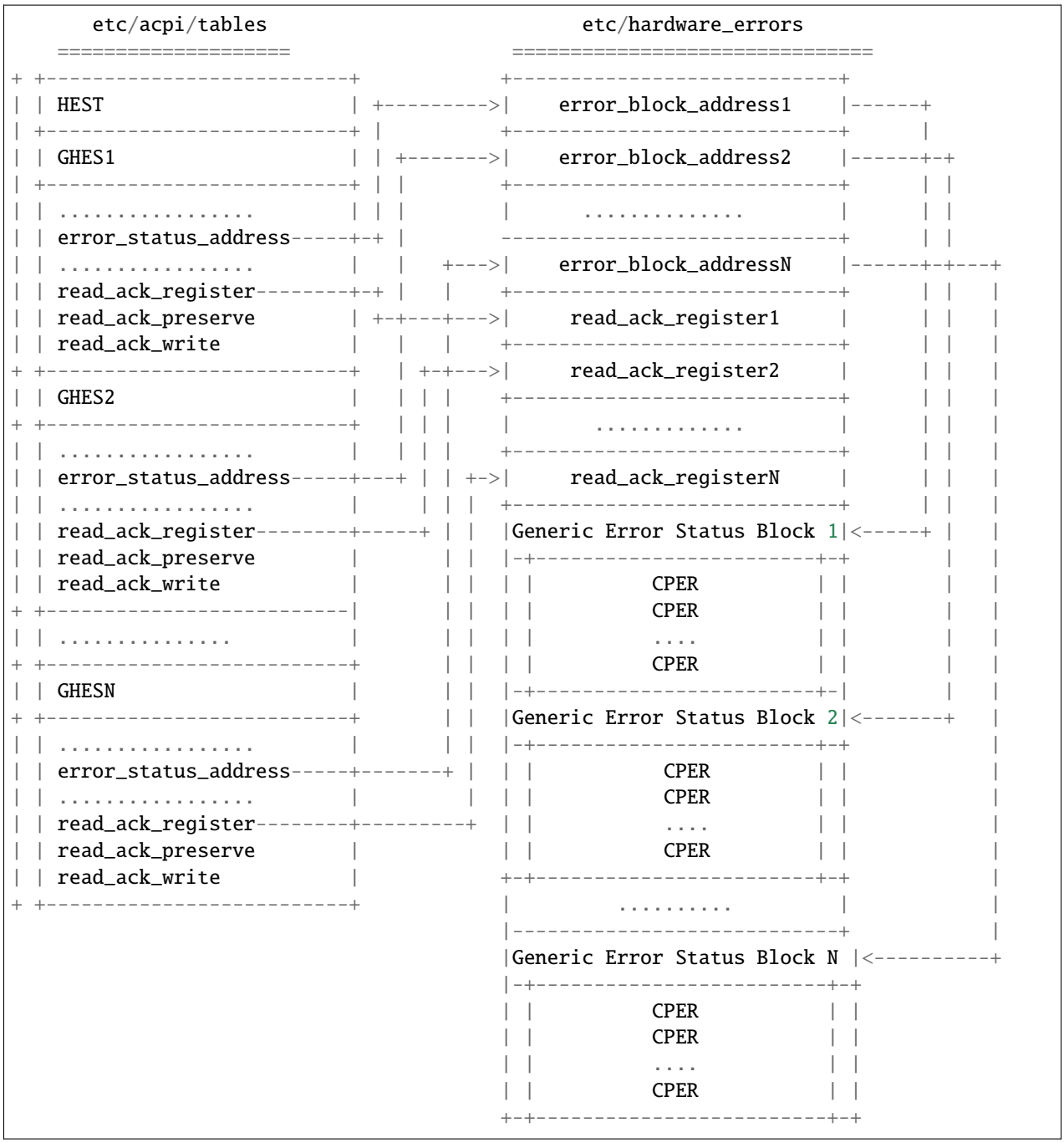
Besides that, migration failure reasons on the swtpm level may include the following:

- the versions of the swtpm on the source and destination sides are incompatible
  - downgrading of TPM state may not be supported
  - the source and destination libtpms were compiled with different compile-time options and the destination side refuses to accept the state

- different migration keys are used on the source and destination side and the destination side cannot decrypt the migrated state (swtpm … –migration-key …)

6.7 APEI tables generating and CPER record

6.7.1 Design Details



(1) QEMU generates the ACPI HEST table. This table goes in the current "etc/acpi/tables" fw_cfg blob. Each error source has different notification types.

(2) A new fw_cfg blob called "etc/hardware_errors" is introduced. QEMU also needs to populate this blob. The "etc/hardware_errors" fw_cfg blob contains an address registers table and an Error Status Data Block table.

(3) The address registers table contains N Error Block Address entries and N Read Ack Register entries. The size for each entry is 8-byte. The Error Status Data Block table contains N Error Status Data Block entries. The size for each entry is 4096(0x1000) bytes. The total size for the "etc/hardware_errors" fw_cfg blob is (N * 8 * 2 + N * 4096) bytes. N is the number of the kinds of hardware error sources.

(4) QEMU generates the ACPI linker/loader script for the firmware. The firmware pre-allocates memory for "etc/acpi/tables", "etc/hardware_errors" and copies blob contents there.

(5) QEMU generates N ADD_POINTER commands, which patch addresses in the "error_status_address" fields of the HEST table with a pointer to the corresponding "address registers" in the "etc/hardware_errors" blob.

(6) QEMU generates N ADD_POINTER commands, which patch addresses in the "read_ack_register" fields of the HEST table with a pointer to the corresponding "read_ack_register" within the "etc/hardware_errors" blob.

(7) QEMU generates N ADD_POINTER commands for the firmware, which patch addresses in the "error_block_address" fields with a pointer to the respective "Error Status Data Block" in the "etc/hardware_errors" blob.

(8) QEMU defines a third and write-only fw_cfg blob which is called "etc/hardware_errors_addr". Through that blob, the firmware can send back the guest-side allocation addresses to QEMU. The "etc/hardware_errors_addr" blob contains a 8-byte entry. QEMU generates a single WRITE_POINTER command for the firmware. The firmware will write back the start address of "etc/hardware_errors" blob to the fw_cfg file "etc/hardware_errors_addr".

(9) When QEMU gets a SIGBUS from the kernel, QEMU writes CPER into corresponding "Error Status Data Block", guest memory, and then injects platform specific interrupt (in case of arm/virt machine it's Synchronous External Abort) as a notification which is necessary for notifying the guest.

(10) This notification (in virtual hardware) will be handled by the guest kernel, on receiving notification, guest APEI driver could read the CPER error and take appropriate action.

(11) kvm_arch_on_sigbus_vcpu() uses source_id as index in "etc/hardware_errors" to find out "Error Status Data Block" entry corresponding to error source. So supported source_id values should be assigned here and not be changed afterwards to make sure that guest will write error into expected "Error Status Data Block" even if guest was migrated to a newer QEMU.

6.8 QEMU<->ACPI BIOS CPU hotplug interface

QEMU supports CPU hotplug via ACPI. This document describes the interface between QEMU and the ACPI BIOS.

ACPI BIOS GPE.2 handler is dedicated for notifying OS about CPU hot-add and hot-remove events.

6.8.1 Legacy ACPI CPU hotplug interface registers

CPU present bitmap for:

- ICH9-LPC (IO port 0x0cd8-0xcf7, 1-byte access)
- PIIX-PM (IO port 0xaf00-0xaf1f, 1-byte access)
- One bit per CPU. Bit position reflects corresponding CPU APIC ID. Read-only.
- The first DWORD in bitmap is used in write mode to switch from legacy to modern CPU hotplug interface, write 0 into it to do switch.

QEMU sets corresponding CPU bit on hot-add event and issues SCI with GPE.2 event set. CPU present map is read by ACPI BIOS GPE.2 handler to notify OS about CPU hot-add events. CPU hot-remove isn't supported.

6.8.2 Modern ACPI CPU hotplug interface registers

Register block base address:

- ICH9-LPC IO port 0x0cd8
- PIIX-PM IO port 0xaf00

Register block size:

- ACPI_CPU_HOTPLUG_REG_LEN = 12

All accesses to registers described below, imply little-endian byte order.

Reserved registers behavior:

- write accesses are ignored
- read accesses return all bits set to 0.

The last stored value in 'CPU selector' must refer to a possible CPU, otherwise

- reads from any register return 0
- writes to any other register are ignored until valid value is stored into it

On QEMU start, 'CPU selector' is initialized to a valid value, on reset it keeps the current value.

Read access behavior

offset [0x0-0x3]
Command data 2: (DWORD access)

If value last stored in 'Command field' is:

0:
reads as 0x0

3:
upper 32 bits of architecture specific CPU ID value

other values:
reserved

offset [0x4]
CPU device status fields: (1 byte access)

bits:

0:
Device is enabled and may be used by guest

1:
Device insert event, used to distinguish device for which no device check event to OSPM was issued. It's valid only when bit 0 is set.

2:
Device remove event, used to distinguish device for which no device eject request to OSPM was issued. Firmware must ignore this bit.

3:
reserved and should be ignored by OSPM

4:
if set to 1, OSPM requests firmware to perform device eject.

5-7:
reserved and should be ignored by OSPM

offset [0x5-0x7]
reserved

offset [0x8]
Command data: (DWORD access)

If value last stored in 'Command field' is one of:

0:
contains 'CPU selector' value of a CPU with pending event[s]

3:
lower 32 bits of architecture specific CPU ID value (in x86 case: APIC ID)

otherwise:
contains 0

Write access behavior

offset [0x0-0x3]
CPU selector: (DWORD access)

Selects active CPU device. All following accesses to other registers will read/store data from/to selected CPU. Valid values: [0 .. max_cpus)

offset [0x4]
CPU device control fields: (1 byte access)

bits:

0:
reserved, OSPM must clear it before writing to register.

1:
if set to 1 clears device insert event, set by OSPM after it has emitted device check event for the selected CPU device

2:
if set to 1 clears device remove event, set by OSPM after it has emitted device eject request for the selected CPU device.

3:
if set to 1 initiates device eject, set by OSPM when it triggers CPU device removal and calls _EJ0 method or by firmware when bit #4 is set. In case bit #4 were set, it's cleared as part of device eject.

4:
if set to 1, OSPM hands over device eject to firmware. Firmware shall issue device eject request as described above (bit #3) and OSPM should not touch device eject bit (#3) in case it's asked firmware to perform CPU device eject.

5-7:
reserved, OSPM must clear them before writing to register

offset[0x5]
Command field: (1 byte access)

value:

0:
selects a CPU device with inserting/removing events and following reads from 'Command data' register return selected CPU ('CPU selector' value). If no CPU with events found, the current 'CPU selector' doesn't change and corresponding insert/remove event flags are not modified.

1:
following writes to 'Command data' register set OST event register in QEMU

2:
following writes to 'Command data' register set OST status register in QEMU

3:
following reads from 'Command data' and 'Command data 2' return architecture specific CPU ID value for currently selected CPU.

other values:
reserved

offset [0x6-0x7]
reserved

offset [0x8]
Command data: (DWORD access)

If last stored 'Command field' value is:

1:
stores value into OST event register

2:
stores value into OST status register, triggers ACPI_DEVICE_OST QMP event from QEMU to external applications with current values of OST event and status registers.

other values:
reserved

6.8.3 Typical usecases

(x86) Detecting and enabling modern CPU hotplug interface

QEMU starts with legacy CPU hotplug interface enabled. Detecting and switching to modern interface is based on the 2 legacy CPU hotplug features:

1. Writes into CPU bitmap are ignored.
2. CPU bitmap always has bit #0 set, corresponding to boot CPU.

Use following steps to detect and enable modern CPU hotplug interface:

1. Store 0x0 to the 'CPU selector' register, attempting to switch to modern mode
2. Store 0x0 to the 'CPU selector' register, to ensure valid selector value
3. Store 0x0 to the 'Command field' register
4. Read the 'Command data 2' register. If read value is 0x0, the modern interface is enabled. Otherwise legacy or no CPU hotplug interface available

Get a cpu with pending event

1. Store 0x0 to the 'CPU selector' register.
2. Store 0x0 to the 'Command field' register.
3. Read the 'CPU device status fields' register.
4. If both bit #1 and bit #2 are clear in the value read, there is no CPU with a pending event and selected CPU remains unchanged.
5. Otherwise, read the 'Command data' register. The value read is the selector of the CPU with the pending event (which is already selected).

Enumerate CPUs present/non present CPUs

1. Set the present CPU count to 0.
2. Set the iterator to 0.
3. Store 0x0 to the 'CPU selector' register, to ensure that it's in a valid state and that access to other registers won't be ignored.
4. Store 0x0 to the 'Command field' register to make 'Command data' register return 'CPU selector' value of selected CPU
5. Read the 'CPU device status fields' register.
6. If bit #0 is set, increment the present CPU count.
7. Increment the iterator.
8. Store the iterator to the 'CPU selector' register.
9. Read the 'Command data' register.
10. If the value read is not zero, goto 05.
11. Otherwise store 0x0 to the 'CPU selector' register, to put it into a valid state and exit. The iterator at this point equals "max_cpus".

6.9 QEMU<->ACPI BIOS memory hotplug interface

ACPI BIOS GPE.3 handler is dedicated for notifying OS about memory hot-add and hot-remove events.

6.9.1 Memory hot-plug interface (IO port 0xa00-0xa17, 1-4 byte access)

Read access behavior

[0x0-0x3]
Lo part of memory device phys address

[0x4-0x7]
Hi part of memory device phys address

[0x8-0xb]
Lo part of memory device size in bytes

[0xc-0xf]
Hi part of memory device size in bytes

[0x10-0x13]
Memory device proximity domain

[0x14]
Memory device status fields

bits:

0:
Device is enabled and may be used by guest

1:
Device insert event, used to distinguish device for which no device check event to OSPM was issued. It's valid only when bit 1 is set.

2:
Device remove event, used to distinguish device for which no device eject request to OSPM was issued.

3-7:
reserved and should be ignored by OSPM

[0x15-0x17]
reserved

Write access behavior

[0x0-0x3]
Memory device slot selector, selects active memory device. All following accesses to other registers in 0xa00-0xa17 region will read/store data from/to selected memory device.

[0x4-0x7]
OST event code reported by OSPM

[0x8-0xb]
OST status code reported by OSPM

[0xc-0x13]
reserved, writes into it are ignored

[0x14]
Memory device control fields

bits:

0:
reserved, OSPM must clear it before writing to register. Due to BUG in versions prior 2.4 that field isn't cleared when other fields are written. Keep it reserved and don't try to reuse it.

1:
if set to 1 clears device insert event, set by OSPM after it has emitted device check event for the selected memory device

2:
if set to 1 clears device remove event, set by OSPM after it has emitted device eject request for the selected memory device

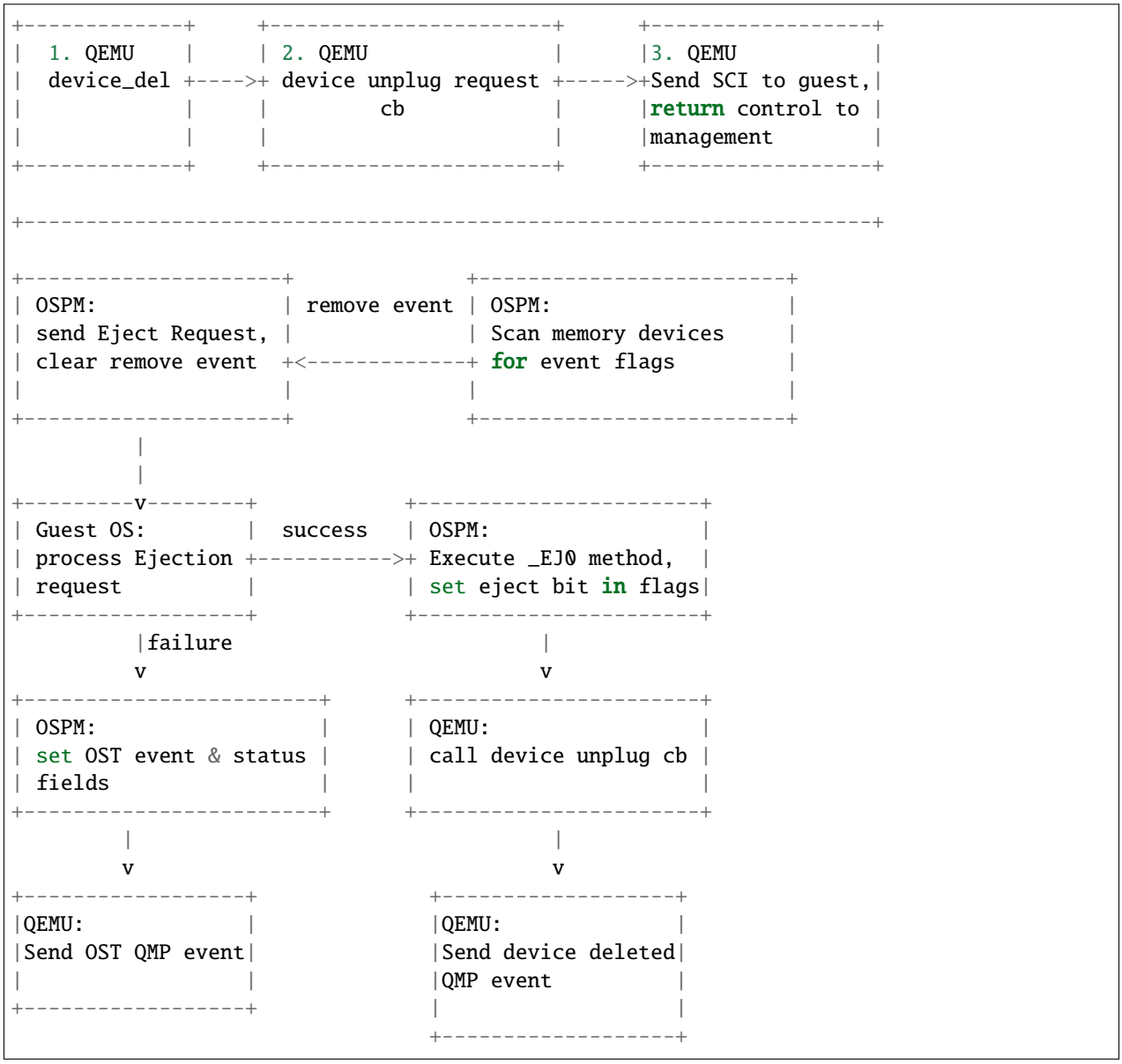
3:
if set to 1 initiates device eject, set by OSPM when it triggers memory device removal and calls _EJ0 method

4-7:
reserved, OSPM must clear them before writing to register

Selecting memory device slot beyond present range has no effect on platform:

- write accesses to memory hot-plug registers not documented above are ignored
- read accesses to memory hot-plug registers not documented above return all bits set to 1.

6.9.2 Memory hot remove process diagram



6.10 QEMU<->ACPI BIOS PCI hotplug interface

QEMU supports PCI hotplug via ACPI, for PCI bus 0. This document describes the interface between QEMU and the ACPI BIOS.

6.10.1 ACPI GPE block (IO ports 0xafe0-0xafe3, byte access)

Generic ACPI GPE block. Bit 1 (GPE.1) used to notify PCI hotplug/eject event to ACPI BIOS, via SCI interrupt.

6.10.2 PCI slot injection notification pending (IO port 0xae00-0xae03, 4-byte access)

Slot injection notification pending. One bit per slot.

Read by ACPI BIOS GPE.1 handler to notify OS of injection events. Read-only.

6.10.3 PCI slot removal notification (IO port 0xae04-0xae07, 4-byte access)

Slot removal notification pending. One bit per slot.

Read by ACPI BIOS GPE.1 handler to notify OS of removal events. Read-only.

6.10.4 PCI device eject (IO port 0xae08-0xae0b, 4-byte access)

Write: Used by ACPI BIOS _EJ0 method to request device removal. One bit per slot.

Read: Hotplug features register. Used by platform to identify features available. Current base feature set (no bits set):

- Read-only "up" register @0xae00, 4-byte access, bit per slot
- Read-only "down" register @0xae04, 4-byte access, bit per slot
- Read/write "eject" register @0xae08, 4-byte access, write: bit per slot eject, read: hotplug feature set
- Read-only hotplug capable register @0xae0c, 4-byte access, bit per slot

6.10.5 PCI removability status (IO port 0xae0c-0xae0f, 4-byte access)

Used by ACPI BIOS _RMV method to indicate removability status to OS. One bit per slot. Read-only.

6.11 QEMU<->ACPI BIOS NVDIMM interface

QEMU supports NVDIMM via ACPI. This document describes the basic concepts of NVDIMM ACPI and the interface between QEMU and the ACPI BIOS.

6.11.1 NVDIMM ACPI Background

NVDIMM is introduced in ACPI 6.0 which defines an NVDIMM root device under _SB scope with a _HID of "ACPI0012". For each NVDIMM present or intended to be supported by platform, platform firmware also exposes an ACPI Namespace Device under the root device.

The NVDIMM child devices under the NVDIMM root device are defined with _ADR corresponding to the NFIT device handle. The NVDIMM root device and the NVDIMM devices can have device specific methods (_DSM) to provide additional functions specific to a particular NVDIMM implementation.

This is an example from ACPI 6.0, a platform contains one NVDIMM:

```
Scope (\_SB){
   Device (NVDR) // Root device
   {
      Name (_HID, "ACPI0012")
      Method (_STA) {...}
      Method (_FIT) {...}
      Method (_DSM, ...) {...}
      Device (NVD)
      {
         Name(_ADR, h) //where h is NFIT Device Handle for this NVDIMM
         Method (_DSM, ...) {...}
      }
   }
}
```

Methods supported on both NVDIMM root device and NVDIMM device

_DSM (Device Specific Method)

It is a control method that enables devices to provide device specific control functions that are consumed by the device driver. The NVDIMM DSM specification can be found at http://pmem.io/documents/NVDIMM_DSM_Interface_Example.pdf

Arguments:

Arg0

A Buffer containing a UUID (16 Bytes)

Arg1

An Integer containing the Revision ID (4 Bytes)

Arg2

An Integer containing the Function Index (4 Bytes)

Arg3

A package containing parameters for the function specified by the UUID, Revision ID, and Function Index

Return Value:

If Function Index = 0, a Buffer containing a function index bitfield. Otherwise, the return value and type depends on the UUID, revision ID and function index which are described in the DSM specification.

Methods on NVDIMM ROOT Device

_FIT(Firmware Interface Table)
It evaluates to a buffer returning data in the format of a series of NFIT Type Structure.

Arguments: None

Return Value: A Buffer containing a list of NFIT Type structure entries.

The detailed definition of the structure can be found at ACPI 6.0: 5.2.25 NVDIMM Firmware Interface Table (NFIT).

6.11.2 QEMU NVDIMM Implementation

QEMU uses 4 bytes IO Port starting from 0x0a18 and a RAM-based memory page for NVDIMM ACPI.

Memory:
QEMU uses BIOS Linker/loader feature to ask BIOS to allocate a memory page and dynamically patch its address into an int32 object named "MEMA" in ACPI.

This page is RAM-based and it is used to transfer data between _DSM method and QEMU. If ACPI has control, this pages is owned by ACPI which writes _DSM input data to it, otherwise, it is owned by QEMU which emulates _DSM access and writes the output data to it.

ACPI writes _DSM Input Data (based on the offset in the page):

[0x0 - 0x3]
4 bytes, NVDIMM Device Handle.

The handle is completely QEMU internal thing, the values in range [1, 0xFFFF] indicate nvdimm device. Other values are reserved for other purposes.

Reserved handles:

- 0 is reserved for nvdimm root device named NVDR.
- 0x10000 is reserved for QEMU internal DSM function called on the root device.

[0x4 - 0x7]
4 bytes, Revision ID, that is the Arg1 of _DSM method.

[0x8 - 0xB]
4 bytes. Function Index, that is the Arg2 of _DSM method.

[0xC - 0xFFF]
4084 bytes, the Arg3 of _DSM method.

QEMU writes Output Data (based on the offset in the page):

[0x0 - 0x3]
4 bytes, the length of result

[0x4 - 0xFFF]
4092 bytes, the DSM result filled by QEMU

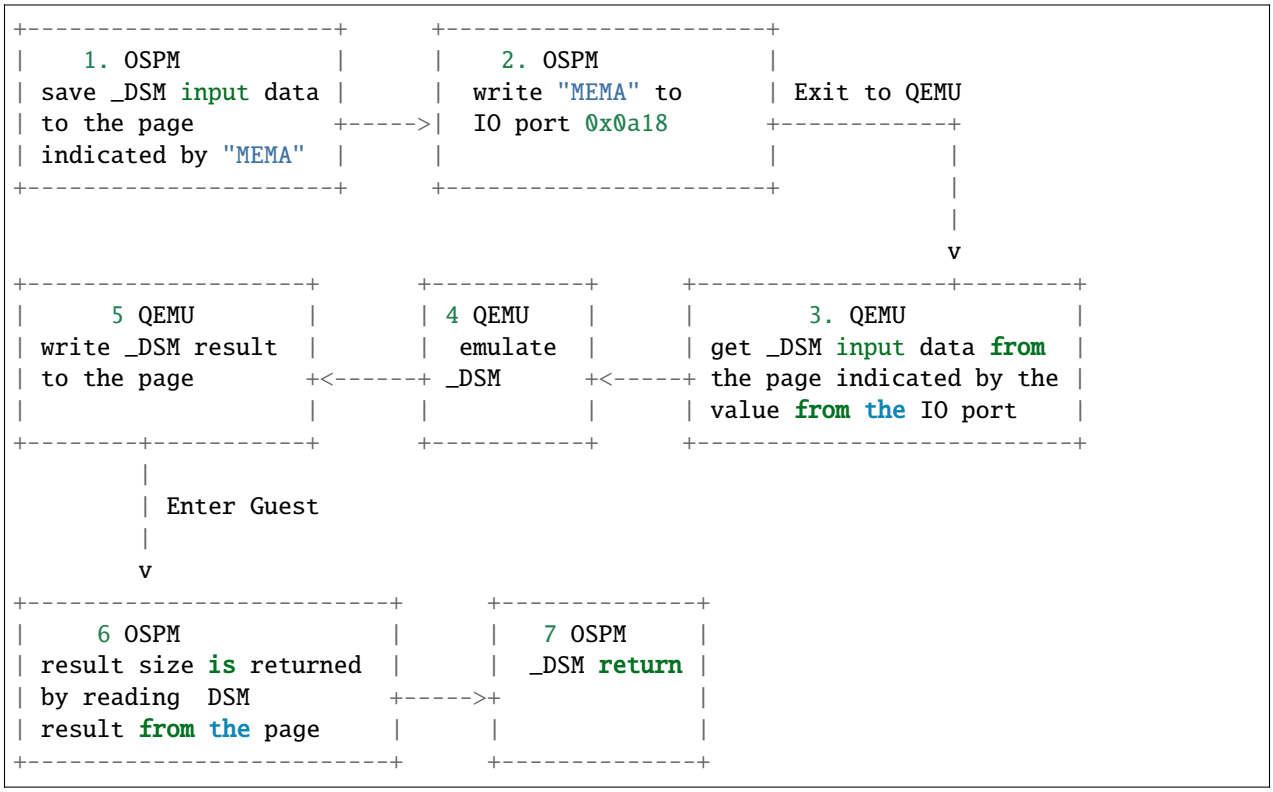
IO Port 0x0a18 - 0xa1b:
ACPI writes the address of the memory page allocated by BIOS to this port then QEMU gets the control and fills the result in the memory page.

Write Access:

[0x0a18 - 0xa1b]
4 bytes, the address of the memory page allocated by BIOS.

6.11.3 _DSM process diagram

"MEMA" indicates the address of memory page allocated by BIOS.



6.11.4 NVDIMM hotplug

ACPI BIOS GPE.4 handler is dedicated for notifying OS about nvdimm device hot-add event.

6.11.5 QEMU internal use only _DSM functions

Read FIT

_FIT method uses _DSM method to fetch NFIT structures blob from QEMU in 1 page sized increments which are then concatenated and returned as _FIT method result.

Input parameters:

Arg0
UUID {set to 648B9CF2-CDA1-4312-8AD9-49C4AF32BD62}

Arg1
Revision ID (set to 1)

Arg2
Function Index, 0x1

Arg3
A package containing a buffer whose layout is as follows:

Field	Length	Offset	Description
offset	4	0	offset in QEMU's NFIT structures blob to read from

Output layout in the dsm memory page:

Field	Length	Offset	Description
length	4	0	length of entire returned data (including this header)
status	4	4	return status codes • 0x0 - success • 0x100 - error caused by NFIT update while read by _FIT wasn't completed • other codes follow Chapter 3 in DSM Spec Rev1
fit data	Varies	8	contains FIT data. This field is present if status field is 0.

The FIT offset is maintained by the OSPM itself, current offset plus the size of the fit data returned by the function is the next offset OSPM should read. When all FIT data has been read out, zero fit data size is returned.

If it returns status code 0x100, OSPM should restart to read FIT (read from offset 0 again).

6.12 ACPI ERST DEVICE

The ACPI ERST device is utilized to support the ACPI Error Record Serialization Table, ERST, functionality. This feature is designed for storing error records in persistent storage for future reference and/or debugging.

The ACPI specification[1], in Chapter "ACPI Platform Error Interfaces (APEI)", and specifically subsection "Error Serialization", outlines a method for storing error records into persistent storage.

The format of error records is described in the UEFI specification[2], in Appendix N "Common Platform Error Record".

While the ACPI specification allows for an NVRAM "mode" (see GET_ERROR_LOG_ADDRESS_RANGE_ATTRIBUTES) where non-volatile RAM is directly exposed for direct access by the OS/guest, this device implements the non-NVRAM "mode". This non-NVRAM "mode" is what is implemented by most BIOS (since flash memory requires programming operations in order to update its contents). Furthermore, as of the time of this writing, Linux only supports the non-NVRAM "mode".

6.12.1 Background/Motivation

Linux uses the persistent storage filesystem, pstore, to record information (eg. dmesg tail) upon panics and shutdowns. Pstore is independent of, and runs before, kdump. In certain scenarios (ie. hosts/guests with root filesystems on NFS/iSCSI where networking software and/or hardware fails, and thus kdump fails), pstore may contain information available for post-mortem debugging.

Two common storage backends for the pstore filesystem are ACPI ERST and UEFI. Most BIOS implement ACPI ERST. UEFI is not utilized in all guests. With QEMU supporting ACPI ERST, it becomes a viable pstore storage backend for virtual machines (as it is now for bare metal machines).

Enabling support for ACPI ERST facilitates a consistent method to capture kernel panic information in a wide range of guests: from resource-constrained microvms to very large guests, and in particular, in direct-boot environments (which would lack UEFI run-time services).

Note that Microsoft Windows also utilizes the ACPI ERST for certain crash information, if available[3].

6.12.2 Configuration|Usage

To use ACPI ERST, a memory-backend-file object and acpi-erst device can be created, for example:

> qemu ... -object memory-backend-file,id=erstnvram,mem-path=acpi-erst.backing,size=0x10000,share=on -device acpi-erst,memdev=erstnvram

For proper operation, the ACPI ERST device needs a memory-backend-file object with the following parameters:

- id: The id of the memory-backend-file object is used to associate this memory with the acpi-erst device.
- size: The size of the ACPI ERST backing storage. This parameter is required.
- mem-path: The location of the ACPI ERST backing storage file. This parameter is also required.
- share: The share=on parameter is required so that updates to the ERST backing store are written to the file.

and ERST device:

- memdev: Is the object id of the memory-backend-file.
- record_size: Specifies the size of the records (or slots) in the backend storage. Must be a power of two value greater than or equal to 4096 (PAGE_SIZE).

6.12.3 PCI Interface

The ERST device is a PCI device with two BARs, one for accessing the programming registers, and the other for accessing the record exchange buffer.

BAR0 contains the programming interface consisting of ACTION and VALUE 64-bit registers. All ERST actions/operations/side effects happen on the write to the ACTION, by design. Any data needed by the action must be placed into VALUE prior to writing ACTION. Reading the VALUE simply returns the register contents, which can be updated by a previous ACTION.

BAR1 contains the 8KiB record exchange buffer, which is the implemented maximum record size.

6.12.4 Backend Storage Format

The backend storage is divided into fixed size "slots", 8KiB in length, with each slot storing a single record. Not all slots need to be occupied, and they need not be occupied in a contiguous fashion. The ability to clear/erase specific records allows for the formation of unoccupied slots.

Slot 0 contains a backend storage header that identifies the contents as ERST and also facilitates efficient access to the records. Depending upon the size of the backend storage, additional slots will be designated to be a part of the slot 0 header. For example, at 8KiB, the slot 0 header can accommodate 1021 records. Thus a storage size of 8MiB (8KiB * 1024) requires an additional slot for use by the header. In this scenario, slot 0 and slot 1 form the backend storage header, and records can be stored starting at slot 2.

Below is an example layout of the backend storage format (for storage size less than 8MiB). The size of the storage is a multiple of 8KiB, and contains N number of slots to store records. The example below shows two records (in CPER format) in the backend storage, while the remaining slots are empty/available.

```
Slot   Record
       <------------------ 8KiB -------------------->
       +-----------------------------------------+
   0   | storage header                          |
       +-----------------------------------------+
   1   | empty/available                         |
       +-----------------------------------------+
   2   | CPER                                    |
       +-----------------------------------------+
   3   | CPER                                    |
       +-----------------------------------------+
 ...   |                                         |
       +-----------------------------------------+
   N   | empty/available                         |
       +-----------------------------------------+
```

The storage header consists of some basic information and an array of CPER record_id's to efficiently access records in the backend storage.

All fields in the header are stored in little endian format.

```
+-----------------------------------------+
| magic                                   | 0x0000
+-----------------------------------------+
| record_offset        | record_size      | 0x0008
+-----------------------------------------+
| record_count         | reserved | version  | 0x0010
+-----------------------------------------+
| record_id[0]                            | 0x0018
+-----------------------------------------+
| record_id[1]                            | 0x0020
+-----------------------------------------+
| record_id[...]                          |
+-----------------------------------------+
| record_id[N]                            | 0x1FF8
+-----------------------------------------+
```

The 'magic' field contains the value 0x524F545354535245.

The 'record_size' field contains the value 0x2000, 8KiB.

The 'record_offset' field points to the first record_id in the array, 0x0018.

The 'version' field contains 0x0100, the first version.

The 'record_count' field contains the number of valid records in the backend storage.

The 'record_id' array fields are the 64-bit record identifiers of the CPER record in the corresponding slot. Stated differently, the location of a CPER record_id in the record_id[] array provides the slot index for the corresponding record in the backend storage.

Note that, for example, with a backend storage less than 8MiB, slot 0 contains the header, so the record_id[0] will never contain a valid CPER record_id. Instead slot 1 is the first available slot and thus record_id_[1] may contain a CPER.

A 'record_id' of all 0s or all 1s indicates an invalid record (ie. the slot is available).

6.12.5 References

[1] "Advanced Configuration and Power Interface Specification",
version 4.0, June 2009.

[2] "Unified Extensible Firmware Interface Specification",
version 2.1, October 2008.

[3] "Windows Hardware Error Architecture", specifically
"Error Record Persistence Mechanism".

6.13 QEMU/Guest Firmware Interface for AMD SEV and SEV-ES

6.13.1 Overview

The guest firmware image (OVMF) may contain some configuration entries which are used by QEMU before the guest launches. These are listed in a GUIDed table at a known location in the firmware image. QEMU parses this table when it loads the firmware image into memory, and then QEMU reads individual entries when their values are needed.

Though nothing in the table structure is SEV-specific, currently all the entries in the table are related to SEV and SEV-ES features.

Table parsing in QEMU

The table is parsed from the footer: first the presence of the table footer GUID (96b582de-1fb2-45f7-baea-a366c55a082d) at 0xffffffd0 is verified. If that is found, two bytes at 0xffffffce are the entire table length.

Then the table is scanned backwards looking for the specific entry GUID.

QEMU files related to parsing and scanning the OVMF table:

- `hw/i386/pc_sysfw_ovmf.c`

The edk2 firmware code that constructs this structure is in the OVMF Reset Vector file.

Table memory layout

GPA	Length	Description
0xffffff80	4	Zero padding
0xffffff84	4	SEV hashes table base address
0xffffff88	4	SEV hashes table size (=0x400)
0xffffff8c	2	SEV hashes table entry length (=0x1a)
0xffffff8e	16	SEV hashes table GUID: 7255371f-3a3b-4b04-927b-1da6efa8d454
0xffffff9e	4	SEV secret block base address
0xffffffa2	4	SEV secret block size (=0xc00)
0xffffffa6	2	SEV secret block entry length (=0x1a)
0xffffffa8	16	SEV secret block GUID: 4c2eb361-7d9b-4cc3-8081-127c90d3d294
0xffffffb8	4	SEV-ES AP reset RIP
0xffffffbc	2	SEV-ES reset block entry length (=0x16)
0xffffffbe	16	SEV-ES reset block entry GUID: 00f771de-1a7e-4fcb-890e-68c77e2fb44e
0xffffffce	2	Length of entire table including table footer GUID and length (=0x72)
0xffffffd0	16	OVMF GUIDed table footer GUID: 96b582de-1fb2-45f7-baea-a366c55a082d
0xffffffe0	8	Application processor entry point code
0xffffffe8	8	“0000VTF0”
0xfffffff0	16	Reset vector code

6.13.2 Table entries description

SEV-ES reset block

Entry GUID: 00f771de-1a7e-4fcb-890e-68c77e2fb44e

For the initial boot of an AP under SEV-ES, the “reset” RIP must be programmed to the RAM area defined by this entry. The entry’s format is:

- IP value [0:15]
- CS segment base [31:16]

A hypervisor reads the CS segment base and IP value. The CS segment base value represents the high order 16-bits of the CS segment base, so the hypervisor must left shift the value of the CS segment base by 16 bits to form the full CS segment base for the CS segment register. It would then program the EIP register with the IP value as read.

SEV secret block

Entry GUID: 4c2eb361-7d9b-4cc3-8081-127c90d3d294

This describes the guest RAM area where the hypervisor should inject the Guest Owner secret (using SEV_LAUNCH_SECRET).

SEV hashes table

Entry GUID: 7255371f-3a3b-4b04-927b-1da6efa8d454

This describes the guest RAM area where the hypervisor should install a table describing the hashes of certain firmware configuration device files that would otherwise be passed in unchecked. The current use is for the kernel, initrd and command line values, but others may be added.

6.14 QEMU Firmware Configuration (fw_cfg) Device

6.14.1 Guest-side Hardware Interface

This hardware interface allows the guest to retrieve various data items (blobs) that can influence how the firmware configures itself, or may contain tables to be installed for the guest OS. Examples include device boot order, ACPI and SMBIOS tables, virtual machine UUID, SMP and NUMA information, kernel/initrd images for direct (Linux) kernel booting, etc.

Selector (Control) Register

- Write only
- Location: platform dependent (IOport or MMIO)
- Width: 16-bit
- Endianness: little-endian (if IOport), or big-endian (if MMIO)

A write to this register sets the index of a firmware configuration item which can subsequently be accessed via the data register.

Setting the selector register will cause the data offset to be set to zero. The data offset impacts which data is accessed via the data register, and is explained below.

Bit14 of the selector register indicates whether the configuration setting is being written. A value of 0 means the item is only being read, and all write access to the data port will be ignored. A value of 1 means the item's data can be overwritten by writes to the data register. In other words, configuration write mode is enabled when the selector value is between 0x4000-0x7fff or 0xc000-0xffff.

Note: As of QEMU v2.4, writes to the fw_cfg data register are no longer supported, and will be ignored (treated as no-ops)!

Note: As of QEMU v2.9, writes are reinstated, but only through the DMA interface (see below). Furthermore, writeability of any specific item is governed independently of Bit14 in the selector key value.

Bit15 of the selector register indicates whether the configuration setting is architecture specific. A value of 0 means the item is a generic configuration item. A value of 1 means the item is specific to a particular architecture. In other words, generic configuration items are accessed with a selector value between 0x0000-0x7fff, and architecture specific configuration items are accessed with a selector value between 0x8000-0xffff.

Data Register

- Read/Write (writes ignored as of QEMU v2.4, but see the DMA interface)
- Location: platform dependent (IOport[1] or MMIO)
- Width: 8-bit (if IOport), 8/16/32/64-bit (if MMIO)
- Endianness: string-preserving

The data register allows access to an array of bytes for each firmware configuration data item. The specific item is selected by writing to the selector register, as described above.

Initially following a write to the selector register, the data offset will be set to zero. Each successful access to the data register will increment the data offset by the appropriate access width.

Each firmware configuration item has a maximum length of data associated with the item. After the data offset has passed the end of this maximum data length, then any reads will return a data value of 0x00, and all writes will be ignored.

An N-byte wide read of the data register will return the next available N bytes of the selected firmware configuration item, as a substring, in increasing address order, similar to memcpy().

Register Locations

x86, x86_64

- Selector Register IOport: 0x510
- Data Register IOport: 0x511
- DMA Address IOport: 0x514

Arm

- Selector Register address: Base + 8 (2 bytes)
- Data Register address: Base + 0 (8 bytes)
- DMA Address address: Base + 16 (8 bytes)

ACPI Interface

The fw_cfg device is defined with ACPI ID `QEMU0002`. Since we expect ACPI tables to be passed into the guest through the fw_cfg device itself, the guest-side firmware can not use ACPI to find fw_cfg. However, once the firmware is finished setting up ACPI tables and hands control over to the guest kernel, the latter can use the fw_cfg ACPI node for a more accurate inventory of in-use IOport or MMIO regions.

[1] On platforms where the data register is exposed as an IOport, its port number will always be one greater than the port number of the selector register. In other words, the two ports overlap, and can not be mapped separately.

Firmware Configuration Items

Signature (Key 0x0000, FW_CFG_SIGNATURE)

The presence of the fw_cfg selector and data registers can be verified by selecting the "signature" item using key 0x0000 (FW_CFG_SIGNATURE), and reading four bytes from the data register. If the fw_cfg device is present, the four bytes read will contain the characters QEMU.

If the DMA interface is available, then reading the DMA Address Register returns 0x51454d5520434647 (QEMU CFG in big-endian format).

Revision / feature bitmap (Key 0x0001, FW_CFG_ID)

A 32-bit little-endian unsigned int, this item is used to check for enabled features.

- Bit 0: traditional interface. Always set.
- Bit 1: DMA interface.

File Directory (Key 0x0019, FW_CFG_FILE_DIR)

Firmware configuration items stored at selector keys 0x0020 or higher (FW_CFG_FILE_FIRST or higher) have an associated entry in a directory structure, which makes it easier for guest-side firmware to identify and retrieve them. The format of this file directory (from fw_cfg.h in the QEMU source tree) is shown here, slightly annotated for clarity:

```
struct FWCfgFiles {         /* the entire file directory fw_cfg item */
    uint32_t count;         /* number of entries, in big-endian format */
    struct FWCfgFile f[];   /* array of file entries, see below */
};

struct FWCfgFile {          /* an individual file entry, 64 bytes total */
    uint32_t size;          /* size of referenced fw_cfg item, big-endian */
    uint16_t select;        /* selector key of fw_cfg item, big-endian */
    uint16_t reserved;
    char name[56];          /* fw_cfg item name, NUL-terminated ascii */
};
```

All Other Data Items

Please consult the QEMU source for the most up-to-date and authoritative list of selector keys and their respective items' purpose, format and writeability.

Ranges

Theoretically, there may be up to 0x4000 generic firmware configuration items, and up to 0x4000 architecturally specific ones.

Selector Reg.	Range Usage
0x0000 - 0x3fff	Generic (0x0000 - 0x3fff, generally RO, possibly RW through the DMA interface in QEMU v2.9+)
0x4000 - 0x7fff	Generic (0x0000 - 0x3fff, RW, ignored in QEMU v2.4+)
0x8000 - 0xbfff	Arch. Specific (0x0000 - 0x3fff, generally RO, possibly RW through the DMA interface in QEMU v2.9+)
0xc000 - 0xffff	Arch. Specific (0x0000 - 0x3fff, RW, ignored in v2.4+)

In practice, the number of allowed firmware configuration items depends on the machine type/version.

6.14.2 Guest-side DMA Interface

If bit 1 of the feature bitmap is set, the DMA interface is present. This does not replace the existing fw_cfg interface, it is an add-on. This interface can be used through the 64-bit wide address register.

The address register is in big-endian format. The value for the register is 0 at startup and after an operation. A write to the least significant half (at offset 4) triggers an operation. This means that operations with 32-bit addresses can be triggered with just one write, whereas operations with 64-bit addresses can be triggered with one 64-bit write or two 32-bit writes, starting with the most significant half (at offset 0).

In this register, the physical address of a `FWCfgDmaAccess` structure in RAM should be written. This is the format of the `FWCfgDmaAccess` structure:

```
typedef struct FWCfgDmaAccess {
    uint32_t control;
    uint32_t length;
    uint64_t address;
} FWCfgDmaAccess;
```

The fields of the structure are in big endian mode, and the field at the lowest address is the `control` field.

The `control` field has the following bits:

- Bit 0: Error
- Bit 1: Read
- Bit 2: Skip
- Bit 3: Select. The upper 16 bits are the selected index.
- Bit 4: Write

When an operation is triggered, if the `control` field has bit 3 set, the upper 16 bits are interpreted as an index of a firmware configuration item. This has the same effect as writing the selector register.

If the `control` field has bit 1 set, a read operation will be performed. `length` bytes for the current selector and offset will be copied into the physical RAM address specified by the `address` field.

If the `control` field has bit 4 set (and not bit 1), a write operation will be performed. `length` bytes will be copied from the physical RAM address specified by the `address` field to the current selector and offset. QEMU prevents starting or finishing the write beyond the end of the item associated with the current selector (i.e., the item cannot be resized). Truncated writes are dropped entirely. Writes to read-only items are also rejected. All of these write errors set bit 0 (the error bit) in the `control` field.

If the `control` field has bit 2 set (and neither bit 1 nor bit 4), a skip operation will be performed. The offset for the current selector will be advanced `length` bytes.

To check the result, read the `control` field:

Error bit set
Something went wrong.

All bits cleared
Transfer finished successfully.

Otherwise
Transfer still in progress (doesn't happen today due to implementation not being async, but may in the future).

6.14.3 Externally Provided Items

Since v2.4, "file" fw_cfg items (i.e., items with selector keys above `FW_CFG_FILE_FIRST`, and with a corresponding entry in the fw_cfg file directory structure) may be inserted via the QEMU command line, using the following syntax:

```
-fw_cfg [name=]<item_name>,file=<path>
```

Or:

```
-fw_cfg [name=]<item_name>,string=<string>
```

Since v5.1, QEMU allows some objects to generate fw_cfg-specific content, the content is then associated with a "file" item using the 'gen_id' option in the command line, using the following syntax:

```
-object <generator-type>,id=<generated_id>,[generator-specific-options] \
-fw_cfg [name=]<item_name>,gen_id=<generated_id>
```

See QEMU man page for more documentation.

Using item_name with plain ASCII characters only is recommended.

Item names beginning with `opt/` are reserved for users. QEMU will never create entries with such names unless explicitly ordered by the user.

To avoid clashes among different users, it is strongly recommended that you use names beginning with `opt/RFQDN/` , where RFQDN is a reverse fully qualified domain name you control. For instance, if SeaBIOS wanted to define additional names, the prefix `opt/org.seabios/` would be appropriate.

For historical reasons, `opt/ovmf/` is reserved for OVMF firmware.

Prefix `opt/org.qemu/` is reserved for QEMU itself.

Use of names not beginning with `opt/` is potentially dangerous and entirely unsupported. QEMU will warn if you try.

Use of names not beginning with `opt/` is tolerated with 'gen_id' (that is, the warning is suppressed), but you must know exactly what you're doing.

All externally provided fw_cfg items are read-only to the guest.

6.15 IBM's Flexible Service Interface (FSI)

The QEMU FSI emulation implements hardware interfaces between ASPEED SOC, FSI master/slave and the end engine.

FSI is a point-to-point two wire interface which is capable of supporting distances of up to 4 meters. FSI interfaces have been used successfully for many years in IBM servers to attach IBM Flexible Support Processors(FSP) to CPUs and IBM ASICs.

FSI allows a service processor access to the internal buses of a host POWER processor to perform configuration or debugging. FSI has long existed in POWER processes and so comes with some baggage, including how it has been integrated into the ASPEED SoC.

Working backwards from the POWER processor, the fundamental pieces of interest for the implementation are: (see the FSI specification for more details)

1. The Common FRU Access Macro (CFAM), an address space containing various "engines" that drive accesses on buses internal and external to the POWER chip. Examples include the SBEFIFO and I2C masters. The engines hang off of an internal Local Bus (LBUS) which is described by the CFAM configuration block.
2. The FSI slave: The slave is the terminal point of the FSI bus for FSI symbols addressed to it. Slaves can be cascaded off of one another. The slave's configuration registers appear in address space of the CFAM to which it is attached.
3. The FSI master: A controller in the platform service processor (e.g. BMC) driving CFAM engine accesses into the POWER chip. At the hardware level FSI is a bit-based protocol supporting synchronous and DMA-driven accesses of engines in a CFAM.
4. The On-Chip Peripheral Bus (OPB): A low-speed bus typically found in POWER processors. This now makes an appearance in the ASPEED SoC due to tight integration of the FSI master IP with the OPB, mainly the existence of an MMIO-mapping of the CFAM address straight onto a sub-region of the OPB address space.
5. An APB-to-OPB bridge enabling access to the OPB from the ARM core in the AST2600. Hardware limitations prevent the OPB from being directly mapped into APB, so all accesses are indirect through the bridge.

The LBUS is modelled to maintain the qdev bus hierarchy and to take advantages of the object model to automatically generate the CFAM configuration block. The configuration block presents engines in the order they are attached to the CFAM's LBUS. Engine implementations should subclass the LBusDevice and set the 'config' member of LBusDeviceClass to match the engine's type.

CFAM designs offer a lot of flexibility, for instance it is possible for a CFAM to be simultaneously driven from multiple FSI links. The modeling is not so complete; it's assumed that each CFAM is attached to a single FSI slave (as a consequence the CFAM subclasses the FSI slave).

As for FSI, its symbols and wire-protocol are not modelled at all. This is not necessary to get FSI off the ground thanks to the mapping of the CFAM address space onto the OPB address space - the models follow this directly and map the CFAM memory region into the OPB's memory region.

The following commands start the `rainier-bmc` machine with built-in FSI model. There are no model specific arguments. Please check this document to learn more about Aspeed `rainier-bmc` machine: (*Aspeed family boards (*-bmc, ast2500-evb, ast2600-evb)*)

```
qemu-system-arm -M rainier-bmc -nographic \
-kernel fitImage-linux.bin \
-dtb aspeed-bmc-ibm-rainier.dtb \
-initrd obmc-phosphor-initramfs.rootfs.cpio.xz \
-drive file=obmc-phosphor-image.rootfs.wic.qcow2,if=sd,index=2 \
-append "rootwait console=ttyS4,115200n8 root=PARTLABEL=rofs-a"
```

The implementation appears as following in the qemu device tree:

```
(qemu) info qtree
bus: main-system-bus
  type System
  ...
  dev: aspeed.apb2opb, id ""
    gpio-out "sysbus-irq" 1
    mmio 000000001e79b000/0000000000001000
    bus: opb.1
      type opb
      dev: fsi.master, id ""
        bus: fsi.bus.1
          type fsi.bus
          dev: cfam.config, id ""
          dev: cfam, id ""
            bus: lbus.1
              type lbus
              dev: scratchpad, id ""
                address = 0 (0x0)
    bus: opb.0
      type opb
      dev: fsi.master, id ""
        bus: fsi.bus.0
          type fsi.bus
          dev: cfam.config, id ""
          dev: cfam, id ""
            bus: lbus.0
              type lbus
              dev: scratchpad, id ""
                address = 0 (0x0)
```

pdbg is a simple application to allow debugging of the host POWER processors from the BMC. (see the pdbg source repository for more details)

```
root@p10bmc:~# pdbg -a getcfam 0x0
p0: 0x0 = 0xc0022d15
```

6.16 VMWare PVSCSI Device Interface

This document describes the VMWare PVSCSI device interface specification, based on the source code of the PVSCSI Linux driver from kernel 3.0.4.

6.16.1 Overview

The interface is based on a memory area shared between hypervisor and VM. The memory area is obtained by driver as a device IO memory resource of `PVSCSI_MEM_SPACE_SIZE` length. The shared memory consists of a registers area and a rings area. The registers area is used to raise hypervisor interrupts and issue device commands. The rings area is used to transfer data descriptors and SCSI commands from VM to hypervisor and to transfer messages produced by hypervisor to VM. Data itself is transferred via virtual scatter-gather DMA.

6.16.2 PVSCSI Device Registers

The length of the registers area is 1 page (`PVSCSI_MEM_SPACE_COMMAND_NUM_PAGES`). The structure of the registers area is described by the `PVSCSIRegOffset` enum. There are registers to issue device commands (with optional short data), issue device interrupts, and control interrupt masking.

6.16.3 PVSCSI Device Rings

There are three rings in shared memory:

Request ring (`struct PVSCSIRingReqDesc *req_ring`)
ring for OS to device requests

Completion ring (`struct PVSCSIRingCmpDesc *cmp_ring`)
ring for device request completions

Message ring (`struct PVSCSIRingMsgDesc *msg_ring`)
ring for messages from device. This ring is optional and the guest might not configure it.

There is a control area (`struct PVSCSIRingsState *rings_state`) used to control rings operation.

6.16.4 PVSCSI Device to Host Interrupts

The following interrupt types are supported by the PVSCSI device:

Completion interrupts (completion ring notifications):

- `PVSCSI_INTR_CMPL_0`
- `PVSCSI_INTR_CMPL_1`

Message interrupts (message ring notifications):

- `PVSCSI_INTR_MSG_0`
- `PVSCSI_INTR_MSG_1`

Interrupts are controlled via the `PVSCSI_REG_OFFSET_INTR_MASK` register. If a bit is set it means the interrupt is enabled, and if it is clear then the interrupt is disabled.

The interrupt modes supported are legacy, MSI and MSI-X. In the case of legacy interrupts, the `PVSCSI_REG_OFFSET_INTR_STATUS` register is used to check which interrupt has arrived. Interrupts are acknowledged when the corresponding bit is written to the interrupt status register.

6.16.5 PVSCSI Device Operation Sequences

Startup sequence

a. Issue `PVSCSI_CMD_ADAPTER_RESET` command

b. Windows driver reads interrupt status register here

c. Issue `PVSCSI_CMD_SETUP_MSG_RING` command with no additional data, check status and disable device messages if error returned (Omitted if device messages disabled by driver configuration)

d. Issue `PVSCSI_CMD_SETUP_RINGS` command, provide rings configuration as `struct PVSCSICmdDescSetupRings`

e. Issue `PVSCSI_CMD_SETUP_MSG_RING` command again, provide rings configuration as `struct PVSCSICmdDescSetupMsgRing`

f. Unmask completion and message (if device messages enabled) interrupts

Shutdown sequence

a. Mask interrupts

b. Flush request ring using `PVSCSI_REG_OFFSET_KICK_NON_RW_IO`

c. Issue `PVSCSI_CMD_ADAPTER_RESET` command

Send request

a. Fill next free request ring descriptor

b. Issue `PVSCSI_REG_OFFSET_KICK_RW_IO` for R/W operations or `PVSCSI_REG_OFFSET_KICK_NON_RW_IO` for other operations

Abort command

a. Issue `PVSCSI_CMD_ABORT_CMD` command

Request completion processing

a. Upon completion interrupt arrival process completion and message (if enabled) rings

6.17 EDU device

This is an educational device for writing (kernel) drivers. Its original intention was to support the Linux kernel lectures taught at the Masaryk University. Students are given this virtual device and are expected to write a driver with I/Os, IRQs, DMAs and such.

The devices behaves very similar to the PCI bridge present in the COMBO6 cards developed under the Liberouter wings. Both PCI device ID and PCI space is inherited from that device.

6.17.1 Command line switches

`-device edu[,dma_mask=mask]`
: `dma_mask` makes the virtual device work with DMA addresses with the given mask. For educational purposes, the device supports only 28 bits (256 MiB) by default. Students shall set dma_mask for the device in the OS driver properly.

6.17.2 PCI specs

PCI ID:
: `1234:11e8`

PCI Region 0:
: I/O memory, 1 MB in size. Users are supposed to communicate with the card through this memory.

6.17.3 MMIO area spec

Only `size == 4` accesses are allowed for addresses < `0x80`. `size == 4` or `size == 8` for the rest.

0x00 (RO)
: [identification] Value is in the form `0xRRrr00edu` where: - `RR` – major version - `rr` – minor version

0x04 (RW)
: [card liveness check] It is a simple value inversion (~ C operator).

0x08 (RW)
: [factorial computation] The stored value is taken and factorial of it is put back here. This happens only after factorial bit in the status register (0x20 below) is cleared.

0x20 (RW)
: [status register] Bitwise OR of:

  0x01
  : computing factorial (RO)

  0x80
  : raise interrupt after finishing factorial computation

0x24 (RO)
: [interrupt status register] It contains values which raised the interrupt (see interrupt raise register below).

0x60 (WO)
: [interrupt raise register] Raise an interrupt. The value will be put to the interrupt status register (using bitwise OR).

0x64 (WO)
: [interrupt acknowledge register] Clear an interrupt. The value will be cleared from the interrupt status register. This needs to be done from the ISR to stop generating interrupts.

0x80 (RW)
: [DMA source address] Where to perform the DMA from.

0x88 (RW)
: [DMA destination address] Where to perform the DMA to.

0x90 (RW)
: [DMA transfer count] The size of the area to perform the DMA on.

0x98 (RW)
[DMA command register] Bitwise OR of:

0x01
start transfer

0x02
direction (0: from RAM to EDU, 1: from EDU to RAM)

0x04
raise interrupt 0x100 after finishing the DMA

6.17.4 IRQ controller

An IRQ is generated when written to the interrupt raise register. The value appears in interrupt status register when the interrupt is raised and has to be written to the interrupt acknowledge register to lower it.

The device supports both INTx and MSI interrupt. By default, INTx is used. Even if the driver disabled INTx and only uses MSI, it still needs to update the acknowledge register at the end of the IRQ handler routine.

6.17.5 DMA controller

One has to specify, source, destination, size, and start the transfer. One 4096 bytes long buffer at offset 0x40000 is available in the EDU device. I.e. one can perform DMA to/from this space when programmed properly.

Example of transferring a 100 byte block to and from the buffer using a given PCI address `addr`:

```
addr     -> DMA source address
0x40000  -> DMA destination address
100      -> DMA transfer count
1        -> DMA command register
while (DMA command register & 1)
    ;
```

```
0x40000  -> DMA source address
addr+100 -> DMA destination address
100      -> DMA transfer count
3        -> DMA command register
while (DMA command register & 1)
    ;
```

6.18 Device Specification for Inter-VM shared memory device

The Inter-VM shared memory device (ivshmem) is designed to share a memory region between multiple QEMU processes running different guests and the host. In order for all guests to be able to pick up the shared memory area, it is modeled by QEMU as a PCI device exposing said memory to the guest as a PCI BAR.

The device can use a shared memory object on the host directly, or it can obtain one from an ivshmem server.

In the latter case, the device can additionally interrupt its peers, and get interrupted by its peers.

For information on configuring the ivshmem device on the QEMU command line, see *Inter-VM Shared Memory device*.

6.18.1 The ivshmem PCI device's guest interface

The device has vendor ID 1af4, device ID 1110, revision 1. Before QEMU 2.6.0, it had revision 0.

PCI BARs

The ivshmem PCI device has two or three BARs:

- BAR0 holds device registers (256 Byte MMIO)
- BAR1 holds MSI-X table and PBA (only ivshmem-doorbell)
- BAR2 maps the shared memory object

There are two ways to use this device:

- If you only need the shared memory part, BAR2 suffices. This way, you have access to the shared memory in the guest and can use it as you see fit.
- If you additionally need the capability for peers to interrupt each other, you need BAR0 and BAR1. You will most likely want to write a kernel driver to handle interrupts. Requires the device to be configured for interrupts, obviously.

Before QEMU 2.6.0, BAR2 can initially be invalid if the device is configured for interrupts. It becomes safely accessible only after the ivshmem server provided the shared memory. These devices have PCI revision 0 rather than 1. Guest software should wait for the IVPosition register (described below) to become non-negative before accessing BAR2.

Revision 0 of the device is not capable to tell guest software whether it is configured for interrupts.

PCI device registers

BAR 0 contains the following registers:

Offset	Size	Access	On reset	Function
0	4	read/write	0	Interrupt Mask bit 0: peer interrupt (rev 0) reserved (rev 1) bit 1..31: reserved
4	4	read/write	0	Interrupt Status bit 0: peer interrupt (rev 0) reserved (rev 1) bit 1..31: reserved
8	4	read-only	0 or ID	IVPosition
12	4	write-only	N/A	Doorbell bit 0..15: vector bit 16..31: peer ID
16	240	none	N/A	reserved

Software should only access the registers as specified in column "Access". Reserved bits should be ignored on read, and preserved on write.

In revision 0 of the device, Interrupt Status and Mask Register together control the legacy INTx interrupt when the device has no MSI-X capability: INTx is asserted when the bit-wise AND of Status and Mask is non-zero and the device has no MSI-X capability. Interrupt Status Register bit 0 becomes 1 when an interrupt request from a peer is received. Reading the register clears it.

IVPosition Register: if the device is not configured for interrupts, this is zero. Else, it is the device's ID (between 0 and 65535).

Before QEMU 2.6.0, the register may read -1 for a short while after reset. These devices have PCI revision 0 rather than 1.

There is no good way for software to find out whether the device is configured for interrupts. A positive IVPosition means interrupts, but zero could be either.

Doorbell Register: writing this register requests to interrupt a peer. The written value's high 16 bits are the ID of the peer to interrupt, and its low 16 bits select an interrupt vector.

If the device is not configured for interrupts, the write is ignored.

If the interrupt hasn't completed setup, the write is ignored. The device is not capable to tell guest software whether setup is complete. Interrupts can regress to this state on migration.

If the peer with the requested ID isn't connected, or it has fewer interrupt vectors connected, the write is ignored. The device is not capable to tell guest software what peers are connected, or how many interrupt vectors are connected.

The peer's interrupt for this vector then becomes pending. There is no way for software to clear the pending bit, and a polling mode of operation is therefore impossible.

If the peer is a revision 0 device without MSI-X capability, its Interrupt Status register is set to 1. This asserts INTx unless masked by the Interrupt Mask register. The device is not capable to communicate the interrupt vector to guest software then.

With multiple MSI-X vectors, different vectors can be used to indicate different events have occurred. The semantics of interrupt vectors are left to the application.

6.18.2 Interrupt infrastructure

When configured for interrupts, the peers share eventfd objects in addition to shared memory. The shared resources are managed by an ivshmem server.

The ivshmem server

The server listens on a UNIX domain socket.

For each new client that connects to the server, the server

- picks an ID,
- creates eventfd file descriptors for the interrupt vectors,
- sends the ID and the file descriptor for the shared memory to the new client,
- sends connect notifications for the new client to the other clients (these contain file descriptors for sending interrupts),
- sends connect notifications for the other clients to the new client, and
- sends interrupt setup messages to the new client (these contain file descriptors for receiving interrupts).

The first client to connect to the server receives ID zero.

When a client disconnects from the server, the server sends disconnect notifications to the other clients.

The next section describes the protocol in detail.

If the server terminates without sending disconnect notifications for its connected clients, the clients can elect to continue. They can communicate with each other normally, but won't receive disconnect notification on disconnect, and no new clients can connect. There is no way for the clients to connect to a restarted server. The device is not capable to tell guest software whether the server is still up.

Example server code is in contrib/ivshmem-server/. Not to be used in production. It assumes all clients use the same number of interrupt vectors.

A standalone client is in contrib/ivshmem-client/. It can be useful for debugging.

The ivshmem Client-Server Protocol

An ivshmem device configured for interrupts connects to an ivshmem server. This section details the protocol between the two.

The connection is one-way: the server sends messages to the client. Each message consists of a single 8 byte little-endian signed number, and may be accompanied by a file descriptor via SCM_RIGHTS. Both client and server close the connection on error.

Note: QEMU currently doesn't close the connection right on error, but only when the character device is destroyed.

On connect, the server sends the following messages in order:

1. The protocol version number, currently zero. The client should close the connection on receipt of versions it can't handle.
2. The client's ID. This is unique among all clients of this server. IDs must be between 0 and 65535, because the Doorbell register provides only 16 bits for them.
3. The number -1, accompanied by the file descriptor for the shared memory.
4. Connect notifications for existing other clients, if any. This is a peer ID (number between 0 and 65535 other than the client's ID), repeated N times. Each repetition is accompanied by one file descriptor. These are for interrupting the peer with that ID using vector 0,..,N-1, in order. If the client is configured for fewer vectors, it closes the extra file descriptors. If it is configured for more, the extra vectors remain unconnected.
5. Interrupt setup. This is the client's own ID, repeated N times. Each repetition is accompanied by one file descriptor. These are for receiving interrupts from peers using vector 0,..,N-1, in order. If the client is configured for fewer vectors, it closes the extra file descriptors. If it is configured for more, the extra vectors remain unconnected.

From then on, the server sends these kinds of messages:

6. Connection / disconnection notification. This is a peer ID.

- If the number comes with a file descriptor, it's a connection notification, exactly like in step 4.
- Else, it's a disconnection notification for the peer with that ID.

Known bugs:

- The protocol changed incompatibly in QEMU 2.5. Before, messages were native endian long, and there was no version number.
- The protocol is poorly designed.

The ivshmem Client-Client Protocol

An ivshmem device configured for interrupts receives eventfd file descriptors for interrupting peers and getting interrupted by peers from the server, as explained in the previous section.

To interrupt a peer, the device writes the 8-byte integer 1 in native byte order to the respective file descriptor.

To receive an interrupt, the device reads and discards as many 8-byte integers as it can.

6.19 PVPANIC DEVICE

pvpanic device is a simulated device, through which a guest panic event is sent to qemu, and a QMP event is generated. This allows management apps (e.g. libvirt) to be notified and respond to the event.

The management app has the option of waiting for GUEST_PANICKED events, and/or polling for guest-panicked RunState, to learn when the pvpanic device has fired a panic event.

The pvpanic device can be implemented as an ISA device (using IOPORT) or as a PCI device.

6.19.1 ISA Interface

pvpanic exposes a single I/O port, by default 0x505. On read, the bits recognized by the device are set. Software should ignore bits it doesn't recognize. On write, the bits not recognized by the device are ignored. Software should set only bits both itself and the device recognize.

Bit Definition

bit 0
: a guest panic has happened and should be processed by the host

bit 1
: a guest panic has happened and will be handled by the guest; the host should record it or report it, but should not affect the execution of the guest.

bit 2 (to be implemented)
: a regular guest shutdown has happened and should be processed by the host

6.19.2 PCI Interface

The PCI interface is similar to the ISA interface except that it uses an MMIO address space provided by its BAR0, 1 byte long. Any machine with a PCI bus can enable a pvpanic device by adding `-device pvpanic-pci` to the command line.

6.19.3 ACPI Interface

pvpanic device is defined with ACPI ID "QEMU0001". Custom methods:

RDPT

To determine whether guest panic notification is supported.

Arguments
: None

Return
: Returns a byte, with the same semantics as the I/O port interface.

WRPT

To send a guest panic event.

Arguments
Arg0 is a byte to be written, with the same semantics as the I/O interface.

Return
None

The ACPI device will automatically refer to the right port in case it is modified.

6.20 QEMU Standard VGA

Exists in two variants, for isa and pci.

command line switches:

`-vga std`
picks isa for -M isapc, otherwise pci

`-device VGA`
pci variant

`-device isa-vga`
isa variant

`-device secondary-vga`
legacy-free pci variant

6.20.1 PCI spec

Applies to the pci variant only for obvious reasons.

PCI ID
`1234:1111`

PCI Region 0
Framebuffer memory, 16 MB in size (by default). Size is tunable via vga_mem_mb property.

PCI Region 1
Reserved (so we have the option to make the framebuffer bar 64bit).

PCI Region 2
MMIO bar, 4096 bytes in size (QEMU 1.3+)

PCI ROM Region
Holds the vgabios (QEMU 0.14+).

The legacy-free variant has no ROM and has `PCI_CLASS_DISPLAY_OTHER` instead of `PCI_CLASS_DISPLAY_VGA`.

6.20.2 IO ports used

Doesn't apply to the legacy-free pci variant, use the MMIO bar instead.

03c0 - 03df
: standard vga ports

01ce
: bochs vbe interface index port

01cf
: bochs vbe interface data port (x86 only)

01d0
: bochs vbe interface data port

6.20.3 Memory regions used

0xe0000000
: Framebuffer memory, isa variant only.

The pci variant used to mirror the framebuffer bar here, QEMU 0.14+ stops doing that (except when in `-M pc-$old` compat mode).

6.20.4 MMIO area spec

Likewise applies to the pci variant only for obvious reasons.

0000 - 03ff
: edid data blob.

0400 - 041f
: vga ioports (`0x3c0` to `0x3df`), remapped 1:1. Word access is supported, bytes are written in little endian order (aka index port first), so indexed registers can be updated with a single mmio write (and thus only one vmexit).

0500 - 0515
: bochs dispi interface registers, mapped flat without index/data ports. Use `(index << 1)` as offset for (16bit) register access.

0600 - 0607
: QEMU extended registers. QEMU 2.2+ only. The pci revision is 2 (or greater) when these registers are present. The registers are 32bit.

0600
: QEMU extended register region size, in bytes.

0604
: framebuffer endianness register. - `0xbebebebe` indicates big endian. - `0x1e1e1e1e` indicates little endian.

6.21 Virtual System Controller

The `virt-ctrl` device is a simple interface defined for the pure virtual machine with no hardware reference implementation to allow the guest kernel to send command to the host hypervisor.

The specification can evolve, the current state is defined as below.

This is a MMIO mapped device using 256 bytes.

Two 32bit registers are defined:

the features register (read-only, address 0x00)
: This register allows the device to report features supported by the controller. The only feature supported for the moment is power control (0x01).

the command register (write-only, address 0x04)
: This register allows the kernel to send the commands to the hypervisor. The implemented commands are part of the power control feature and are reset (1), halt (2) and panic (3). A basic command, no-op (0), is always present and can be used to test the register access. This command has no effect.

6.22 VMCoreInfo device

The `-device vmcoreinfo` will create a `fw_cfg` entry for a guest to store dump details.

6.22.1 etc/vmcoreinfo

A guest may use this `fw_cfg` entry to add information details to QEMU dumps.

The entry of 16 bytes has the following layout, in little-endian:

```
#define VMCOREINFO_FORMAT_NONE 0x0
#define VMCOREINFO_FORMAT_ELF 0x1

struct FWCfgVMCoreInfo {
    uint16_t host_format;  /* formats host supports */
    uint16_t guest_format; /* format guest supplies */
    uint32_t size;         /* size of vmcoreinfo region */
    uint64_t paddr;        /* physical address of vmcoreinfo region */
};
```

Only full write (of 16 bytes) are considered valid for further processing of entry values.

A write of 0 in `guest_format` will disable further processing of vmcoreinfo entry values & content.

You may write a `guest_format` that is not supported by the host, in which case the entry data can be ignored by QEMU (but you may still access it through a debugger, via `vmcoreinfo_realize::vmcoreinfo_state`).

6.22.2 Format & content

As of QEMU 2.11, only `VMCOREINFO_FORMAT_ELF` is supported.

The entry gives location and size of an ELF note that is appended in qemu dumps.

The note format/class must be of the target bitness and the size must be less than 1Mb.

If the ELF note name is `VMCOREINFO`, it is expected to be the Linux vmcoreinfo note (see the kernel documentation for its format). In this case, qemu dump code will read the content as a key=value text file, looking for `NUMBER(phys_base)` key value. The value is expected to be more accurate than architecture guess of the value. This is useful for KASLR-enabled guest with ancient tools not handling the `VMCOREINFO` note.

6.23 Virtual Machine Generation ID Device

The VM generation ID (`vmgenid`) device is an emulated device which exposes a 128-bit, cryptographically random, integer value identifier, referred to as a Globally Unique Identifier, or GUID.

This allows management applications (e.g. libvirt) to notify the guest operating system when the virtual machine is executed with a different configuration (e.g. snapshot execution or creation from a template). The guest operating system notices the change, and is then able to react as appropriate by marking its copies of distributed databases as dirty, re-initializing its random number generator etc.

6.23.1 Requirements

These requirements are extracted from the "How to implement virtual machine generation ID support in a virtualization platform" section of the Microsoft Virtual Machine Generation ID specification dated August 1, 2012.

- **R1a** The generation ID shall live in an 8-byte aligned buffer.
- **R1b** The buffer holding the generation ID shall be in guest RAM, ROM, or device MMIO range.
- **R1c** The buffer holding the generation ID shall be kept separate from areas used by the operating system.
- **R1d** The buffer shall not be covered by an AddressRangeMemory or AddressRangeACPI entry in the E820 or UEFI memory map.
- **R1e** The generation ID shall not live in a page frame that could be mapped with caching disabled. (In other words, regardless of whether the generation ID lives in RAM, ROM or MMIO, it shall only be mapped as cacheable.)
- **R2** to **R5** [These AML requirements are isolated well enough in the Microsoft specification for us to simply refer to them here.]
- **R6** The hypervisor shall expose a _HID (hardware identifier) object in the VMGenId device's scope that is unique to the hypervisor vendor.

6.23.2 QEMU Implementation

The above-mentioned specification does not dictate which ACPI descriptor table will contain the VM Generation ID device. Other implementations (Hyper-V and Xen) put it in the main descriptor table (Differentiated System Description Table or DSDT). For ease of debugging and implementation, we have decided to put it in its own Secondary System Description Table, or SSDT.

The following is a dump of the contents from a running system:

```
# iasl -p ./SSDT -d /sys/firmware/acpi/tables/SSDT

Intel ACPI Component Architecture
ASL+ Optimizing Compiler version 20150717-64
Copyright (c) 2000 - 2015 Intel Corporation

Reading ACPI table from file /sys/firmware/acpi/tables/SSDT - Length
00000198 (0x0000C6)
ACPI: SSDT 0x0000000000000000 0000C6 (v01 BOCHS  VMGENID  00000001 BXPC 00000001)
Acpi table [SSDT] successfully installed and loaded
Pass 1 parse of [SSDT]
Pass 2 parse of [SSDT]
Parsing Deferred Opcodes (Methods/Buffers/Packages/Regions)

Parsing completed
Disassembly completed
ASL Output:    ./SSDT.dsl - 1631 bytes
# cat SSDT.dsl
/*
 * Intel ACPI Component Architecture
 * AML/ASL+ Disassembler version 20150717-64
 * Copyright (c) 2000 - 2015 Intel Corporation
 *
 * Disassembling to symbolic ASL+ operators
 *
 * Disassembly of /sys/firmware/acpi/tables/SSDT, Sun Feb  5 00:19:37 2017
 *
 * Original Table Header:
 *     Signature        "SSDT"
 *     Length           0x000000CA (202)
 *     Revision         0x01
 *     Checksum         0x4B
 *     OEM ID           "BOCHS "
 *     OEM Table ID     "VMGENID"
 *     OEM Revision     0x00000001 (1)
 *     Compiler ID      "BXPC"
 *     Compiler Version 0x00000001 (1)
 */
DdefinitionBlock ("/sys/firmware/acpi/tables/SSDT.aml", "SSDT", 1, "BOCHS ", "VMGENID",␣
↪0x00000001)
{
    Name (VGIA, 0x07FFF000)
    Scope (\_SB)
    {
        Device (VGEN)
        {
            Name (_HID, "QEMUVGID")  // _HID: Hardware ID
            Name (_CID, "VM_Gen_Counter")  // _CID: Compatible ID
            Name (_DDN, "VM_Gen_Counter")  // _DDN: DOS Device Name
            Method (_STA, 0, NotSerialized)  // _STA: Status
            {
                Local0 = 0x0F
                If ((VGIA == Zero))
```

(continues on next page)

(continued from previous page)

```
                {
                    Local0 = Zero
                }

                Return (Local0)
            }

            Method (ADDR, 0, NotSerialized)
            {
                Local0 = Package (0x02) {}
                Index (Local0, Zero) = (VGIA + 0x28)
                Index (Local0, One) = Zero
                Return (Local0)
            }
        }
    }

    Method (\_GPE._E05, 0, NotSerialized)  // _Exx: Edge-Triggered GPE
    {
        Notify (\_SB.VGEN, 0x80) // Status Change
    }
}
```

6.23.3 Design Details:

Requirements R1a through R1e dictate that the memory holding the VM Generation ID must be allocated and owned by the guest firmware, in this case BIOS or UEFI. However, to be useful, QEMU must be able to change the contents of the memory at runtime, specifically when starting a backed-up or snapshotted image. In order to do this, QEMU must know the address that has been allocated.

The mechanism chosen for this memory sharing is writable fw_cfg blobs. These are data object that are visible to both QEMU and guests, and are addressable as sequential files.

More information about fw_cfg can be found in *QEMU Firmware Configuration (fw_cfg) Device*.

Two fw_cfg blobs are used in this case:

`/etc/vmgenid_guid`

- contains the actual VM Generation ID GUID
- read-only to the guest

`/etc/vmgenid_addr`

- contains the address of the downloaded vmgenid blob
- writable by the guest

QEMU sends the following commands to the guest at startup:

1. Allocate memory for vmgenid_guid fw_cfg blob.
2. Write the address of vmgenid_guid into the SSDT (VGIA ACPI variable as shown above in the iasl dump). Note that this change is not propagated back to QEMU.
3. Write the address of vmgenid_guid back to QEMU's copy of vmgenid_addr via the fw_cfg DMA interface.

After step 3, QEMU is able to update the contents of vmgenid_guid at will.

Since BIOS or UEFI does not necessarily run when we wish to change the GUID, the value of VGIA is persisted via the VMState mechanism.

As spelled out in the specification, any change to the GUID executes an ACPI notification. The exact handler to use is not specified, so the vmgenid device uses the first unused one: `\_GPE._E05`.

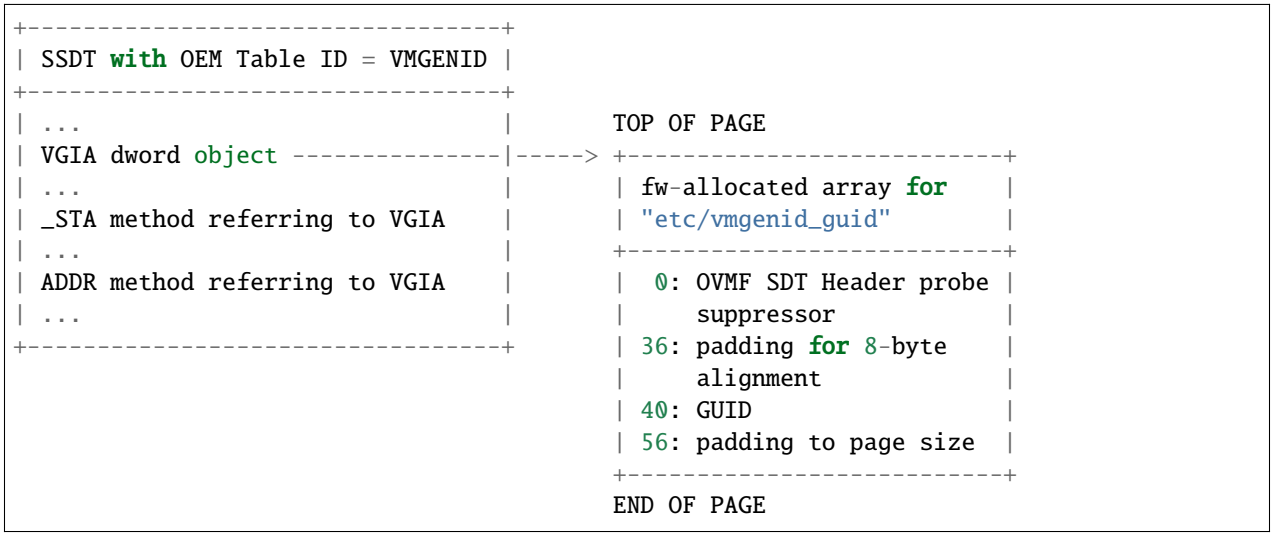
6.23.4 Endian-ness Considerations:

Although not specified in Microsoft's document, it is assumed that the device is expected to use little-endian format.

All GUID passed in via command line or monitor are treated as big-endian. GUID values displayed via monitor are shown in big-endian format.

6.23.5 GUID Storage Format:

In order to implement an OVMF "SDT Header Probe Suppressor", the contents of the vmgenid_guid fw_cfg blob are not simply a 128-bit GUID. There is also significant padding in order to align and fill a memory page, as shown in the following diagram:



6.23.6 Device Usage:

The device has one property, which may be only be set using the command line:

guid
: sets the value of the GUID. A special value `auto` instructs QEMU to generate a new random GUID.

For example:

```
QEMU  -device vmgenid,guid="324e6eaf-d1d1-4bf6-bf41-b9bb6c91fb87"
QEMU  -device vmgenid,guid=auto
```

The property may be queried via QMP/HMP:

```
(QEMU) query-vm-generation-id
{"return": {"guid": "324e6eaf-d1d1-4bf6-bf41-b9bb6c91fb87"}}
```

Setting of this parameter is intentionally left out from the QMP/HMP interfaces. There are no known use cases for changing the GUID once QEMU is running, and adding this capability would greatly increase the complexity.

CHAPTER

SEVEN

DEVELOPER INFORMATION

This section of the manual documents various parts of the internals of QEMU. You only need to read it if you are interested in reading or modifying QEMU's source code.

QEMU is a large and mature project with a number of complex subsystems that can be overwhelming to understand. The development documentation is not comprehensive but hopefully presents enough to get you started. If there are areas that are unclear please reach out either via the IRC channel or mailing list and hopefully we can improve the documentation for future developers.

All developers will want to familiarise themselves with *QEMU Community Processes* and how the community interacts. Please pay particular attention to the *QEMU Coding Style* and *Submitting a Patch* sections to avoid common pitfalls.

If you wish to implement a new hardware model you will want to read through the *The QEMU Object Model (QOM)* documentation to understand how QEMU's object model works.

Those wishing to enhance or add new CPU emulation capabilities will want to read our *TCG Emulation* documentation, especially the overview of the *Translator Internals*.

7.1 QEMU Community Processes

Notes about how to interact with the community and how and where to submit patches.

7.1.1 Code of Conduct

The QEMU community is made up of a mixture of professionals and volunteers from all over the world. Diversity is one of our strengths, but it can also lead to communication issues and unhappiness. To that end, we have a few ground rules that we ask people to adhere to.

- Be welcoming. We are committed to making participation in this project a harassment-free experience for everyone, regardless of level of experience, gender, gender identity and expression, sexual orientation, disability, personal appearance, body size, race, ethnicity, age, religion, or nationality.
- Be respectful. Not all of us will agree all the time. Disagreements, both social and technical, happen all the time and the QEMU community is no exception. When we disagree, we try to understand why. It is important that we resolve disagreements and differing views constructively. Members of the QEMU community should be respectful when dealing with other contributors as well as with people outside the QEMU community and with users of QEMU.

Harassment and other exclusionary behavior are not acceptable. A community where people feel uncomfortable or threatened is neither welcoming nor respectful. Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery
- Personal attacks

- Trolling or insulting/derogatory comments
- Public or private harassment
- Publishing other's private information, such as physical or electronic addresses, without explicit permission

This isn't an exhaustive list of things that you can't do. Rather, take it in the spirit in which it's intended: a guide to make it easier to be excellent to each other.

This code of conduct applies to all spaces managed by the QEMU project. This includes IRC, the mailing lists, the issue tracker, community events, and any other forums created by the project team which the community uses for communication. This code of conduct also applies outside these spaces, when an individual acts as a representative or a member of the project or its community.

By adopting this code of conduct, project maintainers commit themselves to fairly and consistently applying these principles to every aspect of managing this project. If you believe someone is violating the code of conduct, please read the *Conflict Resolution Policy* document for information about how to proceed.

Sources

This document is based on the Fedora Code of Conduct (as of April 2021) and the Contributor Covenant version 1.3.0.

7.1.2 Conflict Resolution Policy

Conflicts in the community can take many forms, from someone having a bad day and using harsh and hurtful language on the mailing list to more serious code of conduct violations (including sexist/racist statements or threats of violence), and everything in between.

For the vast majority of issues, we aim to empower individuals to first resolve conflicts themselves, asking for help when needed, and only after that fails to escalate further. This approach gives people more control over the outcome of their dispute.

How we resolve conflicts

If you are experiencing conflict, please consider first addressing the perceived conflict directly with other involved parties, preferably through a real-time medium such as IRC. You could also try to get a third-party (e.g. a mutual friend, and/or someone with background on the issue, but not involved in the conflict) to intercede or mediate.

If this fails or if you do not feel comfortable proceeding this way, or if the problem requires immediate escalation, report the issue to the QEMU leadership committee by sending an email to qemu@sfconservancy.org, providing references to the misconduct. For very urgent topics, you can also inform one or more members through IRC. The up-to-date list of members is available on the QEMU wiki.

Your report will be treated confidentially by the leadership committee and not be published without your agreement. The QEMU leadership committee will then do its best to review the incident in a timely manner, and will either seek further information, or will make a determination on next steps.

Remedies

Escalating an issue to the QEMU leadership committee may result in actions impacting one or more involved parties. In the event the leadership committee has to intervene, here are some of the ways they might respond:

1. Take no action. For example, if the leadership committee determines the complaint has not been substantiated or is being made in bad faith, or if it is deemed to be outside its purview.
2. A private reprimand, explaining the consequences of continued behavior, to one or more involved individuals.
3. A private reprimand and request for a private or public apology
4. A public reprimand and request for a public apology
5. A public reprimand plus a mandatory cooling off period. The cooling off period may require, for example, one or more of the following: abstaining from maintainer duties; not interacting with people involved, including unsolicited interaction with those enforcing the guidelines and interaction on social media; being denied participation to in-person events. The cooling off period is voluntary but may escalate to a temporary ban in order to enforce it.
6. A temporary or permanent ban from some or all current and future QEMU spaces (mailing lists, IRC, wiki, etc.), possibly including in-person events.

In the event of severe harassment, the leadership committee may advise that the matter be escalated to the relevant local law enforcement agency. It is however not the role of the leadership committee to initiate contact with law enforcement on behalf of any of the community members involved in an incident.

Sources

This document was developed based on the Drupal Conflict Resolution Policy and Process and the Mozilla Consequence Ladder

7.1.3 The Role of Maintainers

Maintainers are a critical part of the project's contributor ecosystem. They come from a wide range of backgrounds from unpaid hobbyists working in their spare time to employees who work on the project as part of their job. Maintainer activities include:

- reviewing patches and suggesting changes
- collecting patches and preparing pull requests
- tending to the long term health of their area
- participating in other project activities

They are also human and subject to the same pressures as everyone else including overload and burnout. Like everyone else they are subject to project's *Code of Conduct* and should also be exemplars of excellent community collaborators.

The MAINTAINERS file

The MAINTAINERS file contains the canonical list of who is a maintainer. The file is machine readable so an appropriately configured git (see *CC the relevant maintainer*) can automatically Cc them on patches that touch their area of code.

The file also describes the status of the area of code to give an idea of how actively that section is maintained.

Table 1: Meaning of support status in MAINTAINERS

Status	Meaning
Supported	Someone is actually paid to look after this.
Maintained	Someone actually looks after it.
Odd Fixes	It has a maintainer but they don't have time to do much other than throw the odd patch in.
Orphan	No current maintainer.
Obsolete	Old obsolete code, should use something else.

Please bear in mind that even if someone is paid to support something it does not mean they are paid to support you. This is open source and the code comes with no warranty and the project makes no guarantees about dealing with bugs or features requests.

Becoming a reviewer

Most maintainers start by becoming subsystem reviewers. While anyone is welcome to review code on the mailing list getting added to the MAINTAINERS file with a line like:

```
R: Random Hacker <rhacker@example.com>
```

marks you as a 'designated reviewer' - expected to provide regular spontaneous feedback. This will ensure that patches touching a given subsystem will automatically be CC'd to you.

Becoming a maintainer

Maintainers are volunteers who put themselves forward or have been asked by others to keep an eye on an area of code. They have generally demonstrated to the community, usually via contributions and code reviews, that they have a good understanding of the subsystem. They are also trusted to make a positive contribution to the project and work well with the other contributors.

The process is simple - simply send a patch to the list that updates the `MAINTAINERS` file. Sometimes this is done as part of a larger series when a new sub-system is being added to the code base. This can also be done by a retiring maintainer who nominates their replacement after discussion with other contributors.

Once the patch is reviewed and merged the only other step is to make sure your GPG key is signed.

Maintainer GPG Keys

GPG is used to sign pull requests so they can be identified as really coming from the maintainer. If your key is not already signed by members of the QEMU community, you should make arrangements to attend a KeySigningParty (for example at KVM Forum) or make alternative arrangements to have your key signed by an attendee. Key signing requires meeting another community member **in person**[1] so please make appropriate arrangements.

7.1.4 QEMU Coding Style

Table of Contents

[1] In recent pandemic times we have had to exercise some flexibility here. Maintainers still need to sign their pull requests though.

Please use the script checkpatch.pl in the scripts directory to check patches before submitting.

Formatting and style

The repository includes a `.editorconfig` file which can help with getting the right settings for your preferred $EDITOR. See https://editorconfig.org/ for details.

Whitespace

Of course, the most important aspect in any coding style is whitespace. Crusty old coders who have trouble spotting the glasses on their noses can tell the difference between a tab and eight spaces from a distance of approximately fifteen parsecs. Many a flamewar has been fought and lost on this issue.

QEMU indents are four spaces. Tabs are never used, except in Makefiles where they have been irreversibly coded into the syntax. Spaces of course are superior to tabs because:

- You have just one way to specify whitespace, not two. Ambiguity breeds mistakes.
- The confusion surrounding 'use tabs to indent, spaces to justify' is gone.
- Tab indents push your code to the right, making your screen seriously unbalanced.
- Tabs will be rendered incorrectly on editors who are misconfigured not to use tab stops of eight positions.
- Tabs are rendered badly in patches, causing off-by-one errors in almost every line.
- It is the QEMU coding style.

Do not leave whitespace dangling off the ends of lines.

Multiline Indent

There are several places where indent is necessary:

- if/else
- while/for
- function definition & call

When breaking up a long line to fit within line width, we need a proper indent for the following lines.

In case of if/else, while/for, align the secondary lines just after the opening parenthesis of the first.

For example:

```
if (a == 1 &&
    b == 2) {

while (a == 1 &&
       b == 2) {
```

In case of function, there are several variants:

- 4 spaces indent from the beginning
- align the secondary lines just after the opening parenthesis of the first

For example:

```
do_something(x, y,
    z);

do_something(x, y,
             z);

do_something(x, do_another(y,
                           z));
```

Line width

Lines should be 80 characters; try not to make them longer.

Sometimes it is hard to do, especially when dealing with QEMU subsystems that use long function or symbol names. If wrapping the line at 80 columns is obviously less readable and more awkward, prefer not to wrap it; better to have an 85 character line than one which is awkwardly wrapped.

Even in that case, try not to make lines much longer than 80 characters. (The checkpatch script will warn at 100 characters, but this is intended as a guard against obviously-overlength lines, not a target.)

Rationale:

- Some people like to tile their 24" screens with a 6x4 matrix of 80x24 xterms and use vi in all of them. The best way to punish them is to let them keep doing it.
- Code and especially patches is much more readable if limited to a sane line length. Eighty is traditional.
- The four-space indentation makes the most common excuse ("But look at all that white space on the left!") moot.
- It is the QEMU coding style.

Naming

Variables are lower_case_with_underscores; easy to type and read. Structured type names are in CamelCase; harder to type but standing out. Enum type names and function type names should also be in CamelCase. Scalar type names are lower_case_with_underscores_ending_with_a_t, like the POSIX uint64_t and family. Note that this last convention contradicts POSIX and is therefore likely to be changed.

Variable Naming Conventions

A number of short naming conventions exist for variables that use common QEMU types. For example, the architecture independent CPUState is often held as a `cs` pointer variable, whereas the concrete CPUArchState is usually held in a pointer called `env`.

Likewise, in device emulation code the common DeviceState is usually called `dev`.

Function Naming Conventions

Wrapped version of standard library or GLib functions use a `qemu_` prefix to alert readers that they are seeing a wrapped version, for example `qemu_strtol` or `qemu_mutex_lock`. Other utility functions that are widely called from across the codebase should not have any prefix, for example `pstrcpy` or bit manipulation functions such as `find_first_bit`.

The `qemu_` prefix is also used for functions that modify global emulator state, for example `qemu_add_vm_change_state_handler`. However, if there is an obvious subsystem-specific prefix it should be used instead.

Public functions from a file or subsystem (declared in headers) tend to have a consistent prefix to show where they came from. For example, `tlb_` for functions from `cputlb.c` or `cpu_` for functions from cpus.c.

If there are two versions of a function to be called with or without a lock held, the function that expects the lock to be already held usually uses the suffix `_locked`.

If a function is a shim designed to deal with compatibility workarounds we use the suffix `_compat`. These are generally not called directly and aliased to the plain function name via the pre-processor. Another common suffix is `_impl`; it is used for the concrete implementation of a function that will not be called directly, but rather through a macro or an inline function.

Block structure

Every indented statement is braced; even if the block contains just one statement. The opening brace is on the line that contains the control flow statement that introduces the new block; the closing brace is on the same line as the else keyword, or on a line by itself if there is no else keyword. Example:

```
if (a == 5) {
    printf("a was 5.\n");
} else if (a == 6) {
    printf("a was 6.\n");
} else {
    printf("a was something else entirely.\n");
}
```

Note that 'else if' is considered a single statement; otherwise a long if/ else if/else if/. . . /else sequence would need an indent for every else statement.

An exception is the opening brace for a function; for reasons of tradition and clarity it comes on a line by itself:

```
void a_function(void)
{
    do_something();
}
```

Rationale: a consistent (except for functions...) bracing style reduces ambiguity and avoids needless churn when lines are added or removed. Furthermore, it is the QEMU coding style.

Declarations

Mixed declarations (interleaving statements and declarations within blocks) are generally not allowed; declarations should be at the beginning of blocks. To avoid accidental re-use it is permissible to declare loop variables inside for loops:

```
for (int i = 0; i < ARRAY_SIZE(thing); i++) {
    /* do something loopy */
}
```

Every now and then, an exception is made for declarations inside a #ifdef or #ifndef block: if the code looks nicer, such declarations can be placed at the top of the block even if there are statements above. On the other hand, however, it's often best to move that #ifdef/#ifndef block to a separate function altogether.

Conditional statements

When comparing a variable for (in)equality with a constant, list the constant on the right, as in:

```
if (a == 1) {
    /* Reads like: "If a equals 1" */
    do_something();
}
```

Rationale: Yoda conditions (as in 'if (1 == a)') are awkward to read. Besides, good compilers already warn users when '==' is mis-typed as '=', even when the constant is on the right.

Comment style

We use traditional C-style /* */ comments and avoid // comments.

Rationale: The // form is valid in C99, so this is purely a matter of consistency of style. The checkpatch script will warn you about this.

Multiline comment blocks should have a row of stars on the left, and the initial /* and terminating */ both on their own lines:

```
/*
 * like
 * this
 */
```

This is the same format required by the Linux kernel coding style.

(Some of the existing comments in the codebase use the GNU Coding Standards form which does not have stars on the left, or other variations; avoid these when writing new comments, but don't worry about converting to the preferred form unless you're editing that comment anyway.)

Rationale: Consistency, and ease of visually picking out a multiline comment from the surrounding code.

Language usage

Preprocessor

Variadic macros

For variadic macros, stick with this C99-like syntax:

```
#define DPRINTF(fmt, ...)                                       \
    do { printf("IRQ: " fmt, ## __VA_ARGS__); } while (0)
```

Include directives

Order include directives as follows:

```
#include "qemu/osdep.h"  /* Always first... */
#include <...>           /* then system headers... */
#include "..."           /* and finally QEMU headers. */
```

The "qemu/osdep.h" header contains preprocessor macros that affect the behavior of core system headers like <stdint.h>. It must be the first include so that core system headers included by external libraries get the preprocessor macros that QEMU depends on.

Do not include "qemu/osdep.h" from header files since the .c file will have already included it.

Headers should normally include everything they need beyond osdep.h. If exceptions are needed for some reason, they must be documented in the header. If all that's needed from a header is typedefs, consider putting those into qemu/typedefs.h instead of including the header.

Cyclic inclusion is forbidden.

Generative Includes

QEMU makes fairly extensive use of the macro pre-processor to instantiate multiple similar functions. While such abuse of the macro processor isn't discouraged it can make debugging and code navigation harder. You should consider carefully if the same effect can be achieved by making it easy for the compiler to constant fold or using python scripting to generate grep friendly code.

If you do use template header files they should be named with the `.c.inc` or `.h.inc` suffix to make it clear they are being included for expansion.

C types

It should be common sense to use the right type, but we have collected a few useful guidelines here.

Scalars

If you're using "int" or "long", odds are good that there's a better type. If a variable is counting something, it should be declared with an unsigned type.

If it's host memory-size related, size_t should be a good choice (use ssize_t only if required). Guest RAM memory offsets must use ram_addr_t, but only for RAM, it may not cover whole guest address space.

If it's file-size related, use off_t. If it's file-offset related (i.e., signed), use off_t. If it's just counting small numbers use "unsigned int"; (on all but oddball embedded systems, you can assume that that type is at least four bytes wide).

In the event that you require a specific width, use a standard type like int32_t, uint32_t, uint64_t, etc. The specific types are mandatory for VMState fields.

Don't use Linux kernel internal types like u32, __u32 or __le32.

Use hwaddr for guest physical addresses except pcibus_t for PCI addresses. In addition, ram_addr_t is a QEMU internal address space that maps guest RAM physical addresses into an intermediate address space that can map to host virtual address spaces. Generally speaking, the size of guest memory can always fit into ram_addr_t but it would not be correct to store an actual guest physical address in a ram_addr_t.

For CPU virtual addresses there are several possible types. vaddr is the best type to use to hold a CPU virtual address in target-independent code. It is guaranteed to be large enough to hold a virtual address for any target, and it does not change size from target to target. It is always unsigned. target_ulong is a type the size of a virtual address on the CPU; this means it may be 32 or 64 bits depending on which target is being built. It should therefore be used only in target-specific code, and in some performance-critical built-per-target core code such as the TLB code. There is also a signed version, target_long. abi_ulong is for the *-user targets, and represents a type the size of 'void *' in that target's ABI. (This may not be the same as the size of a full CPU virtual address in the case of target ABIs which use 32 bit pointers on 64 bit CPUs, like sparc32plus.) Definitions of structures that must match the target's ABI must use this type for anything that on the target is defined to be an 'unsigned long' or a pointer type. There is also a signed version, abi_long.

Of course, take all of the above with a grain of salt. If you're about to use some system interface that requires a type like size_t, pid_t or off_t, use matching types for any corresponding variables.

Also, if you try to use e.g., "unsigned int" as a type, and that conflicts with the signedness of a related variable, sometimes it's best just to use the *wrong* type, if "pulling the thread" and fixing all related variables would be too invasive.

Finally, while using descriptive types is important, be careful not to go overboard. If whatever you're doing causes warnings, or requires casts, then reconsider or ask for help.

Pointers

Ensure that all of your pointers are "const-correct". Unless a pointer is used to modify the pointed-to storage, give it the "const" attribute. That way, the reader knows up-front that this is a read-only pointer. Perhaps more importantly, if we're diligent about this, when you see a non-const pointer, you're guaranteed that it is used to modify the storage it points to, or it is aliased to another pointer that is.

Typedefs

Typedefs are used to eliminate the redundant 'struct' keyword, since type names have a different style than other identifiers ("CamelCase" versus "snake_case"). Each named struct type should have a CamelCase name and a corresponding typedef.

Since certain C compilers choke on duplicated typedefs, you should avoid them and declare a typedef only in one header file. For common types, you can use "include/qemu/typedefs.h" for example. However, as a matter of convenience it is also perfectly fine to use forward struct definitions instead of typedefs in headers and function prototypes; this avoids problems with duplicated typedefs and reduces the need to include headers from other headers.

Reserved namespaces in C and POSIX

Underscore capital, double underscore, and underscore 't' suffixes should be avoided.

Low level memory management

Use of the `malloc/free/realloc/calloc/valloc/memalign/posix_memalign` APIs is not allowed in the QEMU codebase. Instead of these routines, use the GLib memory allocation routines `g_malloc/g_malloc0/g_new/g_new0/g_realloc/g_free` or QEMU's `qemu_memalign/qemu_blockalign/qemu_vfree` APIs.

Please note that `g_malloc` will exit on allocation failure, so there is no need to test for failure (as you would have to with `malloc`). Generally using `g_malloc` on start-up is fine as the result of a failure to allocate memory is going to be a fatal exit anyway. There may be some start-up cases where failing is unreasonable (for example speculatively loading a large debug symbol table).

Care should be taken to avoid introducing places where the guest could trigger an exit by causing a large allocation. For small allocations, of the order of 4k, a failure to allocate is likely indicative of an overloaded host and allowing `g_malloc` to `exit` is a reasonable approach. However for larger allocations where we could realistically fall-back to a smaller one if need be we should use functions like `g_try_new` and check the result. For example this is valid approach for a time/space trade-off like `tlb_mmu_resize_locked` in the SoftMMU TLB code.

If the lifetime of the allocation is within the function and there are multiple exist paths you can also improve the readability of the code by using `g_autofree` and related annotations. See *Automatic memory deallocation* for more details.

Calling `g_malloc` with a zero size is valid and will return NULL.

Prefer `g_new(T, n)` instead of `g_malloc(sizeof(T) * n)` for the following reasons:

- It catches multiplication overflowing size_t;
- It returns T * instead of void *, letting compiler catch more type errors.

Declarations like

```
T *v = g_malloc(sizeof(*v))
```

are acceptable, though.

Memory allocated by `qemu_memalign` or `qemu_blockalign` must be freed with `qemu_vfree`, since breaking this will cause problems on Win32.

String manipulation

Do not use the strncpy function. As mentioned in the man page, it does *not* guarantee a NULL-terminated buffer, which makes it extremely dangerous to use. It also zeros trailing destination bytes out to the specified length. Instead, use this similar function when possible, but note its different signature:

```
void pstrcpy(char *dest, int dest_buf_size, const char *src)
```

Don't use strcat because it can't check for buffer overflows, but:

```
char *pstrcat(char *buf, int buf_size, const char *s)
```

The same limitation exists with sprintf and vsprintf, so use snprintf and vsnprintf.

QEMU provides other useful string functions:

```
int strstart(const char *str, const char *val, const char **ptr)
int stristart(const char *str, const char *val, const char **ptr)
int qemu_strnlen(const char *s, int max_len)
```

There are also replacement character processing macros for isxyz and toxyz, so instead of e.g. isalnum you should use qemu_isalnum.

Because of the memory management rules, you must use g_strdup/g_strndup instead of plain strdup/strndup.

Printf-style functions

Whenever you add a new printf-style function, i.e., one with a format string argument and following "..." in its prototype, be sure to use gcc's printf attribute directive in the prototype.

This makes it so gcc's -Wformat and -Wformat-security options can do their jobs and cross-check format strings with the number and types of arguments.

C standard, implementation defined and undefined behaviors

C code in QEMU should be written to the C11 language specification. A copy of the final version of the C11 standard formatted as a draft, can be downloaded from:

http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf

The C language specification defines regions of undefined behavior and implementation defined behavior (to give compiler authors enough leeway to produce better code). In general, code in QEMU should follow the language specification and avoid both undefined and implementation defined constructs. ("It works fine on the gcc I tested it with" is not a valid argument...) However there are a few areas where we allow ourselves to assume certain behaviors because in practice all the platforms we care about behave in the same way and writing strictly conformant code would be painful. These are:

- you may assume that integers are 2s complement representation
- you may assume that right shift of a signed integer duplicates the sign bit (ie it is an arithmetic shift, not a logical shift)

In addition, QEMU assumes that the compiler does not use the latitude given in C99 and C11 to treat aspects of signed '<<' as undefined, as documented in the GNU Compiler Collection manual starting at version 4.0.

Automatic memory deallocation

QEMU has a mandatory dependency on either the GCC or the Clang compiler. As such it has the freedom to make use of a C language extension for automatically running a cleanup function when a stack variable goes out of scope. This can be used to simplify function cleanup paths, often allowing many goto jumps to be eliminated, through automatic free'ing of memory.

The GLib2 library provides a number of functions/macros for enabling automatic cleanup:

> https://developer.gnome.org/glib/stable/glib-Miscellaneous-Macros.html

Most notably:

- g_autofree - will invoke g_free() on the variable going out of scope
- g_autoptr - for structs / objects, will invoke the cleanup func created by a previous use of G_DEFINE_AUTOPTR_CLEANUP_FUNC. This is supported for most GLib data types and GObjects

For example, instead of

```
int somefunc(void)
{
    int ret = -1;
    char *foo = g_strdup_printf("foo%", "wibble");
    GList *bar = .....

    if (eek) {
       goto cleanup;
    }

    ret = 0;

  cleanup:
    g_free(foo);
    g_list_free(bar);
    return ret;
}
```

Using g_autofree/g_autoptr enables the code to be written as:

```
int somefunc(void)
{
    g_autofree char *foo = g_strdup_printf("foo%", "wibble");
    g_autoptr (GList) bar = .....

    if (eek) {
       return -1;
    }

    return 0;
}
```

While this generally results in simpler, less leak-prone code, there are still some caveats to beware of

- Variables declared with g_auto* MUST always be initialized, otherwise the cleanup function will use uninitialized stack memory

- If a variable declared with g_auto* holds a value which must live beyond the life of the function, that value must be saved and the original variable NULL'd out. This can be simpler using g_steal_pointer

```
char *somefunc(void)
{
    g_autofree char *foo = g_strdup_printf("foo%", "wibble");
    g_autoptr (GList) bar = .....

    if (eek) {
       return NULL;
    }

    return g_steal_pointer(&foo);
}
```

QEMU Specific Idioms

QEMU Object Model Declarations

The QEMU Object Model (QOM) provides a framework for handling objects in the base C language. The first declaration of a storage or class structure should always be the parent and leave a visual space between that declaration and the new code. It is also useful to separate backing for properties (options driven by the user) and internal state to make navigation easier.

For a storage structure the first declaration should always be called "parent_obj" and for a class structure the first member should always be called "parent_class" as below:

```
struct MyDeviceState {
    DeviceState parent_obj;

    /* Properties */
    int prop_a;
    char *prop_b;
    /* Other stuff */
    int internal_state;
};

struct MyDeviceClass {
    DeviceClass parent_class;

    void (*new_fn1)(void);
    bool (*new_fn2)(CPUState *);
};
```

Note that there is no need to provide typedefs for QOM structures since these are generated automatically by the QOM declaration macros. See *The QEMU Object Model (QOM)* for more details.

QEMU GUARD macros

QEMU provides a number of _GUARD macros intended to make the handling of multiple exit paths easier. For example using QEMU_LOCK_GUARD to take a lock will ensure the lock is released on exit from the function.

```
static int my_critical_function(SomeState *s, void *data)
{
    QEMU_LOCK_GUARD(&s->lock);
    do_thing1(data);
    if (check_state2(data)) {
        return -1;
    }
    do_thing3(data);
    return 0;
}
```

will ensure s->lock is released however the function is exited. The equivalent code without _GUARD macro makes us to carefully put qemu_mutex_unlock() on all exit points:

```
static int my_critical_function(SomeState *s, void *data)
{
    qemu_mutex_lock(&s->lock);
    do_thing1(data);
    if (check_state2(data)) {
        qemu_mutex_unlock(&s->lock);
        return -1;
    }
    do_thing3(data);
    qemu_mutex_unlock(&s->lock);
    return 0;
}
```

There are often WITH_ forms of macros which more easily wrap around a block inside a function.

```
WITH_RCU_READ_LOCK_GUARD() {
    QTAILQ_FOREACH_RCU(kid, &bus->children, sibling) {
        err = do_the_thing(kid->child);
        if (err < 0) {
            return err;
        }
    }
}
```

Error handling and reporting

Reporting errors to the human user

Do not use printf(), fprintf() or monitor_printf(). Instead, use error_report() or error_vreport() from error-report.h. This ensures the error is reported in the right place (current monitor or stderr), and in a uniform format.

Use error_printf() & friends to print additional information.

error_report() prints the current location. In certain common cases like command line parsing, the current location is tracked automatically. To manipulate it manually, use the loc_``*``() from error-report.h.

Propagating errors

An error can't always be reported to the user right where it's detected, but often needs to be propagated up the call chain to a place that can handle it. This can be done in various ways.

The most flexible one is Error objects. See error.h for usage information.

Use the simplest suitable method to communicate success / failure to callers. Stick to common methods: non-negative on success / -1 on error, non-negative / -errno, non-null / null, or Error objects.

Example: when a function returns a non-null pointer on success, and it can fail only in one way (as far as the caller is concerned), returning null on failure is just fine, and certainly simpler and a lot easier on the eyes than propagating an Error object through an Error *````* parameter.

Example: when a function's callers need to report details on failure only the function really knows, use Error *````*, and set suitable errors.

Do not report an error to the user when you're also returning an error for somebody else to handle. Leave the reporting to the place that consumes the error returned.

Handling errors

Calling exit() is fine when handling configuration errors during startup. It's problematic during normal operation. In particular, monitor commands should never exit().

Do not call exit() or abort() to handle an error that can be triggered by the guest (e.g., some unimplemented corner case in guest code translation or device emulation). Guests should not be able to terminate QEMU.

Note that &error_fatal is just another way to exit(1), and &error_abort is just another way to abort().

trace-events style

0x prefix

In trace-events files, use a '0x' prefix to specify hex numbers, as in:

```
some_trace(unsigned x, uint64_t y) "x 0x%x y 0x" PRIx64
```

An exception is made for groups of numbers that are hexadecimal by convention and separated by the symbols '.', '/', ':', or ' ' (such as PCI bus id):

```
another_trace(int cssid, int ssid, int dev_num) "bus id: %x.%x.%04x"
```

However, you can use '0x' for such groups if you want. Anyway, be sure that it is obvious that numbers are in hex, ex.:

```
data_dump(uint8_t c1, uint8_t c2, uint8_t c3) "bytes (in hex): %02x %02x %02x"
```

Rationale: hex numbers are hard to read in logs when there is no 0x prefix, especially when (occasionally) the representation doesn't contain any letters and especially in one line with other decimal numbers. Number groups are allowed to not use '0x' because for some things notations like %x.%x.%x are used not only in QEMU. Also dumping raw data bytes with '0x' is less readable.

'#' printf flag

Do not use printf flag '#', like '%#x'.

Rationale: there are two ways to add a '0x' prefix to printed number: '0x%...' and '%#...'. For consistency the only one way should be used. Arguments for '0x%' are:

- it is more popular
- '%#' omits the 0x for the value 0 which makes output inconsistent

7.1.5 Submitting a Patch

QEMU welcomes contributions to fix bugs, add functionality or improve the documentation. However, we get a lot of patches, and so we have some guidelines about submitting them. If you follow these, you'll help make our task of contribution review easier and your change is likely to be accepted and committed faster.

This page seems very long, so if you are only trying to post a quick one-shot fix, the bare minimum we ask is that:

Table 2: Minimal Checklist for Patches

Check	Reason
Patches contain Signed-off-by: Real Name <author@email>	States you are legally able to contribute the code. See *Patch emails must include a Signed-off-by: line*
Sent as patch emails to `qemu-devel@nongnu.org`	The project uses an email list based workflow. See *Submitting your Patches*
Be prepared to respond to review comments	Code that doesn't pass review will not get merged. See *Participating in Code Review*

You do not have to subscribe to post (list policy is to reply-to-all to preserve CCs and keep non-subscribers in the loop on the threads they start), although you may find it easier as a subscriber to pick up good ideas from other posts. If you do subscribe, be prepared for a high volume of email, often over one thousand messages in a week. The list is moderated; first-time posts from an email address (whether or not you subscribed) may be subject to some delay while waiting for a moderator to allow your address.

The larger your contribution is, or if you plan on becoming a long-term contributor, then the more important the rest of this page becomes. Reading the table of contents below should already give you an idea of the basic requirements. Use the table of contents as a reference, and read the parts that you have doubts about.

Table of Contents

Writing your Patches

Use the QEMU coding style

You can run run *scripts/checkpatch.pl <patchfile>* before submitting to check that you are in compliance with our coding standards. Be aware that `checkpatch.pl` is not infallible, though, especially where C preprocessor macros are involved; use some common sense too. See also:

- *QEMU Coding Style*
- Automate a checkpatch run on commit

Base patches against current git master

There's no point submitting a patch which is based on a released version of QEMU because development will have moved on from then and it probably won't even apply to master. We only apply selected bugfixes to release branches and then only as backports once the code has gone into master.

It is also okay to base patches on top of other on-going work that is not yet part of the git master branch. To aid continuous integration tools, such as patchew, you should add a tag line `Based-on: $MESSAGE_ID` to your cover letter to make the series dependency obvious.

Split up long patches

Split up longer patches into a patch series of logical code changes. Each change should compile and execute successfully. For instance, don't add a file to the makefile in patch one and then add the file itself in patch two. (This rule is here so that people can later use tools like git bisect without hitting points in the commit history where QEMU doesn't work for reasons unrelated to the bug they're chasing.) Put documentation first, not last, so that someone reading the series can do a clean-room evaluation of the documentation, then validate that the code matched the documentation. A commit message that mentions "Also, ..." is often a good candidate for splitting into multiple patches. For more thoughts on properly splitting patches and writing good commit messages, see this advice from OpenStack.

Make code motion patches easy to review

If a series requires large blocks of code motion, there are tricks for making the refactoring easier to review. Split up the series so that semantic changes (or even function renames) are done in a separate patch from the raw code motion. Use a one-time setup of `git config diff.renames true; git config diff.algorithm patience` (refer to git-config). The 'diff.renames' property ensures file rename patches will be given in a more compact representation that focuses only on the differences across the file rename, instead of showing the entire old file as a deletion and the new file as an insertion. Meanwhile, the 'diff.algorithm' property ensures that extracting a non-contiguous subset of one file into a new file, but where all extracted parts occur in the same order both before and after the patch, will reduce churn in trying to treat unrelated } lines in the original file as separating hunks of changes.

Ideally, a code motion patch can be reviewed by doing:

```
git format-patch --stdout -1 > patch;
diff -u <(sed -n 's/^-//p' patch) <(sed -n 's/^\+//p' patch)
```

to focus on the few changes that weren't wholesale code motion.

Don't include irrelevant changes

In particular, don't include formatting, coding style or whitespace changes to bits of code that would otherwise not be touched by the patch. (It's OK to fix coding style issues in the immediate area (few lines) of the lines you're changing.) If you think a section of code really does need a reindent or other large-scale style fix, submit this as a separate patch which makes no semantic changes; don't put it in the same patch as your bug fix.

For smaller patches in less frequently changed areas of QEMU, consider using the *Trivial Patches* process.

Write a meaningful commit message

Commit messages should be meaningful and should stand on their own as a historical record of why the changes you applied were necessary or useful.

QEMU follows the usual standard for git commit messages: the first line (which becomes the email subject line) is "subsystem: single line summary of change". Whether the "single line summary of change" starts with a capital is a matter of taste, but we prefer that the summary does not end in a dot. Look at `git shortlog -30` for an idea of sample subject lines. Then there is a blank line and a more detailed description of the patch, another blank and your Signed-off-by: line. Please do not use lines that are longer than 76 characters in your commit message (so that the text still shows up nicely with "git show" in a 80-columns terminal window).

The body of the commit message is a good place to document why your change is important. Don't include comments like "This is a suggestion for fixing this bug" (they can go below the `---` line in the email so they don't go into the final commit message). Make sure the body of the commit message can be read in isolation even if the reader's mailer displays the subject line some distance apart (that is, a body that starts with "... so that" as a continuation of the subject line is harder to follow).

If your patch fixes a commit that is already in the repository, please add an additional line with "Fixes: <at-least-12-digits-of-SHA-commit-id> ("Fixed commit subject")" below the patch description / before your "Signed-off-by:" line in the commit message.

If your patch fixes a bug in the gitlab bug tracker, please add a line with "Resolves: <URL-of-the-bug>" to the commit message, too. Gitlab can close bugs automatically once commits with the "Resolves:" keyword get merged into the master branch of the project. And if your patch addresses a bug in another public bug tracker, you can also use a line with "Buglink: <URL-of-the-bug>" for reference here, too.

Example:

```
Fixes: 14055ce53c2d ("s390x/tcg: avoid overflows in time2tod/tod2time")
Resolves: https://gitlab.com/qemu-project/qemu/-/issues/42
Buglink: https://bugs.launchpad.net/qemu/+bug/1804323``
```

Some other tags that are used in commit messages include "Message-Id:" "Tested-by:", "Acked-by:", "Reported-by:", "Suggested-by:". See `git log` for these keywords for example usage.

Test your patches

Although QEMU uses various *CI* services that attempt to test patches submitted to the list, it still saves everyone time if you have already tested that your patch compiles and works. Because QEMU is such a large project the default configuration won't create a testing pipeline on GitLab when a branch is pushed. See the *CI variable documentation* for details on how to control the running of tests; but it is still wise to also check that your patches work with a full build before submitting a series, especially if your changes might have an unintended effect on other areas of the code you don't normally experiment with. See *Testing in QEMU* for more details on what tests are available.

Also, it is a wise idea to include a testsuite addition as part of your patches - either to ensure that future changes won't regress your new feature, or to add a test which exposes the bug that the rest of your series fixes. Keeping separate commits for the test and the fix allows reviewers to rebase the test to occur first to prove it catches the problem, then again to place it last in the series so that bisection doesn't land on a known-broken state.

Submitting your Patches

The QEMU project uses a public email based workflow for reviewing and merging patches. As a result all contributions to QEMU must be **sent as patches** to the qemu-devel mailing list. Patch contributions should not be posted on the bug tracker, posted on forums, or externally hosted and linked to. (We have other mailing lists too, but all patches must go to qemu-devel, possibly with a Cc: to another list.) `git send-email` (step-by-step setup guide and hints and tips) works best for delivering the patch without mangling it, but attachments can be used as a last resort on a first-time submission.

If you cannot send patch emails

In rare cases it may not be possible to send properly formatted patch emails. You can use sourcehut to send your patches to the QEMU mailing list by following these steps:

1. Register or sign in to your account
2. Add your SSH public key in meta | keys.
3. Publish your git branch using **git push git@git.sr.ht:~USERNAME/qemu HEAD**
4. Send your patches to the QEMU mailing list using the web-based `git-send-email` UI at https://git.sr.ht/~USERNAME/qemu/send-email

This video shows the web-based `git-send-email` workflow. Documentation is available here.

CC the relevant maintainer

Send patches both to the mailing list and CC the maintainer(s) of the files you are modifying. look in the MAINTAINERS file to find out who that is. Also try using scripts/get_maintainer.pl from the repository for learning the most common committers for the files you touched.

Example:

```
~/src/qemu/scripts/get_maintainer.pl -f hw/ide/core.c
```

In fact, you can automate this, via a one-time setup of `git config sendemail.cccmd 'scripts/get_maintainer.pl --nogit-fallback'` (Refer to git-config.)

Do not send as an attachment

Send patches inline so they are easy to reply to with review comments. Do not put patches in attachments.

Use `git format-patch`

Use the right diff format. git format-patch will produce patch emails in the right format (check the documentation to find out how to drive it). You can then edit the cover letter before using `git send-email` to mail the files to the mailing list. (We recommend git send-email because mail clients often mangle patches by wrapping long lines or messing up whitespace. Some distributions do not include send-email in a default install of git; you may need to download additional packages, such as 'git-email' on Fedora-based systems.) Patch series need a cover letter, with shallow threading (all patches in the series are in-reply-to the cover letter, but not to each other); single unrelated patches do not need a cover letter (but if you do send a cover letter, use `--numbered` so the cover and the patch have distinct subject lines). Patches are easier to find if they start a new top-level thread, rather than being buried in-reply-to another existing thread.

Avoid posting large binary blob

If you added binaries to the repository, consider producing the patch emails using `git format-patch --no-binary` and include a link to a git repository to fetch the original commit.

Patch emails must include a `Signed-off-by:` line

Your patches **must** include a Signed-off-by: line. This is a hard requirement because it's how you say "I'm legally okay to contribute this and happy for it to go into QEMU". The process is modelled after the Linux kernel policy.

If you wrote the patch, make sure your "From:" and "Signed-off-by:" lines use the same spelling. It's okay if you subscribe or contribute to the list via more than one address, but using multiple addresses in one commit just confuses things. If someone else wrote the patch, git will include a "From:" line in the body of the email (different from your envelope From:) that will give credit to the correct author; but again, that author's Signed-off-by: line is mandatory, with the same spelling.

There are various tooling options for automatically adding these tags include using `git commit -s` or `git format-patch -s`. For more information see SubmittingPatches 1.12.

Include a meaningful cover letter

This is a requirement for any series with multiple patches (as it aids continuous integration), but optional for an isolated patch. The cover letter explains the overall goal of such a series, and also provides a convenient 0/N email for others to reply to the series as a whole. A one-time setup of `git config format.coverletter auto` (refer to git-config) will generate the cover letter as needed.

When reviewers don't know your goal at the start of their review, they may object to early changes that don't make sense until the end of the series, because they do not have enough context yet at that point of their review. A series where the goal is unclear also risks a higher number of review-fix cycles because the reviewers haven't bought into the idea yet. If the cover letter can explain these points to the reviewer, the process will be smoother patches will get merged faster. Make sure your cover letter includes a diffstat of changes made over the entire series; potential reviewers know what files they are interested in, and they need an easy way determine if your series touches them.

Use the RFC tag if needed

For example, "[PATCH RFC v2]". `git format-patch --subject-prefix=RFC` can help.

"RFC" means "Request For Comments" and is a statement that you don't intend for your patchset to be applied to master, but would like some review on it anyway. Reasons for doing this include:

- the patch depends on some pending kernel changes which haven't yet been accepted, so the QEMU patch series is blocked until that dependency has been dealt with, but is worth reviewing anyway
- the patch set is not finished yet (perhaps it doesn't cover all use cases or work with all targets) but you want early review of a major API change or design structure before continuing

In general, since it's asking other people to do review work on a patchset that the submitter themselves is saying shouldn't be applied, it's best to:

- use it sparingly
- in the cover letter, be clear about why a patch is an RFC, what areas of the patchset you're looking for review on, and why reviewers should care

Consider whether your patch is applicable for stable

If your patch fixes a severe issue or a regression, it may be applicable for stable. In that case, consider adding `Cc: qemu-stable@nongnu.org` to your patch to notify the stable maintainers.

For more details on how QEMU's stable process works, refer to the *QEMU and the stable process* page.

Participating in Code Review

All patches submitted to the QEMU project go through a code review process before they are accepted. This will often mean a series will go through a number of iterations before being picked up by *maintainers*. You therefore should be prepared to read replies to your messages and be willing to act on them.

Maintainers are often willing to manually fix up first-time contributions, since there is a learning curve involved in making an ideal patch submission. However for the best results you should proactively respond to suggestions with changes or justifications for your current approach.

Some areas of code that are well maintained may review patches quickly, lesser-loved areas of code may have a longer delay.

Stay around to fix problems raised in code review

Not many patches get into QEMU straight away – it is quite common that developers will identify bugs, or suggest a cleaner approach, or even just point out code style issues or commit message typos. You'll need to respond to these, and then send a second version of your patches with the issues fixed. This takes a little time and effort on your part, but if you don't do it then your changes will never get into QEMU.

Remember that a maintainer is under no obligation to take your patches. If someone has spent the time reviewing your code and suggesting improvements and you simply re-post without either addressing the comment directly or providing additional justification for the change then it becomes wasted effort. You cannot demand others merge and then fix up your code after the fact.

When replying to comments on your patches **reply to all and not just the sender** – keeping discussion on the mailing list means everybody can follow it. Remember the spirit of the *Code of Conduct* and keep discussions respectful and collaborative and avoid making personal comments.

Pay attention to review comments

Someone took their time to review your work, and it pays to respect that effort; repeatedly submitting a series without addressing all comments from the previous round tends to alienate reviewers and stall your patch. Reviewers aren't always perfect, so it is okay if you want to argue that your code was correct in the first place instead of blindly doing everything the reviewer asked. On the other hand, if someone pointed out a potential issue during review, then even if your code turns out to be correct, it's probably a sign that you should improve your commit message and/or comments in the code explaining why the code is correct.

If you fix issues that are raised during review **resend the entire patch series** not just the one patch that was changed. This allows maintainers to easily apply the fixed series without having to manually identify which patches are relevant. Send the new version as a complete fresh email or series of emails – don't try to make it a followup to version 1. (This helps automatic patch email handling tools distinguish between v1 and v2 emails.)

When resending patches add a version tag

All patches beyond the first version should include a version tag – for example, "[PATCH v2]". This means people can easily identify whether they're looking at the most recent version. (The first version of a patch need not say "v1", just [PATCH] is sufficient.) For patch series, the version applies to the whole series – even if you only change one patch, you resend the entire series and mark it as "v2". Don't try to track versions of different patches in the series separately. git format-patch and git send-email both understand the `-v2` option to make this easier. Send each new revision as a new top-level thread, rather than burying it in-reply-to an earlier revision, as many reviewers are not looking inside deep threads for new patches.

Include version history in patchset revisions

For later versions of patches, include a summary of changes from previous versions, but not in the commit message itself. In an email formatted as a git patch, the commit message is the part above the `---` line, and this will go into the git changelog when the patch is committed. This part should be a self-contained description of what this version of the patch does, written to make sense to anybody who comes back to look at this commit in git in six months' time. The part below the `---` line and above the patch proper (git format-patch puts the diffstat here) is a good place to put remarks for people reading the patch email, and this is where the "changes since previous version" summary belongs. The git-publish script can help with tracking a good summary across versions. Also, the git-backport-diff script can help focus reviewers on what changed between revisions.

Tips and Tricks

Proper use of Reviewed-by: tags can aid review

When reviewing a large series, a reviewer can reply to some of the patches with a Reviewed-by tag, stating that they are happy with that patch in isolation (sometimes conditional on minor cleanup, like fixing whitespace, that doesn't affect code content). You should then update those commit messages by hand to include the Reviewed-by tag, so that in the next revision, reviewers can spot which patches were already clean from the previous round. Conversely, if you significantly modify a patch that was previously reviewed, remove the reviewed-by tag out of the commit message, as well as listing the changes from the previous version, to make it easier to focus a reviewer's attention to your changes.

If your patch seems to have been ignored

If your patchset has received no replies you should "ping" it after a week or two, by sending an email as a reply-to-all to the patch mail, including the word "ping" and ideally also a link to the page for the patch on patchew or lore.kernel.org. It's worth double-checking for reasons why your patch might have been ignored (forgot to CC the maintainer? annoyed people by failing to respond to review comments on an earlier version?), but often for less-maintained areas of QEMU patches do just slip through the cracks. If your ping is also ignored, ping again after another week or so. As the submitter, you are the person with the most motivation to get your patch applied, so you have to be persistent.

Is my patch in?

QEMU has some Continuous Integration machines that try to catch patch submission problems as soon as possible. patchew includes a web interface for tracking the status of various threads that have been posted to the list, and may send you an automated mail if it detected a problem with your patch.

Once your patch has had enough review on list, the maintainer for that area of code will send notification to the list that they are including your patch in a particular staging branch. Periodically, the maintainer then takes care of *Submitting a Pull Request* for aggregating topic branches into mainline QEMU. Generally, you do not need to send a pull request unless you have contributed enough patches to become a maintainer over a particular section of code. Maintainers may further modify your commit, by resolving simple merge conflicts or fixing minor typos pointed out during review, but will always add a Signed-off-by line in addition to yours, indicating that it went through their tree. Occasionally, the maintainer's pull request may hit more difficult merge conflicts, where you may be requested to help rebase and resolve the problems. It may take a couple of weeks between when your patch first had a positive review to when it finally lands in qemu.git; release cycle freezes may extend that time even longer.

Return the favor

Peer review only works if everyone chips in a bit of review time. If everyone submitted more patches than they reviewed, we would have a patch backlog. A good goal is to try to review at least as many patches from others as what you submit. Don't worry if you don't know the code base as well as a maintainer; it's perfectly fine to admit when your review is weak because you are unfamiliar with the code.

7.1.6 Trivial Patches

Overview

Trivial patches that change just a few lines of code sometimes languish on the mailing list even though they require only a small amount of review. This is often the case for patches that do not fall under an actively maintained subsystem and therefore fall through the cracks.

The trivial patches team take on the task of reviewing and building pull requests for patches that:

- Do not fall under an actively maintained subsystem.
- Are single patches or short series (max 2-4 patches).
- Only touch a few lines of code.

You should hint that your patch is a candidate by CCing qemu-trivial@nongnu.org.

Repositories

Since the trivial patch team rotates maintainership there is only one active repository at a time:

- git://github.com/vivier/qemu.git trivial-patches - browse

Workflow

The trivial patches team rotates the duty of collecting trivial patches amongst its members. A team member's job is to:

1. Identify trivial patches on the development mailing list.
2. Review trivial patches, merge them into a git tree, and reply to state that the patch is queued.
3. Send pull requests to the development mailing list once a week.

A single team member can be on duty as long as they like. The suggested time is 1 week before handing off to the next member.

Team

If you would like to join the trivial patches team, contact Laurent Vivier. The current team includes:

- Laurent Vivier

7.1.7 QEMU and the stable process

QEMU stable releases

QEMU stable releases are based upon the last released QEMU version and marked by an additional version number, e.g. 2.10.1. Occasionally, a four-number version is released, if a single urgent fix needs to go on top.

Usually, stable releases are only provided for the last major QEMU release. For example, when QEMU 2.11.0 is released, 2.11.x or 2.11.x.y stable releases are produced only until QEMU 2.12.0 is released, at which point the stable process moves to producing 2.12.x/2.12.x.y releases.

What should go into a stable release?

Generally, the following patches are considered stable material:

- Patches that fix severe issues, like fixes for CVEs
- Patches that fix regressions

If you think the patch would be important for users of the current release (or for a distribution picking fixes), it is usually a good candidate for stable.

How to get a patch into QEMU stable

There are various ways to get a patch into stable:

- Preferred: Make sure that the stable maintainers are on copy when you send the patch by adding

```
Cc: qemu-stable@nongnu.org
```

  to the patch description. By default, this will send a copy of the patch to `qemu-stable@nongnu.org` if you use git send-email, which is where patches that are stable candidates are tracked by the maintainers.

- You can also reply to a patch and put `qemu-stable@nongnu.org` on copy directly in your mail client if you think a previously submitted patch should be considered for a stable release.
- If a maintainer judges the patch appropriate for stable later on (or you notify them), they will add the same line to the patch, meaning that the stable maintainers will be on copy on the maintainer's pull request.
- If you judge an already merged patch suitable for stable, send a mail (preferably as a reply to the most recent patch submission) to `qemu-stable@nongnu.org` along with `qemu-devel@nongnu.org` and appropriate other people (like the patch author or the relevant maintainer) on copy.

Stable release process

When the stable maintainers prepare a new stable release, they will prepare a git branch with a release candidate and send the patches out to `qemu-devel@nongnu.org` for review. If any of your patches are included, please verify that they look fine, especially if the maintainer had to tweak the patch as part of back-porting things across branches. You may also nominate other patches that you think are suitable for inclusion. After review is complete (may involve more release candidates), a new stable release is made available.

7.1.8 Submitting a Pull Request

QEMU welcomes contributions of code, but we generally expect these to be sent as simple patch emails to the mailing list (see our page on *Submitting a Patch* for more details). Generally only existing submaintainers of a tree will need to submit pull requests, although occasionally for a large patch series we might ask a submitter to send a pull request. This page documents our recommendations on pull requests for those people.

A good rule of thumb is not to send a pull request unless somebody asks you to.

Resend the patches with the pull request as emails which are threaded as follow-ups to the pull request itself. The simplest way to do this is to use `git format-patch --cover-letter` to create the emails, and then edit the cover letter to include the pull request details that `git request-pull` outputs.

Use PULL as the subject line tag in both the cover letter and the retransmitted patch mails (for example, by using `--subject-prefix=PULL` in your `git format-patch` command). This helps people to filter in or out the resulting emails (especially useful if they are only CC'd on one email out of the set).

Each patch must have your own Signed-off-by: line as well as that of the original author if the patch was not written by you. This is because with a pull request you're now indicating that the patch has passed via you rather than directly from the original author.

Don't forget to add Reviewed-by: and Acked-by: lines. When other people have reviewed the patches you're putting in the pull request, make sure you've copied their signoffs across. (If you use the patches tool to add patches from email directly to your git repo it will include the tags automatically; if you're updating patches manually or in some other way you'll need to edit the commit messages by hand.)

Don't send pull requests for code that hasn't passed review. A pull request says these patches are ready to go into QEMU now, so they must have passed the standard code review processes. In particular if you've corrected issues in

one round of code review, you need to send your fixed patch series as normal to the list; you can't put it in a pull request until it's gone through. (Extremely trivial fixes may be OK to just fix in passing, but if in doubt err on the side of not.)

Test before sending. This is an obvious thing to say, but make sure everything builds (including that it compiles at each step of the patch series) and that "make check" passes before sending out the pull request. As a submaintainer you're one of QEMU's lines of defense against bad code, so double check the details.

All pull requests must be signed. By "signed" here we mean that the pullreq email should quote a tag which is a GPG-signed tag (as created with 'gpg tag -s ...'). See *Maintainer GPG Keys* for details.

Pull requests not for master should say "not for master" and have "PULL SUBSYSTEM whatever" in the subject tag. If your pull request is targeting a stable branch or some submaintainer tree, please include the string "not for master" in the cover letter email, and make sure the subject tag is "PULL SUBSYSTEM s390/block/whatever" rather than just "PULL". This allows it to be automatically filtered out of the set of pull requests that should be applied to master.

You might be interested in the make-pullreq script which automates some of this process for you and includes a few sanity checks. Note that you must edit it to configure it suitably for your local situation!

7.1.9 Secure Coding Practices

This document covers topics that both developers and security researchers must be aware of so that they can develop safe code and audit existing code properly.

Reporting Security Bugs

For details on how to report security bugs or ask questions about potential security bugs, see the Security Process wiki page.

General Secure C Coding Practices

Most CVEs (security bugs) reported against QEMU are not specific to virtualization or emulation. They are simply C programming bugs. Therefore it's critical to be aware of common classes of security bugs.

There is a wide selection of resources available covering secure C coding. For example, the CERT C Coding Standard covers the most important classes of security bugs.

Instead of describing them in detail here, only the names of the most important classes of security bugs are mentioned:

- Buffer overflows
- Use-after-free and double-free
- Integer overflows
- Format string vulnerabilities

Some of these classes of bugs can be detected by analyzers. Static analysis is performed regularly by Coverity and the most obvious of these bugs are even reported by compilers. Dynamic analysis is possible with valgrind, tsan, and asan.

Input Validation

Inputs from the guest or external sources (e.g. network, files) cannot be trusted and may be invalid. Inputs must be checked before using them in a way that could crash the program, expose host memory to the guest, or otherwise be exploitable by an attacker.

The most sensitive attack surface is device emulation. All hardware register accesses and data read from guest memory must be validated. A typical example is a device that contains multiple units that are selectable by the guest via an index register:

```
typedef struct {
    ProcessingUnit unit[2];
    ...
} MyDeviceState;

static void mydev_writel(void *opaque, uint32_t addr, uint32_t val)
{
    MyDeviceState *mydev = opaque;
    ProcessingUnit *unit;

    switch (addr) {
    case MYDEV_SELECT_UNIT:
        unit = &mydev->unit[val];   <-- this input wasn't validated!
        ...
    }
}
```

If `val` is not in range [0, 1] then an out-of-bounds memory access will take place when `unit` is dereferenced. The code must check that `val` is 0 or 1 and handle the case where it is invalid.

Unexpected Device Accesses

The guest may access device registers in unusual orders or at unexpected moments. Device emulation code must not assume that the guest follows the typical "theory of operation" presented in driver writer manuals. The guest may make nonsense accesses to device registers such as starting operations before the device has been fully initialized.

A related issue is that device emulation code must be prepared for unexpected device register accesses while asynchronous operations are in progress. A well-behaved guest might wait for a completion interrupt before accessing certain device registers. Device emulation code must handle the case where the guest overwrites registers or submits further requests before an ongoing request completes. Unexpected accesses must not cause memory corruption or leaks in QEMU.

Invalid device register accesses can be reported with `qemu_log_mask(LOG_GUEST_ERROR, ...)`. The `-d guest_errors` command-line option enables these log messages.

Live Migration

Device state can be saved to disk image files and shared with other users. Live migration code must validate inputs when loading device state so an attacker cannot gain control by crafting invalid device states. Device state is therefore considered untrusted even though it is typically generated by QEMU itself.

Guest Memory Access Races

Guests with multiple vCPUs may modify guest RAM while device emulation code is running. Device emulation code must copy in descriptors and other guest RAM structures and only process the local copy. This prevents time-of-check-to-time-of-use (TOCTOU) race conditions that could cause QEMU to crash when a vCPU thread modifies guest RAM while device emulation is processing it.

Use of null-co block drivers

The `null-co` block driver is designed for performance: its read accesses are not initialized by default. In case this driver has to be used for security research, it must be used with the `read-zeroes=on` option which fills read buffers with zeroes. Security issues reported with the default (`read-zeroes=off`) will be discarded.

7.2 QEMU Build and Test System

Details about how QEMU's build system works and how it is integrated into our testing infrastructure. You will need to understand some of the basics if you are adding new files and targets to the build.

7.2.1 The QEMU build system architecture

This document aims to help developers understand the architecture of the QEMU build system. As with projects using GNU autotools, the QEMU build system has two stages; first the developer runs the "configure" script to determine the local build environment characteristics, then they run "make" to build the project. This is about where the similarities with GNU autotools end, so try to forget what you know about them.

The two general ways to perform a build are as follows:

- build artifacts outside of QEMU source tree entirely:

```
cd ../
mkdir build
cd build
../qemu/configure
make
```

- build artifacts in a subdir of QEMU source tree:

```
mkdir build
cd build
../configure
make
```

Most of the actual build process uses Meson under the hood, therefore build artifacts cannot be placed in the source tree itself.

Stage 1: configure

The configure script has five tasks:

- detect the host architecture
- list the targets for which to build emulators; the list of targets also affects which firmware binaries and tests to build
- find the compilers (native and cross) used to build executables, firmware and tests. The results are written as either Makefile fragments (`config-host.mak`) or a Meson machine file (`config-meson.cross`)
- create a virtual environment in which all Python code runs during the build, and possibly install packages into it from PyPI
- invoke Meson in the virtual environment, to perform the actual configuration step for the emulator build

The configure script automatically recognizes command line options for which a same-named Meson option exists; dashes in the command line are replaced with underscores.

Almost all QEMU developers that need to modify the build system will only be concerned with Meson, and therefore can skip the rest of this section.

Modifying `configure`

`configure` is a shell script; it uses `#!/bin/sh` and therefore should be compatible with any POSIX shell. It is important to avoid using bash-isms to avoid breaking development platforms where bash is the primary host.

The configure script provides a variety of functions to help writing portable shell code and providing consistent behavior across architectures and operating systems:

`error_exit $MESSAGE $MORE...`
Print $MESSAGE to stderr, followed by $MORE... and then exit from the configure script with non-zero status.

`has $COMMAND`
Determine if $COMMAND exists in the current environment, either as a shell builtin, or executable binary, returning 0 on success. The replacement in Meson is `find_program()`.

`probe_target_compiler $TARGET`
Detect a cross compiler and cross tools for the QEMU target $TARGET (e.g., `$CPU-softmmu`, `$CPU-linux-user`, `$CPU-bsd-user`). If a working compiler is present, return success and set variables `$target_cc`, `$target_ar`, etc. to non-empty values.

`write_target_makefile`
Write a Makefile fragment to stdout, exposing the result of the most `probe_target_compiler` call as the usual Make variables (`CC`, `AR`, `LD`, etc.).

Configure does not generally perform tests for compiler options beyond basic checks to detect the host platform and ensure the compiler is functioning. These are performed using a few more helper functions:

`compile_object $CFLAGS`
Attempt to compile a test program with the system C compiler using $CFLAGS. The test program must have been previously written to a file called $TMPC.

`compile_prog $CFLAGS $LDFLAGS`
Attempt to compile a test program with the system C compiler using $CFLAGS and link it with the system linker using $LDFLAGS. The test program must have been previously written to a file called $TMPC.

`check_define $NAME`
Determine if the macro $NAME is defined by the system C compiler.

do_compiler $CC $ARGS...
: Attempt to run the C compiler $CC, passing it $ARGS... This function does not use flags passed via options such as `--extra-cflags`, and therefore can be used to check for cross compilers. However, most such checks are done at `make` time instead (see for example the `cc-option` macro in `pc-bios/option-rom/Makefile`).

write_c_skeleton
: Write a minimal C program main() function to the temporary file indicated by $TMPC.

Python virtual environments and the build process

An important step in `configure` is to create a Python virtual environment (venv) during the configuration phase. The Python interpreter comes from the `--python` command line option, the `$PYTHON` variable from the environment, or the system PATH, in this order. The venv resides in the `pyvenv` directory in the build tree, and provides consistency in how the build process runs Python code.

At this stage, `configure` also queries the chosen Python interpreter about QEMU's build dependencies. Note that the build process does *not* look for `meson`, `sphinx-build` or `avocado` binaries in the PATH; likewise, there are no options such as `--meson` or `--sphinx-build`. This avoids a potential mismatch, where Meson and Sphinx binaries on the PATH might operate in a different Python environment than the one chosen by the user during the build process. On the other hand, it introduces a potential source of confusion where the user installs a dependency but `configure` is not able to find it. When this happens, the dependency was installed in the `site-packages` directory of another interpreter, or with the wrong `pip` program.

If a package is available for the chosen interpreter, `configure` prepares a small script that invokes it from the venv itself[#distlib]_. If not, `configure` can also optionally install dependencies in the virtual environment with `pip`, either from wheels in `python/wheels` or by downloading the package with PyPI. Downloading can be disabled with `--disable-download`; and anyway, it only happens when a `configure` option (currently, only `--enable-docs`) is explicitly enabled but the dependencies are not present[#pip]_.

The required versions of the packages are stored in a configuration file `pythondeps.toml`. The format is custom to QEMU, but it is documented at the top of the file itself and it should be easy to understand. The requirements should make it possible to use the version that is packaged that is provided by supported distros.

When dependencies are downloaded, instead, `configure` uses a "known good" version that is also listed in `pythondeps.toml`. In this scenario, `pythondeps.toml` behaves like the "lock file" used by `cargo`, `poetry` or other dependency management systems.

Bundled Python packages

Python packages that are **mandatory** dependencies to build QEMU, but are not available in all supported distros, are bundled with the QEMU sources. Currently this includes Meson (outdated in CentOS 8 and derivatives, Ubuntu 20.04 and 22.04, and openSUSE Leap) and tomli (absent in Ubuntu 20.04).

If you need to update these, please do so by modifying and rerunning `python/scripts/vendor.py`. This script embeds the sha256 hash of package sources and checks it. The pypi.org web site provides an easy way to retrieve the sha256 hash of the sources.

Stage 2: Meson

The Meson build system describes the build and install process for:

1) executables, which include:
   - Tools - `qemu-img`, `qemu-nbd`, `qemu-ga` (guest agent), etc
   - System emulators - `qemu-system-$ARCH`
   - Userspace emulators - `qemu-$ARCH`
   - Unit tests
2) documentation
3) ROMs, whether provided as binary blobs in the QEMU distributions or cross compiled under the direction of the configure script
4) other data files, such as icons or desktop files

All executables are built by default, except for some `contrib/` binaries that are known to fail to build on some platforms (for example 32-bit or big-endian platforms). Tests are also built by default, though that might change in the future.

The source code is highly modularized, split across many files to facilitate building of all of these components with as little duplicated compilation as possible. Using the Meson "sourceset" functionality, `meson.build` files group the source files in rules that are enabled according to the available system libraries and to various configuration symbols. Sourcesets belong to one of four groups:

Subsystem sourcesets:
Various subsystems that are common to both tools and emulators have their own sourceset, for example `block_ss` for the block device subsystem, `chardev_ss` for the character device subsystem, etc. These sourcesets are then turned into static libraries as follows:

```
libchardev = static_library('chardev', chardev_ss.sources(),
                            name_suffix: 'fa',
                            build_by_default: false)

chardev = declare_dependency(link_whole: libchardev)
```

As of Meson 0.55.1, the special `.fa` suffix should be used for everything that is used with `link_whole`, to ensure that the link flags are placed correctly in the command line.

Target-independent emulator sourcesets:
Various general purpose helper code is compiled only once and the .o files are linked into all output binaries that need it. This includes error handling infrastructure, standard data structures, platform portability wrapper functions, etc.

Target-independent code lives in the `common_ss`, `system_ss` and `user_ss` sourcesets. `common_ss` is linked into all emulators, `system_ss` only in system emulators, `user_ss` only in user-mode emulators.

Target-dependent emulator sourcesets:
In the target-dependent set lives CPU emulation, some device emulation and much glue code. This sometimes also has to be compiled multiple times, once for each target being built. Target-dependent files are included in the `specific_ss` sourceset.

Each emulator also includes sources for files in the `hw/` and `target/` subdirectories. The subdirectory used for each emulator comes from the target's definition of `TARGET_BASE_ARCH` or (if missing) `TARGET_ARCH`, as found in `default-configs/targets/*.mak`.

Each subdirectory in `hw/` adds one sourceset to the `hw_arch` dictionary, for example:

```
arm_ss = ss.source_set()
arm_ss.add(files('boot.c'), fdt)
...
hw_arch += {'arm': arm_ss}
```

The sourceset is only used for system emulators.

Each subdirectory in `target/` instead should add one sourceset to each of the `target_arch` and `target_system_arch`, which are used respectively for all emulators and for system emulators only. For example:

```
arm_ss = ss.source_set()
arm_system_ss = ss.source_set()
...
target_arch += {'arm': arm_ss}
target_system_arch += {'arm': arm_system_ss}
```

Module sourcesets:

There are two dictionaries for modules: `modules` is used for target-independent modules and `target_modules` is used for target-dependent modules. When modules are disabled the `module` source sets are added to `system_ss` and the `target_modules` source sets are added to `specific_ss`.

Both dictionaries are nested. One dictionary is created per subdirectory, and these per-subdirectory dictionaries are added to the toplevel dictionaries. For example:

```
hw_display_modules = {}
qxl_ss = ss.source_set()
...
hw_display_modules += { 'qxl': qxl_ss }
modules += { 'hw-display': hw_display_modules }
```

Utility sourcesets:

All binaries link with a static library `libqemuutil.a`. This library is built from several sourcesets; most of them however host generated code, and the only two of general interest are `util_ss` and `stub_ss`.

The separation between these two is purely for documentation purposes. `util_ss` contains generic utility files. Even though this code is only linked in some binaries, sometimes it requires hooks only in some of these and depend on other functions that are not fully implemented by all QEMU binaries. `stub_ss` links dummy stubs that will only be linked into the binary if the real implementation is not present. In a way, the stubs can be thought of as a portable implementation of the weak symbols concept.

The following files concur in the definition of which files are linked into each emulator:

`default-configs/devices/*.mak`

The files under `default-configs/devices/` control the boards and devices that are built into each QEMU system emulation targets. They merely contain a list of config variable definitions such as:

```
include arm-softmmu.mak
CONFIG_XLNX_ZYNQMP_ARM=y
CONFIG_XLNX_VERSAL=y
```

`*/Kconfig`

These files are processed together with `default-configs/devices/*.mak` and describe the dependencies between various features, subsystems and device models. They are described in *QEMU and Kconfig*

`default-configs/targets/*.mak`

These files mostly define symbols that appear in the `*-config-target.h` file for each emulator[3]. However, the `TARGET_ARCH` and `TARGET_BASE_ARCH` will also be used to select the `hw/` and `target/` subdirectories that are compiled into each target.

These files rarely need changing unless you are adding a completely new target, or enabling new devices or hardware for a particular system/userspace emulation target

Adding checks

Compiler checks can be as simple as the following:

```
config_host_data.set('HAVE_BTRFS_H', cc.has_header('linux/btrfs.h'))
```

A more complex task such as adding a new dependency usually comprises the following tasks:

- Add a Meson build option to meson_options.txt.
- Add code to perform the actual feature check.
- Add code to include the feature status in `config-host.h`
- Add code to print out the feature status in the configure summary upon completion.

Taking the probe for SDL2_Image as an example, we have the following in `meson_options.txt`:

```
option('sdl_image', type : 'feature', value : 'auto',
       description: 'SDL Image support for icons')
```

Unless the option was given a non-`auto` value (on the configure command line), the detection code must be performed only if the dependency will be used:

```
sdl_image = not_found
if not get_option('sdl_image').auto() or have_system
  sdl_image = dependency('SDL2_image', required: get_option('sdl_image'),
                         method: 'pkg-config')
endif
```

This avoids warnings on static builds of user-mode emulators, for example. Most of the libraries used by system-mode emulators are not available for static linking.

The other supporting code is generally simple:

```
# Create config-host.h (if applicable)
config_host_data.set('CONFIG_SDL_IMAGE', sdl_image.found())

# Summary
summary_info += {'SDL image support': sdl_image.found()}
```

For the configure script to parse the new option, the `scripts/meson-buildoptions.sh` file must be up-to-date; `make update-buildoptions` (or just `make`) will take care of updating it.

[3] This header is included by `qemu/osdep.h` when compiling files from the target-specific sourcesets.

Support scripts

Meson has a special convention for invoking Python scripts: if their first line is `#! /usr/bin/env python3` and the file is *not* executable, find_program() arranges to invoke the script under the same Python interpreter that was used to invoke Meson. This is the most common and preferred way to invoke support scripts from Meson build files, because it automatically uses the value of configure's –python= option.

In case the script is not written in Python, use a `#! /usr/bin/env ...` line and make the script executable.

Scripts written in Python, where it is desirable to make the script executable (for example for test scripts that developers may want to invoke from the command line, such as tests/qapi-schema/test-qapi.py), should be invoked through the `python` variable in meson.build. For example:

```
test('QAPI schema regression tests', python,
     args: files('test-qapi.py'),
     env: test_env, suite: ['qapi-schema', 'qapi-frontend'])
```

This is needed to obey the –python= option passed to the configure script, which may point to something other than the first python3 binary on the path.

By the time Meson runs, Python dependencies are available in the virtual environment and should be invoked through the scripts that `configure` places under `pyvenv`. One way to do so is as follows, using Meson's `find_program` function:

```
sphinx_build = find_program(
     fs.parent(python.full_path()) / 'sphinx-build',
     required: get_option('docs'))
```

Stage 3: Make

The next step in building QEMU is to invoke make. GNU Make is required to build QEMU, and may be installed as `gmake` on some hosts.

The output of Meson is a `build.ninja` file, which is used with the Ninja build tool. However, QEMU's build comprises other components than just the emulators (namely firmware and the tests in `tests/tcg`) which need different cross compilers. The QEMU Makefile wraps both Ninja and the smaller build systems for firmware and tests; it also takes care of running `configure` again when the script changes. Apart from invoking these sub-Makefiles, the resulting build is largely non-recursive.

Tests, whether defined in `meson.build` or not, are also ran by the Makefile with the traditional `make check` phony target, while benchmarks are run with `make bench`. Meson test suites such as `unit` can be ran with `make check-unit`, and `make check-tcg` builds and runs "non-Meson" tests for all targets.

If desired, it is also possible to use `ninja` and `meson test`, respectively to build emulators and run tests defined in meson.build. The main difference is that `make` needs the `-jN` flag in order to enable parallel builds or tests.

Useful make targets

help
Print a help message for the most common build targets.

print-VAR
Print the value of the variable VAR. Useful for debugging the build system.

Important files for the build system

Statically defined files

The following key files are statically defined in the source tree, with the rules needed to build QEMU. Their behaviour is influenced by a number of dynamically created files listed later.

Makefile
The main entry point used when invoking make to build all the components of QEMU. The default 'all' target will naturally result in the build of every component.

***/meson.build**
The meson.build file in the root directory is the main entry point for the Meson build system, and it coordinates the configuration and build of all executables. Build rules for various subdirectories are included in other meson.build files spread throughout the QEMU source tree.

python/scripts/mkvenv.py
A wrapper for the Python `venv` and `distlib.scripts` packages. It handles creating the virtual environment, creating scripts in `pyvenv/bin`, and calling `pip` to install dependencies.

tests/Makefile.include
Rules for external test harnesses. These include the TCG tests and the Avocado-based integration tests.

tests/docker/Makefile.include
Rules for Docker tests. Like `tests/Makefile.include`, this file is included directly by the top level Makefile, anything defined in this file will influence the entire build system.

tests/vm/Makefile.include
Rules for VM-based tests. Like `tests/Makefile.include`, this file is included directly by the top level Makefile, anything defined in this file will influence the entire build system.

Dynamically created files

The following files are generated at run-time in order to control the behaviour of the Makefiles. This avoids the need for QEMU makefiles to go through any pre-processing as seen with autotools, where configure generates `Makefile` from `Makefile.in`.

Built by configure:

config-host.mak
When configure has determined the characteristics of the build host it will write the paths to various tools to this file, for use in `Makefile` and to a smaller extent `meson.build`.

`config-host.mak` is also used as a dependency checking mechanism. If make sees that the modification timestamp on configure is newer than that on `config-host.mak`, then configure will be re-run.

`config-meson.cross`

A Meson "cross file" (or native file) used to communicate the paths to the toolchain and other configuration options.

`config.status`

A small shell script that will invoke configure again with the same environment variables that were set during the first run. It's used to rerun configure after changes to the source code, but it can also be inspected manually to check the contents of the environment.

`Makefile.prereqs`

A set of Makefile dependencies that order the build and execution of firmware and tests after the container images and emulators that they need.

`pc-bios/*/config.mak`, `tests/tcg/config-host.mak`, `tests/tcg/*/config-target.mak`

Configuration variables used to build the firmware and TCG tests, including paths to cross compilation toolchains.

`pyvenv`

A Python virtual environment that is used for all Python code running during the build. Using a virtual environment ensures that even code that is run via `sphinx-build`, `meson` etc. uses the same interpreter and packages.

Built by Meson:

`config-host.h`
Used by C code to determine the properties of the build environment and the set of enabled features for the entire build.

`${TARGET-NAME}-config-devices.mak`
TARGET-NAME is the name of a system emulator. The file is generated by Meson using files under `configs/devices` as input.

`${TARGET-NAME}-config-target.mak`
TARGET-NAME is the name of a system or usermode emulator. The file is generated by Meson using files under `configs/targets` as input.

`$TARGET_NAME-config-target.h`, `$TARGET_NAME-config-devices.h`
Used by C code to determine the properties and enabled features for each target. enabled. They are generated from the contents of the corresponding `*.mak` files using Meson's `configure_file()` function; each target can include them using the `CONFIG_TARGET` and `CONFIG_DEVICES` macro respectively.

`build.ninja`
The build rules.

Built by Makefile:

`Makefile.ninja`
A Makefile include that bridges to ninja for the actual build. The Makefile is mostly a list of targets that Meson included in build.ninja.

`Makefile.mtest`
The Makefile definitions that let "make check" run tests defined in meson.build. The rules are produced from Meson's JSON description of tests (obtained with "meson introspect –tests") through the script scripts/mtest2make.py.

7.2.2 QEMU and Kconfig

QEMU is a very versatile emulator; it can be built for a variety of targets, where each target can emulate various boards and at the same time different targets can share large amounts of code. For example, a POWER and an x86 board can run the same code to emulate a PCI network card, even though the boards use different PCI host bridges, and they can run the same code to emulate a SCSI disk while using different SCSI adapters. Arm, s390 and x86 boards can all present a virtio-blk disk to their guests, but with three different virtio guest interfaces.

Each QEMU target enables a subset of the boards, devices and buses that are included in QEMU's source code. As a result, each QEMU executable only links a small subset of the files that form QEMU's source code; anything that is not needed to support a particular target is culled.

QEMU uses a simple domain-specific language to describe the dependencies between components. This is useful for two reasons:

- new targets and boards can be added without knowing in detail the architecture of the hardware emulation subsystems. Boards only have to list the components they need, and the compiled executable will include all the required dependencies and all the devices that the user can add to that board;
- users can easily build reduced versions of QEMU that support only a subset of boards or devices. For example, by default most targets will include all emulated PCI devices that QEMU supports, but the build process is configurable and it is easy to drop unnecessary (or otherwise unwanted) code to make a leaner binary.

This domain-specific language is based on the Kconfig language that originated in the Linux kernel, though it was heavily simplified and the handling of dependencies is stricter in QEMU.

Unlike Linux, there is no user interface to edit the configuration, which is instead specified in per-target files under the `default-configs/` directory of the QEMU source tree. This is because, unlike Linux, configuration and dependencies can be treated as a black box when building QEMU; the default configuration that QEMU ships with should be okay in almost all cases.

The Kconfig language

Kconfig defines configurable components in files named `hw/*/Kconfig`. Note that configurable components are _not_ visible in C code as preprocessor symbols; they are only visible in the Makefile. Each configurable component defines a Makefile variable whose name starts with `CONFIG_`.

All elements have boolean (true/false) type; truth is written as `y`, while falsehood is written `n`. They are defined in a Kconfig stanza like the following:

```
config ARM_VIRT
   bool
   imply PCI_DEVICES
   imply VFIO_AMD_XGBE
   imply VFIO_XGMAC
   select A15MPCORE
   select ACPI
   select ARM_SMMUV3
```

The `config` keyword introduces a new configuration element. In the example above, Makefiles will have access to a variable named `CONFIG_ARM_VIRT`, with value `y` or `n` (respectively for boolean true and false).

Boolean expressions can be used within the language, whenever `<expr>` is written in the remainder of this section. The `&&`, `||` and `!` operators respectively denote conjunction (AND), disjunction (OR) and negation (NOT).

The `bool` data type declaration is optional, but it is suggested to include it for clarity and future-proofing. After `bool` the following directives can be included:

dependencies: `depends on <expr>`

This defines a dependency for this configurable element. Dependencies evaluate an expression and force the value of the variable to false if the expression is false.

reverse dependencies: `select <symbol> [if <expr>]`

While `depends on` can force a symbol to false, reverse dependencies can be used to force another symbol to true. In the following example, `CONFIG_BAZ` will be true whenever `CONFIG_FOO` is true:

```
config FOO
  select BAZ
```

The optional expression will prevent `select` from having any effect unless it is true.

Note that unlike Linux's Kconfig implementation, QEMU will detect contradictions between `depends on` and `select` statements and prevent you from building such a configuration.

default value: `default <value> [if <expr>]`

Default values are assigned to the config symbol if no other value was set by the user via `default-configs/*.mak` files, and only if `select` or `depends on` directives do not force the value to true or false respectively. `<value>` can be `y` or `n`; it cannot be an arbitrary Boolean expression. However, a condition for applying the default value can be added with `if`.

A configuration element can have any number of default values (usually, if more than one default is present, they will have different conditions). If multiple default values satisfy their condition, only the first defined one is active.

reverse default (weak reverse dependency): `imply <symbol> [if <expr>]`

This is similar to `select` as it applies a lower limit of `y` to another symbol. However, the lower limit is only a default and the "implied" symbol's value may still be set to `n` from a `default-configs/*.mak` files. The following two examples are equivalent:

```
config FOO
  bool
  imply BAZ

config BAZ
  bool
  default y if FOO
```

The next section explains where to use `imply` or `default y`.

Guidelines for writing Kconfig files

Configurable elements in QEMU fall under five broad groups. Each group declares its dependencies in different ways:

subsystems, of which **buses** are a special case

Example:

```
config SCSI
  bool
```

Subsystems always default to false (they have no `default` directive) and are never visible in `default-configs/*.mak` files. It's up to other symbols to `select` whatever subsystems they require.

They sometimes have `select` directives to bring in other required subsystems or buses. For example, `AUX` (the DisplayPort auxiliary channel "bus") selects `I2C` because it can act as an I2C master too.

devices

Example:

```
config MEGASAS_SCSI_PCI
  bool
  default y if PCI_DEVICES
  depends on PCI
  select SCSI
```

Devices are the most complex of the five. They can have a variety of directives that cooperate so that a default configuration includes all the devices that can be accessed from QEMU.

Devices *depend on* the bus that they lie on, for example a PCI device would specify `depends on PCI`. An MMIO device will likely have no `depends on` directive. Devices also *select* the buses that the device provides, for example a SCSI adapter would specify `select SCSI`. Finally, devices are usually `default y` if and only if they have at least one `depends on`; the default could be conditional on a device group.

Devices also select any optional subsystem that they use; for example a video card might specify `select EDID` if it needs to build EDID information and publish it to the guest.

device groups

Example:

```
config PCI_DEVICES
  bool
```

Device groups provide a convenient mechanism to enable/disable many devices in one go. This is useful when a set of devices is likely to be enabled/disabled by several targets. Device groups usually need no directive and are not used in the Makefile either; they only appear as conditions for `default y` directives.

QEMU currently has three device groups, `PCI_DEVICES`, `I2C_DEVICES`, and `TEST_DEVICES`. PCI devices usually have a `default y if PCI_DEVICES` directive rather than just `default y`. This lets some boards (notably s390) easily support a subset of PCI devices, for example only VFIO (passthrough) and virtio-pci devices. `I2C_DEVICES` is similar to `PCI_DEVICES`. It contains i2c devices that users might reasonably want to plug in to an i2c bus on any board (and not ones which are very board-specific or that need to be wired up in a way that can't be done on the command line). `TEST_DEVICES` instead is used for devices that are rarely used on production virtual machines, but provide useful hooks to test QEMU or KVM.

boards

Example:

```
config SUN4M
  bool
  default y
  depends on SPARC && !SPARC64
  imply TCX
  imply CG3
  select CS4231
  select ECCMEMCTL
  select EMPTY_SLOT
  select ESCC
  select ESP
```

(continues on next page)

(continued from previous page)

```
  select FDC
  select SLAVIO
  select LANCE
  select M48T59
  select STP2000
```

Boards specify their constituent devices using `imply` and `select` directives. A device should be listed under `select` if the board cannot be started at all without it. It should be listed under `imply` if (depending on the QEMU command line) the board may or may not be started without it. Boards default to true, but also have a `depends on` clause to limit them to the appropriate targets. For some targets, not all boards may be supported by hardware virtualization, in which case they also depend on the `TCG` symbol, Other symbols that are commonly used as dependencies for boards include libraries (such as `FDT`) or `TARGET_BIG_ENDIAN` (possibly negated).

Boards are listed for convenience in the `default-configs/*.mak` for the target they apply to.

internal elements

Example:

```
config ECCMEMCTL
  bool
  select ECC
```

Internal elements group code that is useful in several boards or devices. They are usually enabled with `select` and in turn select other elements; they are never visible in `default-configs/*.mak` files, and often not even in the Makefile.

Writing and modifying default configurations

In addition to the Kconfig files under hw/, each target also includes a file called `default-configs/TARGETNAME-softmmu.mak`. These files initialize some Kconfig variables to non-default values and provide the starting point to turn on devices and subsystems.

A file in `default-configs/` looks like the following example:

```
# Default configuration for alpha-softmmu

# Uncomment the following lines to disable these optional devices:
#
#CONFIG_PCI_DEVICES=n
#CONFIG_TEST_DEVICES=n

# Boards:
#
CONFIG_DP264=y
```

The first part, consisting of commented-out `=n` assignments, tells the user which devices or device groups are implied by the boards. The second part, consisting of `=y` assignments, tells the user which boards are supported by the target. The user will typically modify the default configuration by uncommenting lines in the first group, or commenting out lines in the second group.

It is also possible to run QEMU's configure script with the `--without-default-devices` option. When this is done, everything defaults to `n` unless it is `selected` or explicitly switched on in the `.mak` files. In other words, `default` and

`imply` directives are disabled. When QEMU is built with this option, the user will probably want to change some lines in the first group, for example like this:

```
CONFIG_PCI_DEVICES=y
#CONFIG_TEST_DEVICES=n
```

and/or pick a subset of the devices in those device groups. Without further modifications to `configs/devices/`, a system emulator built without default devices might not do much more than start an empty machine, and even then only if `--nodefaults` is specified on the command line. Starting a VM *without* `--nodefaults` is allowed to fail, but should never abort. Failures in `make check` with `--without-default-devices` are considered bugs in the test code: the tests should either use `--nodefaults`, and should be skipped if a necessary device is not present in the build. Such failures should not be worked around with `select` directives.

Right now there is no single place that lists all the optional devices for `CONFIG_PCI_DEVICES` and `CONFIG_TEST_DEVICES`. In the future, we expect that `.mak` files will be automatically generated, so that they will include all these symbols and some help text on what they do.

Kconfig.host

In some special cases, a configurable element depends on host features that are detected by QEMU's configure or `meson.build` scripts; for example some devices depend on the availability of KVM or on the presence of a library on the host.

These symbols should be listed in `Kconfig.host` like this:

```
config TPM
  bool
```

and also listed as follows in the top-level meson.build's host_kconfig variable:

```
host_kconfig = \
  (have_tpm ? ['CONFIG_TPM=y'] : []) + \
  (host_os == 'linux' ? ['CONFIG_LINUX=y'] : []) + \
  (have_ivshmem ? ['CONFIG_IVSHMEM=y'] : []) + \
  ...
```

7.2.3 QEMU Documentation

QEMU's documentation is written in reStructuredText format and built using the Sphinx documentation generator. We generate both the HTML manual and the manpages from the some documentation sources.

hxtool and .hx files

The documentation for QEMU command line options and Human Monitor Protocol (HMP) commands is written in files with the `.hx` suffix. These are processed in two ways:

- `scripts/hxtool` creates C header files from them, which are included in QEMU to do things like handle the `--help` option output
- a Sphinx extension in `docs/sphinx/hxtool.py` generates rST output to be included in the HTML or manpage documentation

The syntax of these `.hx` files is simple. It is broadly an alternation of C code put into the C output and rST format text put into the documentation. A few special directives are recognised; these are all-caps and must be at the beginning of the line.

`HXCOMM` is the comment marker. The line, including any arbitrary text after the marker, is discarded and appears neither in the C output nor the documentation output.

`SRST` starts a reStructuredText section. Following lines are put into the documentation verbatim, and discarded from the C output. The alternative form `SRST()` is used to define a label which can be referenced from elsewhere in the rST documentation. The label will take the form `<DOCNAME-HXFILE-LABEL>`, where `DOCNAME` is the name of the top level rST file, `HXFILE` is the filename of the .hx file without the `.hx` extension, and `LABEL` is the text provided within the `SRST()` directive. For example, `<system/invocation-qemu-options-initrd>`.

`ERST` ends the documentation section started with `SRST`, and switches back to a C code section.

`DEFHEADING()` defines a heading that should appear in both the `--help` output and in the documentation. This directive should be in the C code block. If there is a string inside the brackets, this is the heading to use. If this string is empty, it produces a blank line in the `--help` output and is ignored for the rST output.

`ARCHHEADING()` is a variant of `DEFHEADING()` which produces the heading only if the specified guest architecture was compiled into QEMU. This should be avoided in new documentation.

Within C code sections, you should check the comments at the top of the file to see what the expected usage is, because this varies between files. For instance in `qemu-options.hx` we use the `DEF()` macro to define each option and specify its `--help` text, but in `hmp-commands.hx` the C code sections are elements of an array of structs of type `HMPCommand` which define the name, behaviour and help text for each monitor command.

In the file `qemu-options.hx`, do not try to explicitly define a reStructuredText label within a documentation section. This file is included into two separate Sphinx documents, and some versions of Sphinx will complain about the duplicate label that results. Use the `SRST()` directive documented above, to emit an unambiguous label.

7.2.4 Testing in QEMU

This document describes the testing infrastructure in QEMU.

Testing with "make check"

The "make check" testing family includes most of the C based tests in QEMU. For a quick help, run `make check-help` from the source tree.

The usual way to run these tests is:

```
make check
```

which includes QAPI schema tests, unit tests, QTests and some iotests. Different sub-types of "make check" tests will be explained below.

Before running tests, it is best to build QEMU programs first. Some tests expect the executables to exist and will fail with obscure messages if they cannot find them.

Unit tests

Unit tests, which can be invoked with `make check-unit`, are simple C tests that typically link to individual QEMU object files and exercise them by calling exported functions.

If you are writing new code in QEMU, consider adding a unit test, especially for utility modules that are relatively stateless or have few dependencies. To add a new unit test:

1. Create a new source file. For example, `tests/unit/foo-test.c`.
2. Write the test. Normally you would include the header file which exports the module API, then verify the interface behaves as expected from your test. The test code should be organized with the glib testing framework. Copying and modifying an existing test is usually a good idea.
3. Add the test to `tests/unit/meson.build`. The unit tests are listed in a dictionary called `tests`. The values are any additional sources and dependencies to be linked with the test. For a simple test whose source is in `tests/unit/foo-test.c`, it is enough to add an entry like:

```
{
  ...
  'foo-test': [],
  ...
}
```

Since unit tests don't require environment variables, the simplest way to debug a unit test failure is often directly invoking it or even running it under `gdb`. However there can still be differences in behavior between `make` invocations and your manual run, due to `$MALLOC_PERTURB_` environment variable (which affects memory reclamation and catches invalid pointers better) and gtester options. If necessary, you can run

```
make check-unit V=1
```

and copy the actual command line which executes the unit test, then run it from the command line.

QTest

QTest is a device emulation testing framework. It can be very useful to test device models; it could also control certain aspects of QEMU (such as virtual clock stepping), with a special purpose "qtest" protocol. Refer to *QTest Device Emulation Testing Framework* for more details.

QTest cases can be executed with

```
make check-qtest
```

Writing portable test cases

Both unit tests and qtests can run on POSIX hosts as well as Windows hosts. Care must be taken when writing portable test cases that can be built and run successfully on various hosts. The following list shows some best practices:

- Use portable APIs from glib whenever necessary, e.g.: g_setenv(), g_mkdtemp(), g_mkdir().
- Avoid using hardcoded /tmp for temporary file directory. Use g_get_tmp_dir() instead.
- Bear in mind that Windows has different special string representation for stdin/stdout/stderr and null devices. For example if your test case uses "/dev/fd/2" and "/dev/null" on Linux, remember to use "2" and "nul" on Windows instead. Also IO redirection does not work on Windows, so avoid using "2>nul" whenever necessary.

- If your test cases uses the blkdebug feature, use relative path to pass the config and image file paths in the command line as Windows absolute path contains the delimiter ":" which will confuse the blkdebug parser.
- Use double quotes in your extra QEMU command line in your test cases instead of single quotes, as Windows does not drop single quotes when passing the command line to QEMU.
- Windows opens a file in text mode by default, while a POSIX compliant implementation treats text files and binary files the same. So if your test cases opens a file to write some data and later wants to compare the written data with the original one, be sure to pass the letter 'b' as part of the mode string to fopen(), or O_BINARY flag for the open() call.
- If a certain test case can only run on POSIX or Linux hosts, use a proper #ifdef in the codes. If the whole test suite cannot run on Windows, disable the build in the meson.build file.

QAPI schema tests

The QAPI schema tests validate the QAPI parser used by QMP, by feeding predefined input to the parser and comparing the result with the reference output.

The input/output data is managed under the `tests/qapi-schema` directory. Each test case includes four files that have a common base name:

- `${casename}.json` - the file contains the JSON input for feeding the parser
- `${casename}.out` - the file contains the expected stdout from the parser
- `${casename}.err` - the file contains the expected stderr from the parser
- `${casename}.exit` - the expected error code

Consider adding a new QAPI schema test when you are making a change on the QAPI parser (either fixing a bug or extending/modifying the syntax). To do this:

1. Add four files for the new case as explained above. For example:

   ```
   $EDITOR tests/qapi-schema/foo.{json,out,err,exit}
   ```

2. Add the new test in `tests/Makefile.include`. For example:

   ```
   qapi-schema += foo.json
   ```

check-block

`make check-block` runs a subset of the block layer iotests (the tests that are in the "auto" group). See the "QEMU iotests" section below for more information.

QEMU iotests

QEMU iotests, under the directory `tests/qemu-iotests`, is the testing framework widely used to test block layer related features. It is higher level than "make check" tests and 99% of the code is written in bash or Python scripts. The testing success criteria is golden output comparison, and the test files are named with numbers.

To run iotests, make sure QEMU is built successfully, then switch to the `tests/qemu-iotests` directory under the build directory, and run `./check` with desired arguments from there.

By default, "raw" format and "file" protocol is used; all tests will be executed, except the unsupported ones. You can override the format and protocol with arguments:

```
# test with qcow2 format
./check -qcow2
# or test a different protocol
./check -nbd
```

It's also possible to list test numbers explicitly:

```
# run selected cases with qcow2 format
./check -qcow2 001 030 153
```

Cache mode can be selected with the "-c" option, which may help reveal bugs that are specific to certain cache mode.

More options are supported by the `./check` script, run `./check -h` for help.

Writing a new test case

Consider writing a tests case when you are making any changes to the block layer. An iotest case is usually the choice for that. There are already many test cases, so it is possible that extending one of them may achieve the goal and save the boilerplate to create one. (Unfortunately, there isn't a 100% reliable way to find a related one out of hundreds of tests. One approach is using `git grep`.)

Usually an iotest case consists of two files. One is an executable that produces output to stdout and stderr, the other is the expected reference output. They are given the same number in file names. E.g. Test script `055` and reference output `055.out`.

In rare cases, when outputs differ between cache mode `none` and others, a `.out.nocache` file is added. In other cases, when outputs differ between image formats, more than one `.out` files are created ending with the respective format names, e.g. `178.out.qcow2` and `178.out.raw`.

There isn't a hard rule about how to write a test script, but a new test is usually a (copy and) modification of an existing case. There are a few commonly used ways to create a test:

- A Bash script. It will make use of several environmental variables related to the testing procedure, and could source a group of `common.*` libraries for some common helper routines.
- A Python unittest script. Import `iotests` and create a subclass of `iotests.QMPTestCase`, then call `iotests.main` method. The downside of this approach is that the output is too scarce, and the script is considered harder to debug.
- A simple Python script without using unittest module. This could also import `iotests` for launching QEMU and utilities etc, but it doesn't inherit from `iotests.QMPTestCase` therefore doesn't use the Python unittest execution. This is a combination of 1 and 2.

Pick the language per your preference since both Bash and Python have comparable library support for invoking and interacting with QEMU programs. If you opt for Python, it is strongly recommended to write Python 3 compatible code.

Both Python and Bash frameworks in iotests provide helpers to manage test images. They can be used to create and clean up images under the test directory. If no I/O or any protocol specific feature is needed, it is often more convenient to use the pseudo block driver, `null-co://`, as the test image, which doesn't require image creation or cleaning up. Avoid system-wide devices or files whenever possible, such as `/dev/null` or `/dev/zero`. Otherwise, image locking implications have to be considered. For example, another application on the host may have locked the file, possibly leading to a test failure. If using such devices are explicitly desired, consider adding `locking=off` option to disable image locking.

Debugging a test case

The following options to the `check` script can be useful when debugging a failing test:

- `-gdb` wraps every QEMU invocation in a `gdbserver`, which waits for a connection from a gdb client. The options given to `gdbserver` (e.g. the address on which to listen for connections) are taken from the `$GDB_OPTIONS` environment variable. By default (if `$GDB_OPTIONS` is empty), it listens on `localhost:12345`. It is possible to connect to it for example with `gdb -iex "target remote $addr"`, where `$addr` is the address `gdbserver` listens on. If the `-gdb` option is not used, `$GDB_OPTIONS` is ignored, regardless of whether it is set or not.
- `-valgrind` attaches a valgrind instance to QEMU. If it detects warnings, it will print and save the log in `$TEST_DIR/<valgrind_pid>.valgrind`. The final command line will be `valgrind --log-file=$TEST_DIR/ <valgrind_pid>.valgrind --error-exitcode=99 $QEMU ...`
- `-d` (debug) just increases the logging verbosity, showing for example the QMP commands and answers.
- `-p` (print) redirects QEMU's stdout and stderr to the test output, instead of saving it into a log file in `$TEST_DIR/qemu-machine-<random_string>`.

Test case groups

"Tests may belong to one or more test groups, which are defined in the form of a comment in the test source file. By convention, test groups are listed in the second line of the test file, after the "#!/..." line, like this:

```
#!/usr/bin/env python3
# group: auto quick
#
...
```

Another way of defining groups is creating the tests/qemu-iotests/group.local file. This should be used only for downstream (this file should never appear in upstream). This file may be used for defining some downstream test groups or for temporarily disabling tests, like this:

```
# groups for some company downstream process
#
# ci - tests to run on build
# down - our downstream tests, not for upstream
#
# Format of each line is:
# TEST_NAME TEST_GROUP [TEST_GROUP ]...

013 ci
210 disabled
215 disabled
our-ugly-workaround-test down ci
```

Note that the following group names have a special meaning:

- quick: Tests in this group should finish within a few seconds.
- auto: Tests in this group are used during "make check" and should be runnable in any case. That means they should run with every QEMU binary (also non-x86), with every QEMU configuration (i.e. must not fail if an optional feature is not compiled in - but reporting a "skip" is ok), work at least with the qcow2 file format, work with all kind of host filesystems and users (e.g. "nobody" or "root") and must not take too much memory and disk space (since CI pipelines tend to fail otherwise).
- disabled: Tests in this group are disabled and ignored by check.

Container based tests

Introduction

The container testing framework in QEMU utilizes public images to build and test QEMU in predefined and widely accessible Linux environments. This makes it possible to expand the test coverage across distros, toolchain flavors and library versions. The support was originally written for Docker although we also support Podman as an alternative container runtime. Although many of the target names and scripts are prefixed with "docker" the system will automatically run on whichever is configured.

The container images are also used to augment the generation of tests for testing TCG. See *Testing with "make check-tcg"* for more details.

Docker Prerequisites

Install "docker" with the system package manager and start the Docker service on your development machine, then make sure you have the privilege to run Docker commands. Typically it means setting up passwordless `sudo docker` command or login as root. For example:

```
$ sudo yum install docker
$ # or `apt-get install docker` for Ubuntu, etc.
$ sudo systemctl start docker
$ sudo docker ps
```

The last command should print an empty table, to verify the system is ready.

An alternative method to set up permissions is by adding the current user to "docker" group and making the docker daemon socket file (by default `/var/run/docker.sock`) accessible to the group:

```
$ sudo groupadd docker
$ sudo usermod $USER -a -G docker
$ sudo chown :docker /var/run/docker.sock
```

Note that any one of above configurations makes it possible for the user to exploit the whole host with Docker bind mounting or other privileged operations. So only do it on development machines.

Podman Prerequisites

Install "podman" with the system package manager.

```
$ sudo dnf install podman
$ podman ps
```

The last command should print an empty table, to verify the system is ready.

Quickstart

From source tree, type `make docker-help` to see the help. Testing can be started without configuring or building QEMU (`configure` and `make` are done in the container, with parameters defined by the make target):

```
make docker-test-build@centos8
```

This will create a container instance using the `centos8` image (the image is downloaded and initialized automatically), in which the `test-build` job is executed.

Registry

The QEMU project has a container registry hosted by GitLab at `registry.gitlab.com/qemu-project/qemu` which will automatically be used to pull in pre-built layers. This avoids unnecessary strain on the distro archives created by multiple developers running the same container build steps over and over again. This can be overridden locally by using the `NOCACHE` build option:

```
make docker-image-debian-arm64-cross NOCACHE=1
```

Images

Along with many other images, the `centos8` image is defined in a Dockerfile in `tests/docker/dockerfiles/`, called `centos8.docker`. `make docker-help` command will list all the available images.

A `.pre` script can be added beside the `.docker` file, which will be executed before building the image under the build context directory. This is mainly used to do necessary host side setup. One such setup is `binfmt_misc`, for example, to make qemu-user powered cross build containers work.

Most of the existing Dockerfiles were written by hand, simply by creating a a new `.docker` file under the `tests/docker/dockerfiles/` directory. This has led to an inconsistent set of packages being present across the different containers.

Thus going forward, QEMU is aiming to automatically generate the Dockerfiles using the `lcitool` program provided by the `libvirt-ci` project:

> https://gitlab.com/libvirt/libvirt-ci

`libvirt-ci` contains an `lcitool` program as well as a list of mappings to distribution package names for a wide variety of third party projects. `lcitool` applies the mappings to a list of build pre-requisites in `tests/lcitool/projects/qemu.yml`, determines the list of native packages to install on each distribution, and uses them to generate build environments (dockerfiles and Cirrus CI variable files) that are consistent across OS distribution.

Adding new build pre-requisites

When preparing a patch series that adds a new build pre-requisite to QEMU, the prerequisites should to be added to `tests/lcitool/projects/qemu.yml` in order to make the dependency available in the CI build environments.

In the simple case where the pre-requisite is already known to `libvirt-ci` the following steps are needed:

- Edit `tests/lcitool/projects/qemu.yml` and add the pre-requisite
- Run `make lcitool-refresh` to re-generate all relevant build environment manifests

It may be that `libvirt-ci` does not know about the new pre-requisite. If that is the case, some extra preparation steps will be required first to contribute the mapping to the `libvirt-ci` project:

- Fork the `libvirt-ci` project on gitlab
- Add an entry for the new build prerequisite to `lcitool/facts/mappings.yml`, listing its native package name on as many OS distros as practical. Run `python -m pytest --regenerate-output` and check that the changes are correct.
- Commit the `mappings.yml` change together with the regenerated test files, and submit a merge request to the `libvirt-ci` project. Please note in the description that this is a new build pre-requisite desired for use with QEMU.
- CI pipeline will run to validate that the changes to `mappings.yml` are correct, by attempting to install the newly listed package on all OS distributions supported by `libvirt-ci`.
- Once the merge request is accepted, go back to QEMU and update the `tests/lcitool/libvirt-ci` submodule to point to a commit that contains the `mappings.yml` update. Then add the prerequisite and run `make lcitool-refresh`.
- Please also trigger gitlab container generation pipelines on your change for as many OS distros as practical to make sure that there are no obvious breakages when adding the new pre-requisite. Please see CI documentation page on how to trigger gitlab CI pipelines on your change.
- Please also trigger gitlab container generation pipelines on your change for as many OS distros as practical to make sure that there are no obvious breakages when adding the new pre-requisite. Please see CI documentation page on how to trigger gitlab CI pipelines on your change.

For enterprise distros that default to old, end-of-life versions of the Python runtime, QEMU uses a separate set of mappings that work with more recent versions. These can be found in `tests/lcitool/mappings.yml`. Modifying this file should not be necessary unless the new pre-requisite is a Python library or tool.

Adding new OS distros

In some cases `libvirt-ci` will not know about the OS distro that is desired to be tested. Before adding a new OS distro, discuss the proposed addition:

- Send a mail to qemu-devel, copying people listed in the MAINTAINERS file for `Build and test automation`.

  There are limited CI compute resources available to QEMU, so the cost/benefit tradeoff of adding new OS distros needs to be considered.

- File an issue at https://gitlab.com/libvirt/libvirt-ci/-/issues pointing to the qemu-devel mail thread in the archives.

  This alerts other people who might be interested in the work to avoid duplication, as well as to get feedback from libvirt-ci maintainers on any tips to ease the addition

Assuming there is agreement to add a new OS distro then

- Fork the `libvirt-ci` project on gitlab
- Add metadata under `lcitool/facts/targets/` for the new OS distro. There might be code changes required if the OS distro uses a package format not currently known. The `libvirt-ci` maintainers can advise on this when the issue is filed.
- Edit the `lcitool/facts/mappings.yml` change to add entries for the new OS, listing the native package names for as many packages as practical. Run `python -m pytest --regenerate-output` and check that the changes are correct.
- Commit the changes to `lcitool/facts` and the regenerated test files, and submit a merge request to the `libvirt-ci` project. Please note in the description that this is a new build pre-requisite desired for use with QEMU

- CI pipeline will run to validate that the changes to `mappings.yml` are correct, by attempting to install the newly listed package on all OS distributions supported by `libvirt-ci`.
- Once the merge request is accepted, go back to QEMU and update the `libvirt-ci` submodule to point to a commit that contains the `mappings.yml` update.

Tests

Different tests are added to cover various configurations to build and test QEMU. Docker tests are the executables under `tests/docker` named `test-*`. They are typically shell scripts and are built on top of a shell library, `tests/docker/common.rc`, which provides helpers to find the QEMU source and build it.

The full list of tests is printed in the `make docker-help` help.

Debugging a Docker test failure

When CI tasks, maintainers or yourself report a Docker test failure, follow the below steps to debug it:

1. Locally reproduce the failure with the reported command line. E.g. run `make docker-test-mingw@fedora-win64-cross J=8`.
2. Add "V=1" to the command line, try again, to see the verbose output.
3. Further add "DEBUG=1" to the command line. This will pause in a shell prompt in the container right before testing starts. You could either manually build QEMU and run tests from there, or press Ctrl-D to let the Docker testing continue.
4. If you press Ctrl-D, the same building and testing procedure will begin, and will hopefully run into the error again. After that, you will be dropped to the prompt for debug.

Options

Various options can be used to affect how Docker tests are done. The full list is in the `make docker` help text. The frequently used ones are:

- `V=1`: the same as in top level `make`. It will be propagated to the container and enable verbose output.
- `J=$N`: the number of parallel tasks in make commands in the container, similar to the `-j $N` option in top level `make`. (The `-j` option in top level `make` will not be propagated into the container.)
- `DEBUG=1`: enables debug. See the previous "Debugging a Docker test failure" section.

Thread Sanitizer

Thread Sanitizer (TSan) is a tool which can detect data races. QEMU supports building and testing with this tool.

For more information on TSan:

https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual

Thread Sanitizer in Docker

TSan is currently supported in the ubuntu2204 docker.

The test-tsan test will build using TSan and then run make check.

```
make docker-test-tsan@ubuntu2204
```

TSan warnings under docker are placed in files located at build/tsan/.

We recommend using DEBUG=1 to allow launching the test from inside the docker, and to allow review of the warnings generated by TSan.

Building and Testing with TSan

It is possible to build and test with TSan, with a few additional steps. These steps are normally done automatically in the docker.

There is a one time patch needed in clang-9 or clang-10 at this time:

```
sed -i 's/^const/static const/g' \
    /usr/lib/llvm-10/lib/clang/10.0.0/include/sanitizer/tsan_interface.h
```

To configure the build for TSan:

```
../configure --enable-tsan --cc=clang-10 --cxx=clang++-10 \
             --disable-werror --extra-cflags="-O0"
```

The runtime behavior of TSAN is controlled by the TSAN_OPTIONS environment variable.

More information on the TSAN_OPTIONS can be found here:

https://github.com/google/sanitizers/wiki/ThreadSanitizerFlags

For example:

```
export TSAN_OPTIONS=suppressions=<path to qemu>/tests/tsan/suppressions.tsan \
                    detect_deadlocks=false history_size=7 exitcode=0 \
                    log_path=<build path>/tsan/tsan_warning
```

The above exitcode=0 has TSan continue without error if any warnings are found. This allows for running the test and then checking the warnings afterwards. If you want TSan to stop and exit with error on warnings, use exitcode=66.

TSan Suppressions

Keep in mind that for any data race warning, although there might be a data race detected by TSan, there might be no actual bug here. TSan provides several different mechanisms for suppressing warnings. In general it is recommended to fix the code if possible to eliminate the data race rather than suppress the warning.

A few important files for suppressing warnings are:

tests/tsan/suppressions.tsan - Has TSan warnings we wish to suppress at runtime. The comment on each suppression will typically indicate why we are suppressing it. More information on the file format can be found here:

https://github.com/google/sanitizers/wiki/ThreadSanitizerSuppressions

tests/tsan/ignore.tsan - Has TSan warnings we wish to disable at compile time for test or debug. Add flags to configure to enable:

"–extra-cflags=-fsanitize-blacklist=<src path>/tests/tsan/ignore.tsan"

More information on the file format can be found here under "Blacklist Format":

https://github.com/google/sanitizers/wiki/ThreadSanitizerFlags

TSan Annotations

include/qemu/tsan.h defines annotations. See this file for more descriptions of the annotations themselves. Annotations can be used to suppress TSan warnings or give TSan more information so that it can detect proper relationships between accesses of data.

Annotation examples can be found here:

https://github.com/llvm/llvm-project/tree/master/compiler-rt/test/tsan/

Good files to start with are: annotate_happens_before.cpp and ignore_race.cpp

The full set of annotations can be found here:

https://github.com/llvm/llvm-project/blob/master/compiler-rt/lib/tsan/rtl/tsan_interface_ann.cpp

docker-binfmt-image-debian-% targets

It is possible to combine Debian's bootstrap scripts with a configured `binfmt_misc` to bootstrap a number of Debian's distros including experimental ports not yet supported by a released OS. This can simplify setting up a rootfs by using docker to contain the foreign rootfs rather than manually invoking chroot.

Setting up `binfmt_misc`

You can use the script `qemu-binfmt-conf.sh` to configure a QEMU user binary to automatically run binaries for the foreign architecture. While the scripts will try their best to work with dynamically linked QEMU's a statically linked one will present less potential complications when copying into the docker image. Modern kernels support the `F` (fix binary) flag which will open the QEMU executable on setup and avoids the need to find and re-open in the chroot environment. This is triggered with the `--persistent` flag.

Example invocation

For example to setup the HPPA ports builds of Debian:

```
make docker-binfmt-image-debian-sid-hppa \
  DEB_TYPE=sid DEB_ARCH=hppa \
  DEB_URL=http://ftp.ports.debian.org/debian-ports/ \
  DEB_KEYRING=/usr/share/keyrings/debian-ports-archive-keyring.gpg \
  EXECUTABLE=(pwd)/qemu-hppa V=1
```

The `DEB_` variables are substitutions used by `debian-bootstrap.pre` which is called to do the initial debootstrap of the rootfs before it is copied into the container. The second stage is run as part of the build. The final image will be tagged as `qemu/debian-sid-hppa`.

VM testing

This test suite contains scripts that bootstrap various guest images that have necessary packages to build QEMU. The basic usage is documented in `Makefile` help which is displayed with `make vm-help`.

Quickstart

Run `make vm-help` to list available make targets. Invoke a specific make command to run build test in an image. For example, `make vm-build-freebsd` will build the source tree in the FreeBSD image. The command can be executed from either the source tree or the build dir; if the former, `./configure` is not needed. The command will then generate the test image in `./tests/vm/` under the working directory.

Note: images created by the scripts accept a well-known RSA key pair for SSH access, so they SHOULD NOT be exposed to external interfaces if you are concerned about attackers taking control of the guest and potentially exploiting a QEMU security bug to compromise the host.

QEMU binaries

By default, `qemu-system-x86_64` is searched in $PATH to run the guest. If there isn't one, or if it is older than 2.10, the test won't work. In this case, provide the QEMU binary in env var: `QEMU=/path/to/qemu-2.10+`.

Likewise the path to `qemu-img` can be set in QEMU_IMG environment variable.

Make jobs

The `-j$X` option in the make command line is not propagated into the VM, specify `J=$X` to control the make jobs in the guest.

Debugging

Add `DEBUG=1` and/or `V=1` to the make command to allow interactive debugging and verbose output. If this is not enough, see the next section. `V=1` will be propagated down into the make jobs in the guest.

Manual invocation

Each guest script is an executable script with the same command line options. For example to work with the netbsd guest, use `$QEMU_SRC/tests/vm/netbsd`:

```
$ cd $QEMU_SRC/tests/vm

# To bootstrap the image
$ ./netbsd --build-image --image /var/tmp/netbsd.img
<...>

# To run an arbitrary command in guest (the output will not be echoed unless
# --debug is added)
$ ./netbsd --debug --image /var/tmp/netbsd.img uname -a

# To build QEMU in guest
```

(continues on next page)

(continued from previous page)

```
$ ./netbsd --debug --image /var/tmp/netbsd.img --build-qemu $QEMU_SRC

# To get to an interactive shell
$ ./netbsd --interactive --image /var/tmp/netbsd.img sh
```

Adding new guests

Please look at existing guest scripts for how to add new guests.

Most importantly, create a subclass of BaseVM and implement `build_image()` method and define `BUILD_SCRIPT`, then finally call `basevm.main()` from the script's `main()`.

- Usually in `build_image()`, a template image is downloaded from a predefined URL. `BaseVM._download_with_cache()` takes care of the cache and the checksum, so consider using it.
- Once the image is downloaded, users, SSH server and QEMU build deps should be set up:
  - Root password set to `BaseVM.ROOT_PASS`
  - User `BaseVM.GUEST_USER` is created, and password set to `BaseVM.GUEST_PASS`
  - SSH service is enabled and started on boot, `$QEMU_SRC/tests/keys/id_rsa.pub` is added to ssh's `authorized_keys` file of both root and the normal user
  - DHCP client service is enabled and started on boot, so that it can automatically configure the virtio-net-pci NIC and communicate with QEMU user net (10.0.2.2)
  - Necessary packages are installed to untar the source tarball and build QEMU
- Write a proper `BUILD_SCRIPT` template, which should be a shell script that untars a raw virtio-blk block device, which is the tarball data blob of the QEMU source tree, then configure/build it. Running "make check" is also recommended.

Image fuzzer testing

An image fuzzer was added to exercise format drivers. Currently only qcow2 is supported. To start the fuzzer, run

```
tests/image-fuzzer/runner.py -c '[["qemu-img", "info", "$test_img"]]' /tmp/test qcow2
```

Alternatively, some command different from `qemu-img info` can be tested, by changing the `-c` option.

Integration tests using the Avocado Framework

The `tests/avocado` directory hosts integration tests. They're usually higher level tests, and may interact with external resources and with various guest operating systems.

These tests are written using the Avocado Testing Framework (which must be installed separately) in conjunction with a the `avocado_qemu.Test` class, implemented at `tests/avocado/avocado_qemu`.

Tests based on `avocado_qemu.Test` can easily:

- Customize the command line arguments given to the convenience `self.vm` attribute (a QEMUMachine instance)
- Interact with the QEMU monitor, send QMP commands and check their results
- Interact with the guest OS, using the convenience console device (which may be useful to assert the effectiveness and correctness of command line arguments or QMP commands)

- Interact with external data files that accompany the test itself (see `self.get_data()`)
- Download (and cache) remote data files, such as firmware and kernel images
- Have access to a library of guest OS images (by means of the `avocado.utils.vmimage` library)
- Make use of various other test related utilities available at the test class itself and at the utility library:
  - http://avocado-framework.readthedocs.io/en/latest/api/test/avocado.html#avocado.Test
  - http://avocado-framework.readthedocs.io/en/latest/api/utils/avocado.utils.html

Running tests

You can run the avocado tests simply by executing:

```
make check-avocado
```

This involves the automatic installation, from PyPI, of all the necessary avocado-framework dependencies into the QEMU venv within the build tree (at `./pyvenv`). Test results are also saved within the build tree (at `tests/results`).

Note: the build environment must be using a Python 3 stack, and have the `venv` and `pip` packages installed. If necessary, make sure `configure` is called with `--python=` and that those modules are available. On Debian and Ubuntu based systems, depending on the specific version, they may be on packages named `python3-venv` and `python3-pip`.

It is also possible to run tests based on tags using the `make check-avocado` command and the `AVOCADO_TAGS` environment variable:

```
make check-avocado AVOCADO_TAGS=quick
```

Note that tags separated with commas have an AND behavior, while tags separated by spaces have an OR behavior. For more information on Avocado tags, see:

https://avocado-framework.readthedocs.io/en/latest/guides/user/chapters/tags.html

To run a single test file, a couple of them, or a test within a file using the `make check-avocado` command, set the `AVOCADO_TESTS` environment variable with the test files or test names. To run all tests from a single file, use:

```
make check-avocado AVOCADO_TESTS=$FILEPATH
```

The same is valid to run tests from multiple test files:

```
make check-avocado AVOCADO_TESTS='$FILEPATH1 $FILEPATH2'
```

To run a single test within a file, use:

```
make check-avocado AVOCADO_TESTS=$FILEPATH:$TESTCLASS.$TESTNAME
```

The same is valid to run single tests from multiple test files:

```
make check-avocado AVOCADO_TESTS='$FILEPATH1:$TESTCLASS1.$TESTNAME1 $FILEPATH2:
↪$TESTCLASS2.$TESTNAME2'
```

The scripts installed inside the virtual environment may be used without an "activation". For instance, the Avocado test runner may be invoked by running:

```
pyvenv/bin/avocado run $OPTION1 $OPTION2 tests/avocado/
```

Note that if `make check-avocado` was not executed before, it is possible to create the Python virtual environment with the dependencies needed running:

```
make check-venv
```

It is also possible to run tests from a single file or a single test within a test file. To run tests from a single file within the build tree, use:

```
pyvenv/bin/avocado run tests/avocado/$TESTFILE
```

To run a single test within a test file, use:

```
pyvenv/bin/avocado run tests/avocado/$TESTFILE:$TESTCLASS.$TESTNAME
```

Valid test names are visible in the output from any previous execution of Avocado or `make check-avocado`, and can also be queried using:

```
pyvenv/bin/avocado list tests/avocado
```

Manual Installation

To manually install Avocado and its dependencies, run:

```
pip install --user avocado-framework
```

Alternatively, follow the instructions on this link:

https://avocado-framework.readthedocs.io/en/latest/guides/user/chapters/installing.html

Overview

The `tests/avocado/avocado_qemu` directory provides the `avocado_qemu` Python module, containing the `avocado_qemu.Test` class. Here's a simple usage example:

```
from avocado_qemu import QemuSystemTest


class Version(QemuSystemTest):
    """
    :avocado: tags=quick
    """
    def test_qmp_human_info_version(self):
        self.vm.launch()
        res = self.vm.cmd('human-monitor-command',
                          command_line='info version')
        self.assertRegex(res, r'^(\d+\.\d+\.\d)')
```

To execute your test, run:

```
avocado run version.py
```

Tests may be classified according to a convention by using docstring directives such as `:avocado: tags=TAG1,TAG2`. To run all tests in the current directory, tagged as "quick", run:

```
avocado run -t quick .
```

The avocado_qemu.Test base test class

The avocado_qemu.Test class has a number of characteristics that are worth being mentioned right away.

First of all, it attempts to give each test a ready to use QEMUMachine instance, available at self.vm. Because many tests will tweak the QEMU command line, launching the QEMUMachine (by using self.vm.launch()) is left to the test writer.

The base test class has also support for tests with more than one QEMUMachine. The way to get machines is through the self.get_vm() method which will return a QEMUMachine instance. The self.get_vm() method accepts arguments that will be passed to the QEMUMachine creation and also an optional name attribute so you can identify a specific machine and get it more than once through the tests methods. A simple and hypothetical example follows:

```
from avocado_qemu import QemuSystemTest


class MultipleMachines(QemuSystemTest):
    def test_multiple_machines(self):
        first_machine = self.get_vm()
        second_machine = self.get_vm()
        self.get_vm(name='third_machine').launch()

        first_machine.launch()
        second_machine.launch()

        first_res = first_machine.cmd(
            'human-monitor-command',
            command_line='info version')

        second_res = second_machine.cmd(
            'human-monitor-command',
            command_line='info version')

        third_res = self.get_vm(name='third_machine').cmd(
            'human-monitor-command',
            command_line='info version')

        self.assertEqual(first_res, second_res, third_res)
```

At test "tear down", avocado_qemu.Test handles all the QEMUMachines shutdown.

The avocado_qemu.LinuxTest base test class

The avocado_qemu.LinuxTest is further specialization of the avocado_qemu.Test class, so it contains all the characteristics of the later plus some extra features.

First of all, this base class is intended for tests that need to interact with a fully booted and operational Linux guest. At this time, it uses a Fedora 31 guest image. The most basic example looks like this:

```
from avocado_qemu import LinuxTest


class SomeTest(LinuxTest):

    def test(self):
        self.launch_and_wait()
        self.ssh_command('some_command_to_be_run_in_the_guest')
```

Please refer to tests that use avocado_qemu.LinuxTest under tests/avocado for more examples.

QEMUMachine

The QEMUMachine API is already widely used in the Python iotests, device-crash-test and other Python scripts. It's a wrapper around the execution of a QEMU binary, giving its users:

- the ability to set command line arguments to be given to the QEMU binary
- a ready to use QMP connection and interface, which can be used to send commands and inspect its results, as well as asynchronous events
- convenience methods to set commonly used command line arguments in a more succinct and intuitive way

QEMU binary selection

The QEMU binary used for the self.vm QEMUMachine instance will primarily depend on the value of the qemu_bin parameter. If it's not explicitly set, its default value will be the result of a dynamic probe in the same source tree. A suitable binary will be one that targets the architecture matching host machine.

Based on this description, test writers will usually rely on one of the following approaches:

1) Set qemu_bin, and use the given binary
2) Do not set qemu_bin, and use a QEMU binary named like "qemu-system-${arch}", either in the current working directory, or in the current source tree.

The resulting qemu_bin value will be preserved in the avocado_qemu.Test as an attribute with the same name.

Attribute reference

Test

Besides the attributes and methods that are part of the base `avocado.Test` class, the following attributes are available on any `avocado_qemu.Test` instance.

vm

A QEMUMachine instance, initially configured according to the given `qemu_bin` parameter.

arch

The architecture can be used on different levels of the stack, e.g. by the framework or by the test itself. At the framework level, it will currently influence the selection of a QEMU binary (when one is not explicitly given).

Tests are also free to use this attribute value, for their own needs. A test may, for instance, use the same value when selecting the architecture of a kernel or disk image to boot a VM with.

The `arch` attribute will be set to the test parameter of the same name. If one is not given explicitly, it will either be set to `None`, or, if the test is tagged with one (and only one) `:avocado: tags=arch:VALUE` tag, it will be set to `VALUE`.

cpu

The cpu model that will be set to all QEMUMachine instances created by the test.

The `cpu` attribute will be set to the test parameter of the same name. If one is not given explicitly, it will either be set to `None ``, or, if the test is tagged with one (and only one) ``:avocado: tags=cpu:VALUE` tag, it will be set to `VALUE`.

machine

The machine type that will be set to all QEMUMachine instances created by the test.

The `machine` attribute will be set to the test parameter of the same name. If one is not given explicitly, it will either be set to `None`, or, if the test is tagged with one (and only one) `:avocado: tags=machine:VALUE` tag, it will be set to `VALUE`.

qemu_bin

The preserved value of the `qemu_bin` parameter or the result of the dynamic probe for a QEMU binary in the current working directory or source tree.

LinuxTest

Besides the attributes present on the `avocado_qemu.Test` base class, the `avocado_qemu.LinuxTest` adds the following attributes:

distro

The name of the Linux distribution used as the guest image for the test. The name should match the **Provider** column on the list of images supported by the avocado.utils.vmimage library:

https://avocado-framework.readthedocs.io/en/latest/guides/writer/libs/vmimage.html#supported-images

distro_version

The version of the Linux distribution as the guest image for the test. The name should match the **Version** column on the list of images supported by the avocado.utils.vmimage library:

https://avocado-framework.readthedocs.io/en/latest/guides/writer/libs/vmimage.html#supported-images

distro_checksum

The sha256 hash of the guest image file used for the test.

If this value is not set in the code or by a test parameter (with the same name), no validation on the integrity of the image will be performed.

Parameter reference

To understand how Avocado parameters are accessed by tests, and how they can be passed to tests, please refer to:

```
https://avocado-framework.readthedocs.io/en/latest/guides/writer/chapters/writing.html
↪#accessing-test-parameters
```

Parameter values can be easily seen in the log files, and will look like the following:

```
PARAMS (key=qemu_bin, path=*, default=./qemu-system-x86_64) => './qemu-system-x86_64
```

Test

arch

The architecture that will influence the selection of a QEMU binary (when one is not explicitly given).

Tests are also free to use this parameter value, for their own needs. A test may, for instance, use the same value when selecting the architecture of a kernel or disk image to boot a VM with.

This parameter has a direct relation with the `arch` attribute. If not given, it will default to None.

cpu

The cpu model that will be set to all QEMUMachine instances created by the test.

machine

The machine type that will be set to all QEMUMachine instances created by the test.

qemu_bin

The exact QEMU binary to be used on QEMUMachine.

LinuxTest

Besides the parameters present on the `avocado_qemu.Test` base class, the `avocado_qemu.LinuxTest` adds the following parameters:

distro

The name of the Linux distribution used as the guest image for the test. The name should match the **Provider** column on the list of images supported by the avocado.utils.vmimage library:

https://avocado-framework.readthedocs.io/en/latest/guides/writer/libs/vmimage.html#supported-images

distro_version

The version of the Linux distribution as the guest image for the test. The name should match the **Version** column on the list of images supported by the avocado.utils.vmimage library:

https://avocado-framework.readthedocs.io/en/latest/guides/writer/libs/vmimage.html#supported-images

distro_checksum

The sha256 hash of the guest image file used for the test.

If this value is not set in the code or by this parameter no validation on the integrity of the image will be performed.

Skipping tests

The Avocado framework provides Python decorators which allow for easily skip tests running under certain conditions. For example, on the lack of a binary on the test system or when the running environment is a CI system. For further information about those decorators, please refer to:

```
https://avocado-framework.readthedocs.io/en/latest/guides/writer/chapters/writing.html
↪#skipping-tests
```

While the conditions for skipping tests are often specifics of each one, there are recurring scenarios identified by the QEMU developers and the use of environment variables became a kind of standard way to enable/disable tests.

Here is a list of the most used variables:

AVOCADO_ALLOW_LARGE_STORAGE

Tests which are going to fetch or produce assets considered *large* are not going to run unless that `AVOCADO_ALLOW_LARGE_STORAGE=1` is exported on the environment.

The definition of *large* is a bit arbitrary here, but it usually means an asset which occupies at least 1GB of size on disk when uncompressed.

SPEED

Tests which have a long runtime will not be run unless `SPEED=slow` is exported on the environment.

The definition of *long* is a bit arbitrary here, and it depends on the usefulness of the test too. A unique test is worth spending more time on, small variations on existing tests perhaps less so. As a rough guide, a test or set of similar tests which take more than 100 seconds to complete.

AVOCADO_ALLOW_UNTRUSTED_CODE

There are tests which will boot a kernel image or firmware that can be considered not safe to run on the developer's workstation, thus they are skipped by default. The definition of *not safe* is also arbitrary but usually it means a blob which either its source or build process aren't public available.

You should export `AVOCADO_ALLOW_UNTRUSTED_CODE=1` on the environment in order to allow tests which make use of those kind of assets.

AVOCADO_TIMEOUT_EXPECTED

The Avocado framework has a timeout mechanism which interrupts tests to avoid the test suite of getting stuck. The timeout value can be set via test parameter or property defined in the test class, for further details:

```
https://avocado-framework.readthedocs.io/en/latest/guides/writer/chapters/writing.html
↪#setting-a-test-timeout
```

Even though the timeout can be set by the test developer, there are some tests that may not have a well-defined limit of time to finish under certain conditions. For example, tests that take longer to execute when QEMU is compiled with debug flags. Therefore, the `AVOCADO_TIMEOUT_EXPECTED` variable has been used to determine whether those tests should run or not.

QEMU_TEST_FLAKY_TESTS

Some tests are not working reliably and thus are disabled by default. This includes tests that don't run reliably on GitLab's CI which usually expose real issues that are rarely seen on developer machines due to the constraints of the CI environment. If you encounter a similar situation then raise a bug and then mark the test as shown on the code snippet below:

```
# See https://gitlab.com/qemu-project/qemu/-/issues/nnnn
@skipUnless(os.getenv('QEMU_TEST_FLAKY_TESTS'), 'Test is unstable on GitLab')
def test(self):
    do_something()
```

You can also add `:avocado: tags=flaky` to the test meta-data so only the flaky tests can be run as a group:

```
env QEMU_TEST_FLAKY_TESTS=1 ./pyvenv/bin/avocado \
   run tests/avocado -filter-by-tags=flaky
```

Tests should not live in this state forever and should either be fixed or eventually removed.

Uninstalling Avocado

If you've followed the manual installation instructions above, you can easily uninstall Avocado. Start by listing the packages you have installed:

```
pip list --user
```

And remove any package you want with:

```
pip uninstall <package_name>
```

If you've used `make check-avocado`, the Python virtual environment where Avocado is installed will be cleaned up as part of `make check-clean`.

Testing with "make check-tcg"

The check-tcg tests are intended for simple smoke tests of both linux-user and softmmu TCG functionality. However to build test programs for guest targets you need to have cross compilers available. If your distribution supports cross compilers you can do something as simple as:

```
apt install gcc-aarch64-linux-gnu
```

The configure script will automatically pick up their presence. Sometimes compilers have slightly odd names so the availability of them can be prompted by passing in the appropriate configure option for the architecture in question, for example:

```
$(configure) --cross-cc-aarch64=aarch64-cc
```

There is also a `--cross-cc-cflags-ARCH` flag in case additional compiler flags are needed to build for a given target.

If you have the ability to run containers as the user the build system will automatically use them where no system compiler is available. For architectures where we also support building QEMU we will generally use the same container to build tests. However there are a number of additional containers defined that have a minimal cross-build environment that is only suitable for building test cases. Sometimes we may use a bleeding edge distribution for compiler features needed for test cases that aren't yet in the LTS distros we support for QEMU itself.

See *Container based tests* for more details.

Running subset of tests

You can build the tests for one architecture:

```
make build-tcg-tests-$TARGET
```

And run with:

```
make run-tcg-tests-$TARGET
```

Adding `V=1` to the invocation will show the details of how to invoke QEMU for the test which is useful for debugging tests.

TCG test dependencies

The TCG tests are deliberately very light on dependencies and are either totally bare with minimal gcc lib support (for system-mode tests) or just glibc (for linux-user tests). This is because getting a cross compiler to work with additional libraries can be challenging.

Other TCG Tests

There are a number of out-of-tree test suites that are used for more extensive testing of processor features.

KVM Unit Tests

The KVM unit tests are designed to run as a Guest OS under KVM but there is no reason why they can't exercise the TCG as well. It provides a minimal OS kernel with hooks for enabling the MMU as well as reporting test results via a special device:

```
https://git.kernel.org/pub/scm/virt/kvm/kvm-unit-tests.git
```

Linux Test Project

The LTP is focused on exercising the syscall interface of a Linux kernel. It checks that syscalls behave as documented and strives to exercise as many corner cases as possible. It is a useful test suite to run to exercise QEMU's linux-user code:

```
https://linux-test-project.github.io/
```

GCC gcov support

`gcov` is a GCC tool to analyze the testing coverage by instrumenting the tested code. To use it, configure QEMU with `--enable-gcov` option and build. Then run the tests as usual.

If you want to gather coverage information on a single test the `make clean-gcda` target can be used to delete any existing coverage information before running a single test.

You can generate a HTML coverage report by executing `make coverage-html` which will create `meson-logs/coveragereport/index.html`.

Further analysis can be conducted by running the `gcov` command directly on the various .gcda output files. Please read the `gcov` documentation for more information.

7.2.5 ACPI/SMBIOS avocado tests using biosbits

Introduction

Biosbits is a software written by Josh Triplett that can be downloaded from https://biosbits.org/. The github codebase can be found here. It is a software that executes the bios components such as acpi and smbios tables directly through acpica bios interpreter (a freely available C based library written by Intel, downloadable from https://acpica.org/ and is included with biosbits) without an operating system getting involved in between. Bios-bits has python integration with grub so actual routines that executes bios components can be written in python instead of bash-ish (grub's native scripting language). There are several advantages to directly testing the bios in a real physical machine or in a VM as opposed to indirectly discovering bios issues through the operating system (the OS). Operating systems tend to bypass bios problems and hide them from the end user. We have more control of what we wanted to test and how by being as close to the bios on a running system as possible without a complicated software component such as an operating system coming in between. Another issue is that we cannot exercise bios components such as ACPI and SMBIOS without being in the highest hardware privilege level, ring 0 for example in case of x86. Since the OS executes from ring 0 whereas normal user land software resides in unprivileged ring 3, operating system must be modified in order to write our test routines that exercise and test the bios. This is not possible in all cases. Lastly, test frameworks and routines are preferably written using a high level scripting language such as python. OSes and OS modules are generally written using low level languages such as C and low level assembly machine language. Writing test routines in a low level language makes things more cumbersome. These and other reasons makes using bios-bits very attractive for testing bioses. More details on the inspiration for developing biosbits and its real life uses can be found in[1] and[2].

For QEMU, we maintain a fork of bios bits in gitlab along with all the dependent submodules here. This fork contains numerous fixes, a newer acpica and changes specific to running this avocado QEMU tests using bits. The author of this document is the sole maintainer of the QEMU fork of bios bits repository. For more information, please see author's FOSDEM talk on this bios-bits based test framework.

Description of the test framework

Under the directory `tests/avocado/`, `acpi-bits.py` is a QEMU avocado test that drives all this.

A brief description of the various test files follows.

Under `tests/avocado/` as the root we have:

```
├── acpi-bits
│   ├── bits-config
│   └── bits-cfg.txt
```

(continues on next page)

[1] https://blog.linuxplumbersconf.org/2011/ocw/system/presentations/867/original/bits.pdf
[2] https://www.youtube.com/watch?v=36QIepyUuhg

(continued from previous page)

```
│  ├── bits-tests
│  │   ├── smbios.py2
│  │   ├── testacpi.py2
│  │   └── testcpuid.py2
├── acpi-bits.py
```

- `tests/avocado`:

  `acpi-bits.py`: This is the main python avocado test script that generates a biosbits iso. It then spawns a QEMU VM with it, collects the log and reports test failures. This is the script one would be interested in if they wanted to add or change some component of the log parsing, add a new command line to alter how QEMU is spawned etc. Test writers typically would not need to modify this script unless they wanted to enhance or change the log parsing for their tests. In order to enable debugging, you can set **V=1** environment variable. This enables verbose mode for the test and also dumps the entire log from bios bits and more information in case failure happens. You can also set **BITS_DEBUG=1** to turn on debug mode. It will enable verbose logs and also retain the temporary work directory the test used for you to inspect and run the specific commands manually.

  In order to run this test, please perform the following steps from the QEMU build directory:

  ```
  $ make check-venv (needed only the first time to create the venv)
  $ ./pyvenv/bin/avocado run -t acpi tests/avocado
  ```

  The above will run all acpi avocado tests including this one. In order to run the individual tests, perform the following:

  ```
  $ ./pyvenv/bin/avocado run tests/avocado/acpi-bits.py --tap -
  ```

  The above will produce output in tap format. You can omit "–tap -" in the end and it will produce output like the following:

  ```
  $ ./pyvenv/bin/avocado run tests/avocado/acpi-bits.py
  Fetching asset from tests/avocado/acpi-bits.py:AcpiBitsTest.test_acpi_
  ↪smbios_bits
  JOB ID     : eab225724da7b64c012c65705dc2fa14ab1defef
  JOB LOG    : /home/anisinha/avocado/job-results/job-2022-10-10T17.58-
  ↪eab2257/job.log
  (1/1) tests/avocado/acpi-bits.py:AcpiBitsTest.test_acpi_smbios_bits: PASS␣
  ↪(33.09 s)
  RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 |␣
  ↪CANCEL 0
  JOB TIME   : 39.22 s
  ```

  You can inspect the log file for more information about the run or in order to diagnoze issues. If you pass V=1 in the environment, more diagnostic logs would be found in the test log.

- `tests/avocado/acpi-bits/bits-config`:

  This location contains biosbits configuration files that determine how the software runs the tests.

  `bits-config.txt`: This is the biosbits config file that determines what tests or actions are performed by bits. The description of the config options are provided in the file itself.

- `tests/avocado/acpi-bits/bits-tests`:

  This directory contains biosbits python based tests that are run from within the biosbits environment in the spawned VM. New additions of test cases can be made in the appropriate test file. For exam-

ple, new acpi tests can go into testacpi.py2 and one would call testsuite.add_test() to register the new test so that it gets executed as a part of the ACPI tests. It might be occasionally necessary to disable some subtests or add a new test that belongs to a test suite not already present in this directory. To do this, please clone the bits source from https://gitlab.com/qemu-project/biosbits-bits/-/tree/qemu-bits. Note that this is the "qemu-bits" branch and not the "bits" branch of the repository. "qemu-bits" is the branch where we have made all the QEMU specific enhancements and we must use the source from this branch only. Copy the test suite/script that needs modification (addition of new tests or disabling them) from python directory into this directory. For example, in order to change cpuid related tests, copy the following file into this directory and rename it with .py2 extension: https://gitlab.com/qemu-project/biosbits-bits/-/blob/qemu-bits/python/testcpuid.py Then make your additions and changes here. Therefore, the steps are:

(a) Copy unmodified test script to this directory from bits source.

(b) Add a SPDX license header.

(c) Perform modifications to the test.

Commits (a), (b) and (c) preferably should go under separate commits so that the original test script and the changes we have made are separated and clear. (a) and (b) can sometimes be combined into a single step.

The test framework will then use your modified test script to run the test. No further changes would be needed. Please check the logs to make sure that appropriate changes have taken effect.

The tests have an extension .py2 in order to indicate that:

(a) They are python2.7 based scripts and not python 3 scripts.

(b) They are run from within the bios bits VM and is not subjected to QEMU build/test python script maintenance and dependency resolutions.

(c) They need not be loaded by avocado framework when running tests.

Author: Ani Sinha <anisinha@redhat.com>

References:

7.2.6 QTest Device Emulation Testing Framework

Qtest Driver Framework

In order to test a specific driver, plain libqos tests need to take care of booting QEMU with the right machine and devices. This makes each test "hardcoded" for a specific configuration, reducing the possible coverage that it can reach.

For example, the sdhci device is supported on both x86_64 and ARM boards, therefore a generic sdhci test should test all machines and drivers that support that device. Using only libqos APIs, the test has to manually take care of covering all the setups, and build the correct command line.

This also introduces backward compatibility issues: if a device/driver command line name is changed, all tests that use that will not work properly anymore and need to be adjusted.

The aim of qgraph is to create a graph of drivers, machines and tests such that a test aimed to a certain driver does not have to care of booting the right QEMU machine, pick the right device, build the command line and so on. Instead, it only defines what type of device it is testing (interface in qgraph terms) and the framework takes care of covering all supported types of devices and machine architectures.

Following the above example, an interface would be `sdhci`, so the sdhci-test should only care of linking its qgraph node with that interface. In this way, if the command line of a sdhci driver is changed, only the respective qgraph driver node has to be adjusted.

QGraph concepts

The graph is composed by nodes that represent machines, drivers, tests and edges that define the relationships between them (`CONSUMES`, `PRODUCES`, and `CONTAINS`).

Nodes

A node can be of four types:

- **QNODE_MACHINE**: for example `arm/raspi2b`
- **QNODE_DRIVER**: for example `generic-sdhci`
- **QNODE_INTERFACE**: for example `sdhci` (interface for all `-sdhci` drivers). An interface is not explicitly created, it will be automatically instantiated when a node consumes or produces it. An interface is simply a struct that abstracts the various drivers for the same type of device, and offers an API to the nodes that use it ("consume" relation in qgraph terms) that is implemented/backed up by the drivers that implement it ("produce" relation in qgraph terms).
- **QNODE_TEST**: for example `sdhci-test`. A test consumes an interface and tests the functions provided by it.

Notes for the nodes:

- QNODE_MACHINE: each machine struct must have a `QGuestAllocator` and implement `get_driver()` to return the allocator mapped to the interface "memory". The function can also return `NULL` if the allocator is not set.
- QNODE_DRIVER: driver names must be unique, and machines and nodes planned to be "consumed" by other nodes must match QEMU drivers name, otherwise they won't be discovered

Edges

An edge relation between two nodes (drivers or machines) `X` and `Y` can be:

- `X CONSUMES Y`: `Y` can be plugged into `X`
- `X PRODUCES Y`: `X` provides the interface `Y`
- `X CONTAINS Y`: `Y` is part of `X` component

Execution steps

The basic framework steps are the following:

- All nodes and edges are created in their respective machine/driver/test files
- The framework starts QEMU and asks for a list of available devices and machines (note that only machines and "consumed" nodes are mapped 1:1 with QEMU devices)
- The framework walks the graph starting from the available machines and performs a Depth First Search for tests
- Once a test is found, the path is walked again and all drivers are allocated accordingly and the final interface is passed to the test

- The test is executed
- Unused objects are cleaned and the path discovery is continued

Depending on the QEMU binary used, only some drivers/machines will be available and only test that are reached by them will be executed.

Command line

Command line is built by using node names and optional arguments passed by the user when building the edges.

There are three types of command line arguments:

- `in node` : created from the node name. For example, machines will have `-M <machine>` to its command line, while devices `-device <device>`. It is automatically done by the framework.
- `after node` : added as additional argument to the node name. This argument is added optionally when creating edges, by setting the parameter `after_cmd_line` and `extra_edge_opts` in `QOSGraphEdgeOptions`. The framework automatically adds a comma before `extra_edge_opts`, because it is going to add attributes after the destination node pointed by the edge containing these options, and automatically adds a space before `after_cmd_line`, because it adds an additional device, not an attribute.
- `before node` : added as additional argument to the node name. This argument is added optionally when creating edges, by setting the parameter `before_cmd_line` in `QOSGraphEdgeOptions`. This attribute is going to add attributes before the destination node pointed by the edge containing these options. It is helpful to commands that are not node-representable, such as `-fdsev` or `-netdev`.

While adding command line in edges is always used, not all nodes names are used in every path walk: this is because the contained or produced ones are already added by QEMU, so only nodes that "consumes" will be used to build the command line. Also, nodes that will have `{ "abstract" : true }` as QMP attribute will loose their command line, since they are not proper devices to be added in QEMU.

Example:

```
QOSGraphEdgeOptions opts = {
    .before_cmd_line = "-drive id=drv0,if=none,file=null-co://,"
                       "file.read-zeroes=on,format=raw",
    .after_cmd_line = "-device scsi-hd,bus=vs0.0,drive=drv0",

    opts.extra_device_opts = "id=vs0";
};

qos_node_create_driver("virtio-scsi-device",
                       virtio_scsi_device_create);
qos_node_consumes("virtio-scsi-device", "virtio-bus", &opts);
```

Will produce the following command line: `-drive id=drv0,if=none,file=null-co://, -device virtio-scsi-device,id=vs0 -device scsi-hd,bus=vs0.0,drive=drv0`

Troubleshooting unavailable tests

If there is no path from an available machine to a test then that test will be unavailable and won't execute. This can happen if a test or driver did not set up its qgraph node correctly. It can also happen if the necessary machine type or device is missing from the QEMU binary because it was compiled out or otherwise.

It is possible to troubleshoot unavailable tests by running:

```
$ QTEST_QEMU_BINARY=build/qemu-system-x86_64 build/tests/qtest/qos-test --verbose
# ALL QGRAPH EDGES: {
#   src='virtio-net'
#      |-> dest='virtio-net-tests/vhost-user/multiqueue' type=2 (node=0x559142109e30)
#      |-> dest='virtio-net-tests/vhost-user/migrate' type=2 (node=0x559142109d00)
#   src='virtio-net-pci'
#      |-> dest='virtio-net' type=1 (node=0x55914210d740)
#   src='pci-bus'
#      |-> dest='virtio-net-pci' type=2 (node=0x55914210d880)
#   src='pci-bus-pc'
#      |-> dest='pci-bus' type=1 (node=0x559142103f40)
#   src='i440FX-pcihost'
#      |-> dest='pci-bus-pc' type=0 (node=0x55914210ac70)
#   src='x86_64/pc'
#      |-> dest='i440FX-pcihost' type=0 (node=0x5591421117f0)
#   src=''
#      |-> dest='x86_64/pc' type=0 (node=0x559142111600)
#      |-> dest='arm/raspi2b' type=0 (node=0x559142110740)
...
# }
# ALL QGRAPH NODES: {
#   name='virtio-net-tests/announce-self' type=3 cmd_line='(null)' [available]
#   name='arm/raspi2b' type=0 cmd_line='-M raspi2b ' [UNAVAILABLE]
...
# }
```

The `virtio-net-tests/announce-self` test is listed as "available" in the "ALL QGRAPH NODES" output. This means the test will execute. We can follow the qgraph path in the "ALL QGRAPH EDGES" output as follows: '' -> 'x86_64/pc' -> 'i440FX-pcihost' -> 'pci-bus-pc' -> 'pci-bus' -> 'virtio-net-pci' -> 'virtio-net'. The root of the qgraph is '' and the depth first search begins there.

The `arm/raspi2b` machine node is listed as "UNAVAILABLE". Although it is reachable from the root via '' -> 'arm/raspi2b' the node is unavailable because the QEMU binary did not list it when queried by the framework. This is expected because we used the `qemu-system-x86_64` binary which does not support ARM machine types.

If a test is unexpectedly listed as "UNAVAILABLE", first check that the "ALL QGRAPH EDGES" output reports edge connectivity from the root ('') to the test. If there is no connectivity then the qgraph nodes were not set up correctly and the driver or test code is incorrect. If there is connectivity, check the availability of each node in the path in the "ALL QGRAPH NODES" output. The first unavailable node in the path is the reason why the test is unavailable. Typically this is because the QEMU binary lacks support for the necessary machine type or device.

Creating a new driver and its interface

Here we continue the sdhci use case, with the following scenario:

- sdhci-test aims to test the read[q,w], writeq functions offered by the sdhci drivers.
- The current sdhci device is supported by both x86_64/pc and ARM (in this example we focus on the arm-raspi2b) machines.
- QEMU offers 2 types of drivers: QSDHCI_MemoryMapped for ARM and QSDHCI_PCI for x86_64/pc. Both implement the read[q,w], writeq functions.

In order to implement such scenario in qgraph, the test developer needs to:

- Create the x86_64/pc machine node. This machine uses the pci-bus architecture so it contains a PCI driver, pci-bus-pc. The actual path is

  x86_64/pc --contains--> 1440FX-pcihost --contains--> pci-bus-pc --produces--> pci-bus.

  For the sake of this example, we do not focus on the PCI interface implementation.

- Create the sdhci-pci driver node, representing QSDHCI_PCI. The driver uses the PCI bus (and its API), so it must consume the pci-bus generic interface (which abstracts all the pci drivers available)

  sdhci-pci --consumes--> pci-bus

- Create an arm/raspi2b machine node. This machine contains a generic-sdhci memory mapped sdhci driver node, representing QSDHCI_MemoryMapped.

  arm/raspi2b --contains--> generic-sdhci

- Create the sdhci interface node. This interface offers the functions that are shared by all sdhci devices. The interface is produced by sdhci-pci and generic-sdhci, the available architecture-specific drivers.

  sdhci-pci --produces--> sdhci

  generic-sdhci --produces--> sdhci

- Create the sdhci-test test node. The test consumes the sdhci interface, using its API. It doesn't need to look at the supported machines or drivers.

  sdhci-test --consumes--> sdhci

arm-raspi2b machine, simplified from tests/qtest/libqos/arm-raspi2-machine.c:

```
#include "qgraph.h"

struct QRaspi2Machine {
    QOSGraphObject obj;
    QGuestAllocator alloc;
    QSDHCI_MemoryMapped sdhci;
};

static void *raspi2_get_driver(void *object, const char *interface)
{
    QRaspi2Machine *machine = object;
    if (!g_strcmp0(interface, "memory")) {
        return &machine->alloc;
    }
```

(continues on next page)

(continued from previous page)

```
    fprintf(stderr, "%s not present in arm/raspi2b\n", interface);
    g_assert_not_reached();
}

static QOSGraphObject *raspi2_get_device(void *obj,
                                         const char *device)
{
    QRaspi2Machine *machine = obj;
    if (!g_strcmp0(device, "generic-sdhci")) {
        return &machine->sdhci.obj;
    }

    fprintf(stderr, "%s not present in arm/raspi2b\n", device);
    g_assert_not_reached();
}

static void *qos_create_machine_arm_raspi2(QTestState *qts)
{
    QRaspi2Machine *machine = g_new0(QRaspi2Machine, 1);

    alloc_init(&machine->alloc, ...);

    /* Get node(s) contained inside (CONTAINS) */
    machine->obj.get_device = raspi2_get_device;

    /* Get node(s) produced (PRODUCES) */
    machine->obj.get_driver = raspi2_get_driver;

    /* free the object */
    machine->obj.destructor = raspi2_destructor;
    qos_init_sdhci_mm(&machine->sdhci, ...);
    return &machine->obj;
}

static void raspi2_register_nodes(void)
{
    /* arm/raspi2b --contains--> generic-sdhci */
    qos_node_create_machine("arm/raspi2b",
                            qos_create_machine_arm_raspi2);
    qos_node_contains("arm/raspi2b", "generic-sdhci", NULL);
}

libqos_init(raspi2_register_nodes);
```

x86_64/pc machine, simplified from `tests/qtest/libqos/x86_64_pc-machine.c`:

```
#include "qgraph.h"

struct i440FX_pcihost {
    QOSGraphObject obj;
    QPCIBusPC pci;
};
```

(continues on next page)

(continued from previous page)

```
struct QX86PCMachine {
    QOSGraphObject obj;
    QGuestAllocator alloc;
    i440FX_pcihost bridge;
};

/* i440FX_pcihost */

static QOSGraphObject *i440FX_host_get_device(void *obj,
                                              const char *device)
{
    i440FX_pcihost *host = obj;
    if (!g_strcmp0(device, "pci-bus-pc")) {
        return &host->pci.obj;
    }
    fprintf(stderr, "%s not present in i440FX-pcihost\n", device);
    g_assert_not_reached();
}

/* x86_64/pc machine */

static void *pc_get_driver(void *object, const char *interface)
{
    QX86PCMachine *machine = object;
    if (!g_strcmp0(interface, "memory")) {
        return &machine->alloc;
    }

    fprintf(stderr, "%s not present in x86_64/pc\n", interface);
    g_assert_not_reached();
}

static QOSGraphObject *pc_get_device(void *obj, const char *device)
{
    QX86PCMachine *machine = obj;
    if (!g_strcmp0(device, "i440FX-pcihost")) {
        return &machine->bridge.obj;
    }

    fprintf(stderr, "%s not present in x86_64/pc\n", device);
    g_assert_not_reached();
}

static void *qos_create_machine_pc(QTestState *qts)
{
    QX86PCMachine *machine = g_new0(QX86PCMachine, 1);

    /* Get node(s) contained inside (CONTAINS) */
    machine->obj.get_device = pc_get_device;

    /* Get node(s) produced (PRODUCES) */
```

(continues on next page)

(continued from previous page)

```
    machine->obj.get_driver = pc_get_driver;

    /* free the object */
    machine->obj.destructor = pc_destructor;
    pc_alloc_init(&machine->alloc, qts, ALLOC_NO_FLAGS);

    /* Get node(s) contained inside (CONTAINS) */
    machine->bridge.obj.get_device = i440FX_host_get_device;

    return &machine->obj;
}

static void pc_machine_register_nodes(void)
{
    /* x86_64/pc --contains--> 1440FX-pcihost --contains-->
     * pci-bus-pc [--produces--> pci-bus (in pci.h)] */
    qos_node_create_machine("x86_64/pc", qos_create_machine_pc);
    qos_node_contains("x86_64/pc", "i440FX-pcihost", NULL);

    /* contained drivers don't need a constructor,
     * they will be init by the parent */
    qos_node_create_driver("i440FX-pcihost", NULL);
    qos_node_contains("i440FX-pcihost", "pci-bus-pc", NULL);
}

libqos_init(pc_machine_register_nodes);
```

`sdhci` taken from `tests/qtest/libqos/sdhci.c`:

```
/* Interface node, offers the sdhci API */
struct QSDHCI {
    uint16_t (*readw)(QSDHCI *s, uint32_t reg);
    uint64_t (*readq)(QSDHCI *s, uint32_t reg);
    void (*writeq)(QSDHCI *s, uint32_t reg, uint64_t val);
    /* other fields */
};

/* Memory Mapped implementation of QSDHCI */
struct QSDHCI_MemoryMapped {
    QOSGraphObject obj;
    QSDHCI sdhci;
    /* other driver-specific fields */
};

/* PCI implementation of QSDHCI */
struct QSDHCI_PCI {
    QOSGraphObject obj;
    QSDHCI sdhci;
    /* other driver-specific fields */
};

/* Memory mapped implementation of QSDHCI */
```

(continues on next page)

(continued from previous page)

```
static void *sdhci_mm_get_driver(void *obj, const char *interface)
{
    QSDHCI_MemoryMapped *smm = obj;
    if (!g_strcmp0(interface, "sdhci")) {
        return &smm->sdhci;
    }
    fprintf(stderr, "%s not present in generic-sdhci\n", interface);
    g_assert_not_reached();
}

void qos_init_sdhci_mm(QSDHCI_MemoryMapped *sdhci, QTestState *qts,
                       uint32_t addr, QSDHCIProperties *common)
{
    /* Get node contained inside (CONTAINS) */
    sdhci->obj.get_driver = sdhci_mm_get_driver;

    /* SDHCI interface API */
    sdhci->sdhci.readw = sdhci_mm_readw;
    sdhci->sdhci.readq = sdhci_mm_readq;
    sdhci->sdhci.writeq = sdhci_mm_writeq;
    sdhci->qts = qts;
}

/* PCI implementation of QSDHCI */

static void *sdhci_pci_get_driver(void *object,
                                  const char *interface)
{
    QSDHCI_PCI *spci = object;
    if (!g_strcmp0(interface, "sdhci")) {
        return &spci->sdhci;
    }

    fprintf(stderr, "%s not present in sdhci-pci\n", interface);
    g_assert_not_reached();
}

static void *sdhci_pci_create(void *pci_bus,
                              QGuestAllocator *alloc,
                              void *addr)
{
    QSDHCI_PCI *spci = g_new0(QSDHCI_PCI, 1);
    QPCIBus *bus = pci_bus;
    uint64_t barsize;

    qpci_device_init(&spci->dev, bus, addr);

    /* SDHCI interface API */
    spci->sdhci.readw = sdhci_pci_readw;
    spci->sdhci.readq = sdhci_pci_readq;
    spci->sdhci.writeq = sdhci_pci_writeq;
```

(continues on next page)

(continued from previous page)

```
    /* Get node(s) produced (PRODUCES) */
    spci->obj.get_driver = sdhci_pci_get_driver;

    spci->obj.start_hw = sdhci_pci_start_hw;
    spci->obj.destructor = sdhci_destructor;
    return &spci->obj;
}

static void qsdhci_register_nodes(void)
{
    QOSGraphEdgeOptions opts = {
        .extra_device_opts = "addr=04.0",
    };

    /* generic-sdhci */
    /* generic-sdhci --produces--> sdhci */
    qos_node_create_driver("generic-sdhci", NULL);
    qos_node_produces("generic-sdhci", "sdhci");

    /* sdhci-pci */
    /* sdhci-pci --produces--> sdhci
     * sdhci-pci --consumes--> pci-bus */
    qos_node_create_driver("sdhci-pci", sdhci_pci_create);
    qos_node_produces("sdhci-pci", "sdhci");
    qos_node_consumes("sdhci-pci", "pci-bus", &opts);
}

libqos_init(qsdhci_register_nodes);
```

In the above example, all possible types of relations are created:

```
x86_64/pc --contains--> 1440FX-pcihost --contains--> pci-bus-pc
                                                       |
            sdhci-pci --consumes--> pci-bus <--produces--+
               |
               +--produces--+
                            |
                            v
                          sdhci
                            ^
                            |
                            +--produces-- +
                                          |
            arm/raspi2b --contains--> generic-sdhci
```

or inverting the consumes edge in consumed_by:

```
x86_64/pc --contains--> 1440FX-pcihost --contains--> pci-bus-pc
                                                       |
          sdhci-pci <--consumed by-- pci-bus <--produces--+
              |
```

(continues on next page)

(continued from previous page)

```
              +--produces--+
                           |
                           v
                         sdhci
                           ^
                           |
                           +--produces-- +
                                         |
          arm/raspi2b --contains--> generic-sdhci
```

Adding a new test

Given the above setup, adding a new test is very simple. `sdhci-test`, taken from `tests/qtest/sdhci-test.c`:

```
static void check_capab_sdma(QSDHCI *s, bool supported)
{
    uint64_t capab, capab_sdma;

    capab = s->readq(s, SDHC_CAPAB);
    capab_sdma = FIELD_EX64(capab, SDHC_CAPAB, SDMA);
    g_assert_cmpuint(capab_sdma, ==, supported);
}

static void test_registers(void *obj, void *data,
                           QGuestAllocator *alloc)
{
    QSDHCI *s = obj;

    /* example test */
    check_capab_sdma(s, s->props.capab.sdma);
}

static void register_sdhci_test(void)
{
    /* sdhci-test --consumes--> sdhci */
    qos_add_test("registers", "sdhci", test_registers, NULL);
}

libqos_init(register_sdhci_test);
```

Here a new test is created, consuming `sdhci` interface node and creating a valid path from both machines to a test. Final graph will be like this:

```
x86_64/pc --contains--> 1440FX-pcihost --contains--> pci-bus-pc
                                                       |
             sdhci-pci --consumes--> pci-bus <--produces--+
                |
                +--produces--+
                             |
                             v
                           sdhci <--consumes-- sdhci-test
```

(continues on next page)

(continued from previous page)

```
                            ^
                            |
                            +--produces-- +
                                          |
            arm/raspi2b --contains--> generic-sdhci
```

or inverting the consumes edge in consumed_by:

```
x86_64/pc --contains--> 1440FX-pcihost --contains--> pci-bus-pc
                                                       |
          sdhci-pci <--consumed by-- pci-bus <--produces--+
              |
              +--produces--+
                           |
                           v
                          sdhci --consumed by--> sdhci-test
                           ^
                           |
                           +--produces-- +
                                         |
          arm/raspi2b --contains--> generic-sdhci
```

Assuming there the binary is `QTEST_QEMU_BINARY=./qemu-system-x86_64` a valid test path will be: `/x86_64/pc/1440FX-pcihost/pci-bus-pc/pci-bus/sdhci-pc/sdhci/sdhci-test`

and for the binary `QTEST_QEMU_BINARY=./qemu-system-arm`:

`/arm/raspi2b/generic-sdhci/sdhci/sdhci-test`

Additional examples are also in `test-qgraph.c`

Qgraph API reference

struct **QOSGraphEdgeOptions**

Edge options to be passed to the contains/consumes *_args function.

Definition

```
struct QOSGraphEdgeOptions {
  void *arg;
  uint32_t size_arg;
  const char *extra_device_opts;
  const char *before_cmd_line;
  const char *after_cmd_line;
  const char *edge_name;
};
```

Members

arg

optional arg that will be used by dest edge

size_arg

arg size that will be used by dest edge

extra_device_opts
optional additional command line for dest edge, used to add additional attributes *after* the node command line, the framework automatically prepends "," to this argument.

before_cmd_line
optional additional command line for dest edge, used to add additional attributes *before* the node command line, usually other non-node represented commands, like "-fdsev synt"

after_cmd_line
optional extra command line to be added after the device command. This option is used to add other devices command line that depend on current node. Automatically prepends " " to this argument

edge_name
optional edge to differentiate multiple devices with same node name

struct **QOSGraphTestOptions**

Test options to be passed to the test functions.

Definition

```
struct QOSGraphTestOptions {
  QOSGraphEdgeOptions edge;
  void *arg;
  QOSBeforeTest before;
  bool subprocess;
};
```

Members

edge
edge arguments that will be used by test. Note that test *does not* use edge_name, and uses instead arg and size_arg as data arg for its test function.

arg
if **before** is non-NULL, pass **arg** there. Otherwise pass it to the test function.

before
executed before the test. Used to add additional parameters to the command line and modify the argument to the test function.

subprocess
run the test in a subprocess.

struct **QOSGraphObject**

Each driver, test or machine of this framework will have a QOSGraphObject as first field.

Definition

```
struct QOSGraphObject {
  QOSGetDriver get_driver;
  QOSGetDevice get_device;
  QOSStartFunct start_hw;
  QOSDestructorFunc destructor;
  GDestroyNotify free;
};
```

Members

get_driver
see **get_device**

get_device
: Once a machine-to-test path has been found, the framework traverses it again and allocates all the nodes, using the provided constructor. To satisfy their relations, i.e. for produces or contains, where a struct constructor needs an external parameter represented by the previous node, the framework will call **get_device** (for contains) or **get_driver** (for produces), depending on the edge type, passing them the name of the next node to be taken and getting from them the corresponding pointer to the actual structure of the next node to be used in the path.

start_hw
: This function is executed after all the path objects have been allocated, but before the test is run. It starts the hw, setting the initial configurations (*_device_enable) and making it ready for the test.

destructor
: Opposite to the node constructor, destroys the object. This function is called after the test has been executed, and performs a complete cleanup of each node allocated field. In case no constructor is provided, no destructor will be called.

free
: free the memory associated to the QOSGraphObject and its contained children

Description

This set of functions offered by QOSGraphObject are executed in different stages of the framework:

void **qos_graph_init**(void)

: initialize the framework, creates two hash tables: one for the nodes and another for the edges.

Parameters

void
: no arguments

void **qos_graph_destroy**(void)

: deallocates all the hash tables, freeing all nodes and edges.

Parameters

void
: no arguments

void **qos_node_destroy**(void *key)

: removes and frees a node from the nodes hash table.

Parameters

void *key
: Name of the node

void **qos_edge_destroy**(void *key)

: removes and frees an edge from the edges hash table.

Parameters

void *key
: Name of the node

void **qos_add_test**(const char *name, const char *interface, QOSTestFunc test_func, *QOSGraphTestOptions* *opts)

: adds a test node **name** to the nodes hash table.

Parameters

const char *name
: Name of the test

const char *interface
Name of the interface node it consumes

QOSTestFunc test_func
Actual test to perform

QOSGraphTestOptions *opts
Facultative options (see QOSGraphTestOptions)

Description

The test will consume a **interface** node, and once the graph walking algorithm has found it, the **test_func** will be executed. It also has the possibility to add an optional **opts** (see QOSGraphTestOptions).

For tests, opts->edge.arg and size_arg represent the arg to pass to **test_func**

void **qos_node_create_machine**(const char *name, QOSCreateMachineFunc function)
creates the machine **name** and adds it to the node hash table.

Parameters

const char *name
Name of the machine

QOSCreateMachineFunc function
Machine constructor

Description

This node will be of type QNODE_MACHINE and have **function** as constructor

void **qos_node_create_machine_args**(const char *name, QOSCreateMachineFunc function, const char *opts)
same as qos_node_create_machine, but with the possibility to add an optional ", **opts**" after -M machine command line.

Parameters

const char *name
Name of the machine

QOSCreateMachineFunc function
Machine constructor

const char *opts
Optional additional command line

void **qos_node_create_driver**(const char *name, QOSCreateDriverFunc function)
creates the driver **name** and adds it to the node hash table.

Parameters

const char *name
Name of the driver

QOSCreateDriverFunc function
Driver constructor

Description

This node will be of type QNODE_DRIVER and have **function** as constructor

void **qos_node_create_driver_named**(const char *name, const char *qemu_name, QOSCreateDriverFunc function)
behaves as qos_node_create_driver() with the extension of allowing to specify a different node name vs. associated QEMU device name.

Parameters

const char *name
Custom, unique name of the node to be created

const char *qemu_name
Actual (official) QEMU driver name the node shall be associated with

QOSCreateDriverFunc function
Driver constructor

Description

Use this function instead of qos_node_create_driver() if you need to create several instances of the same QEMU device. You are free to choose a custom node name, however the chosen node name must always be unique.

void **qos_node_contains**(const char *container, const char *contained, *QOSGraphEdgeOptions* *opts, ...)

creates one or more edges of type QEDGE_CONTAINS and adds them to the edge list mapped to **container** in the edge hash table.

Parameters

const char *container
Source node that "contains"

const char *contained
Destination node that "is contained"

QOSGraphEdgeOptions *opts
Facultative options (see `QOSGraphEdgeOptions`)

...
variable arguments

Description

The edges will have **container** as source and **contained** as destination.

If **opts** is NULL, a single edge will be added with no options. If **opts** is non-NULL, the arguments after **contained** represent a NULL-terminated list of `QOSGraphEdgeOptions` structs, and an edge will be added for each of them.

This function can be useful when there are multiple devices with the same node name contained in a machine/other node

For example, if `arm/raspi2b` contains 2 `generic-sdhci` devices, the right commands will be:

```
qos_node_create_machine("arm/raspi2b");
qos_node_create_driver("generic-sdhci", constructor);
// assume rest of the fields are set NULL
QOSGraphEdgeOptions op1 = { .edge_name = "emmc" };
QOSGraphEdgeOptions op2 = { .edge_name = "sdcard" };
qos_node_contains("arm/raspi2b", "generic-sdhci", &op1, &op2, NULL);
```

Of course this also requires that the **container**'s get_device function should implement a case for "emmc" and "sdcard".

For contains, op1.arg and op1.size_arg represent the arg to pass to **contained** constructor to properly initialize it.

void **qos_node_produces**(const char *producer, const char *interface)

creates an edge of type QEDGE_PRODUCES and adds it to the edge list mapped to **producer** in the edge hash table.

Parameters

const char *producer
Source node that "produces"

const char *interface
Interface node that "is produced"

Description

This edge will have **producer** as source and **interface** as destination.

void **qos_node_consumes**(const char *consumer, const char *interface, *QOSGraphEdgeOptions* *opts)

creates an edge of type QEDGE_CONSUMED_BY and adds it to the edge list mapped to **interface** in the edge hash table.

Parameters

const char *consumer
Node that "consumes"

const char *interface
Interface node that "is consumed by"

QOSGraphEdgeOptions *opts
Facultative options (see `QOSGraphEdgeOptions`)

Description

This edge will have **interface** as source and **consumer** as destination. It also has the possibility to add an optional **opts** (see `QOSGraphEdgeOptions`)

void **qos_invalidate_command_line**(void)

invalidates current command line, so that qgraph framework cannot try to cache the current command line and forces QEMU to restart.

Parameters

void
no arguments

const char ***qos_get_current_command_line**(void)

return the command line required by the machine and driver objects. This is the same string that was passed to the test's "before" callback, if any.

Parameters

void
no arguments

void ***qos_allocate_objects**(QTestState *qts, QGuestAllocator **p_alloc)

Parameters

QTestState *qts
The `QTestState` that will be referred to by the machine object.

QGuestAllocator **p_alloc
Where to store the allocator for the machine object, or `NULL`.

Description

Allocate driver objects for the current test path, but relative to the QTestState **qts**.

Returns a test object just like the one that was passed to the test function, but relative to **qts**.

void **qos_object_destroy**(*QOSGraphObject* *obj)

calls the destructor for **obj**

Parameters

QOSGraphObject *obj
A *QOSGraphObject* to destroy

void **qos_object_queue_destroy**(*QOSGraphObject* *obj)

queue the destructor for **obj** so that it is called at the end of the test

Parameters

QOSGraphObject *obj
A *QOSGraphObject* to destroy

void **qos_object_start_hw**(*QOSGraphObject* *obj)

calls the start_hw function for **obj**

Parameters

QOSGraphObject *obj
A *QOSGraphObject* containing the start_hw function

QOSGraphObject ***qos_machine_new**(QOSGraphNode *node, QTestState *qts)

instantiate a new machine node

Parameters

QOSGraphNode *node
Machine node to be instantiated

QTestState *qts
A `QTestState` that will be referred to by the machine object.

Description

Returns a machine object.

QOSGraphObject ***qos_driver_new**(QOSGraphNode *node, *QOSGraphObject* *parent, QGuestAllocator *alloc, void *arg)

instantiate a new driver node

Parameters

QOSGraphNode *node
A driver node to be instantiated

QOSGraphObject *parent
A *QOSGraphObject* to be consumed by the new driver node

QGuestAllocator *alloc
An allocator to be used by the new driver node.

void *arg
The argument for the consumed-by edge to **node**.

Description

Calls the constructor for the driver object.

void **qos_dump_graph**(void)

prints all currently existing nodes and edges to stdout. Just for debugging purposes.

Parameters

void

no arguments

Description

All qtests add themselves to the overall qos graph by calling qgraph functions that add device nodes and edges between the individual graph nodes for tests. As the actual graph is assmbled at runtime by the qos subsystem, it is sometimes not obvious how the overall graph looks like. E.g. when writing new tests it may happen that those new tests are simply ignored by the qtest framework.

This function allows to identify problems in the created qgraph. Keep in mind: only tests with a path down from the actual test case node (leaf) up to the graph's root node are actually executed by the qtest framework. And the qtest framework uses QMP to automatically check which QEMU drivers are actually currently available, and accordingly qos marks certain paths as 'unavailable' in such cases (e.g. when QEMU was compiled without support for a certain feature).

QTest is a device emulation testing framework. It can be very useful to test device models; it could also control certain aspects of QEMU (such as virtual clock stepping), with a special purpose "qtest" protocol. Refer to *QTest Protocol* for more details of the protocol.

QTest cases can be executed with

```
make check-qtest
```

The QTest library is implemented by `tests/qtest/libqtest.c` and the API is defined in `tests/qtest/libqtest.h`.

Consider adding a new QTest case when you are introducing a new virtual hardware, or extending one if you are adding functionalities to an existing virtual device.

On top of libqtest, a higher level library, `libqos`, was created to encapsulate common tasks of device drivers, such as memory management and communicating with system buses or devices. Many virtual device tests use libqos instead of directly calling into libqtest. Libqos also offers the Qgraph API to increase each test coverage and automate QEMU command line arguments and devices setup. Refer to *Qtest Driver Framework* for Qgraph explanation and API.

Steps to add a new QTest case are:

1. Create a new source file for the test. (More than one file can be added as necessary.) For example, `tests/qtest/foo-test.c`.
2. Write the test code with the glib and libqtest/libqos API. See also existing tests and the library headers for reference.
3. Register the new test in `tests/qtest/meson.build`. Add the test executable name to an appropriate `qtests_*` variable. There is one variable per architecture, plus `qtests_generic` for tests that can be run for all architectures. For example:

   ```
   qtests_generic = [
     ...
     'foo-test',
     ...
   ]
   ```

4. If the test has more than one source file or needs to be linked with any dependency other than `qemuutil` and `qos`, list them in the `qtests` dictionary. For example a test that needs to use the `QIO` library will have an entry like:

```
{
  ...
  'foo-test': [io],
  ...
}
```

Debugging a QTest failure is slightly harder than the unit test because the tests look up QEMU program names in the environment variables, such as `QTEST_QEMU_BINARY` and `QTEST_QEMU_IMG`, and also because it is not easy to attach gdb to the QEMU process spawned from the test. But manual invoking and using gdb on the test is still simple to do: find out the actual command from the output of

```
make check-qtest V=1
```

which you can run manually.

QTest Protocol

Line based protocol, request/response based. Server can send async messages so clients should always handle many async messages before the response comes in.

Valid requests

Clock management:

The qtest client is completely in charge of the QEMU_CLOCK_VIRTUAL. qtest commands let you adjust the value of the clock (monotonically). All the commands return the current value of the clock in nanoseconds.

```
> clock_step
< OK VALUE
```

Advance the clock to the next deadline. Useful when waiting for asynchronous events.

```
> clock_step NS
< OK VALUE
```

Advance the clock by NS nanoseconds.

```
> clock_set NS
< OK VALUE
```

Advance the clock to NS nanoseconds (do nothing if it's already past).

PIO and memory access:

```
> outb ADDR VALUE
< OK
```

```
> outw ADDR VALUE
< OK
```

```
> outl ADDR VALUE
< OK
```

```
> inb ADDR
< OK VALUE
```

```
> inw ADDR
< OK VALUE
```

```
> inl ADDR
< OK VALUE
```

```
> writeb ADDR VALUE
< OK
```

```
> writew ADDR VALUE
< OK
```

```
> writel ADDR VALUE
< OK
```

```
> writeq ADDR VALUE
< OK
```

```
> readb ADDR
< OK VALUE
```

```
> readw ADDR
< OK VALUE
```

```
> readl ADDR
< OK VALUE
```

```
> readq ADDR
< OK VALUE
```

```
> read ADDR SIZE
< OK DATA
```

```
> write ADDR SIZE DATA
< OK
```

```
> b64read ADDR SIZE
< OK B64_DATA
```

```
> b64write ADDR SIZE B64_DATA
< OK
```

```
> memset ADDR SIZE VALUE
< OK
```

ADDR, SIZE, VALUE are all integers parsed with strtoul() with a base of 0. For 'memset' a zero size is permitted and does nothing.

DATA is an arbitrarily long hex number prefixed with '0x'. If it's smaller than the expected size, the value will be zero filled at the end of the data sequence.

B64_DATA is an arbitrarily long base64 encoded string. If the sizes do not match, the data will be truncated.

IRQ management:

```
> irq_intercept_in QOM-PATH
< OK
```

```
> irq_intercept_out QOM-PATH
< OK
```

Attach to the gpio-in (resp. gpio-out) pins exported by the device at QOM-PATH. When the pin is triggered, one of the following async messages will be printed to the qtest stream:

```
IRQ raise NUM
IRQ lower NUM
```

where NUM is an IRQ number. For the PC, interrupts can be intercepted simply with "irq_intercept_in ioapic" (note that IRQ0 comes out with NUM=0 even though it is remapped to GSI 2).

Setting interrupt level:

```
> set_irq_in QOM-PATH NAME NUM LEVEL
< OK
```

where NAME is the name of the irq/gpio list, NUM is an IRQ number and LEVEL is an signed integer IRQ level.

Forcibly set the given interrupt pin to the given level.

libqtest API reference

QTestState *qtest_initf(const char *fmt, ...)

Parameters

const char *fmt
Format for creating other arguments to pass to QEMU, formatted like sprintf().

...
variable arguments

Description

Convenience wrapper around qtest_init().

Return

`QTestState` instance.

QTestState *qtest_vinitf(const char *fmt, va_list ap)

Parameters

const char *fmt
Format for creating other arguments to pass to QEMU, formatted like vsprintf().

va_list ap
Format arguments.

Description

Convenience wrapper around qtest_init().

Return

`QTestState` instance.

QTestState *qtest_init(const char *extra_args)

Parameters

const char *extra_args
other arguments to pass to QEMU. CAUTION: these arguments are subject to word splitting and shell evaluation.

Return

`QTestState` instance.

QTestState *qtest_init_with_env(const char *var, const char *extra_args)

Parameters

const char *var
Environment variable from where to take the QEMU binary

const char *extra_args
Other arguments to pass to QEMU. CAUTION: these arguments are subject to word splitting and shell evaluation.

Description

Like qtest_init(), but use a different environment variable for the QEMU binary.

Return

`QTestState` instance.

QTestState *qtest_init_without_qmp_handshake(const char *extra_args)

Parameters

const char *extra_args
other arguments to pass to QEMU. CAUTION: these arguments are subject to word splitting and shell evaluation.

Return

`QTestState` instance.

QTestState *qtest_init_with_serial(const char *extra_args, int *sock_fd)

Parameters

const char *extra_args
other arguments to pass to QEMU. CAUTION: these arguments are subject to word splitting and shell evaluation.

int *sock_fd
pointer to store the socket file descriptor for connection with serial.

Return

`QTestState` instance.

void qtest_wait_qemu(QTestState *s)

Parameters

QTestState *s
`QTestState` instance to operate on.

Description

Wait for the QEMU process to terminate. It is safe to call this function multiple times.

void qtest_kill_qemu(QTestState *s)

Parameters

QTestState *s
`QTestState` instance to operate on.

Description

Kill the QEMU process and wait for it to terminate. It is safe to call this function multiple times. Normally qtest_quit() is used instead because it also frees QTestState. Use qtest_kill_qemu() when you just want to kill QEMU and qtest_quit() will be called later.

void qtest_quit(QTestState *s)

Parameters

QTestState *s
`QTestState` instance to operate on.

Description

Shut down the QEMU process associated to **s**.

QDict *qtest_qmp_fds(QTestState *s, int *fds, size_t fds_num, const char *fmt, ...)

Parameters

QTestState *s
`QTestState` instance to operate on.

int *fds
array of file descriptors

size_t fds_num
number of elements in **fds**

const char *fmt
QMP message to send to qemu, formatted like qobject_from_jsonf_nofail(). See parse_interpolation() for what's supported after '%'.

...
variable arguments

Description

Sends a QMP message to QEMU with fds and returns the response.

QDict ***qtest_qmp**(QTestState *s, const char *fmt, ...)

Parameters

QTestState *s
`QTestState` instance to operate on.

const char *fmt
QMP message to send to qemu, formatted like qobject_from_jsonf_nofail(). See parse_interpolation() for what's supported after '%'.

...
variable arguments

Description

Sends a QMP message to QEMU and returns the response.

void **qtest_qmp_send**(QTestState *s, const char *fmt, ...)

Parameters

QTestState *s
`QTestState` instance to operate on.

const char *fmt
QMP message to send to qemu, formatted like qobject_from_jsonf_nofail(). See parse_interpolation() for what's supported after '%'.

...
variable arguments

Description

Sends a QMP message to QEMU and leaves the response in the stream.

void **qtest_qmp_send_raw**(QTestState *s, const char *fmt, ...)

Parameters

QTestState *s
`QTestState` instance to operate on.

const char *fmt
text to send, formatted like sprintf()

...
variable arguments

Description

Sends text to the QMP monitor verbatim. Need not be valid JSON; this is useful for negative tests.

int **qtest_socket_server**(const char *socket_path)

Parameters

const char *socket_path
the UNIX domain socket path

Description

Create and return a listen socket file descriptor, or abort on failure.

QDict ***qtest_vqmp_fds**(QTestState *s, int *fds, size_t fds_num, const char *fmt, va_list ap)

Parameters

QTestState *s
`QTestState` instance to operate on.

int *fds
array of file descriptors

size_t fds_num
number of elements in **fds**

const char *fmt
QMP message to send to QEMU, formatted like qobject_from_jsonf_nofail(). See parse_interpolation() for what's supported after '%'.

va_list ap
QMP message arguments

Description

Sends a QMP message to QEMU with fds and returns the response.

QDict ***qtest_vqmp**(QTestState *s, const char *fmt, va_list ap)

Parameters

QTestState *s
`QTestState` instance to operate on.

const char *fmt
QMP message to send to QEMU, formatted like qobject_from_jsonf_nofail(). See parse_interpolation() for what's supported after '%'.

va_list ap
QMP message arguments

Description

Sends a QMP message to QEMU and returns the response.

void **qtest_qmp_vsend_fds**(QTestState *s, int *fds, size_t fds_num, const char *fmt, va_list ap)

Parameters

QTestState *s
`QTestState` instance to operate on.

int *fds
array of file descriptors

size_t fds_num
number of elements in **fds**

const char *fmt
QMP message to send to QEMU, formatted like qobject_from_jsonf_nofail(). See parse_interpolation() for what's supported after '%'.

va_list ap
QMP message arguments

Description

Sends a QMP message to QEMU and leaves the response in the stream.

void **qtest_qmp_vsend**(QTestState *s, const char *fmt, va_list ap)

Parameters

QTestState *s
QTestState instance to operate on.

const char *fmt
QMP message to send to QEMU, formatted like qobject_from_jsonf_nofail(). See parse_interpolation() for what's supported after '%'.

va_list ap
QMP message arguments

Description

Sends a QMP message to QEMU and leaves the response in the stream.

QDict *__qtest_qmp_receive_dict__(QTestState *s)

Parameters

QTestState *s
QTestState instance to operate on.

Description

Reads a QMP message from QEMU and returns the response.

QDict *__qtest_qmp_receive__(QTestState *s)

Parameters

QTestState *s
QTestState instance to operate on.

Description

Reads a QMP message from QEMU and returns the response.

If a callback is registered with qtest_qmp_set_event_callback, it will be invoked for every event seen, otherwise events will be buffered until a call to one of the qtest_qmp_eventwait family of functions.

void **qtest_qmp_set_event_callback**(QTestState *s, QTestQMPEventCallback cb, void *opaque)

Parameters

QTestState *s
QTestSTate instance to operate on

QTestQMPEventCallback cb
callback to invoke for events

void *opaque
data to pass to **cb**

Description

Register a callback to be invoked whenever an event arrives

void **qtest_qmp_eventwait**(QTestState *s, const char *event)

Parameters

QTestState *s
QTestState instance to operate on.

const char *event
event to wait for.

Description

Continuously polls for QMP responses until it receives the desired event.

Any callback registered with qtest_qmp_set_event_callback will be invoked for every event seen.

QDict ***qtest_qmp_eventwait_ref**(QTestState *s, const char *event)

Parameters

QTestState *s
QTestState instance to operate on.

const char *event
event to wait for.

Description

Continuously polls for QMP responses until it receives the desired event.

Any callback registered with qtest_qmp_set_event_callback will be invoked for every event seen.

Returns a copy of the event for further investigation.

QDict ***qtest_qmp_event_ref**(QTestState *s, const char *event)

Parameters

QTestState *s
QTestState instance to operate on.

const char *event
event to return.

Description

Removes non-matching events from the buffer that was set by qtest_qmp_receive, until an event bearing the given name is found, and returns it. If no event matches, clears the buffer and returns NULL.

char ***qtest_hmp**(QTestState *s, const char *fmt, ...)

Parameters

QTestState *s
QTestState instance to operate on.

const char *fmt
HMP command to send to QEMU, formats arguments like sprintf().

`...`
: variable arguments

Description

Send HMP command to QEMU via QMP's human-monitor-command. QMP events are discarded.

Return

the command's output. The caller should g_free() it.

char *qtest_vhmp(QTestState *s, const char *fmt, va_list ap)

Parameters

`QTestState *s`
: `QTestState` instance to operate on.

`const char *fmt`
: HMP command to send to QEMU, formats arguments like vsprintf().

`va_list ap`
: HMP command arguments

Description

Send HMP command to QEMU via QMP's human-monitor-command. QMP events are discarded.

Return

the command's output. The caller should g_free() it.

bool qtest_get_irq(QTestState *s, int num)

Parameters

`QTestState *s`
: `QTestState` instance to operate on.

`int num`
: Interrupt to observe.

Return

The level of the **num** interrupt.

void qtest_irq_intercept_in(QTestState *s, const char *string)

Parameters

`QTestState *s`
: `QTestState` instance to operate on.

`const char *string`
: QOM path of a device.

Description

Associate qtest irqs with the GPIO-in pins of the device whose path is specified by **string**.

void qtest_irq_intercept_out(QTestState *s, const char *string)

Parameters

`QTestState *s`
: `QTestState` instance to operate on.

const char *string
QOM path of a device.

Description

Associate qtest irqs with the GPIO-out pins of the device whose path is specified by **string**.

void **qtest_irq_intercept_out_named**(QTestState *s, const char *qom_path, const char *name)

Parameters

QTestState *s
QTestState instance to operate on.

const char *qom_path
QOM path of a device.

const char *name
Name of the GPIO out pin

Description

Associate a qtest irq with the named GPIO-out pin of the device whose path is specified by **string** and whose name is **name**.

void **qtest_set_irq_in**(QTestState *s, const char *string, const char *name, int irq, int level)

Parameters

QTestState *s
QTestState instance to operate on.

const char *string
QOM path of a device

const char *name
IRQ name

int irq
IRQ number

int level
IRQ level

Description

Force given device/irq GPIO-in pin to the given level.

void **qtest_outb**(QTestState *s, uint16_t addr, uint8_t value)

Parameters

QTestState *s
QTestState instance to operate on.

uint16_t addr
I/O port to write to.

uint8_t value
Value being written.

Description

Write an 8-bit value to an I/O port.

void **qtest_outw**(QTestState *s, uint16_t addr, uint16_t value)

Parameters

QTestState *s
QTestState instance to operate on.

uint16_t addr
I/O port to write to.

uint16_t value
Value being written.

Description

Write a 16-bit value to an I/O port.

void **qtest_outl**(QTestState *s, uint16_t addr, uint32_t value)

Parameters

QTestState *s
QTestState instance to operate on.

uint16_t addr
I/O port to write to.

uint32_t value
Value being written.

Description

Write a 32-bit value to an I/O port.

uint8_t **qtest_inb**(QTestState *s, uint16_t addr)

Parameters

QTestState *s
QTestState instance to operate on.

uint16_t addr
I/O port to read from.

Description

Returns an 8-bit value from an I/O port.

uint16_t **qtest_inw**(QTestState *s, uint16_t addr)

Parameters

QTestState *s
QTestState instance to operate on.

uint16_t addr
I/O port to read from.

Description

Returns a 16-bit value from an I/O port.

uint32_t **qtest_inl**(QTestState *s, uint16_t addr)

Parameters

QTestState *s
QTestState instance to operate on.

uint16_t addr
I/O port to read from.

Description

Returns a 32-bit value from an I/O port.

void **qtest_writeb**(QTestState *s, uint64_t addr, uint8_t value)

Parameters

QTestState *s
QTestState instance to operate on.

uint64_t addr
Guest address to write to.

uint8_t value
Value being written.

Description

Writes an 8-bit value to memory.

void **qtest_writew**(QTestState *s, uint64_t addr, uint16_t value)

Parameters

QTestState *s
QTestState instance to operate on.

uint64_t addr
Guest address to write to.

uint16_t value
Value being written.

Description

Writes a 16-bit value to memory.

void **qtest_writel**(QTestState *s, uint64_t addr, uint32_t value)

Parameters

QTestState *s
QTestState instance to operate on.

uint64_t addr
Guest address to write to.

uint32_t value
Value being written.

Description

Writes a 32-bit value to memory.

void **qtest_writeq**(QTestState *s, uint64_t addr, uint64_t value)

Parameters

QTestState *s
QTestState instance to operate on.

uint64_t addr
Guest address to write to.

uint64_t value
Value being written.

Description

Writes a 64-bit value to memory.

uint8_t **qtest_readb**(QTestState *s, uint64_t addr)

Parameters

QTestState *s
QTestState instance to operate on.

uint64_t addr
Guest address to read from.

Description

Reads an 8-bit value from memory.

Return

Value read.

uint16_t **qtest_readw**(QTestState *s, uint64_t addr)

Parameters

QTestState *s
QTestState instance to operate on.

uint64_t addr
Guest address to read from.

Description

Reads a 16-bit value from memory.

Return

Value read.

uint32_t **qtest_readl**(QTestState *s, uint64_t addr)

Parameters

QTestState *s
QTestState instance to operate on.

uint64_t addr
Guest address to read from.

Description

Reads a 32-bit value from memory.

Return

Value read.

uint64_t **qtest_readq**(QTestState *s, uint64_t addr)

Parameters

QTestState *s
: QTestState instance to operate on.

uint64_t addr
: Guest address to read from.

Description

Reads a 64-bit value from memory.

Return

Value read.

void **qtest_memread**(QTestState *s, uint64_t addr, void *data, size_t size)

Parameters

QTestState *s
: QTestState instance to operate on.

uint64_t addr
: Guest address to read from.

void *data
: Pointer to where memory contents will be stored.

size_t size
: Number of bytes to read.

Description

Read guest memory into a buffer.

uint64_t **qtest_rtas_call**(QTestState *s, const char *name, uint32_t nargs, uint64_t args, uint32_t nret, uint64_t ret)

Parameters

QTestState *s
: QTestState instance to operate on.

const char *name
: name of the command to call.

uint32_t nargs
: Number of args.

uint64_t args
: Guest address to read args from.

uint32_t nret
: Number of return value.

uint64_t ret
: Guest address to write return values to.

Description

Call an RTAS function

void **qtest_bufread**(QTestState *s, uint64_t addr, void *data, size_t size)

Parameters

QTestState *s
: QTestState instance to operate on.

uint64_t addr
: Guest address to read from.

void *data
: Pointer to where memory contents will be stored.

size_t size
: Number of bytes to read.

Description

Read guest memory into a buffer and receive using a base64 encoding.

void **qtest_memwrite**(QTestState *s, uint64_t addr, const void *data, size_t size)

Parameters

QTestState *s
: QTestState instance to operate on.

uint64_t addr
: Guest address to write to.

const void *data
: Pointer to the bytes that will be written to guest memory.

size_t size
: Number of bytes to write.

Description

Write a buffer to guest memory.

void **qtest_bufwrite**(QTestState *s, uint64_t addr, const void *data, size_t size)

Parameters

QTestState *s
: QTestState instance to operate on.

uint64_t addr
: Guest address to write to.

const void *data
: Pointer to the bytes that will be written to guest memory.

size_t size
: Number of bytes to write.

Description

Write a buffer to guest memory and transmit using a base64 encoding.

void **qtest_memset**(QTestState *s, uint64_t addr, uint8_t patt, size_t size)

Parameters

QTestState *s
: QTestState instance to operate on.

uint64_t addr
Guest address to write to.

uint8_t patt
Byte pattern to fill the guest memory region with.

size_t size
Number of bytes to write.

Description

Write a pattern to guest memory.

int64_t **qtest_clock_step_next**(QTestState *s)

Parameters

QTestState *s
`QTestState` instance to operate on.

Description

Advance the QEMU_CLOCK_VIRTUAL to the next deadline.

Return

The current value of the QEMU_CLOCK_VIRTUAL in nanoseconds.

int64_t **qtest_clock_step**(QTestState *s, int64_t step)

Parameters

QTestState *s
QTestState instance to operate on.

int64_t step
Number of nanoseconds to advance the clock by.

Description

Advance the QEMU_CLOCK_VIRTUAL by **step** nanoseconds.

Return

The current value of the QEMU_CLOCK_VIRTUAL in nanoseconds.

int64_t **qtest_clock_set**(QTestState *s, int64_t val)

Parameters

QTestState *s
QTestState instance to operate on.

int64_t val
Nanoseconds value to advance the clock to.

Description

Advance the QEMU_CLOCK_VIRTUAL to **val** nanoseconds since the VM was launched.

Return

The current value of the QEMU_CLOCK_VIRTUAL in nanoseconds.

bool **qtest_big_endian**(QTestState *s)

Parameters

QTestState *s
: QTestState instance to operate on.

Return

True if the architecture under test has a big endian configuration.

const char ***qtest_get_arch**(void)

Parameters

void
: no arguments

Return

The architecture for the QEMU executable under test.

bool **qtest_has_accel**(const char *accel_name)

Parameters

const char *accel_name
: Accelerator name to check for.

Return

true if the accelerator is built in.

void **qtest_add_func**(const char *str, void (*fn)(void))

Parameters

const char *str
: Test case path.

void (*fn)(void)
: Test case function

Description

Add a GTester testcase with the given name and function. The path is prefixed with the architecture under test, as returned by qtest_get_arch().

void **qtest_add_data_func**(const char *str, const void *data, void (*fn)(const void*))

Parameters

const char *str
: Test case path.

const void *data
: Test case data

void (*fn)(const void *)
: Test case function

Description

Add a GTester testcase with the given name, data and function. The path is prefixed with the architecture under test, as returned by qtest_get_arch().

void **qtest_add_data_func_full**(const char *str, void *data, void (*fn)(const void*), GDestroyNotify data_free_func)

Parameters

const char *str
Test case path.

void *data
Test case data

void (*fn)(const void *)
Test case function

GDestroyNotify data_free_func
GDestroyNotify for data

Description

Add a GTester testcase with the given name, data and function. The path is prefixed with the architecture under test, as returned by qtest_get_arch().

data is passed to **data_free_func**() on test completion.

qtest_add

```
qtest_add (testpath, Fixture, tdata, fsetup, ftest, fteardown)
```

Parameters

testpath
Test case path

Fixture
Fixture type

tdata
Test case data

fsetup
Test case setup function

ftest
Test case function

fteardown
Test case teardown function

Description

Add a GTester testcase with the given name, data and functions. The path is prefixed with the architecture under test, as returned by qtest_get_arch().

void **qtest_add_abrt_handler**(GHookFunc fn, const void *data)

Parameters

GHookFunc fn
Handler function

const void *data
Argument that is passed to the handler

Description

Add a handler function that is invoked on SIGABRT. This can be used to terminate processes and perform other cleanup. The handler can be removed with qtest_remove_abrt_handler().

void **qtest_remove_abrt_handler**(void *data)

Parameters

void *data
Argument previously passed to qtest_add_abrt_handler()

Description

Remove an abrt handler that was previously added with qtest_add_abrt_handler().

QDict ***qtest_vqmp_assert_success_ref**(QTestState *qts, const char *fmt, va_list args)

Parameters

QTestState *qts
QTestState instance to operate on

const char *fmt
QMP message to send to qemu, formatted like qobject_from_jsonf_nofail(). See parse_interpolation() for what's supported after '%'.

va_list args
variable arguments for **fmt**

Description

Sends a QMP message to QEMU, asserts that a 'return' key is present in the response, and returns the response.

void **qtest_vqmp_assert_success**(QTestState *qts, const char *fmt, va_list args)

Parameters

QTestState *qts
QTestState instance to operate on

const char *fmt
QMP message to send to qemu, formatted like qobject_from_jsonf_nofail(). See parse_interpolation() for what's supported after '%'.

va_list args
variable arguments for **fmt**

Description

Sends a QMP message to QEMU and asserts that a 'return' key is present in the response.

QDict ***qtest_vqmp_fds_assert_success_ref**(QTestState *qts, int *fds, size_t nfds, const char *fmt, va_list args)

Parameters

QTestState *qts
QTestState instance to operate on

int *fds
the file descriptors to send

size_t nfds
number of **fds** to send

const char *fmt
QMP message to send to qemu, formatted like qobject_from_jsonf_nofail(). See parse_interpolation() for what's supported after '%'.

va_list args
variable arguments for **fmt**

Description

Sends a QMP message with file descriptors to QEMU, asserts that a 'return' key is present in the response, and returns the response.

void **qtest_vqmp_fds_assert_success**(QTestState *qts, int *fds, size_t nfds, const char *fmt, va_list args)

Parameters

QTestState *qts
QTestState instance to operate on

int *fds
the file descriptors to send

size_t nfds
number of **fds** to send

const char *fmt
QMP message to send to qemu, formatted like qobject_from_jsonf_nofail(). See parse_interpolation() for what's supported after '%'.

va_list args
variable arguments for **fmt**

Description

Sends a QMP message with file descriptors to QEMU and asserts that a 'return' key is present in the response.

QDict *qtest_qmp_assert_failure_ref(QTestState *qts, const char *fmt, ...)

Parameters

QTestState *qts
QTestState instance to operate on

const char *fmt
QMP message to send to qemu, formatted like qobject_from_jsonf_nofail(). See parse_interpolation() for what's supported after '%'.

...
variable arguments

Description

Sends a QMP message to QEMU, asserts that an 'error' key is present in the response, and returns the response.

QDict *qtest_vqmp_assert_failure_ref(QTestState *qts, const char *fmt, va_list args)

Parameters

QTestState *qts
QTestState instance to operate on

const char *fmt
QMP message to send to qemu, formatted like qobject_from_jsonf_nofail(). See parse_interpolation() for what's supported after '%'.

`va_list args`
: variable arguments for **fmt**

Description

Sends a QMP message to QEMU, asserts that an 'error' key is present in the response, and returns the response.

QDict *`qtest_qmp_assert_success_ref`(QTestState *qts, const char *fmt, ...)

Parameters

`QTestState *qts`
: QTestState instance to operate on

`const char *fmt`
: QMP message to send to qemu, formatted like qobject_from_jsonf_nofail(). See parse_interpolation() for what's supported after '%'.

`...`
: variable arguments

Description

Sends a QMP message to QEMU, asserts that a 'return' key is present in the response, and returns the response.

void `qtest_qmp_assert_success`(QTestState *qts, const char *fmt, ...)

Parameters

`QTestState *qts`
: QTestState instance to operate on

`const char *fmt`
: QMP message to send to qemu, formatted like qobject_from_jsonf_nofail(). See parse_interpolation() for what's supported after '%'.

`...`
: variable arguments

Description

Sends a QMP message to QEMU and asserts that a 'return' key is present in the response.

QDict *`qtest_qmp_fds_assert_success_ref`(QTestState *qts, int *fds, size_t nfds, const char *fmt, ...)

Parameters

`QTestState *qts`
: QTestState instance to operate on

`int *fds`
: the file descriptors to send

`size_t nfds`
: number of **fds** to send

`const char *fmt`
: QMP message to send to qemu, formatted like qobject_from_jsonf_nofail(). See parse_interpolation() for what's supported after '%'.

`...`
: variable arguments

Description

Sends a QMP message with file descriptors to QEMU, asserts that a 'return' key is present in the response, and returns the response.

void **qtest_qmp_fds_assert_success**(QTestState *qts, int *fds, size_t nfds, const char *fmt, ...)

Parameters

QTestState *qts
QTestState instance to operate on

int *fds
the file descriptors to send

size_t nfds
number of **fds** to send

const char *fmt
QMP message to send to qemu, formatted like qobject_from_jsonf_nofail(). See parse_interpolation() for what's supported after '%'.

...
variable arguments

Description

Sends a QMP message with file descriptors to QEMU and asserts that a 'return' key is present in the response.

void **qtest_cb_for_every_machine**(void (*cb)(const char *machine), bool skip_old_versioned)

Parameters

void (*cb)(const char *machine)
Pointer to the callback function

bool skip_old_versioned
true if versioned old machine types should be skipped

Call a callback function for every name of all available machines.

char ***qtest_resolve_machine_alias**(const char *var, const char *alias)

Parameters

const char *var
Environment variable from where to take the QEMU binary

const char *alias
The alias to resolve

Return

the machine type corresponding to the alias if any, otherwise NULL.

bool **qtest_has_machine**(const char *machine)

Parameters

const char *machine
The machine to look for

Return

true if the machine is available in the target binary.

bool **qtest_has_machine_with_env**(const char *var, const char *machine)

Parameters

const char *var
Environment variable from where to take the QEMU binary

const char *machine
The machine to look for

Return

true if the machine is available in the specified binary.

bool **qtest_has_device**(const char *device)

Parameters

const char *device
The device to look for

Return

true if the device is available in the target binary.

void **qtest_qmp_device_add_qdict**(QTestState *qts, const char *drv, const QDict *arguments)

Parameters

QTestState *qts
QTestState instance to operate on

const char *drv
Name of the device that should be added

const QDict *arguments
QDict with properties for the device to initialize

Description

Generic hot-plugging test via the device_add QMP command with properties supplied in form of QDict. Use NULL for empty properties list.

void **qtest_qmp_device_add**(QTestState *qts, const char *driver, const char *id, const char *fmt, ...)

Parameters

QTestState *qts
QTestState instance to operate on

const char *driver
Name of the device that should be added

const char *id
Identification string

const char *fmt
QMP message to send to qemu, formatted like qobject_from_jsonf_nofail(). See parse_interpolation() for what's supported after '%'.

...
variable arguments

Description

Generic hot-plugging test via the device_add QMP command.

void **qtest_qmp_add_client**(QTestState *qts, const char *protocol, int fd)

Parameters

QTestState *qts
QTestState instance to operate on

const char *protocol
the protocol to add to

int fd
the client file-descriptor

Description

Call QMP `getfd` (on Windows `get-win32-socket`) followed by `add_client` with the given **fd**.

void **qtest_qmp_device_del_send**(QTestState *qts, const char *id)

Parameters

QTestState *qts
QTestState instance to operate on

const char *id
Identification string

Description

Generic hot-unplugging test via the device_del QMP command.

void **qtest_qmp_device_del**(QTestState *qts, const char *id)

Parameters

QTestState *qts
QTestState instance to operate on

const char *id
Identification string

Description

Generic hot-unplugging test via the device_del QMP command. Waiting for command completion event.

bool **qtest_probe_child**(QTestState *s)

Parameters

QTestState *s
QTestState instance to operate on.

Return

true if the child is still alive.

void **qtest_set_expected_status**(QTestState *s, int status)

Parameters

QTestState *s
QTestState instance to operate on.

int status
an expected exit status.

Description

Set expected exit status of the child.

void **qtest_qom_set_bool**(QTestState *s, const char *path, const char *property, bool value)

Parameters

QTestState *s
: QTestState instance to operate on.

const char *path
: Path to the property being set.

const char *property
: Property being set.

bool value
: Value to set the property.

Description

Set the property with passed in value.

bool **qtest_qom_get_bool**(QTestState *s, const char *path, const char *property)

Parameters

QTestState *s
: QTestState instance to operate on.

const char *path
: Path to the property being retrieved.

const char *property
: Property from where the value is being retrieved.

Return

Value retrieved from property.

pid_t **qtest_pid**(QTestState *s)

Parameters

QTestState *s
: QTestState instance to operate on.

Return

the PID of the QEMU process, or <= 0

bool **have_qemu_img**(void)

Parameters

void
: no arguments

Return

true if “qemu-img” is available.

bool **mkimg**(const char *file, const char *fmt, unsigned size_mb)

Parameters

const char *file
File name of the image that should be created

const char *fmt
Format, e.g. "qcow2" or "raw"

unsigned size_mb
Size of the image in megabytes

Description

Create a disk image with qemu-img. Note that the QTEST_QEMU_IMG environment variable must point to the qemu-img file.

Return

true if the image has been created successfully.

7.2.7 CI

Most of QEMU's CI is run on GitLab's infrastructure although a number of other CI services are used for specialised purposes. The most up to date information about them and their status can be found on the project wiki testing page.

Definition of terms

This section defines the terms used in this document and correlates them with what is currently used on QEMU.

Automated tests

An automated test is written on a test framework using its generic test functions/classes. The test framework can run the tests and report their success or failure[1].

An automated test has essentially three parts:

1. The test initialization of the parameters, where the expected parameters, like inputs and expected results, are set up;
2. The call to the code that should be tested;
3. An assertion, comparing the result from the previous call with the expected result set during the initialization of the parameters. If the result matches the expected result, the test has been successful; otherwise, it has failed.

[1] Sommerville, Ian (2016). Software Engineering. p. 233.

Unit testing

A unit test is responsible for exercising individual software components as a unit, like interfaces, data structures, and functionality, uncovering errors within the boundaries of a component. The verification effort is in the smallest software unit and focuses on the internal processing logic and data structures. A test case of unit tests should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow[2].

On QEMU, unit testing is represented by the 'check-unit' target from 'make'.

Functional testing

A functional test focuses on the functional requirement of the software. Deriving sets of input conditions, the functional tests should fully exercise all the functional requirements for a program. Functional testing is complementary to other testing techniques, attempting to find errors like incorrect or missing functions, interface errors, behavior errors, and initialization and termination errors[3].

On QEMU, functional testing is represented by the 'check-qtest' target from 'make'.

System testing

System tests ensure all application elements mesh properly while the overall functionality and performance are achieved[4]. Some or all system components are integrated to create a complete system to be tested as a whole. System testing ensures that components are compatible, interact correctly, and transfer the right data at the right time across their interfaces. As system testing focuses on interactions, use case-based testing is a practical approach to system testing[5]. Note that, in some cases, system testing may require interaction with third-party software, like operating system images, databases, networks, and so on.

On QEMU, system testing is represented by the 'check-avocado' target from 'make'.

Flaky tests

A flaky test is defined as a test that exhibits both a passing and a failing result with the same code on different runs. Some usual reasons for an intermittent/flaky test are async wait, concurrency, and test order dependency[6].

Gating

A gate restricts the move of code from one stage to another on a test/deployment pipeline. The step move is granted with approval. The approval can be a manual intervention or a set of tests succeeding[7].

On QEMU, the gating process happens during the pull request. The approval is done by the project leader running its own set of tests. The pull request gets merged when the tests succeed.

[2] Pressman, Roger S. & Maxim, Bruce R. (2020). Software Engineering, A Practitioner's Approach. p. 48, 376, 378, 381.
[3] Pressman, Roger S. & Maxim, Bruce R. (2020). Software Engineering, A Practitioner's Approach. p. 388.
[4] Pressman, Roger S. & Maxim, Bruce R. (2020). Software Engineering, A Practitioner's Approach. Software Engineering, p. 377.
[5] Sommerville, Ian (2016). Software Engineering. p. 59, 232, 240.
[6] Luo, Qingzhou, et al. An empirical analysis of flaky tests. Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014.
[7] Humble, Jez & Farley, David (2010). Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment, p. 122.

Continuous Integration (CI)

Continuous integration (CI) requires the builds of the entire application and the execution of a comprehensive set of automated tests every time there is a need to commit any set of changes[8]. The automated tests can be composed of the unit, functional, system, and other tests.

Keynotes about continuous integration (CI)[9]:

1. System tests may depend on external software (operating system images, firmware, database, network).
2. It may take a long time to build and test. It may be impractical to build the system being developed several times per day.
3. If the development platform is different from the target platform, it may not be possible to run system tests in the developer's private workspace. There may be differences in hardware, operating system, or installed software. Therefore, more time is required for testing the system.

References

Custom CI/CD variables

QEMU CI pipelines can be tuned by setting some CI environment variables.

Set variable globally in the user's CI namespace

Variables can be set globally in the user's CI namespace setting.

For further information about how to set these variables, please refer to:

```
https://docs.gitlab.com/ee/ci/variables/#add-a-cicd-variable-to-a-project
```

Set variable manually when pushing a branch or tag to the user's repository

Variables can be set manually when pushing a branch or tag, using git-push command line arguments.

Example setting the QEMU_CI_EXAMPLE_VAR variable:

```
git push -o ci.variable="QEMU_CI_EXAMPLE_VAR=value" myrepo mybranch
```

For further information about how to set these variables, please refer to:

```
https://docs.gitlab.com/ee/user/project/push_options.html#push-options-for-gitlab-cicd
```

[8] Humble, Jez & Farley, David (2010). Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment, p. 55.
[9] Sommerville, Ian (2016). Software Engineering. p. 743.

Setting aliases in your git config

You can use aliases to make it easier to push branches with different CI configurations. For example define an alias for triggering CI:

```
git config --local alias.push-ci "push -o ci.variable=QEMU_CI=1"
git config --local alias.push-ci-now "push -o ci.variable=QEMU_CI=2"
```

Which lets you run:

```
git push-ci
```

to create the pipeline, or:

```
git push-ci-now
```

to create and run the pipeline

Variable naming and grouping

The variables used by QEMU's CI configuration are grouped together in a handful of namespaces

- QEMU_JOB_nnnn - variables to be defined in individual jobs or templates, to influence the shared rules defined in the .base_job_template.
- QEMU_CI_nnn - variables to be set by contributors in their repository CI settings, or as git push variables, to influence which jobs get run in a pipeline
- QEMU_CI_CONTAINER_TAG - the tag used to publish containers in stage 1, for use by build jobs in stage 2. Defaults to 'latest', but if running pipelines for different branches concurrently, it should be overridden per pipeline.
- QEMU_CI_UPSTREAM - gitlab namespace that is considered to be the 'upstream'. This defaults to 'qemu-project'. Contributors may choose to override this if they are modifying rules in base.yml and need to validate how they will operate when in an upstream context, as opposed to their fork context.
- nnn - other misc variables not falling into the above categories, or using different names for historical reasons and not yet converted.

Maintainer controlled job variables

The following variables may be set when defining a job in the CI configuration file.

QEMU_JOB_CIRRUS

The job makes use of Cirrus CI infrastructure, requiring the configuration setup for cirrus-run to be present in the repository

QEMU_JOB_OPTIONAL

The job is expected to be successful in general, but is not run by default due to need to conserve limited CI resources. It is available to be started manually by the contributor in the CI pipelines UI.

QEMU_JOB_ONLY_FORKS

The job results are only of interest to contributors prior to submitting code. They are not required as part of the gating CI pipeline.

QEMU_JOB_SKIPPED

The job is not reliably successful in general, so is not currently suitable to be run by default. Ideally this should be a temporary marker until the problems can be addressed, or the job permanently removed.

QEMU_JOB_PUBLISH

The job is for publishing content after a branch has been merged into the upstream default branch.

QEMU_JOB_AVOCADO

The job runs the Avocado integration test suite

Contributor controlled runtime variables

The following variables may be set by contributors to control job execution

QEMU_CI

By default, no pipelines will be created on contributor forks in order to preserve CI credits

Set this variable to 1 to create the pipelines, but leave all the jobs to be manually started from the UI

Set this variable to 2 to create the pipelines and run all the jobs immediately, as was the historical behaviour

QEMU_CI_AVOCADO_TESTING

By default, tests using the Avocado framework are not run automatically in the pipelines (because multiple artifacts have to be downloaded, and if these artifacts are not already cached, downloading them make the jobs reach the timeout limit). Set this variable to have the tests using the Avocado framework run automatically.

Other misc variables

These variables are primarily to control execution of jobs on private runners

AARCH64_RUNNER_AVAILABLE

If you've got access to an aarch64 host that can be used as a gitlab-CI runner, you can set this variable to enable the tests that require this kind of host. The runner should be tagged with "aarch64".

AARCH32_RUNNER_AVAILABLE

If you've got access to an armhf host or an arch64 host that can run aarch32 EL0 code to be used as a gitlab-CI runner, you can set this variable to enable the tests that require this kind of host. The runner should be tagged with "aarch32".

S390X_RUNNER_AVAILABLE

If you've got access to an IBM Z host that can be used as a gitlab-CI runner, you can set this variable to enable the tests that require this kind of host. The runner should be tagged with "s390x".

CENTOS_STREAM_8_x86_64_RUNNER_AVAILABLE

If you've got access to a CentOS Stream 8 x86_64 host that can be used as a gitlab-CI runner, you can set this variable to enable the tests that require this kind of host. The runner should be tagged with both "centos_stream_8" and "x86_64".

CCACHE_DISABLE

The jobs are configured to use "ccache" by default since this typically reduces compilation time, at the cost of increased storage. If the use of "ccache" is suspected to be hurting the overall job execution time, setting the "CCACHE_DISABLE=1" env variable to disable it.

Jobs on Custom Runners

Besides the jobs run under the various CI systems listed before, there are a number additional jobs that will run before an actual merge. These use the same GitLab CI's service/framework already used for all other GitLab based CI jobs, but rely on additional systems, not the ones provided by GitLab as "shared runners".

The architecture of GitLab's CI service allows different machines to be set up with GitLab's "agent", called gitlab-runner, which will take care of running jobs created by events such as a push to a branch. Here, the combination of a machine, properly configured with GitLab's gitlab-runner, is called a "custom runner".

The GitLab CI jobs definition for the custom runners are located under:

```
.gitlab-ci.d/custom-runners.yml
```

Custom runners entail custom machines. To see a list of the machines currently deployed in the QEMU GitLab CI and their maintainers, please refer to the QEMU wiki.

Machine Setup Howto

For all Linux based systems, the setup can be mostly automated by the execution of two Ansible playbooks. Create an `inventory` file under `scripts/ci/setup`, such as this:

```
fully.qualified.domain
other.machine.hostname
```

You may need to set some variables in the inventory file itself. One very common need is to tell Ansible to use a Python 3 interpreter on those hosts. This would look like:

```
fully.qualified.domain ansible_python_interpreter=/usr/bin/python3
other.machine.hostname ansible_python_interpreter=/usr/bin/python3
```

Build environment

The `scripts/ci/setup/build-environment.yml` Ansible playbook will set up machines with the environment needed to perform builds and run QEMU tests. This playbook consists on the installation of various required packages (and a general package update while at it). It currently covers a number of different Linux distributions, but it can be expanded to cover other systems.

The minimum required version of Ansible successfully tested in this playbook is 2.8.0 (a version check is embedded within the playbook itself). To run the playbook, execute:

```
cd scripts/ci/setup
ansible-playbook -i inventory build-environment.yml
```

Please note that most of the tasks in the playbook require superuser privileges, such as those from the `root` account or those obtained by `sudo`. If necessary, please refer to `ansible-playbook` options such as `--become`, `--become-method`, `--become-user` and `--ask-become-pass`.

gitlab-runner setup and registration

The gitlab-runner agent needs to be installed on each machine that will run jobs. The association between a machine and a GitLab project happens with a registration token. To find the registration token for your repository/project, navigate on GitLab's web UI to:

- Settings (the gears-like icon at the bottom of the left hand side vertical toolbar), then
- CI/CD, then
- Runners, and click on the "Expand" button, then
- Under "Set up a specific Runner manually", look for the value under "And this registration token:"

Copy the `scripts/ci/setup/vars.yml.template` file to `scripts/ci/setup/vars.yml`. Then, set the `gitlab_runner_registration_token` variable to the value obtained earlier.

To run the playbook, execute:

```
cd scripts/ci/setup
ansible-playbook -i inventory gitlab-runner.yml
```

Following the registration, it's necessary to configure the runner tags, and optionally other configurations on the GitLab UI. Navigate to:

- Settings (the gears like icon), then
- CI/CD, then
- Runners, and click on the "Expand" button, then
- "Runners activated for this project", then
- Click on the "Edit" icon (next to the "Lock" Icon)

Tags are very important as they are used to route specific jobs to specific types of runners, so it's a good idea to double check that the automatically created tags are consistent with the OS and architecture. For instance, an Ubuntu 20.04 aarch64 system should have tags set as:

```
ubuntu_20.04,aarch64
```

Because the job definition at `.gitlab-ci.d/custom-runners.yml` would contain:

```
ubuntu-20.04-aarch64-all:
 tags:
 - ubuntu_20.04
 - aarch64
```

It's also recommended to:

- increase the "Maximum job timeout" to something like `2h`
- give it a better Description

7.2.8 How to use the QAPI code generator

Introduction

QAPI is a native C API within QEMU which provides management-level functionality to internal and external users. For external users/processes, this interface is made available by a JSON-based wire format for the QEMU Monitor Protocol (QMP) for controlling qemu, as well as the QEMU Guest Agent (QGA) for communicating with the guest. The remainder of this document uses "Client JSON Protocol" when referring to the wire contents of a QMP or QGA connection.

To map between Client JSON Protocol interfaces and the native C API, we generate C code from a QAPI schema. This document describes the QAPI schema language, and how it gets mapped to the Client JSON Protocol and to C. It additionally provides guidance on maintaining Client JSON Protocol compatibility.

The QAPI schema language

The QAPI schema defines the Client JSON Protocol's commands and events, as well as types used by them. Forward references are allowed.

It is permissible for the schema to contain additional types not used by any commands or events, for the side effect of generated C code used internally.

There are several kinds of types: simple types (a number of built-in types, such as `int` and `str`; as well as enumerations), arrays, complex types (structs and unions), and alternate types (a choice between other types).

Schema syntax

Syntax is loosely based on JSON. Differences:

- Comments: start with a hash character (#) that is not part of a string, and extend to the end of the line.
- Strings are enclosed in `'single quotes'`, not `"double quotes"`.
- Strings are restricted to printable ASCII, and escape sequences to just \\.
- Numbers and `null` are not supported.

A second layer of syntax defines the sequences of JSON texts that are a correctly structured QAPI schema. We provide a grammar for this syntax in an EBNF-like notation:

- Production rules look like `non-terminal = expression`
- Concatenation: expression `A B` matches expression `A`, then `B`
- Alternation: expression `A | B` matches expression `A` or `B`
- Repetition: expression `A...` matches zero or more occurrences of expression `A`
- Repetition: expression `A, ...` matches zero or more occurrences of expression `A` separated by `,`
- Grouping: expression `( A )` matches expression `A`
- JSON's structural characters are terminals: `{ } [ ] : ,`
- JSON's literal names are terminals: `false true`
- String literals enclosed in `'single quotes'` are terminal, and match this JSON string, with a leading * stripped off
- When JSON object member's name starts with *, the member is optional.
- The symbol `STRING` is a terminal, and matches any JSON string
- The symbol `BOOL` is a terminal, and matches JSON `false` or `true`
- ALL-CAPS words other than `STRING` are non-terminals

The order of members within JSON objects does not matter unless explicitly noted.

A QAPI schema consists of a series of top-level expressions:

```
SCHEMA = TOP-LEVEL-EXPR...
```

The top-level expressions are all JSON objects. Code and documentation is generated in schema definition order. Code order should not matter.

A top-level expressions is either a directive or a definition:

```
TOP-LEVEL-EXPR = DIRECTIVE | DEFINITION
```

There are two kinds of directives and six kinds of definitions:

```
DIRECTIVE = INCLUDE | PRAGMA
DEFINITION = ENUM | STRUCT | UNION | ALTERNATE | COMMAND | EVENT
```

These are discussed in detail below.

Built-in Types

The following types are predefined, and map to C as follows:

Schema	C	JSON
`str`	`char *`	any JSON string, UTF-8
`number`	`double`	any JSON number
`int`	`int64_t`	a JSON number without fractional part that fits into the C integer type
`int8`	`int8_t`	likewise
`int16`	`int16_t`	likewise
`int32`	`int32_t`	likewise
`int64`	`int64_t`	likewise
`uint8`	`uint8_t`	likewise
`uint16`	`uint16_t`	likewise
`uint32`	`uint32_t`	likewise
`uint64`	`uint64_t`	likewise
`size`	`uint64_t`	like `uint64_t`, except `StringInputVisitor` accepts size suffixes
`bool`	`bool`	JSON `true` or `false`
`null`	`QNull *`	JSON `null`
`any`	`QObject *`	any JSON value
`QType`	`QType`	JSON string matching enum `QType` values

Include directives

Syntax:

```
INCLUDE = { 'include': STRING }
```

The QAPI schema definitions can be modularized using the 'include' directive:

```
{ 'include': 'path/to/file.json' }
```

The directive is evaluated recursively, and include paths are relative to the file using the directive. Multiple includes of the same file are idempotent.

As a matter of style, it is a good idea to have all files be self-contained, but at the moment, nothing prevents an included file from making a forward reference to a type that is only introduced by an outer file. The parser may be made stricter in the future to prevent incomplete include files.

Pragma directives

Syntax:

```
PRAGMA = { 'pragma': {
               '*doc-required': BOOL,
               '*command-name-exceptions': [ STRING, ... ],
               '*command-returns-exceptions': [ STRING, ... ],
               '*documentation-exceptions': [ STRING, ... ],
               '*member-name-exceptions': [ STRING, ... ] } }
```

The pragma directive lets you control optional generator behavior.

Pragma's scope is currently the complete schema. Setting the same pragma to different values in parts of the schema doesn't work.

Pragma 'doc-required' takes a boolean value. If true, documentation is required. Default is false.

Pragma 'command-name-exceptions' takes a list of commands whose names may contain "_" instead of "-". Default is none.

Pragma 'command-returns-exceptions' takes a list of commands that may violate the rules on permitted return types. Default is none.

Pragma 'documentation-exceptions' takes a list of types, commands, and events whose members / arguments need not be documented. Default is none.

Pragma 'member-name-exceptions' takes a list of types whose member names may contain uppercase letters, and "_" instead of "-". Default is none.

Enumeration types

Syntax:

```
ENUM = { 'enum': STRING,
         'data': [ ENUM-VALUE, ... ],
         '*prefix': STRING,
         '*if': COND,
         '*features': FEATURES }
ENUM-VALUE = STRING
           | { 'name': STRING,
               '*if': COND,
               '*features': FEATURES }
```

Member 'enum' names the enum type.

Each member of the 'data' array defines a value of the enumeration type. The form STRING is shorthand for { 'name': STRING }. The 'name' values must be be distinct.

Example:

```
{ 'enum': 'MyEnum', 'data': [ 'value1', 'value2', 'value3' ] }
```

Nothing prevents an empty enumeration, although it is probably not useful.

On the wire, an enumeration type's value is represented by its (string) name. In C, it's represented by an enumeration constant. These are of the form PREFIX_NAME, where PREFIX is derived from the enumeration type's name, and NAME from the value's name. For the example above, the generator maps 'MyEnum' to MY_ENUM and 'value1' to VALUE1, resulting in the enumeration constant MY_ENUM_VALUE1. The optional 'prefix' member overrides PREFIX.

The generated C enumeration constants have values 0, 1, ..., N-1 (in QAPI schema order), where N is the number of values. There is an additional enumeration constant PREFIX__MAX with value N.

Do not use string or an integer type when an enumeration type can do the job satisfactorily.

The optional 'if' member specifies a conditional. See *Configuring the schema* below for more on this.

The optional 'features' member specifies features. See *Features* below for more on this.

Type references and array types

Syntax:

```
TYPE-REF = STRING | ARRAY-TYPE
ARRAY-TYPE = [ STRING ]
```

A string denotes the type named by the string.

A one-element array containing a string denotes an array of the type named by the string. Example: `['int']` denotes an array of `int`.

Struct types

Syntax:

```
STRUCT = { 'struct': STRING,
           'data': MEMBERS,
           '*base': STRING,
           '*if': COND,
           '*features': FEATURES }
MEMBERS = { MEMBER, ... }
MEMBER = STRING : TYPE-REF
       | STRING : { 'type': TYPE-REF,
                    '*if': COND,
                    '*features': FEATURES }
```

Member 'struct' names the struct type.

Each MEMBER of the 'data' object defines a member of the struct type.

The MEMBER's STRING name consists of an optional * prefix and the struct member name. If * is present, the member is optional.

The MEMBER's value defines its properties, in particular its type. The form *TYPE-REF* is shorthand for `{ 'type': TYPE-REF }`.

Example:

```
{ 'struct': 'MyType',
  'data': { 'member1': 'str', 'member2': ['int'], '*member3': 'str' } }
```

A struct type corresponds to a struct in C, and an object in JSON. The C struct's members are generated in QAPI schema order.

The optional 'base' member names a struct type whose members are to be included in this type. They go first in the C struct.

Example:

```
{ 'struct': 'BlockdevOptionsGenericFormat',
  'data': { 'file': 'str' } }
{ 'struct': 'BlockdevOptionsGenericCOWFormat',
  'base': 'BlockdevOptionsGenericFormat',
  'data': { '*backing': 'str' } }
```

An example BlockdevOptionsGenericCOWFormat object on the wire could use both members like this:

```
{ "file": "/some/place/my-image",
  "backing": "/some/place/my-backing-file" }
```

The optional 'if' member specifies a conditional. See *Configuring the schema* below for more on this.

The optional 'features' member specifies features. See *Features* below for more on this.

Union types

Syntax:

```
UNION = { 'union': STRING,
          'base': ( MEMBERS | STRING ),
          'discriminator': STRING,
          'data': BRANCHES,
          '*if': COND,
          '*features': FEATURES }
BRANCHES = { BRANCH, ... }
BRANCH = STRING : TYPE-REF
       | STRING : { 'type': TYPE-REF, '*if': COND }
```

Member 'union' names the union type.

The 'base' member defines the common members. If it is a *MEMBERS* object, it defines common members just like a struct type's 'data' member defines struct type members. If it is a STRING, it names a struct type whose members are the common members.

Member 'discriminator' must name a non-optional enum-typed member of the base struct. That member's value selects a branch by its name. If no such branch exists, an empty branch is assumed.

Each BRANCH of the 'data' object defines a branch of the union. A union must have at least one branch.

The BRANCH's STRING name is the branch name. It must be a value of the discriminator enum type.

The BRANCH's value defines the branch's properties, in particular its type. The type must a struct type. The form *TYPE-REF* is shorthand for `{ 'type': TYPE-REF }`.

In the Client JSON Protocol, a union is represented by an object with the common members (from the base type) and the selected branch's members. The two sets of member names must be disjoint.

Example:

```
{ 'enum': 'BlockdevDriver', 'data': [ 'file', 'qcow2' ] }
{ 'union': 'BlockdevOptions',
  'base': { 'driver': 'BlockdevDriver', '*read-only': 'bool' },
  'discriminator': 'driver',
  'data': { 'file': 'BlockdevOptionsFile',
            'qcow2': 'BlockdevOptionsQcow2' } }
```

Resulting in these JSON objects:

```
{ "driver": "file", "read-only": true,
  "filename": "/some/place/my-image" }
{ "driver": "qcow2", "read-only": false,
  "backing": "/some/place/my-image", "lazy-refcounts": true }
```

The order of branches need not match the order of the enum values. The branches need not cover all possible enum values. In the resulting generated C data types, a union is represented as a struct with the base members in QAPI schema order, and then a union of structures for each branch of the struct.

The optional 'if' member specifies a conditional. See *Configuring the schema* below for more on this.

The optional 'features' member specifies features. See *Features* below for more on this.

Alternate types

Syntax:

```
ALTERNATE = { 'alternate': STRING,
              'data': ALTERNATIVES,
              '*if': COND,
              '*features': FEATURES }
ALTERNATIVES = { ALTERNATIVE, ... }
ALTERNATIVE = STRING : STRING
            | STRING : { 'type': STRING, '*if': COND }
```

Member 'alternate' names the alternate type.

Each ALTERNATIVE of the 'data' object defines a branch of the alternate. An alternate must have at least one branch.

The ALTERNATIVE's STRING name is the branch name.

The ALTERNATIVE's value defines the branch's properties, in particular its type. The form STRING is shorthand for `{ 'type': STRING }`.

Example:

```
{ 'alternate': 'BlockdevRef',
  'data': { 'definition': 'BlockdevOptions',
            'reference': 'str' } }
```

An alternate type is like a union type, except there is no discriminator on the wire. Instead, the branch to use is inferred from the value. An alternate can only express a choice between types represented differently on the wire.

If a branch is typed as the 'bool' built-in, the alternate accepts true and false; if it is typed as any of the various numeric built-ins, it accepts a JSON number; if it is typed as a 'str' built-in or named enum type, it accepts a JSON string; if it is typed as the 'null' built-in, it accepts JSON null; and if it is typed as a complex type (struct or union), it accepts a JSON object.

The example alternate declaration above allows using both of the following example objects:

```
{ "file": "my_existing_block_device_id" }
{ "file": { "driver": "file",
            "read-only": false,
            "filename": "/tmp/mydisk.qcow2" } }
```

The optional 'if' member specifies a conditional. See *Configuring the schema* below for more on this.

The optional 'features' member specifies features. See *Features* below for more on this.

Commands

Syntax:

```
COMMAND = { 'command': STRING,
            (
            '*data': ( MEMBERS | STRING ),
            |
            'data': STRING,
            'boxed': true,
            )
            '*returns': TYPE-REF,
            '*success-response': false,
            '*gen': false,
            '*allow-oob': true,
            '*allow-preconfig': true,
            '*coroutine': true,
            '*if': COND,
            '*features': FEATURES }
```

Member 'command' names the command.

Member 'data' defines the arguments. It defaults to an empty *MEMBERS* object.

If 'data' is a *MEMBERS* object, then MEMBERS defines arguments just like a struct type's 'data' defines struct type members.

If 'data' is a STRING, then STRING names a complex type whose members are the arguments. A union type requires `'boxed': true`.

Member 'returns' defines the command's return type. It defaults to an empty struct type. It must normally be a complex type or an array of a complex type. To return anything else, the command must be listed in pragma 'commands-returns-exceptions'. If you do this, extending the command to return additional information will be harder. Use of the pragma for new commands is strongly discouraged.

A command's error responses are not specified in the QAPI schema. Error conditions should be documented in comments.

In the Client JSON Protocol, the value of the "execute" or "exec-oob" member is the command name. The value of the "arguments" member then has to conform to the arguments, and the value of the success response's "return" member will conform to the return type.

Some example commands:

```
{ 'command': 'my-first-command',
  'data': { 'arg1': 'str', '*arg2': 'str' } }
{ 'struct': 'MyType', 'data': { '*value': 'str' } }
{ 'command': 'my-second-command',
  'returns': [ 'MyType' ] }
```

which would validate this Client JSON Protocol transaction:

```
=> { "execute": "my-first-command",
     "arguments": { "arg1": "hello" } }
<= { "return": { } }
=> { "execute": "my-second-command" }
<= { "return": [ { "value": "one" }, { } ] }
```

The generator emits a prototype for the C function implementing the command. The function itself needs to be written by hand. See section *Code generated for commands* for examples.

The function returns the return type. When member 'boxed' is absent, it takes the command arguments as arguments one by one, in QAPI schema order. Else it takes them wrapped in the C struct generated for the complex argument type. It takes an additional `Error **` argument in either case.

The generator also emits a marshalling function that extracts arguments for the user's function out of an input QDict, calls the user's function, and if it succeeded, builds an output QObject from its return value. This is for use by the QMP monitor core.

In rare cases, QAPI cannot express a type-safe representation of a corresponding Client JSON Protocol command. You then have to suppress generation of a marshalling function by including a member 'gen' with boolean value false, and instead write your own function. For example:

```
{ 'command': 'netdev_add',
  'data': {'type': 'str', 'id': 'str'},
  'gen': false }
```

Please try to avoid adding new commands that rely on this, and instead use type-safe unions.

Normally, the QAPI schema is used to describe synchronous exchanges, where a response is expected. But in some cases, the action of a command is expected to change state in a way that a successful response is not possible (although the command will still return an error object on failure). When a successful reply is not possible, the command definition includes the optional member 'success-response' with boolean value false. So far, only QGA makes use of this member.

Member 'allow-oob' declares whether the command supports out-of-band (OOB) execution. It defaults to false. For example:

```
{ 'command': 'migrate_recover',
  'data': { 'uri': 'str' }, 'allow-oob': true }
```

See the *QEMU Machine Protocol Specification* for out-of-band execution syntax and semantics.

Commands supporting out-of-band execution can still be executed in-band.

When a command is executed in-band, its handler runs in the main thread with the BQL held.

When a command is executed out-of-band, its handler runs in a dedicated monitor I/O thread with the BQL *not* held.

An OOB-capable command handler must satisfy the following conditions:

- It terminates quickly.
- It does not invoke system calls that may block.
- It does not access guest RAM that may block when userfaultfd is enabled for postcopy live migration.
- It takes only "fast" locks, i.e. all critical sections protected by any lock it takes also satisfy the conditions for OOB command handler code.

The restrictions on locking limit access to shared state. Such access requires synchronization, but OOB commands can't take the BQL or any other "slow" lock.

When in doubt, do not implement OOB execution support.

Member 'allow-preconfig' declares whether the command is available before the machine is built. It defaults to false. For example:

```
{ 'enum': 'QMPCapability',
  'data': [ 'oob' ] }
{ 'command': 'qmp_capabilities',
```

(continues on next page)

(continued from previous page)

```
  'data': { '*enable': [ 'QMPCapability' ] },
  'allow-preconfig': true }
```

QMP is available before the machine is built only when QEMU was started with –preconfig.

Member 'coroutine' tells the QMP dispatcher whether the command handler is safe to be run in a coroutine. It defaults to false. If it is true, the command handler is called from coroutine context and may yield while waiting for an external event (such as I/O completion) in order to avoid blocking the guest and other background operations.

Coroutine safety can be hard to prove, similar to thread safety. Common pitfalls are:

- The BQL isn't held across `qemu_coroutine_yield()`, so operations that used to assume that they execute atomically may have to be more careful to protect against changes in the global state.
- Nested event loops (`AIO_WAIT_WHILE()` etc.) are problematic in coroutine context and can easily lead to deadlocks. They should be replaced by yielding and reentering the coroutine when the condition becomes false.

Since the command handler may assume coroutine context, any callers other than the QMP dispatcher must also call it in coroutine context. In particular, HMP commands calling such a QMP command handler must be marked `.coroutine = true` in hmp-commands.hx.

It is an error to specify both `'coroutine': true` and `'allow-oob': true` for a command. We don't currently have a use case for both together and without a use case, it's not entirely clear what the semantics should be.

The optional 'if' member specifies a conditional. See *Configuring the schema* below for more on this.

The optional 'features' member specifies features. See *Features* below for more on this.

Events

Syntax:

```
EVENT = { 'event': STRING,
          (
          '*data': ( MEMBERS | STRING ),
          |
          'data': STRING,
          'boxed': true,
          )
          '*if': COND,
          '*features': FEATURES }
```

Member 'event' names the event. This is the event name used in the Client JSON Protocol.

Member 'data' defines the event-specific data. It defaults to an empty MEMBERS object.

If 'data' is a MEMBERS object, then MEMBERS defines event-specific data just like a struct type's 'data' defines struct type members.

If 'data' is a STRING, then STRING names a complex type whose members are the event-specific data. A union type requires `'boxed': true`.

An example event is:

```
{ 'event': 'EVENT_C',
  'data': { '*a': 'int', 'b': 'str' } }
```

Resulting in this JSON object:

```
{ "event": "EVENT_C",
  "data": { "b": "test string" },
  "timestamp": { "seconds": 1267020223, "microseconds": 435656 } }
```

The generator emits a function to send the event. When member 'boxed' is absent, it takes event-specific data one by one, in QAPI schema order. Else it takes them wrapped in the C struct generated for the complex type. See section *Code generated for events* for examples.

The optional 'if' member specifies a conditional. See *Configuring the schema* below for more on this.

The optional 'features' member specifies features. See *Features* below for more on this.

Features

Syntax:

```
FEATURES = [ FEATURE, ... ]
FEATURE = STRING
        | { 'name': STRING, '*if': COND }
```

Sometimes, the behaviour of QEMU changes compatibly, but without a change in the QMP syntax (usually by allowing values or operations that previously resulted in an error). QMP clients may still need to know whether the extension is available.

For this purpose, a list of features can be specified for definitions, enumeration values, and struct members. Each feature list member can either be `{ 'name': STRING, '*if': COND }`, or STRING, which is shorthand for `{ 'name': STRING }`.

The optional 'if' member specifies a conditional. See *Configuring the schema* below for more on this.

Example:

```
{ 'struct': 'TestType',
  'data': { 'number': 'int' },
  'features': [ 'allow-negative-numbers' ] }
```

The feature strings are exposed to clients in introspection, as explained in section *Client JSON Protocol introspection*.

Intended use is to have each feature string signal that this build of QEMU shows a certain behaviour.

Special features

Feature "deprecated" marks a command, event, enum value, or struct member as deprecated. It is not supported elsewhere so far. Interfaces so marked may be withdrawn in future releases in accordance with QEMU's deprecation policy.

Feature "unstable" marks a command, event, enum value, or struct member as unstable. It is not supported elsewhere so far. Interfaces so marked may be withdrawn or changed incompatibly in future releases.

Naming rules and reserved names

All names must begin with a letter, and contain only ASCII letters, digits, hyphen, and underscore. There are two exceptions: enum values may start with a digit, and names that are downstream extensions (see section *Downstream extensions*) start with underscore.

Names beginning with `q_` are reserved for the generator, which uses them for munging QMP names that resemble C keywords or other problematic strings. For example, a member named `default` in qapi becomes `q_default` in the generated C code.

Types, commands, and events share a common namespace. Therefore, generally speaking, type definitions should always use CamelCase for user-defined type names, while built-in types are lowercase.

Type names ending with `List` are reserved for the generator, which uses them for array types.

Command names, member names within a type, and feature names should be all lower case with words separated by a hyphen. However, some existing older commands and complex types use underscore; when extending them, consistency is preferred over blindly avoiding underscore.

Event names should be ALL_CAPS with words separated by underscore.

Member name `u` and names starting with `has-` or `has_` are reserved for the generator, which uses them for unions and for tracking optional members.

Names beginning with `x-` used to signify "experimental". This convention has been replaced by special feature "unstable".

Pragmas `command-name-exceptions` and `member-name-exceptions` let you violate naming rules. Use for new code is strongly discouraged. See *Pragma directives* for details.

Downstream extensions

QAPI schema names that are externally visible, say in the Client JSON Protocol, need to be managed with care. Names starting with a downstream prefix of the form __RFQDN_ are reserved for the downstream who controls the valid, reverse fully qualified domain name RFQDN. RFQDN may only contain ASCII letters, digits, hyphen and period.

Example: Red Hat, Inc. controls redhat.com, and may therefore add a downstream command `__com.redhat_drive-mirror`.

Configuring the schema

Syntax:

```
COND = STRING
     | { 'all: [ COND, ... ] }
     | { 'any: [ COND, ... ] }
     | { 'not': COND }
```

All definitions take an optional 'if' member. Its value must be a string, or an object with a single member 'all', 'any' or 'not'.

The C code generated for the definition will then be guarded by an #if preprocessing directive with an operand generated from that condition:

- STRING will generate defined(STRING)
- { 'all': [COND, ...] } will generate (COND && ...)

- { 'any': [COND, ...] } will generate (COND || ...)
- { 'not': COND } will generate !COND

Example: a conditional struct

```
{ 'struct': 'IfStruct', 'data': { 'foo': 'int' },
  'if': { 'all': [ 'CONFIG_FOO', 'HAVE_BAR' ] } }
```

gets its generated code guarded like this:

```
#if defined(CONFIG_FOO) && defined(HAVE_BAR)
... generated code ...
#endif /* defined(HAVE_BAR) && defined(CONFIG_FOO) */
```

Individual members of complex types can also be made conditional. This requires the longhand form of MEMBER.

Example: a struct type with unconditional member 'foo' and conditional member 'bar'

```
{ 'struct': 'IfStruct',
  'data': { 'foo': 'int',
            'bar': { 'type': 'int', 'if': 'IFCOND'} } }
```

A union's discriminator may not be conditional.

Likewise, individual enumeration values may be conditional. This requires the longhand form of *ENUM-VALUE*.

Example: an enum type with unconditional value 'foo' and conditional value 'bar'

```
{ 'enum': 'IfEnum',
  'data': [ 'foo',
            { 'name' : 'bar', 'if': 'IFCOND' } ] }
```

Likewise, features can be conditional. This requires the longhand form of *FEATURE*.

Example: a struct with conditional feature 'allow-negative-numbers'

```
{ 'struct': 'TestType',
  'data': { 'number': 'int' },
  'features': [ { 'name': 'allow-negative-numbers',
                  'if': 'IFCOND' } ] }
```

Please note that you are responsible to ensure that the C code will compile with an arbitrary combination of conditions, since the generator is unable to check it at this point.

The conditions apply to introspection as well, i.e. introspection shows a conditional entity only when the condition is satisfied in this particular build.

Documentation comments

A multi-line comment that starts and ends with a ## line is a documentation comment.

If the documentation comment starts like

```
##
# @SYMBOL:
```

it documents the definition of SYMBOL, else it's free-form documentation.

See below for more on *Definition documentation*.

Free-form documentation may be used to provide additional text and structuring content.

Headings and subheadings

A free-form documentation comment containing a line which starts with some = symbols and then a space defines a section heading:

```
##
# = This is a top level heading
#
# This is a free-form comment which will go under the
# top level heading.
##

##
# == This is a second level heading
##
```

A heading line must be the first line of the documentation comment block.

Section headings must always be correctly nested, so you can only define a third-level heading inside a second-level heading, and so on.

Documentation markup

Documentation comments can use most rST markup. In particular, a `::` literal block can be used for examples:

```
# ::
#
#   Text of the example, may span
#   multiple lines
```

* starts an itemized list:

```
# * First item, may span
#   multiple lines
# * Second item
```

You can also use - instead of *.

A decimal number followed by `.` starts a numbered list:

```
# 1. First item, may span
#    multiple lines
# 2. Second item
```

The actual number doesn't matter.

Lists of either kind must be preceded and followed by a blank line. If a list item's text spans multiple lines, then the second and subsequent lines must be correctly indented to line up with the first character of the first line.

The usual ****strong****, **emphasized** and ``\`\`literal\`\`` markup should be used. If you need a single literal *, you will need to backslash-escape it.

Use `@foo` to reference a name in the schema. This is an rST extension. It is rendered the same way as ``\`\`foo\`\``, but carries additional meaning.

Example:

```
##
# Some text foo with **bold** and *emphasis*
#
# 1. with a list
# 2. like that
#
# And some code:
#
# ::
#
#   $ echo foo
#   -> do this
#   <- get that
##
```

For legibility, wrap text paragraphs so every line is at most 70 characters long.

Separate sentences with two spaces.

Definition documentation

Definition documentation, if present, must immediately precede the definition it documents.

When documentation is required (see *pragma* 'doc-required'), every definition must have documentation.

Definition documentation starts with a line naming the definition, followed by an optional overview, a description of each argument (for commands and events), member (for structs and unions), branch (for alternates), or value (for enums), a description of each feature (if any), and finally optional tagged sections.

Descriptions start with '@name:'. The description text must be indented like this:

```
# @name: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed
#     do eiusmod tempor incididunt ut labore et dolore magna aliqua.
```

Extensions added after the definition was first released carry a "(since x.y.z)" comment.

The feature descriptions must be preceded by a blank line and then a line "Features:", like this:

```
#
# Features:
#
# @feature: Description text
```

A tagged section begins with a paragraph that starts with one of the following words: "Note:"/"Notes:", "Since:", "Example:"/"Examples:", "Returns:", "Errors:", "TODO:". It ends with the start of a new section.

The second and subsequent lines of tagged sections must be indented like this:

```
# Note: Ut enim ad minim veniam, quis nostrud exercitation ullamco
#     laboris nisi ut aliquip ex ea commodo consequat.
#
#     Duis aute irure dolor in reprehenderit in voluptate velit esse
#     cillum dolore eu fugiat nulla pariatur.
```

"Returns" and "Errors" sections are only valid for commands. They document the success and the error response, respectively.

A "Since: x.y.z" tagged section lists the release that introduced the definition.

An "Example" or "Examples" section is rendered entirely as literal fixed-width text. "TODO" sections are not rendered at all (they are for developers, not users of QMP). In other sections, the text is formatted, and rST markup can be used.

For example:

```
##
# @BlockStats:
#
# Statistics of a virtual block device or a block backing device.
#
# @device: If the stats are for a virtual block device, the name
#     corresponding to the virtual block device.
#
# @node-name: The node name of the device.  (Since 2.3)
#
# ... more members ...
#
# Since: 0.14
##
{ 'struct': 'BlockStats',
  'data': {'*device': 'str', '*node-name': 'str',
           ... more members ... } }

##
# @query-blockstats:
#
# Query the @BlockStats for all virtual block devices.
#
# @query-nodes: If true, the command will query all the block nodes
#     ... explain, explain ...
#     (Since 2.3)
#
# Returns: A list of @BlockStats for each virtual block devices.
#
# Since: 0.14
#
# Example:
#
#     -> { "execute": "query-blockstats" }
#     <- {
#           ... lots of output ...
#        }
##
```

(continues on next page)

(continued from previous page)

```
{ 'command': 'query-blockstats',
  'data': { '*query-nodes': 'bool' },
  'returns': ['BlockStats'] }
```

Markup pitfalls

A blank line is required between list items and paragraphs. Without it, the list may not be recognized, resulting in garbled output. Good example:

```
# An event's state is modified if:
#
# - its name matches the @name pattern, and
# - if @vcpu is given, the event has the "vcpu" property.
```

Without the blank line this would be a single paragraph.

Indentation matters. Bad example:

```
# @none: None (no memory side cache in this proximity domain,
#              or cache associativity unknown)
#     (since 5.0)
```

The last line's de-indent is wrong. The second and subsequent lines need to line up with each other, like this:

```
# @none: None (no memory side cache in this proximity domain,
#     or cache associativity unknown)
#     (since 5.0)
```

Section tags are case-sensitive and end with a colon. They are only recognized after a blank line. Good example:

```
#
# Since: 7.1
```

Bad examples (all ordinary paragraphs):

```
# since: 7.1

# Since 7.1

# Since : 7.1
```

Likewise, member descriptions require a colon. Good example:

```
# @interface-id: Interface ID
```

Bad examples (all ordinary paragraphs):

```
# @interface-id   Interface ID

# @interface-id : Interface ID
```

Undocumented members are not flagged, yet. Instead, the generated documentation describes them as "Not documented". Think twice before adding more undocumented members.

When you change documentation comments, please check the generated documentation comes out as intended!

Client JSON Protocol introspection

Clients of a Client JSON Protocol commonly need to figure out what exactly the server (QEMU) supports.

For this purpose, QMP provides introspection via command query-qmp-schema. QGA currently doesn't support introspection.

While Client JSON Protocol wire compatibility should be maintained between qemu versions, we cannot make the same guarantees for introspection stability. For example, one version of qemu may provide a non-variant optional member of a struct, and a later version rework the member to instead be non-optional and associated with a variant. Likewise, one version of qemu may list a member with open-ended type 'str', and a later version could convert it to a finite set of strings via an enum type; or a member may be converted from a specific type to an alternate that represents a choice between the original type and something else.

query-qmp-schema returns a JSON array of SchemaInfo objects. These objects together describe the wire ABI, as defined in the QAPI schema. There is no specified order to the SchemaInfo objects returned; a client must search for a particular name throughout the entire array to learn more about that name, but is at least guaranteed that there will be no collisions between type, command, and event names.

However, the SchemaInfo can't reflect all the rules and restrictions that apply to QMP. It's interface introspection (figuring out what's there), not interface specification. The specification is in the QAPI schema. To understand how QMP is to be used, you need to study the QAPI schema.

Like any other command, query-qmp-schema is itself defined in the QAPI schema, along with the SchemaInfo type. This text attempts to give an overview how things work. For details you need to consult the QAPI schema.

SchemaInfo objects have common members "name", "meta-type", "features", and additional variant members depending on the value of meta-type.

Each SchemaInfo object describes a wire ABI entity of a certain meta-type: a command, event or one of several kinds of type.

SchemaInfo for commands and events have the same name as in the QAPI schema.

Command and event names are part of the wire ABI, but type names are not. Therefore, the SchemaInfo for types have auto-generated meaningless names. For readability, the examples in this section use meaningful type names instead.

Optional member "features" exposes the entity's feature strings as a JSON array of strings.

To examine a type, start with a command or event using it, then follow references by name.

QAPI schema definitions not reachable that way are omitted.

The SchemaInfo for a command has meta-type "command", and variant members "arg-type", "ret-type" and "allow-oob". On the wire, the "arguments" member of a client's "execute" command must conform to the object type named by "arg-type". The "return" member that the server passes in a success response conforms to the type named by "ret-type". When "allow-oob" is true, it means the command supports out-of-band execution. It defaults to false.

If the command takes no arguments, "arg-type" names an object type without members. Likewise, if the command returns nothing, "ret-type" names an object type without members.

Example: the SchemaInfo for command query-qmp-schema

```
{ "name": "query-qmp-schema", "meta-type": "command",
  "arg-type": "q_empty", "ret-type": "SchemaInfoList" }

  Type "q_empty" is an automatic object type without members, and type
  "SchemaInfoList" is the array of SchemaInfo type.
```

The SchemaInfo for an event has meta-type "event", and variant member "arg-type". On the wire, a "data" member that the server passes in an event conforms to the object type named by "arg-type".

If the event carries no additional information, "arg-type" names an object type without members. The event may not have a data member on the wire then.

Each command or event defined with 'data' as MEMBERS object in the QAPI schema implicitly defines an object type.

Example: the SchemaInfo for EVENT_C from section *Events*

```
{ "name": "EVENT_C", "meta-type": "event",
  "arg-type": "q_obj-EVENT_C-arg" }

Type "q_obj-EVENT_C-arg" is an implicitly defined object type with
the two members from the event's definition.
```

The SchemaInfo for struct and union types has meta-type "object" and variant member "members".

The SchemaInfo for a union type additionally has variant members "tag" and "variants".

"members" is a JSON array describing the object's common members, if any. Each element is a JSON object with members "name" (the member's name), "type" (the name of its type), "features" (a JSON array of feature strings), and "default". The latter two are optional. The member is optional if "default" is present. Currently, "default" can only have value null. Other values are reserved for future extensions. The "members" array is in no particular order; clients must search the entire object when learning whether a particular member is supported.

Example: the SchemaInfo for MyType from section *Struct types*

```
{ "name": "MyType", "meta-type": "object",
  "members": [
      { "name": "member1", "type": "str" },
      { "name": "member2", "type": "int" },
      { "name": "member3", "type": "str", "default": null } ] }
```

"features" exposes the command's feature strings as a JSON array of strings.

Example: the SchemaInfo for TestType from section *Features*:

```
{ "name": "TestType", "meta-type": "object",
  "members": [
      { "name": "number", "type": "int" } ],
  "features": ["allow-negative-numbers"] }
```

"tag" is the name of the common member serving as type tag. "variants" is a JSON array describing the object's variant members. Each element is a JSON object with members "case" (the value of type tag this element applies to) and "type" (the name of an object type that provides the variant members for this type tag value). The "variants" array is in no particular order, and is not guaranteed to list cases in the same order as the corresponding "tag" enum type.

Example: the SchemaInfo for union BlockdevOptions from section *Union types*

```
{ "name": "BlockdevOptions", "meta-type": "object",
  "members": [
      { "name": "driver", "type": "BlockdevDriver" },
      { "name": "read-only", "type": "bool", "default": null } ],
  "tag": "driver",
  "variants": [
```

(continues on next page)

(continued from previous page)

```
      { "case": "file", "type": "BlockdevOptionsFile" },
      { "case": "qcow2", "type": "BlockdevOptionsQcow2" } ] }
```

Note that base types are "flattened": its members are included in the "members" array.

The SchemaInfo for an alternate type has meta-type "alternate", and variant member "members". "members" is a JSON array. Each element is a JSON object with member "type", which names a type. Values of the alternate type conform to exactly one of its member types. There is no guarantee on the order in which "members" will be listed.

Example: the SchemaInfo for BlockdevRef from section *Alternate types*

```
{ "name": "BlockdevRef", "meta-type": "alternate",
  "members": [
      { "type": "BlockdevOptions" },
      { "type": "str" } ] }
```

The SchemaInfo for an array type has meta-type "array", and variant member "element-type", which names the array's element type. Array types are implicitly defined. For convenience, the array's name may resemble the element type; however, clients should examine member "element-type" instead of making assumptions based on parsing member "name".

Example: the SchemaInfo for ['str']

```
{ "name": "[str]", "meta-type": "array",
  "element-type": "str" }
```

The SchemaInfo for an enumeration type has meta-type "enum" and variant member "members".

"members" is a JSON array describing the enumeration values. Each element is a JSON object with member "name" (the member's name), and optionally "features" (a JSON array of feature strings). The "members" array is in no particular order; clients must search the entire array when learning whether a particular value is supported.

Example: the SchemaInfo for MyEnum from section *Enumeration types*

```
{ "name": "MyEnum", "meta-type": "enum",
  "members": [
    { "name": "value1" },
    { "name": "value2" },
    { "name": "value3" }
  ] }
```

The SchemaInfo for a built-in type has the same name as the type in the QAPI schema (see section *Built-in Types*), with one exception detailed below. It has variant member "json-type" that shows how values of this type are encoded on the wire.

Example: the SchemaInfo for str

```
{ "name": "str", "meta-type": "builtin", "json-type": "string" }
```

The QAPI schema supports a number of integer types that only differ in how they map to C. They are identical as far as SchemaInfo is concerned. Therefore, they get all mapped to a single type "int" in SchemaInfo.

As explained above, type names are not part of the wire ABI. Not even the names of built-in types. Clients should examine member "json-type" instead of hard-coding names of built-in types.

Compatibility considerations

Maintaining backward compatibility at the Client JSON Protocol level while evolving the schema requires some care. This section is about syntactic compatibility, which is necessary, but not sufficient, for actual compatibility.

Clients send commands with argument data, and receive command responses with return data and events with event data.

Adding opt-in functionality to the send direction is backwards compatible: adding commands, optional arguments, enumeration values, union and alternate branches; turning an argument type into an alternate of that type; making mandatory arguments optional. Clients oblivious of the new functionality continue to work.

Incompatible changes include removing commands, command arguments, enumeration values, union and alternate branches, adding mandatory command arguments, and making optional arguments mandatory.

The specified behavior of an absent optional argument should remain the same. With proper documentation, this policy still allows some flexibility; for example, when an optional 'buffer-size' argument is specified to default to a sensible buffer size, the actual default value can still be changed. The specified default behavior is not the exact size of the buffer, only that the default size is sensible.

Adding functionality to the receive direction is generally backwards compatible: adding events, adding return and event data members. Clients are expected to ignore the ones they don't know.

Removing "unreachable" stuff like events that can't be triggered anymore, optional return or event data members that can't be sent anymore, and return or event data member (enumeration) values that can't be sent anymore makes no difference to clients, except for introspection. The latter can conceivably confuse clients, so tread carefully.

Incompatible changes include removing return and event data members.

Any change to a command definition's 'data' or one of the types used there (recursively) needs to consider send direction compatibility.

Any change to a command definition's 'return', an event definition's 'data', or one of the types used there (recursively) needs to consider receive direction compatibility.

Any change to types used in both contexts need to consider both.

Enumeration type values and complex and alternate type members may be reordered freely. For enumerations and alternate types, this doesn't affect the wire encoding. For complex types, this might make the implementation emit JSON object members in a different order, which the Client JSON Protocol permits.

Since type names are not visible in the Client JSON Protocol, types may be freely renamed. Even certain refactorings are invisible, such as splitting members from one type into a common base type.

Code generation

The QAPI code generator qapi-gen.py generates code and documentation from the schema. Together with the core QAPI libraries, this code provides everything required to take JSON commands read in by a Client JSON Protocol server, unmarshal the arguments into the underlying C types, call into the corresponding C function, map the response back to a Client JSON Protocol response to be returned to the user, and introspect the commands.

As an example, we'll use the following schema, which describes a single complex user-defined type, along with command which takes a list of that type as a parameter, and returns a single element of that type. The user is responsible for writing the implementation of qmp_my_command(); everything else is produced by the generator.

```
$ cat example-schema.json
{ 'struct': 'UserDefOne',
  'data': { 'integer': 'int', '*string': 'str', '*flag': 'bool' } }
```

(continues on next page)

(continued from previous page)

```
{ 'command': 'my-command',
  'data': { 'arg1': ['UserDefOne'] },
  'returns': 'UserDefOne' }

{ 'event': 'MY_EVENT' }
```

We run qapi-gen.py like this:

```
$ python scripts/qapi-gen.py --output-dir="qapi-generated" \
--prefix="example-" example-schema.json
```

For a more thorough look at generated code, the testsuite includes tests/qapi-schema/qapi-schema-tests.json that covers more examples of what the generator will accept, and compiles the resulting C code as part of 'make check-unit'.

Code generated for QAPI types

The following files are created:

$(prefix)qapi-types.h
: C types corresponding to types defined in the schema

$(prefix)qapi-types.c
: Cleanup functions for the above C types

The $(prefix) is an optional parameter used as a namespace to keep the generated code from one schema/code-generation separated from others so code can be generated/used from multiple schemas without clobbering previously created code.

Example:

```
$ cat qapi-generated/example-qapi-types.h
[Uninteresting stuff omitted...]

#ifndef EXAMPLE_QAPI_TYPES_H
#define EXAMPLE_QAPI_TYPES_H

#include "qapi/qapi-builtin-types.h"

typedef struct UserDefOne UserDefOne;

typedef struct UserDefOneList UserDefOneList;

typedef struct q_obj_my_command_arg q_obj_my_command_arg;

struct UserDefOne {
    int64_t integer;
    char *string;
    bool has_flag;
    bool flag;
};

void qapi_free_UserDefOne(UserDefOne *obj);
G_DEFINE_AUTOPTR_CLEANUP_FUNC(UserDefOne, qapi_free_UserDefOne)
```

(continues on next page)

(continued from previous page)

```
struct UserDefOneList {
    UserDefOneList *next;
    UserDefOne *value;
};

void qapi_free_UserDefOneList(UserDefOneList *obj);
G_DEFINE_AUTOPTR_CLEANUP_FUNC(UserDefOneList, qapi_free_UserDefOneList)

struct q_obj_my_command_arg {
    UserDefOneList *arg1;
};

#endif /* EXAMPLE_QAPI_TYPES_H */
$ cat qapi-generated/example-qapi-types.c
[Uninteresting stuff omitted...]

void qapi_free_UserDefOne(UserDefOne *obj)
{
    Visitor *v;

    if (!obj) {
        return;
    }

    v = qapi_dealloc_visitor_new();
    visit_type_UserDefOne(v, NULL, &obj, NULL);
    visit_free(v);
}

void qapi_free_UserDefOneList(UserDefOneList *obj)
{
    Visitor *v;

    if (!obj) {
        return;
    }

    v = qapi_dealloc_visitor_new();
    visit_type_UserDefOneList(v, NULL, &obj, NULL);
    visit_free(v);
}

[Uninteresting stuff omitted...]
```

For a modular QAPI schema (see section *Include directives*), code for each sub-module SUBDIR/SUBMODULE.json is actually generated into

```
SUBDIR/$(prefix)qapi-types-SUBMODULE.h
SUBDIR/$(prefix)qapi-types-SUBMODULE.c
```

If qapi-gen.py is run with option –builtins, additional files are created:

qapi-builtin-types.h

C types corresponding to built-in types

qapi-builtin-types.c
Cleanup functions for the above C types

Code generated for visiting QAPI types

These are the visitor functions used to walk through and convert between a native QAPI C data structure and some other format (such as QObject); the generated functions are named visit_type_FOO() and visit_type_FOO_members().

The following files are generated:

$(prefix)qapi-visit.c
Visitor function for a particular C type, used to automagically convert QObjects into the corresponding C type and vice-versa, as well as for deallocating memory for an existing C type

$(prefix)qapi-visit.h
Declarations for previously mentioned visitor functions

Example:

```
$ cat qapi-generated/example-qapi-visit.h
[Uninteresting stuff omitted...]

#ifndef EXAMPLE_QAPI_VISIT_H
#define EXAMPLE_QAPI_VISIT_H

#include "qapi/qapi-builtin-visit.h"
#include "example-qapi-types.h"


bool visit_type_UserDefOne_members(Visitor *v, UserDefOne *obj, Error **errp);

bool visit_type_UserDefOne(Visitor *v, const char *name,
                 UserDefOne **obj, Error **errp);

bool visit_type_UserDefOneList(Visitor *v, const char *name,
                 UserDefOneList **obj, Error **errp);

bool visit_type_q_obj_my_command_arg_members(Visitor *v, q_obj_my_command_arg *obj,␣
↪Error **errp);

#endif /* EXAMPLE_QAPI_VISIT_H */
$ cat qapi-generated/example-qapi-visit.c
[Uninteresting stuff omitted...]

bool visit_type_UserDefOne_members(Visitor *v, UserDefOne *obj, Error **errp)
{
    bool has_string = !!obj->string;

    if (!visit_type_int(v, "integer", &obj->integer, errp)) {
        return false;
    }
    if (visit_optional(v, "string", &has_string)) {
        if (!visit_type_str(v, "string", &obj->string, errp)) {
```

(continues on next page)

(continued from previous page)

```
            return false;
        }
    }
    if (visit_optional(v, "flag", &obj->has_flag)) {
        if (!visit_type_bool(v, "flag", &obj->flag, errp)) {
            return false;
        }
    }
    return true;
}

bool visit_type_UserDefOne(Visitor *v, const char *name,
                 UserDefOne **obj, Error **errp)
{
    bool ok = false;

    if (!visit_start_struct(v, name, (void **)obj, sizeof(UserDefOne), errp)) {
        return false;
    }
    if (!*obj) {
        /* incomplete */
        assert(visit_is_dealloc(v));
        ok = true;
        goto out_obj;
    }
    if (!visit_type_UserDefOne_members(v, *obj, errp)) {
        goto out_obj;
    }
    ok = visit_check_struct(v, errp);
out_obj:
    visit_end_struct(v, (void **)obj);
    if (!ok && visit_is_input(v)) {
        qapi_free_UserDefOne(*obj);
        *obj = NULL;
    }
    return ok;
}

bool visit_type_UserDefOneList(Visitor *v, const char *name,
                 UserDefOneList **obj, Error **errp)
{
    bool ok = false;
    UserDefOneList *tail;
    size_t size = sizeof(**obj);

    if (!visit_start_list(v, name, (GenericList **)obj, size, errp)) {
        return false;
    }

    for (tail = *obj; tail;
         tail = (UserDefOneList *)visit_next_list(v, (GenericList *)tail, size)) {
        if (!visit_type_UserDefOne(v, NULL, &tail->value, errp)) {
```

(continues on next page)

(continued from previous page)

```
            goto out_obj;
        }
    }

    ok = visit_check_list(v, errp);
out_obj:
    visit_end_list(v, (void **)obj);
    if (!ok && visit_is_input(v)) {
        qapi_free_UserDefOneList(*obj);
        *obj = NULL;
    }
    return ok;
}

bool visit_type_q_obj_my_command_arg_members(Visitor *v, q_obj_my_command_arg *obj,␣
↪Error **errp)
{
    if (!visit_type_UserDefOneList(v, "arg1", &obj->arg1, errp)) {
        return false;
    }
    return true;
}

[Uninteresting stuff omitted...]
```

For a modular QAPI schema (see section *Include directives*), code for each sub-module SUBDIR/SUBMODULE.json is actually generated into

```
SUBDIR/$(prefix)qapi-visit-SUBMODULE.h
SUBDIR/$(prefix)qapi-visit-SUBMODULE.c
```

If qapi-gen.py is run with option –builtins, additional files are created:

qapi-builtin-visit.h
: Visitor functions for built-in types

qapi-builtin-visit.c
: Declarations for these visitor functions

Code generated for commands

These are the marshaling/dispatch functions for the commands defined in the schema. The generated code provides qmp_marshal_COMMAND(), and declares qmp_COMMAND() that the user must implement.

The following files are generated:

$(prefix)qapi-commands.c
: Command marshal/dispatch functions for each QMP command defined in the schema

$(prefix)qapi-commands.h
: Function prototypes for the QMP commands specified in the schema

$(prefix)qapi-commands.trace-events
: Trace event declarations, see *Tracing*.

$(prefix)qapi-init-commands.h
Command initialization prototype

$(prefix)qapi-init-commands.c
Command initialization code

Example:

```
$ cat qapi-generated/example-qapi-commands.h
[Uninteresting stuff omitted...]

#ifndef EXAMPLE_QAPI_COMMANDS_H
#define EXAMPLE_QAPI_COMMANDS_H

#include "example-qapi-types.h"

UserDefOne *qmp_my_command(UserDefOneList *arg1, Error **errp);
void qmp_marshal_my_command(QDict *args, QObject **ret, Error **errp);

#endif /* EXAMPLE_QAPI_COMMANDS_H */

$ cat qapi-generated/example-qapi-commands.trace-events
# AUTOMATICALLY GENERATED, DO NOT MODIFY

qmp_enter_my_command(const char *json) "%s"
qmp_exit_my_command(const char *result, bool succeeded) "%s %d"

$ cat qapi-generated/example-qapi-commands.c
[Uninteresting stuff omitted...]

static void qmp_marshal_output_UserDefOne(UserDefOne *ret_in,
                                  QObject **ret_out, Error **errp)
{
    Visitor *v;

    v = qobject_output_visitor_new_qmp(ret_out);
    if (visit_type_UserDefOne(v, "unused", &ret_in, errp)) {
        visit_complete(v, ret_out);
    }
    visit_free(v);
    v = qapi_dealloc_visitor_new();
    visit_type_UserDefOne(v, "unused", &ret_in, NULL);
    visit_free(v);
}

void qmp_marshal_my_command(QDict *args, QObject **ret, Error **errp)
{
    Error *err = NULL;
    bool ok = false;
    Visitor *v;
    UserDefOne *retval;
    q_obj_my_command_arg arg = {0};

    v = qobject_input_visitor_new_qmp(QOBJECT(args));
```

(continues on next page)

(continued from previous page)

```
    if (!visit_start_struct(v, NULL, NULL, 0, errp)) {
        goto out;
    }
    if (visit_type_q_obj_my_command_arg_members(v, &arg, errp)) {
        ok = visit_check_struct(v, errp);
    }
    visit_end_struct(v, NULL);
    if (!ok) {
        goto out;
    }

    if (trace_event_get_state_backends(TRACE_QMP_ENTER_MY_COMMAND)) {
        g_autoptr(GString) req_json = qobject_to_json(QOBJECT(args));

        trace_qmp_enter_my_command(req_json->str);
    }

    retval = qmp_my_command(arg.arg1, &err);
    if (err) {
        trace_qmp_exit_my_command(error_get_pretty(err), false);
        error_propagate(errp, err);
        goto out;
    }

    qmp_marshal_output_UserDefOne(retval, ret, errp);

    if (trace_event_get_state_backends(TRACE_QMP_EXIT_MY_COMMAND)) {
        g_autoptr(GString) ret_json = qobject_to_json(*ret);

        trace_qmp_exit_my_command(ret_json->str, true);
    }

out:
    visit_free(v);
    v = qapi_dealloc_visitor_new();
    visit_start_struct(v, NULL, NULL, 0, NULL);
    visit_type_q_obj_my_command_arg_members(v, &arg, NULL);
    visit_end_struct(v, NULL);
    visit_free(v);
}

[Uninteresting stuff omitted...]
$ cat qapi-generated/example-qapi-init-commands.h
[Uninteresting stuff omitted...]
#ifndef EXAMPLE_QAPI_INIT_COMMANDS_H
#define EXAMPLE_QAPI_INIT_COMMANDS_H

#include "qapi/qmp/dispatch.h"

void example_qmp_init_marshal(QmpCommandList *cmds);

#endif /* EXAMPLE_QAPI_INIT_COMMANDS_H */
```

(continues on next page)

(continued from previous page)

```
$ cat qapi-generated/example-qapi-init-commands.c
[Uninteresting stuff omitted...]
void example_qmp_init_marshal(QmpCommandList *cmds)
{
    QTAILQ_INIT(cmds);

    qmp_register_command(cmds, "my-command",
                         qmp_marshal_my_command, 0, 0);
}
[Uninteresting stuff omitted...]
```

For a modular QAPI schema (see section *Include directives*), code for each sub-module SUBDIR/SUBMODULE.json is actually generated into:

```
SUBDIR/$(prefix)qapi-commands-SUBMODULE.h
SUBDIR/$(prefix)qapi-commands-SUBMODULE.c
```

Code generated for events

This is the code related to events defined in the schema, providing qapi_event_send_EVENT().

The following files are created:

$(prefix)qapi-events.h
: Function prototypes for each event type

$(prefix)qapi-events.c
: Implementation of functions to send an event

$(prefix)qapi-emit-events.h
: Enumeration of all event names, and common event code declarations

$(prefix)qapi-emit-events.c
: Common event code definitions

Example:

```
$ cat qapi-generated/example-qapi-events.h
[Uninteresting stuff omitted...]

#ifndef EXAMPLE_QAPI_EVENTS_H
#define EXAMPLE_QAPI_EVENTS_H

#include "qapi/util.h"
#include "example-qapi-types.h"

void qapi_event_send_my_event(void);

#endif /* EXAMPLE_QAPI_EVENTS_H */
$ cat qapi-generated/example-qapi-events.c
[Uninteresting stuff omitted...]

void qapi_event_send_my_event(void)
{
```

(continues on next page)

(continued from previous page)

```
    QDict *qmp;

    qmp = qmp_event_build_dict("MY_EVENT");

    example_qapi_event_emit(EXAMPLE_QAPI_EVENT_MY_EVENT, qmp);

    qobject_unref(qmp);
}

[Uninteresting stuff omitted...]
$ cat qapi-generated/example-qapi-emit-events.h
[Uninteresting stuff omitted...]

#ifndef EXAMPLE_QAPI_EMIT_EVENTS_H
#define EXAMPLE_QAPI_EMIT_EVENTS_H

#include "qapi/util.h"

typedef enum example_QAPIEvent {
    EXAMPLE_QAPI_EVENT_MY_EVENT,
    EXAMPLE_QAPI_EVENT__MAX,
} example_QAPIEvent;

#define example_QAPIEvent_str(val) \
    qapi_enum_lookup(&example_QAPIEvent_lookup, (val))

extern const QEnumLookup example_QAPIEvent_lookup;

void example_qapi_event_emit(example_QAPIEvent event, QDict *qdict);

#endif /* EXAMPLE_QAPI_EMIT_EVENTS_H */
$ cat qapi-generated/example-qapi-emit-events.c
[Uninteresting stuff omitted...]

const QEnumLookup example_QAPIEvent_lookup = {
    .array = (const char *const[]) {
        [EXAMPLE_QAPI_EVENT_MY_EVENT] = "MY_EVENT",
    },
    .size = EXAMPLE_QAPI_EVENT__MAX
};

[Uninteresting stuff omitted...]
```

For a modular QAPI schema (see section *Include directives*), code for each sub-module SUBDIR/SUBMODULE.json is actually generated into

```
SUBDIR/$(prefix)qapi-events-SUBMODULE.h
SUBDIR/$(prefix)qapi-events-SUBMODULE.c
```

Code generated for introspection

The following files are created:

$(prefix)qapi-introspect.c
: Defines a string holding a JSON description of the schema

$(prefix)qapi-introspect.h
: Declares the above string

Example:

```
$ cat qapi-generated/example-qapi-introspect.h
[Uninteresting stuff omitted...]

#ifndef EXAMPLE_QAPI_INTROSPECT_H
#define EXAMPLE_QAPI_INTROSPECT_H

#include "qapi/qmp/qlit.h"

extern const QLitObject example_qmp_schema_qlit;

#endif /* EXAMPLE_QAPI_INTROSPECT_H */
$ cat qapi-generated/example-qapi-introspect.c
[Uninteresting stuff omitted...]

const QLitObject example_qmp_schema_qlit = QLIT_QLIST(((QLitObject[]) {
    QLIT_QDICT(((QLitDictEntry[]) {
        { "arg-type", QLIT_QSTR("0"), },
        { "meta-type", QLIT_QSTR("command"), },
        { "name", QLIT_QSTR("my-command"), },
        { "ret-type", QLIT_QSTR("1"), },
        {}
    })),
    QLIT_QDICT(((QLitDictEntry[]) {
        { "arg-type", QLIT_QSTR("2"), },
        { "meta-type", QLIT_QSTR("event"), },
        { "name", QLIT_QSTR("MY_EVENT"), },
        {}
    })),
    /* "0" = q_obj_my-command-arg */
    QLIT_QDICT(((QLitDictEntry[]) {
        { "members", QLIT_QLIST(((QLitObject[]) {
            QLIT_QDICT(((QLitDictEntry[]) {
                { "name", QLIT_QSTR("arg1"), },
                { "type", QLIT_QSTR("[1]"), },
                {}
            })),
            {}
        })), },
        { "meta-type", QLIT_QSTR("object"), },
        { "name", QLIT_QSTR("0"), },
        {}
    })),
```

(continues on next page)

(continued from previous page)

```
    /* "1" = UserDefOne */
    QLIT_QDICT(((QLitDictEntry[]) {
        { "members", QLIT_QLIST(((QLitObject[]) {
            QLIT_QDICT(((QLitDictEntry[]) {
                { "name", QLIT_QSTR("integer"), },
                { "type", QLIT_QSTR("int"), },
                {}
            })),
            QLIT_QDICT(((QLitDictEntry[]) {
                { "default", QLIT_QNULL, },
                { "name", QLIT_QSTR("string"), },
                { "type", QLIT_QSTR("str"), },
                {}
            })),
            QLIT_QDICT(((QLitDictEntry[]) {
                { "default", QLIT_QNULL, },
                { "name", QLIT_QSTR("flag"), },
                { "type", QLIT_QSTR("bool"), },
                {}
            })),
            {}
        })), },
        { "meta-type", QLIT_QSTR("object"), },
        { "name", QLIT_QSTR("1"), },
        {}
    })),
    /* "2" = q_empty */
    QLIT_QDICT(((QLitDictEntry[]) {
        { "members", QLIT_QLIST(((QLitObject[]) {
            {}
        })), },
        { "meta-type", QLIT_QSTR("object"), },
        { "name", QLIT_QSTR("2"), },
        {}
    })),
    QLIT_QDICT(((QLitDictEntry[]) {
        { "element-type", QLIT_QSTR("1"), },
        { "meta-type", QLIT_QSTR("array"), },
        { "name", QLIT_QSTR("[1]"), },
        {}
    })),
    QLIT_QDICT(((QLitDictEntry[]) {
        { "json-type", QLIT_QSTR("int"), },
        { "meta-type", QLIT_QSTR("builtin"), },
        { "name", QLIT_QSTR("int"), },
        {}
    })),
    QLIT_QDICT(((QLitDictEntry[]) {
        { "json-type", QLIT_QSTR("string"), },
        { "meta-type", QLIT_QSTR("builtin"), },
        { "name", QLIT_QSTR("str"), },
        {}
```

(continues on next page)

(continued from previous page)

```
    })),
    QLIT_QDICT(((QLitDictEntry[]) {
        { "json-type", QLIT_QSTR("boolean"), },
        { "meta-type", QLIT_QSTR("builtin"), },
        { "name", QLIT_QSTR("bool"), },
        {}
    })),
    {}
}));

[Uninteresting stuff omitted...]
```

7.2.9 Fuzzing

This document describes the virtual-device fuzzing infrastructure in QEMU and how to use it to implement additional fuzzers.

Basics

Fuzzing operates by passing inputs to an entry point/target function. The fuzzer tracks the code coverage triggered by the input. Based on these findings, the fuzzer mutates the input and repeats the fuzzing.

To fuzz QEMU, we rely on libfuzzer. Unlike other fuzzers such as AFL, libfuzzer is an *in-process* fuzzer. For the developer, this means that it is their responsibility to ensure that state is reset between fuzzing-runs.

Building the fuzzers

To build the fuzzers, install a recent version of clang: Configure with (substitute the clang binaries with the version you installed). Here, enable-sanitizers, is optional but it allows us to reliably detect bugs such as out-of-bounds accesses, use-after-frees, double-frees etc.:

```
CC=clang-8 CXX=clang++-8 /path/to/configure --enable-fuzzing \
                                            --enable-sanitizers
```

Fuzz targets are built similarly to system targets:

```
make qemu-fuzz-i386
```

This builds `./qemu-fuzz-i386`

The first option to this command is: `--fuzz-target=FUZZ_NAME` To list all of the available fuzzers run `qemu-fuzz-i386` with no arguments.

For example:

```
./qemu-fuzz-i386 --fuzz-target=virtio-scsi-fuzz
```

Internally, libfuzzer parses all arguments that do not begin with "`--`". Information about these is available by passing `-help=1`

Now the only thing left to do is wait for the fuzzer to trigger potential crashes.

Useful libFuzzer flags

As mentioned above, libFuzzer accepts some arguments. Passing `-help=1` will list the available arguments. In particular, these arguments might be helpful:

- `CORPUS_DIR/` : Specify a directory as the last argument to libFuzzer. libFuzzer stores each "interesting" input in this corpus directory. The next time you run libFuzzer, it will read all of the inputs from the corpus, and continue fuzzing from there. You can also specify multiple directories. libFuzzer loads existing inputs from all specified directories, but will only write new ones to the first one specified.
- `-max_len=4096` : specify the maximum byte-length of the inputs libFuzzer will generate.
- `-close_fd_mask={1,2,3}` : close, stderr, or both. Useful for targets that trigger many debug/error messages, or create output on the serial console.
- `-jobs=4 -workers=4` : These arguments configure libFuzzer to run 4 fuzzers in parallel (4 fuzzing jobs in 4 worker processes). Alternatively, with only `-jobs=N`, libFuzzer automatically spawns a number of workers less than or equal to half the available CPU cores. Replace 4 with a number appropriate for your machine. Make sure to specify a `CORPUS_DIR`, which will allow the parallel fuzzers to share information about the interesting inputs they find.
- `-use_value_profile=1` : For each comparison operation, libFuzzer computes `(caller_pc&4095) | (popcnt(Arg1 ^ Arg2) << 12)` and places this in the coverage table. Useful for targets with "magic" constants. If Arg1 came from the fuzzer's input and Arg2 is a magic constant, then each time the Hamming distance between Arg1 and Arg2 decreases, libFuzzer adds the input to the corpus.
- `-shrink=1` : Tries to make elements of the corpus "smaller". Might lead to better coverage performance, depending on the target.

Note that libFuzzer's exact behavior will depend on the version of clang and libFuzzer used to build the device fuzzers.

Generating Coverage Reports

Code coverage is a crucial metric for evaluating a fuzzer's performance. libFuzzer's output provides a "cov: " column that provides a total number of unique blocks/edges covered. To examine coverage on a line-by-line basis we can use Clang coverage:

1. Configure libFuzzer to store a corpus of all interesting inputs (see CORPUS_DIR above)
2. `./configure` the QEMU build with

```
--enable-fuzzing \
--extra-cflags="-fprofile-instr-generate -fcoverage-mapping"
```

3. Re-run the fuzzer. Specify $CORPUS_DIR/* as an argument, telling libfuzzer to execute all of the inputs in $CORPUS_DIR and exit. Once the process exits, you should find a file, "default.profraw" in the working directory.
4. Execute these commands to generate a detailed HTML coverage-report:

```
llvm-profdata merge -output=default.profdata default.profraw
llvm-cov show ./path/to/qemu-fuzz-i386 -instr-profile=default.profdata \
--format html -output-dir=/path/to/output/report
```

Adding a new fuzzer

Coverage over virtual devices can be improved by adding additional fuzzers. Fuzzers are kept in `tests/qtest/fuzz/` and should be added to `tests/qtest/fuzz/meson.build`

Fuzzers can rely on both qtest and libqos to communicate with virtual devices.

1. Create a new source file. For example `tests/qtest/fuzz/foo-device-fuzz.c`.
2. Write the fuzzing code using the libqtest/libqos API. See existing fuzzers for reference.
3. Add the fuzzer to `tests/qtest/fuzz/meson.build`.

Fuzzers can be more-or-less thought of as special qtest programs which can modify the qtest commands and/or qtest command arguments based on inputs provided by libfuzzer. Libfuzzer passes a byte array and length. Commonly the fuzzer loops over the byte-array interpreting it as a list of qtest commands, addresses, or values.

The Generic Fuzzer

Writing a fuzz target can be a lot of effort (especially if a device driver has not be built-out within libqos). Many devices can be fuzzed to some degree, without any device-specific code, using the generic-fuzz target.

The generic-fuzz target is capable of fuzzing devices over their PIO, MMIO, and DMA input-spaces. To apply the generic-fuzz to a device, we need to define two env-variables, at minimum:

- `QEMU_FUZZ_ARGS=` is the set of QEMU arguments used to configure a machine, with the device attached. For example, if we want to fuzz the virtio-net device attached to a pc-i440fx machine, we can specify:

```
QEMU_FUZZ_ARGS="-M pc -nodefaults -netdev user,id=user0 \
-device virtio-net,netdev=user0"
```

- `QEMU_FUZZ_OBJECTS=` is a set of space-delimited strings used to identify the MemoryRegions that will be fuzzed. These strings are compared against MemoryRegion names and MemoryRegion owner names, to decide whether each MemoryRegion should be fuzzed. These strings support globbing. For the virtio-net example, we could use one of

```
QEMU_FUZZ_OBJECTS='virtio-net'
QEMU_FUZZ_OBJECTS='virtio*'
QEMU_FUZZ_OBJECTS='virtio* pcspk' # Fuzz the virtio devices and the speaker
QEMU_FUZZ_OBJECTS='*' # Fuzz the whole machine``
```

The `"info mtree"` and `"info qom-tree"` monitor commands can be especially useful for identifying the `MemoryRegion` and `Object` names used for matching.

As a generic rule-of-thumb, the more `MemoryRegions`/Devices we match, the greater the input-space, and the smaller the probability of finding crashing inputs for individual devices. As such, it is usually a good idea to limit the fuzzer to only a few `MemoryRegions`.

To ensure that these env variables have been configured correctly, we can use:

```
./qemu-fuzz-i386 --fuzz-target=generic-fuzz -runs=0
```

The output should contain a complete list of matched MemoryRegions.

OSS-Fuzz

QEMU is continuously fuzzed on OSS-Fuzz. By default, the OSS-Fuzz build will try to fuzz every fuzz-target. Since the generic-fuzz target requires additional information provided in environment variables, we pre-define some generic-fuzz configs in `tests/qtest/fuzz/generic_fuzz_configs.h`. Each config must specify:

- `.name`: To identify the fuzzer config
- `.args` OR `.argfunc`: A string or pointer to a function returning a string. These strings are used to specify the `QEMU_FUZZ_ARGS` environment variable. `argfunc` is useful when the config relies on e.g. a dynamically created temp directory, or a free tcp/udp port.
- `.objects`: A string that specifies the `QEMU_FUZZ_OBJECTS` environment variable.

To fuzz additional devices/device configuration on OSS-Fuzz, send patches for either a new device-specific fuzzer or a new generic-fuzz config.

Build details:

- The Dockerfile that sets up the environment for building QEMU's fuzzers on OSS-Fuzz can be fund in the OSS-Fuzz repository __(https://github.com/google/oss-fuzz/blob/master/projects/qemu/Dockerfile)
- The script responsible for building the fuzzers can be found in the QEMU source tree at `scripts/oss-fuzz/build.sh`

Building Crash Reproducers

When we find a crash, we should try to create an independent reproducer, that can be used on a non-fuzzer build of QEMU. This filters out any potential false-positives, and improves the debugging experience for developers. Here are the steps for building a reproducer for a crash found by the generic-fuzz target.

- Ensure the crash reproduces:

```
qemu-fuzz-i386 --fuzz-target... ./crash-...
```

- Gather the QTest output for the crash:

```
QEMU_FUZZ_TIMEOUT=0 QTEST_LOG=1 FUZZ_SERIALIZE_QTEST=1 \
qemu-fuzz-i386 --fuzz-target... ./crash-... &> /tmp/trace
```

- Reorder and clean-up the resulting trace:

```
scripts/oss-fuzz/reorder_fuzzer_qtest_trace.py /tmp/trace > /tmp/reproducer
```

- Get the arguments needed to start qemu, and provide a path to qemu:

```
less /tmp/trace # The args should be logged at the top of this file
export QEMU_ARGS="-machine ..."
export QEMU_PATH="path/to/qemu-system"
```

- Ensure the crash reproduces in qemu-system:

```
$QEMU_PATH $QEMU_ARGS -qtest stdio < /tmp/reproducer
```

- From the crash output, obtain some string that identifies the crash. This can be a line in the stack-trace, for example:

```
export CRASH_TOKEN="hw/usb/hcd-xhci.c:1865"
```

- Minimize the reproducer:

```
scripts/oss-fuzz/minimize_qtest_trace.py -M1 -M2 \
  /tmp/reproducer /tmp/reproducer-minimized
```

- Confirm that the minimized reproducer still crashes:

```
$QEMU_PATH $QEMU_ARGS -qtest stdio < /tmp/reproducer-minimized
```

- Create a one-liner reproducer that can be sent over email:

```
./scripts/oss-fuzz/output_reproducer.py -bash /tmp/reproducer-minimized
```

- Output the C source code for a test case that will reproduce the bug:

```
./scripts/oss-fuzz/output_reproducer.py -owner "John Smith <john@smith.com>"\
  -name "test_function_name" /tmp/reproducer-minimized
```

- Report the bug and send a patch with the C reproducer upstream

Implementation Details / Fuzzer Lifecycle

The fuzzer has two entrypoints that libfuzzer calls. libfuzzer provides it's own `main()`, which performs some setup, and calls the entrypoints:

`LLVMFuzzerInitialize`: called prior to fuzzing. Used to initialize all of the necessary state

`LLVMFuzzerTestOneInput`: called for each fuzzing run. Processes the input and resets the state at the end of each run.

In more detail:

`LLVMFuzzerInitialize` parses the arguments to the fuzzer (must start with two dashes, so they are ignored by libfuzzer `main()`). Currently, the arguments select the fuzz target. Then, the qtest client is initialized. If the target requires qos, qgraph is set up and the QOM/LIBQOS modules are initialized. Then the QGraph is walked and the QEMU cmd_line is determined and saved.

After this, the `vl.c:main` is called to set up the guest. There are target-specific hooks that can be called before and after main, for additional setup(e.g. PCI setup, or VM snapshotting).

`LLVMFuzzerTestOneInput`: Uses qtest/qos functions to act based on the fuzz input. It is also responsible for manually calling `main_loop_wait` to ensure that bottom halves are executed and any cleanup required before the next input.

Since the same process is reused for many fuzzing runs, QEMU state needs to be reset at the end of each run. For example, this can be done by rebooting the VM, after each run.

- *Pros*: Straightforward and fast for simple fuzz targets.
- *Cons*: Depending on the device, does not reset all device state. If the device requires some initialization prior to being ready for fuzzing (common for QOS-based targets), this initialization needs to be done after each reboot.
- *Example target*: `i440fx-qtest-reboot-fuzz`

7.2.10 Control-Flow Integrity (CFI)

This document describes the current control-flow integrity (CFI) mechanism in QEMU. How it can be enabled, its benefits and deficiencies, and how it affects new and existing code in QEMU

Basics

CFI is a hardening technique that focusing on guaranteeing that indirect function calls have not been altered by an attacker. The type used in QEMU is a forward-edge control-flow integrity that ensures function calls performed through function pointers, always call a "compatible" function. A compatible function is a function with the same signature of the function pointer declared in the source code.

This type of CFI is entirely compiler-based and relies on the compiler knowing the signature of every function and every function pointer used in the code. As of now, the only compiler that provides support for CFI is Clang.

CFI is best used on production binaries, to protect against unknown attack vectors.

In case of a CFI violation (i.e. call to a non-compatible function) QEMU will terminate abruptly, to stop the possible attack.

Building with CFI

NOTE: CFI requires the use of link-time optimization. Therefore, when CFI is selected, LTO will be automatically enabled.

To build with CFI, the minimum requirement is Clang 6+. If you are planning to also enable fuzzing, then Clang 11+ is needed (more on this later).

Given the use of LTO, a version of AR that supports LLVM IR is required. The easies way of doing this is by selecting the AR provided by LLVM:

```
AR=llvm-ar-9 CC=clang-9 CXX=clang++-9 /path/to/configure --enable-cfi
```

CFI is enabled on every binary produced.

If desired, an additional flag to increase the verbosity of the output in case of a CFI violation is offered (`--enable-debug-cfi`).

Using QEMU built with CFI

A binary with CFI will work exactly like a standard binary. In case of a CFI violation, the binary will terminate with an illegal instruction signal.

Incompatible code with CFI

As mentioned above, CFI is entirely compiler-based and therefore relies on compile-time knowledge of the code. This means that, while generally supported for most code, some specific use pattern can break CFI compatibility, and create false-positives. The two main patterns that can cause issues are:

- Just-in-time compiled code: since such code is created at runtime, the jump to the buffer containing JIT code will fail.
- Libraries loaded dynamically, e.g. with dlopen/dlsym, since the library was not known at compile time.

Current areas of QEMU that are not entirely compatible with CFI are:

1. TCG, since the idea of TCG is to pre-compile groups of instructions at runtime to speed-up interpretation, quite similarly to a JIT compiler
2. TCI, where the interpreter has to interpret the generic *call* operation
3. Plugins, since a plugin is implemented as an external library
4. Modules, since they are implemented as an external library
5. Directly calling signal handlers from the QEMU source code, since the signal handler may have been provided by an external library or even plugged at runtime.

Disabling CFI for a specific function

If you are working on function that is performing a call using an incompatible way, as described before, you can selectively disable CFI checks for such function by using the decorator `QEMU_DISABLE_CFI` at function definition, and add an explanation on why the function is not compatible with CFI. An example of the use of `QEMU_DISABLE_CFI` is provided here:

```
/*
 * Disable CFI checks.
 * TCG creates binary blobs at runtime, with the transformed code.
 * A TB is a blob of binary code, created at runtime and called with an
 * indirect function call. Since such function did not exist at compile time,
 * the CFI runtime has no way to verify its signature and would fail.
 * TCG is not considered a security-sensitive part of QEMU so this does not
 * affect the impact of CFI in environment with high security requirements
 */
QEMU_DISABLE_CFI
static inline tcg_target_ulong cpu_tb_exec(CPUState *cpu, TranslationBlock *itb)
```

NOTE: CFI needs to be disabled at the **caller** function, (i.e. a compatible cfi function that calls a non-compatible one), since the check is performed when the function call is performed.

CFI and fuzzing

There is generally no advantage of using CFI and fuzzing together, because they target different environments (production for CFI, debug for fuzzing).

CFI could be used in conjunction with fuzzing to identify a broader set of bugs that may not end immediately in a segmentation fault or triggering an assertion. However, other sanitizers such as address and ub sanitizers can identify such bugs in a more precise way than CFI.

There is, however, an interesting use case in using CFI in conjunction with fuzzing, that is to make sure that CFI is not triggering any false positive in remote-but-possible parts of the code.

CFI can be enabled with fuzzing, but with some caveats: 1. Fuzzing relies on the linker performing function wrapping at link-time. The standard BFD linker does not support function wrapping when LTO is also enabled. The workaround is to use LLVM's lld linker. 2. Fuzzing also relies on a custom linker script, which is only supported by lld with version 11+.

In other words, to compile with fuzzing and CFI, clang 11+ is required, and lld needs to be used as a linker:

```
AR=llvm-ar-11 CC=clang-11 CXX=clang++-11 /path/to/configure --enable-cfi \
                          -enable-fuzzing --extra-ldflags="-fuse-ld=lld"
```

and then, compile the fuzzers as usual.

7.3 Internal QEMU APIs

Details about how QEMU's various internal APIs. Most of these are generated from in-code annotations to function prototypes.

7.3.1 Bitwise operations

The header `qemu/bitops.h` provides utility functions for performing bitwise operations.

void **set_bit**(long nr, unsigned long *addr)

Set a bit in memory

Parameters

long nr
the bit to set

unsigned long *addr
the address to start counting from

void **set_bit_atomic**(long nr, unsigned long *addr)

Set a bit in memory atomically

Parameters

long nr
the bit to set

unsigned long *addr
the address to start counting from

void **clear_bit**(long nr, unsigned long *addr)

Clears a bit in memory

Parameters

long nr
Bit to clear

unsigned long *addr
Address to start counting from

void **clear_bit_atomic**(long nr, unsigned long *addr)

Clears a bit in memory atomically

Parameters

long nr
Bit to clear

unsigned long *addr
Address to start counting from

void **change_bit**(long nr, unsigned long *addr)

Toggle a bit in memory

Parameters

long nr
Bit to change

unsigned long *addr
Address to start counting from

int **test_and_set_bit**(long nr, unsigned long *addr)
Set a bit and return its old value

Parameters

long nr
Bit to set

unsigned long *addr
Address to count from

int **test_and_clear_bit**(long nr, unsigned long *addr)
Clear a bit and return its old value

Parameters

long nr
Bit to clear

unsigned long *addr
Address to count from

int **test_and_change_bit**(long nr, unsigned long *addr)
Change a bit and return its old value

Parameters

long nr
Bit to change

unsigned long *addr
Address to count from

int **test_bit**(long nr, const unsigned long *addr)
Determine whether a bit is set

Parameters

long nr
bit number to test

const unsigned long *addr
Address to start counting from

unsigned long **find_last_bit**(const unsigned long *addr, unsigned long size)
find the last set bit in a memory region

Parameters

const unsigned long *addr
The address to start the search at

unsigned long size
The maximum size to search

Description

Returns the bit number of the last set bit, or **size** if there is no set bit in the bitmap.

unsigned long **find_next_bit**(const unsigned long *addr, unsigned long size, unsigned long offset)

find the next set bit in a memory region

Parameters

`const unsigned long *addr`
The address to base the search on

`unsigned long size`
The bitmap size in bits

`unsigned long offset`
The bitnumber to start searching at

Description

Returns the bit number of the next set bit, or **size** if there are no further set bits in the bitmap.

unsigned long **find_next_zero_bit**(const unsigned long *addr, unsigned long size, unsigned long offset)

find the next cleared bit in a memory region

Parameters

`const unsigned long *addr`
The address to base the search on

`unsigned long size`
The bitmap size in bits

`unsigned long offset`
The bitnumber to start searching at

Description

Returns the bit number of the next cleared bit, or **size** if there are no further clear bits in the bitmap.

unsigned long **find_first_bit**(const unsigned long *addr, unsigned long size)

find the first set bit in a memory region

Parameters

`const unsigned long *addr`
The address to start the search at

`unsigned long size`
The maximum size to search

Description

Returns the bit number of the first set bit, or **size** if there is no set bit in the bitmap.

unsigned long **find_first_zero_bit**(const unsigned long *addr, unsigned long size)

find the first cleared bit in a memory region

Parameters

`const unsigned long *addr`
The address to start the search at

`unsigned long size`
The maximum size to search

Description

Returns the bit number of the first cleared bit, or **size** if there is no clear bit in the bitmap.

uint8_t **rol8**(uint8_t word, unsigned int shift)

rotate an 8-bit value left

Parameters

uint8_t word
value to rotate

unsigned int shift
bits to roll

uint8_t **ror8**(uint8_t word, unsigned int shift)

rotate an 8-bit value right

Parameters

uint8_t word
value to rotate

unsigned int shift
bits to roll

uint16_t **rol16**(uint16_t word, unsigned int shift)

rotate a 16-bit value left

Parameters

uint16_t word
value to rotate

unsigned int shift
bits to roll

uint16_t **ror16**(uint16_t word, unsigned int shift)

rotate a 16-bit value right

Parameters

uint16_t word
value to rotate

unsigned int shift
bits to roll

uint32_t **rol32**(uint32_t word, unsigned int shift)

rotate a 32-bit value left

Parameters

uint32_t word
value to rotate

unsigned int shift
bits to roll

uint32_t **ror32**(uint32_t word, unsigned int shift)

rotate a 32-bit value right

Parameters

uint32_t word
value to rotate

unsigned int shift
bits to roll

uint64_t **rol64**(uint64_t word, unsigned int shift)

rotate a 64-bit value left

Parameters

uint64_t word
value to rotate

unsigned int shift
bits to roll

uint64_t **ror64**(uint64_t word, unsigned int shift)

rotate a 64-bit value right

Parameters

uint64_t word
value to rotate

unsigned int shift
bits to roll

uint32_t **hswap32**(uint32_t h)

swap 16-bit halfwords within a 32-bit value

Parameters

uint32_t h
value to swap

uint64_t **hswap64**(uint64_t h)

swap 16-bit halfwords within a 64-bit value

Parameters

uint64_t h
value to swap

uint64_t **wswap64**(uint64_t h)

swap 32-bit words within a 64-bit value

Parameters

uint64_t h
value to swap

uint32_t **extract32**(uint32_t value, int start, int length)

Parameters

uint32_t value
the value to extract the bit field from

int start
the lowest bit in the bit field (numbered from 0)

int length
the length of the bit field

Description

Extract from the 32 bit input **value** the bit field specified by the **start** and **length** parameters, and return it. The bit field must lie entirely within the 32 bit word. It is valid to request that all 32 bits are returned (ie **length** 32 and **start** 0).

Return

the value of the bit field extracted from the input value.

uint8_t **extract8**(uint8_t value, int start, int length)

Parameters

`uint8_t value`
: the value to extract the bit field from

`int start`
: the lowest bit in the bit field (numbered from 0)

`int length`
: the length of the bit field

Description

Extract from the 8 bit input **value** the bit field specified by the **start** and **length** parameters, and return it. The bit field must lie entirely within the 8 bit word. It is valid to request that all 8 bits are returned (ie **length** 8 and **start** 0).

Return

the value of the bit field extracted from the input value.

uint16_t **extract16**(uint16_t value, int start, int length)

Parameters

`uint16_t value`
: the value to extract the bit field from

`int start`
: the lowest bit in the bit field (numbered from 0)

`int length`
: the length of the bit field

Description

Extract from the 16 bit input **value** the bit field specified by the **start** and **length** parameters, and return it. The bit field must lie entirely within the 16 bit word. It is valid to request that all 16 bits are returned (ie **length** 16 and **start** 0).

Return

the value of the bit field extracted from the input value.

uint64_t **extract64**(uint64_t value, int start, int length)

Parameters

`uint64_t value`
: the value to extract the bit field from

`int start`
: the lowest bit in the bit field (numbered from 0)

`int length`
: the length of the bit field

Description

Extract from the 64 bit input **value** the bit field specified by the **start** and **length** parameters, and return it. The bit field must lie entirely within the 64 bit word. It is valid to request that all 64 bits are returned (ie **length** 64 and **start** 0).

Return

the value of the bit field extracted from the input value.

int32_t **sextract32**(uint32_t value, int start, int length)

Parameters

`uint32_t value`
: the value to extract the bit field from

`int start`
: the lowest bit in the bit field (numbered from 0)

`int length`
: the length of the bit field

Description

Extract from the 32 bit input **value** the bit field specified by the **start** and **length** parameters, and return it, sign extended to an int32_t (ie with the most significant bit of the field propagated to all the upper bits of the return value). The bit field must lie entirely within the 32 bit word. It is valid to request that all 32 bits are returned (ie **length** 32 and **start** 0).

Return

the sign extended value of the bit field extracted from the input value.

int64_t **sextract64**(uint64_t value, int start, int length)

Parameters

`uint64_t value`
: the value to extract the bit field from

`int start`
: the lowest bit in the bit field (numbered from 0)

`int length`
: the length of the bit field

Description

Extract from the 64 bit input **value** the bit field specified by the **start** and **length** parameters, and return it, sign extended to an int64_t (ie with the most significant bit of the field propagated to all the upper bits of the return value). The bit field must lie entirely within the 64 bit word. It is valid to request that all 64 bits are returned (ie **length** 64 and **start** 0).

Return

the sign extended value of the bit field extracted from the input value.

uint32_t **deposit32**(uint32_t value, int start, int length, uint32_t fieldval)

Parameters

`uint32_t value`
: initial value to insert bit field into

`int start`
: the lowest bit in the bit field (numbered from 0)

int length
the length of the bit field

uint32_t fieldval
the value to insert into the bit field

Description

Deposit **fieldval** into the 32 bit **value** at the bit field specified by the **start** and **length** parameters, and return the modified **value**. Bits of **value** outside the bit field are not modified. Bits of **fieldval** above the least significant **length** bits are ignored. The bit field must lie entirely within the 32 bit word. It is valid to request that all 32 bits are modified (ie **length** 32 and **start** 0).

Return

the modified **value**.

uint64_t **deposit64**(uint64_t value, int start, int length, uint64_t fieldval)

Parameters

uint64_t value
initial value to insert bit field into

int start
the lowest bit in the bit field (numbered from 0)

int length
the length of the bit field

uint64_t fieldval
the value to insert into the bit field

Description

Deposit **fieldval** into the 64 bit **value** at the bit field specified by the **start** and **length** parameters, and return the modified **value**. Bits of **value** outside the bit field are not modified. Bits of **fieldval** above the least significant **length** bits are ignored. The bit field must lie entirely within the 64 bit word. It is valid to request that all 64 bits are modified (ie **length** 64 and **start** 0).

Return

the modified **value**.

uint32_t **half_shuffle32**(uint32_t x)

Parameters

uint32_t x
32-bit value (of which only the bottom 16 bits are of interest)

Description

Given an input value:

```
xxxx xxxx xxxx xxxx ABCD EFGH IJKL MNOP
```

return the value where the bottom 16 bits are spread out into the odd bits in the word, and the even bits are zeroed:

```
0A0B 0C0D 0E0F 0G0H 0I0J 0K0L 0M0N 0O0P
```

Any bits set in the top half of the input are ignored.

Return

the shuffled bits.

uint64_t **half_shuffle64**(uint64_t x)

Parameters

uint64_t x
64-bit value (of which only the bottom 32 bits are of interest)

Description

Given an input value:

```
xxxx xxxx xxxx .... xxxx xxxx ABCD EFGH IJKL MNOP QRST UVWX YZab cdef
```

return the value where the bottom 32 bits are spread out into the odd bits in the word, and the even bits are zeroed:

```
0A0B 0C0D 0E0F 0G0H 0I0J 0K0L 0M0N .... 0U0V 0W0X 0Y0Z 0a0b 0c0d 0e0f
```

Any bits set in the top half of the input are ignored.

Return

the shuffled bits.

uint32_t **half_unshuffle32**(uint32_t x)

Parameters

uint32_t x
32-bit value (of which only the odd bits are of interest)

Description

Given an input value:

```
xAxB xCxD xExF xGxH xIxJ xKxL xMxN xOxP
```

return the value where all the odd bits are compressed down into the low half of the word, and the high half is zeroed:

```
0000 0000 0000 0000 ABCD EFGH IJKL MNOP
```

Any even bits set in the input are ignored.

Return

the unshuffled bits.

uint64_t **half_unshuffle64**(uint64_t x)

Parameters

uint64_t x
64-bit value (of which only the odd bits are of interest)

Description

Given an input value:

```
xAxB xCxD xExF xGxH xIxJ xKxL xMxN .... xUxV xWxX xYxZ xaxb xcxd xexf
```

return the value where all the odd bits are compressed down into the low half of the word, and the high half is zeroed:

```
0000 0000 0000 .... 0000 0000 ABCD EFGH IJKL MNOP QRST UVWX YZab cdef
```

Any even bits set in the input are ignored.

Return

the unshuffled bits.

7.3.2 Load and Store APIs

QEMU internally has multiple families of functions for performing loads and stores. This document attempts to enumerate them all and indicate when to use them. It does not provide detailed documentation of each API – for that you should look at the documentation comments in the relevant header files.

ld*_p and st*_p

These functions operate on a host pointer, and should be used when you already have a pointer into host memory (corresponding to guest ram or a local buffer). They deal with doing accesses with the desired endianness and with correctly handling potentially unaligned pointer values.

Function names follow the pattern:

load: `ld{sign}{size}_{endian}_p(ptr)`

store: `st{size}_{endian}_p(ptr, val)`

sign

- (empty) : for 32 or 64 bit sizes
- `u` : unsigned
- `s` : signed

size

- `b` : 8 bits
- `w` : 16 bits
- `24` : 24 bits
- `l` : 32 bits
- `q` : 64 bits

endian

- `he` : host endian
- `be` : big endian
- `le` : little endian

The `_{endian}` infix is omitted for target-endian accesses.

The target endian accessors are only available to source files which are built per-target.

There are also functions which take the size as an argument:

load: `ldn{endian}_p(ptr, sz)`

which performs an unsigned load of `sz` bytes from `ptr` as an `{endian}` order value and returns it in a uint64_t.

store: `stn{endian}_p(ptr, sz, val)`

which stores `val` to `ptr` as an `{endian}` order value of size `sz` bytes.

Regexes for git grep:

- `\<ld[us]\?[bwlq]\(_[hbl]e\)\?_p\>`
- `\<st[bwlq]\(_[hbl]e\)\?_p\>`
- `\<st24\(_[hbl]e\)\?_p\>`
- `\<ldn_\([hbl]e\)\?_p\>`
- `\<stn_\([hbl]e\)\?_p\>`

cpu_{ld,st}*_mmu

These functions operate on a guest virtual address, plus a context known as a "mmu index" which controls how that virtual address is translated, plus a `MemOp` which contains alignment requirements among other things. The `MemOp` and mmu index are combined into a single argument of type `MemOpIdx`.

The meaning of the indexes are target specific, but specifying a particular index might be necessary if, for instance, the helper requires a "always as non-privileged" access rather than the default access for the current state of the guest CPU.

These functions may cause a guest CPU exception to be taken (e.g. for an alignment fault or MMU fault) which will result in guest CPU state being updated and control longjmp'ing out of the function call. They should therefore only be used in code that is implementing emulation of the guest CPU.

The `retaddr` parameter is used to control unwinding of the guest CPU state in case of a guest CPU exception. This is passed to `cpu_restore_state()`. Therefore the value should either be 0, to indicate that the guest CPU state is already synchronized, or the result of `GETPC()` from the top level `HELPER(foo)` function, which is a return address into the generated code[1].

Function names follow the pattern:

load: `cpu_ld{size}{end}_mmu(env, ptr, oi, retaddr)`

store: `cpu_st{size}{end}_mmu(env, ptr, val, oi, retaddr)`

size

- `b` : 8 bits
- `w` : 16 bits
- `l` : 32 bits
- `q` : 64 bits

end

- (empty) : for target endian, or 8 bit sizes
- `_be` : big endian
- `_le` : little endian

Regexes for git grep:

- `\<cpu_ld[bwlq]\(_[bl]e\)\?_mmu\>`
- `\<cpu_st[bwlq]\(_[bl]e\)\?_mmu\>`

[1] Note that `GETPC()` should be used with great care: calling it in other functions that are *not* the top level `HELPER(foo)` will cause unexpected behavior. Instead, the value of `GETPC()` should be read from the helper and passed if needed to the functions that the helper calls.

cpu_{ld,st}*_mmuidx_ra

These functions work like the `cpu_{ld,st}_mmu` functions except that the `mmuidx` parameter is not combined with a `MemOp`, and therefore there is no required alignment supplied or enforced.

Function names follow the pattern:

load: `cpu_ld{sign}{size}{end}_mmuidx_ra(env, ptr, mmuidx, retaddr)`

store: `cpu_st{size}{end}_mmuidx_ra(env, ptr, val, mmuidx, retaddr)`

sign

- (empty) : for 32 or 64 bit sizes
- `u` : unsigned
- `s` : signed

size

- `b` : 8 bits
- `w` : 16 bits
- `l` : 32 bits
- `q` : 64 bits

end

- (empty) : for target endian, or 8 bit sizes
- `_be` : big endian
- `_le` : little endian

Regexes for git grep:

- `\<cpu_ld[us]\?[bwlq]\(_[bl]e\)\?_mmuidx_ra\>`
- `\<cpu_st[bwlq]\(_[bl]e\)\?_mmuidx_ra\>`

cpu_{ld,st}*_data_ra

These functions work like the `cpu_{ld,st}_mmuidx_ra` functions except that the `mmuidx` parameter is taken from the current mode of the guest CPU, as determined by `cpu_mmu_index(env, false)`.

These are generally the preferred way to do accesses by guest virtual address from helper functions, unless the access should be performed with a context other than the default, or alignment should be enforced for the access.

Function names follow the pattern:

load: `cpu_ld{sign}{size}{end}_data_ra(env, ptr, ra)`

store: `cpu_st{size}{end}_data_ra(env, ptr, val, ra)`

sign

- (empty) : for 32 or 64 bit sizes
- `u` : unsigned
- `s` : signed

size

- `b` : 8 bits

- `w` : 16 bits
- `l` : 32 bits
- `q` : 64 bits

end

- (empty) : for target endian, or 8 bit sizes
- `_be` : big endian
- `_le` : little endian

Regexes for git grep:

- `\<cpu_ld[us]\?[bwlq]\(_[bl]e\)\?_data_ra\>`
- `\<cpu_st[bwlq]\(_[bl]e\)\?_data_ra\>`

cpu_{ld,st}*_data

These functions work like the `cpu_{ld,st}_data_ra` functions except that the `retaddr` parameter is 0, and thus does not unwind guest CPU state.

This means they must only be used from helper functions where the translator has saved all necessary CPU state. These functions are the right choice for calls made from hooks like the CPU `do_interrupt` hook or when you know for certain that the translator had to save all the CPU state anyway.

Function names follow the pattern:

load: `cpu_ld{sign}{size}{end}_data(env, ptr)`

store: `cpu_st{size}{end}_data(env, ptr, val)`

sign

- (empty) : for 32 or 64 bit sizes
- `u` : unsigned
- `s` : signed

size

- `b` : 8 bits
- `w` : 16 bits
- `l` : 32 bits
- `q` : 64 bits

end

- (empty) : for target endian, or 8 bit sizes
- `_be` : big endian
- `_le` : little endian

Regexes for git grep:

- `\<cpu_ld[us]\?[bwlq]\(_[bl]e\)\?_data\>`
- `\<cpu_st[bwlq]\(_[bl]e\)\?_data\+\>`

cpu_ld*_code

These functions perform a read for instruction execution. The `mmuidx` parameter is taken from the current mode of the guest CPU, as determined by `cpu_mmu_index(env, true)`. The `retaddr` parameter is 0, and thus does not unwind guest CPU state, because CPU state is always synchronized while translating instructions. Any guest CPU exception that is raised will indicate an instruction execution fault rather than a data read fault.

In general these functions should not be used directly during translation. There are wrapper functions that are to be used which also take care of plugins for tracing.

Function names follow the pattern:

load: `cpu_ld{sign}{size}_code(env, ptr)`

sign

- (empty) : for 32 or 64 bit sizes
- `u` : unsigned
- `s` : signed

size

- `b` : 8 bits
- `w` : 16 bits
- `l` : 32 bits
- `q` : 64 bits

Regexes for git grep:

- `\<cpu_ld[us]\?[bwlq]_code\>`

translator_ld*

These functions are a wrapper for `cpu_ld*_code` which also perform any actions required by any tracing plugins. They are only to be called during the translator callback `translate_insn`.

There is a set of functions ending in `_swap` which, if the parameter is true, returns the value in the endianness that is the reverse of the guest native endianness, as determined by `TARGET_BIG_ENDIAN`.

Function names follow the pattern:

load: `translator_ld{sign}{size}(env, ptr)`

swap: `translator_ld{sign}{size}_swap(env, ptr, swap)`

sign

- (empty) : for 32 or 64 bit sizes
- `u` : unsigned
- `s` : signed

size

- `b` : 8 bits
- `w` : 16 bits
- `l` : 32 bits
- `q` : 64 bits

Regexes for git grep:

- `\<translator_ld[us]\?[bwlq]\(_swap\)\?\>`

helper_{ld,st}*_mmu

These functions are intended primarily to be called by the code generated by the TCG backend. Like the `cpu_{ld,st}_mmu` functions they perform accesses by guest virtual address, with a given `MemOpIdx`.

They differ from `cpu_{ld,st}_mmu` in that they take the endianness of the operation only from the MemOpIdx, and loads extend the return value to the size of a host general register (`tcg_target_ulong`).

load: `helper_ld{sign}{size}_mmu(env, addr, opindex, retaddr)`

store: `helper_{size}_mmu(env, addr, val, opindex, retaddr)`

sign

- (empty) : for 32 or 64 bit sizes
- `u` : unsigned
- `s` : signed

size

- `b` : 8 bits
- `w` : 16 bits
- `l` : 32 bits
- `q` : 64 bits

Regexes for git grep:

- `\<helper_ld[us]\?[bwlq]_mmu\>`
- `\<helper_st[bwlq]_mmu\>`

address_space_*

These functions are the primary ones to use when emulating CPU or device memory accesses. They take an AddressSpace, which is the way QEMU defines the view of memory that a device or CPU has. (They generally correspond to being the "master" end of a hardware bus or bus fabric.)

Each CPU has an AddressSpace. Some kinds of CPU have more than one AddressSpace (for instance Arm guest CPUs have an AddressSpace for the Secure world and one for NonSecure if they implement TrustZone). Devices which can do DMA-type operations should generally have an AddressSpace. There is also a "system address space" which typically has all the devices and memory that all CPUs can see. (Some older device models use the "system address space" rather than properly modelling that they have an AddressSpace of their own.)

Functions are provided for doing byte-buffer reads and writes, and also for doing one-data-item loads and stores.

In all cases the caller provides a MemTxAttrs to specify bus transaction attributes, and can check whether the memory transaction succeeded using a MemTxResult return code.

`address_space_read(address_space, addr, attrs, buf, len)`

`address_space_write(address_space, addr, attrs, buf, len)`

`address_space_rw(address_space, addr, attrs, buf, len, is_write)`

`address_space_ld{sign}{size}_{endian}(address_space, addr, attrs, txresult)`

`address_space_st{size}_{endian}(address_space, addr, val, attrs, txresult)`

sign

- (empty) : for 32 or 64 bit sizes
- `u` : unsigned

(No signed load operations are provided.)

size

- `b` : 8 bits
- `w` : 16 bits
- `l` : 32 bits
- `q` : 64 bits

endian

- `le` : little endian
- `be` : big endian

The `_{endian}` suffix is omitted for byte accesses.

Regexes for git grep:

- `\<address_space_\(read\|write\|rw\)\>`
- `\<address_space_ldu\?[bwql]\(_[lb]e\)\?\>`
- `\<address_space_st[bwql]\(_[lb]e\)\?\>`

address_space_write_rom

This function performs a write by physical address like `address_space_write`, except that if the write is to a ROM then the ROM contents will be modified, even though a write by the guest CPU to the ROM would be ignored. This is used for non-guest writes like writes from the gdb debug stub or initial loading of ROM contents.

Note that portions of the write which attempt to write data to a device will be silently ignored – only real RAM and ROM will be written to.

Regexes for git grep:

- `address_space_write_rom`

{ld,st}*_phys

These are functions which are identical to `address_space_{ld,st}*`, except that they always pass `MEMTXATTRS_UNSPECIFIED` for the transaction attributes, and ignore whether the transaction succeeded or failed.

The fact that they ignore whether the transaction succeeded means they should not be used in new code, unless you know for certain that your code will only be used in a context where the CPU or device doing the access has no way to report such an error.

`load:  ld{sign}{size}_{endian}_phys`

`store:  st{size}_{endian}_phys`

sign

- (empty) : for 32 or 64 bit sizes
- u : unsigned

(No signed load operations are provided.)

size

- b : 8 bits
- w : 16 bits
- l : 32 bits
- q : 64 bits

endian

- le : little endian
- be : big endian

The _{endian}_ infix is omitted for byte accesses.

Regexes for git grep:

- \<ldu\?[bwlq]\(_[bl]e\)\?_phys\>
- \<st[bwlq]\(_[bl]e\)\?_phys\>

cpu_physical_memory_*

These are convenience functions which are identical to address_space_* but operate specifically on the system address space, always pass a MEMTXATTRS_UNSPECIFIED set of memory attributes and ignore whether the memory transaction succeeded or failed. For new code they are better avoided:

- there is likely to be behaviour you need to model correctly for a failed read or write operation
- a device should usually perform operations on its own AddressSpace rather than using the system address space

cpu_physical_memory_read

cpu_physical_memory_write

cpu_physical_memory_rw

Regexes for git grep:

- \<cpu_physical_memory_\(read\|write\|rw\)\>

cpu_memory_rw_debug

Access CPU memory by virtual address for debug purposes.

This function is intended for use by the GDB stub and similar code. It takes a virtual address, converts it to a physical address via an MMU lookup using the current settings of the specified CPU, and then performs the access (using address_space_rw for reads or cpu_physical_memory_write_rom for writes). This means that if the access is a write to a ROM then this function will modify the contents (whereas a normal guest CPU access would ignore the write attempt).

cpu_memory_rw_debug

dma_memory_*

These behave like address_space_*, except that they perform a DMA barrier operation first.

TODO: We should provide guidance on when you need the DMA barrier operation and when it's OK to use address_space_*, and make sure our existing code is doing things correctly.

dma_memory_read

dma_memory_write

dma_memory_rw

Regexes for git grep:

- \<dma_memory_\(read\|write\|rw\)\>
- \<ldu\?[bwlq]\(_[bl]e\)\?_dma\>
- \<st[bwlq]\(_[bl]e\)\?_dma\>

pci_dma_* and {ld,st}*_pci_dma

These functions are specifically for PCI device models which need to perform accesses where the PCI device is a bus master. You pass them a PCIDevice * and they will do dma_memory_* operations on the correct address space for that device.

pci_dma_read

pci_dma_write

pci_dma_rw

load: ld{sign}{size}_{endian}_pci_dma

store: st{size}_{endian}_pci_dma

sign

- (empty) : for 32 or 64 bit sizes
- u : unsigned

(No signed load operations are provided.)

size

- b : 8 bits
- w : 16 bits
- l : 32 bits
- q : 64 bits

endian

- le : little endian
- be : big endian

The _{endian}_ infix is omitted for byte accesses.

Regexes for git grep:

- \<pci_dma_\(read\|write\|rw\)\>
- \<ldu\?[bwlq]\(_[bl]e\)\?_pci_dma\>

- `\<st[bwlq]\(_[bl]e\)\?_pci_dma\>`

7.3.3 The memory API

The memory API models the memory and I/O buses and controllers of a QEMU machine. It attempts to allow modelling of:

- ordinary RAM
- memory-mapped I/O (MMIO)
- memory controllers that can dynamically reroute physical memory regions to different destinations

The memory model provides support for

- tracking RAM changes by the guest
- setting up coalesced memory for kvm
- setting up ioeventfd regions for kvm

Memory is modelled as an acyclic graph of MemoryRegion objects. Sinks (leaves) are RAM and MMIO regions, while other nodes represent buses, memory controllers, and memory regions that have been rerouted.

In addition to MemoryRegion objects, the memory API provides AddressSpace objects for every root and possibly for intermediate MemoryRegions too. These represent memory as seen from the CPU or a device's viewpoint.

Types of regions

There are multiple types of memory regions (all represented by a single C type MemoryRegion):

- RAM: a RAM region is simply a range of host memory that can be made available to the guest. You typically initialize these with memory_region_init_ram(). Some special purposes require the variants memory_region_init_resizeable_ram(), memory_region_init_ram_from_file(), or memory_region_init_ram_ptr().
- MMIO: a range of guest memory that is implemented by host callbacks; each read or write causes a callback to be called on the host. You initialize these with memory_region_init_io(), passing it a MemoryRegionOps structure describing the callbacks.
- ROM: a ROM memory region works like RAM for reads (directly accessing a region of host memory), and forbids writes. You initialize these with memory_region_init_rom().
- ROM device: a ROM device memory region works like RAM for reads (directly accessing a region of host memory), but like MMIO for writes (invoking a callback). You initialize these with memory_region_init_rom_device().
- IOMMU region: an IOMMU region translates addresses of accesses made to it and forwards them to some other target memory region. As the name suggests, these are only needed for modelling an IOMMU, not for simple devices. You initialize these with memory_region_init_iommu().
- container: a container simply includes other memory regions, each at a different offset. Containers are useful for grouping several regions into one unit. For example, a PCI BAR may be composed of a RAM region and an MMIO region.

  A container's subregions are usually non-overlapping. In some cases it is useful to have overlapping regions; for example a memory controller that can overlay a subregion of RAM with MMIO or ROM, or a PCI controller that does not prevent card from claiming overlapping BARs.

  You initialize a pure container with memory_region_init().

- alias: a subsection of another region. Aliases allow a region to be split apart into discontiguous regions. Examples of uses are memory banks used when the guest address space is smaller than the amount of RAM addressed, or a memory controller that splits main memory to expose a "PCI hole". You can also create aliases to avoid trying to add the original region to multiple parents via *memory_region_add_subregion*.

  Aliases may point to any type of region, including other aliases, but an alias may not point back to itself, directly or indirectly. You initialize these with memory_region_init_alias().

- reservation region: a reservation region is primarily for debugging. It claims I/O space that is not supposed to be handled by QEMU itself. The typical use is to track parts of the address space which will be handled by the host kernel when KVM is enabled. You initialize these by passing a NULL callback parameter to memory_region_init_io().

It is valid to add subregions to a region which is not a pure container (that is, to an MMIO, RAM or ROM region). This means that the region will act like a container, except that any addresses within the container's region which are not claimed by any subregion are handled by the container itself (ie by its MMIO callbacks or RAM backing). However it is generally possible to achieve the same effect with a pure container one of whose subregions is a low priority "background" region covering the whole address range; this is often clearer and is preferred. Subregions cannot be added to an alias region.

Migration

Where the memory region is backed by host memory (RAM, ROM and ROM device memory region types), this host memory needs to be copied to the destination on migration. These APIs which allocate the host memory for you will also register the memory so it is migrated:

- memory_region_init_ram()
- memory_region_init_rom()
- memory_region_init_rom_device()

For most devices and boards this is the correct thing. If you have a special case where you need to manage the migration of the backing memory yourself, you can call the functions:

- memory_region_init_ram_nomigrate()
- memory_region_init_rom_nomigrate()
- memory_region_init_rom_device_nomigrate()

which only initialize the MemoryRegion and leave handling migration to the caller.

The functions:

- memory_region_init_resizeable_ram()
- memory_region_init_ram_from_file()
- memory_region_init_ram_from_fd()
- memory_region_init_ram_ptr()
- memory_region_init_ram_device_ptr()

are for special cases only, and so they do not automatically register the backing memory for migration; the caller must manage migration if necessary.

Region names

Regions are assigned names by the constructor. For most regions these are only used for debugging purposes, but RAM regions also use the name to identify live migration sections. This means that RAM region names need to have ABI stability.

Region lifecycle

A region is created by one of the memory_region_init*() functions and attached to an object, which acts as its owner or parent. QEMU ensures that the owner object remains alive as long as the region is visible to the guest, or as long as the region is in use by a virtual CPU or another device. For example, the owner object will not die between an address_space_map operation and the corresponding address_space_unmap.

After creation, a region can be added to an address space or a container with memory_region_add_subregion(), and removed using memory_region_del_subregion().

Various region attributes (read-only, dirty logging, coalesced mmio, ioeventfd) can be changed during the region lifecycle. They take effect as soon as the region is made visible. This can be immediately, later, or never.

Destruction of a memory region happens automatically when the owner object dies.

If however the memory region is part of a dynamically allocated data structure, you should call object_unparent() to destroy the memory region before the data structure is freed. For an example see VFIOMSIXInfo and VFIOQuirk in hw/vfio/pci.c.

You must not destroy a memory region as long as it may be in use by a device or CPU. In order to do this, as a general rule do not create or destroy memory regions dynamically during a device's lifetime, and only call object_unparent() in the memory region owner's instance_finalize callback. The dynamically allocated data structure that contains the memory region then should obviously be freed in the instance_finalize callback as well.

If you break this rule, the following situation can happen:

- the memory region's owner had a reference taken via memory_region_ref (for example by address_space_map)
- the region is unparented, and has no owner anymore
- when address_space_unmap is called, the reference to the memory region's owner is leaked.

There is an exception to the above rule: it is okay to call object_unparent at any time for an alias or a container region. It is therefore also okay to create or destroy alias and container regions dynamically during a device's lifetime.

This exceptional usage is valid because aliases and containers only help QEMU building the guest's memory map; they are never accessed directly. memory_region_ref and memory_region_unref are never called on aliases or containers, and the above situation then cannot happen. Exploiting this exception is rarely necessary, and therefore it is discouraged, but nevertheless it is used in a few places.

For regions that "have no owner" (NULL is passed at creation time), the machine object is actually used as the owner. Since instance_finalize is never called for the machine object, you must never call object_unparent on regions that have no owner, unless they are aliases or containers.

Overlapping regions and priority

Usually, regions may not overlap each other; a memory address decodes into exactly one target. In some cases it is useful to allow regions to overlap, and sometimes to control which of an overlapping regions is visible to the guest. This is done with memory_region_add_subregion_overlap(), which allows the region to overlap any other region in the same container, and specifies a priority that allows the core to decide which of two regions at the same address are visible (highest wins). Priority values are signed, and the default value is zero. This means that you can use memory_region_add_subregion_overlap() both to specify a region that must sit 'above' any others (with a positive priority) and also a background region that sits 'below' others (with a negative priority).

If the higher priority region in an overlap is a container or alias, then the lower priority region will appear in any "holes" that the higher priority region has left by not mapping subregions to that area of its address range. (This applies recursively – if the subregions are themselves containers or aliases that leave holes then the lower priority region will appear in these holes too.)

For example, suppose we have a container A of size 0x8000 with two subregions B and C. B is a container mapped at 0x2000, size 0x4000, priority 2; C is an MMIO region mapped at 0x0, size 0x6000, priority 1. B currently has two of its own subregions: D of size 0x1000 at offset 0 and E of size 0x1000 at offset 0x2000. As a diagram:

```
      0      1000   2000   3000   4000   5000   6000   7000   8000
      |------|------|------|------|------|------|------|------|
A:    [                                                      ]
C:    [CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC]
B:                  [                           ]
D:                  [DDDDD]
E:                                [EEEEE]
```

The regions that will be seen within this address range then are:

```
[CCCCCCCCCCCC][DDDDD][CCCCC][EEEEE][CCCCC]
```

Since B has higher priority than C, its subregions appear in the flat map even where they overlap with C. In ranges where B has not mapped anything C's region appears.

If B had provided its own MMIO operations (ie it was not a pure container) then these would be used for any addresses in its range not handled by D or E, and the result would be:

```
[CCCCCCCCCCCC][DDDDD][BBBBB][EEEEE][BBBBB]
```

Priority values are local to a container, because the priorities of two regions are only compared when they are both children of the same container. This means that the device in charge of the container (typically modelling a bus or a memory controller) can use them to manage the interaction of its child regions without any side effects on other parts of the system. In the example above, the priorities of D and E are unimportant because they do not overlap each other. It is the relative priority of B and C that causes D and E to appear on top of C: D and E's priorities are never compared against the priority of C.

Visibility

The memory core uses the following rules to select a memory region when the guest accesses an address:

- all direct subregions of the root region are matched against the address, in descending priority order
  - if the address lies outside the region offset/size, the subregion is discarded
  - if the subregion is a leaf (RAM or MMIO), the search terminates, returning this leaf region
  - if the subregion is a container, the same algorithm is used within the subregion (after the address is adjusted by the subregion offset)
  - if the subregion is an alias, the search is continued at the alias target (after the address is adjusted by the subregion offset and alias offset)
  - if a recursive search within a container or alias subregion does not find a match (because of a "hole" in the container's coverage of its address range), then if this is a container with its own MMIO or RAM backing the search terminates, returning the container itself. Otherwise we continue with the next subregion in priority order
- if none of the subregions match the address then the search terminates with no match found

Example memory map

```
system_memory: container@0-2^48-1
 |
 +---- lomem: alias@0-0xdfffffff ---> #ram (0-0xdfffffff)
 |
 +---- himem: alias@0x100000000-0x11fffffff ---> #ram (0xe0000000-0xffffffff)
 |
 +---- vga-window: alias@0xa0000-0xbffff ---> #pci (0xa0000-0xbffff)
 |      (prio 1)
 |
 +---- pci-hole: alias@0xe0000000-0xffffffff ---> #pci (0xe0000000-0xffffffff)

pci (0-2^32-1)
 |
 +--- vga-area: container@0xa0000-0xbffff
 |      |
 |      +--- alias@0x00000-0x7fff  ---> #vram (0x010000-0x017fff)
 |      |
 |      +--- alias@0x08000-0xffff  ---> #vram (0x020000-0x027fff)
 |
 +---- vram: ram@0xe1000000-0xe1ffffff
 |
 +---- vga-mmio: mmio@0xe2000000-0xe200ffff

ram: ram@0x00000000-0xffffffff
```

This is a (simplified) PC memory map. The 4GB RAM block is mapped into the system address space via two aliases: "lomem" is a 1:1 mapping of the first 3.5GB; "himem" maps the last 0.5GB at address 4GB. This leaves 0.5GB for the so-called PCI hole, that allows a 32-bit PCI bus to exist in a system with 4GB of memory.

The memory controller diverts addresses in the range 640K-768K to the PCI address space. This is modelled using the "vga-window" alias, mapped at a higher priority so it obscures the RAM at the same addresses. The vga window can

be removed by programming the memory controller; this is modelled by removing the alias and exposing the RAM underneath.

The pci address space is not a direct child of the system address space, since we only want parts of it to be visible (we accomplish this using aliases). It has two subregions: vga-area models the legacy vga window and is occupied by two 32K memory banks pointing at two sections of the framebuffer. In addition the vram is mapped as a BAR at address e1000000, and an additional BAR containing MMIO registers is mapped after it.

Note that if the guest maps a BAR outside the PCI hole, it would not be visible as the pci-hole alias clips it to a 0.5GB range.

MMIO Operations

MMIO regions are provided with ->read() and ->write() callbacks, which are sufficient for most devices. Some devices change behaviour based on the attributes used for the memory transaction, or need to be able to respond that the access should provoke a bus error rather than completing successfully; those devices can use the ->read_with_attrs() and ->write_with_attrs() callbacks instead.

In addition various constraints can be supplied to control how these callbacks are called:

- .valid.min_access_size, .valid.max_access_size define the access sizes (in bytes) which the device accepts; accesses outside this range will have device and bus specific behaviour (ignored, or machine check)
- .valid.unaligned specifies that the *device being modelled* supports unaligned accesses; if false, unaligned accesses will invoke the appropriate bus or CPU specific behaviour.
- .impl.min_access_size, .impl.max_access_size define the access sizes (in bytes) supported by the *implementation*; other access sizes will be emulated using the ones available. For example a 4-byte write will be emulated using four 1-byte writes, if .impl.max_access_size = 1.
- .impl.unaligned specifies that the *implementation* supports unaligned accesses; if false, unaligned accesses will be emulated by two aligned accesses.

API Reference

struct **MemoryRegionSection**

describes a fragment of a `MemoryRegion`

Definition

```
struct MemoryRegionSection {
  Int128 size;
  MemoryRegion *mr;
  FlatView *fv;
  hwaddr offset_within_region;
  hwaddr offset_within_address_space;
  bool readonly;
  bool nonvolatile;
  bool unmergeable;
};
```

Members

size

the size of the section; will not exceed **mr**'s boundaries

mr

the region, or `NULL` if empty

fv
: the flat view of the address space the region is mapped in

offset_within_region
: the beginning of the section, relative to **mr**'s start

offset_within_address_space
: the address of the first byte of the section relative to the region's address space

readonly
: writes to this section are ignored

nonvolatile
: this section is non-volatile

unmergeable
: this section should not get merged with adjacent sections

struct **MemoryListener**
: callbacks structure for updates to the physical memory map

Definition

```
struct MemoryListener {
  void (*begin)(MemoryListener *listener);
  void (*commit)(MemoryListener *listener);
  void (*region_add)(MemoryListener *listener, MemoryRegionSection *section);
  void (*region_del)(MemoryListener *listener, MemoryRegionSection *section);
  void (*region_nop)(MemoryListener *listener, MemoryRegionSection *section);
  void (*log_start)(MemoryListener *listener, MemoryRegionSection *section, int old, int␣
↪new);
  void (*log_stop)(MemoryListener *listener, MemoryRegionSection *section, int old, int␣
↪new);
  void (*log_sync)(MemoryListener *listener, MemoryRegionSection *section);
  void (*log_sync_global)(MemoryListener *listener, bool last_stage);
  void (*log_clear)(MemoryListener *listener, MemoryRegionSection *section);
  bool (*log_global_start)(MemoryListener *listener, Error **errp);
  void (*log_global_stop)(MemoryListener *listener);
  void (*log_global_after_sync)(MemoryListener *listener);
  void (*eventfd_add)(MemoryListener *listener, MemoryRegionSection *section, bool match_
↪data, uint64_t data, EventNotifier *e);
  void (*eventfd_del)(MemoryListener *listener, MemoryRegionSection *section, bool match_
↪data, uint64_t data, EventNotifier *e);
  void (*coalesced_io_add)(MemoryListener *listener, MemoryRegionSection *section,␣
↪hwaddr addr, hwaddr len);
  void (*coalesced_io_del)(MemoryListener *listener, MemoryRegionSection *section,␣
↪hwaddr addr, hwaddr len);
  unsigned priority;
  const char *name;
};
```

Members

begin
: Called at the beginning of an address space update transaction. Followed by calls to *MemoryListener.region_add()*, *MemoryListener.region_del()*, *MemoryListener.region_nop()*, *MemoryListener.log_start()* and *MemoryListener.log_stop()* in increasing address order.

listener: The *MemoryListener*.

commit
Called at the end of an address space update transaction, after the last call to *MemoryListener.region_add()*, *MemoryListener.region_del()* or *MemoryListener.region_nop()*, *MemoryListener.log_start()* and *MemoryListener.log_stop()*.

listener: The *MemoryListener*.

region_add
Called during an address space update transaction, for a section of the address space that is new in this address space space since the last transaction.

listener: The *MemoryListener*. **section**: The new *MemoryRegionSection*.

region_del
Called during an address space update transaction, for a section of the address space that has disappeared in the address space since the last transaction.

listener: The *MemoryListener*. **section**: The old *MemoryRegionSection*.

region_nop
Called during an address space update transaction, for a section of the address space that is in the same place in the address space as in the last transaction.

listener: The *MemoryListener*. **section**: The *MemoryRegionSection*.

log_start
Called during an address space update transaction, after one of *MemoryListener.region_add()*, *MemoryListener.region_del()* or *MemoryListener.region_nop()*, if dirty memory logging clients have become active since the last transaction.

listener: The *MemoryListener*. **section**: The *MemoryRegionSection*. **old**: A bitmap of dirty memory logging clients that were active in the previous transaction. **new**: A bitmap of dirty memory logging clients that are active in the current transaction.

log_stop
Called during an address space update transaction, after one of *MemoryListener.region_add()*, *MemoryListener.region_del()* or *MemoryListener.region_nop()* and possibly after *MemoryListener.log_start()*, if dirty memory logging clients have become inactive since the last transaction.

listener: The *MemoryListener*. **section**: The *MemoryRegionSection*. **old**: A bitmap of dirty memory logging clients that were active in the previous transaction. **new**: A bitmap of dirty memory logging clients that are active in the current transaction.

log_sync
Called by memory_region_snapshot_and_clear_dirty() and memory_global_dirty_log_sync(), before accessing QEMU's "official" copy of the dirty memory bitmap for a *MemoryRegionSection*.

listener: The *MemoryListener*. **section**: The *MemoryRegionSection*.

log_sync_global
This is the global version of **log_sync** when the listener does not have a way to synchronize the log with finer granularity. When the listener registers with **log_sync_global** defined, then its **log_sync** must be NULL. Vice versa.

listener: The *MemoryListener*. **last_stage**: The last stage to synchronize the log during migration. The caller should guarantee that the synchronization with true for **last_stage** is triggered for once after all VCPUs have been stopped.

`log_clear`
Called before reading the dirty memory bitmap for a *MemoryRegionSection*.

listener: The *MemoryListener*. **section**: The *MemoryRegionSection*.

`log_global_start`
Called by memory_global_dirty_log_start(), which enables the `DIRTY_LOG_MIGRATION` client on all memory regions in the address space. *MemoryListener.log_global_start()* is also called when a *MemoryListener* is added, if global dirty logging is active at that time.

listener: The *MemoryListener*. **errp**: pointer to Error*, to store an error if it happens.

Return: true on success, else false setting **errp** with error.

`log_global_stop`
Called by memory_global_dirty_log_stop(), which disables the `DIRTY_LOG_MIGRATION` client on all memory regions in the address space.

listener: The *MemoryListener*.

`log_global_after_sync`
Called after reading the dirty memory bitmap for any *MemoryRegionSection*.

listener: The *MemoryListener*.

`eventfd_add`
Called during an address space update transaction, for a section of the address space that has had a new ioeventfd registration since the last transaction.

listener: The *MemoryListener*. **section**: The new *MemoryRegionSection*. **match_data**: The **match_data** parameter for the new ioeventfd. **data**: The **data** parameter for the new ioeventfd. **e**: The `EventNotifier` parameter for the new ioeventfd.

`eventfd_del`
Called during an address space update transaction, for a section of the address space that has dropped an ioeventfd registration since the last transaction.

listener: The *MemoryListener*. **section**: The new *MemoryRegionSection*. **match_data**: The **match_data** parameter for the dropped ioeventfd. **data**: The **data** parameter for the dropped ioeventfd. **e**: The `EventNotifier` parameter for the dropped ioeventfd.

`coalesced_io_add`
Called during an address space update transaction, for a section of the address space that has had a new coalesced MMIO range registration since the last transaction.

listener: The *MemoryListener*. **section**: The new *MemoryRegionSection*. **addr**: The starting address for the coalesced MMIO range. **len**: The length of the coalesced MMIO range.

`coalesced_io_del`
Called during an address space update transaction, for a section of the address space that has dropped a coalesced MMIO range since the last transaction.

listener: The *MemoryListener*. **section**: The new *MemoryRegionSection*. **addr**: The starting address for the coalesced MMIO range. **len**: The length of the coalesced MMIO range.

`priority`
Govern the order in which memory listeners are invoked. Lower priorities are invoked earlier for "add" or "start" callbacks, and later for "delete" or "stop" callbacks.

`name`
Name of the listener. It can be used in contexts where we'd like to identify one memory listener with the rest.

Description

Allows a component to adjust to changes in the guest-visible memory map. Use with memory_listener_register() and memory_listener_unregister().

struct **AddressSpace**

describes a mapping of addresses to `MemoryRegion` objects

Definition

```
struct AddressSpace {
};
```

Members

flatview_cb

Typedef: callback for flatview_for_each_range()

Syntax

```
bool flatview_cb (Int128 start, Int128 len, const MemoryRegion *mr, hwaddr
offset_in_region, void *opaque)
```

Parameters

Int128 start
start address of the range within the FlatView

Int128 len
length of the range in bytes

const MemoryRegion *mr
MemoryRegion covering this range

hwaddr offset_in_region
offset of the first byte of the range within **mr**

void *opaque
data pointer passed to flatview_for_each_range()

Return

true to stop the iteration, false to keep going.

void **flatview_for_each_range**(FlatView *fv, *flatview_cb* cb, void *opaque)

Iterate through a FlatView

Parameters

FlatView *fv
the FlatView to iterate through

flatview_cb cb
function to call for each range

void *opaque
opaque data pointer to pass to **cb**

Description

A FlatView is made up of a list of non-overlapping ranges, each of which is a slice of a MemoryRegion. This function iterates through each range in **fv**, calling **cb**. The callback function can terminate iteration early by returning 'true'.

MemoryRegionSection ***memory_region_section_new_copy**(*MemoryRegionSection* *s)

Copy a memory region section

Parameters

MemoryRegionSection *s
the *MemoryRegionSection* to copy

Description

Allocate memory for a new copy, copy the memory region section, and properly take a reference on all relevant members.

void **memory_region_section_free_copy**(*MemoryRegionSection* *s)

Free a copied memory region section

Parameters

MemoryRegionSection *s
the *MemoryRegionSection* to copy

Description

Free a copy of a memory section created via memory_region_section_new_copy(). properly dropping references on all relevant members.

void **memory_region_init**(MemoryRegion *mr, *Object* *owner, const char *name, uint64_t size)

Initialize a memory region

Parameters

MemoryRegion *mr
the MemoryRegion to be initialized

Object *owner
the object that tracks the region's reference count

const char *name
used for debugging; not visible to the user or ABI

uint64_t size
size of the region; any subregions beyond this size will be clipped

Description

The region typically acts as a container for other memory regions. Use memory_region_add_subregion() to add subregions.

void **memory_region_ref**(MemoryRegion *mr)

Add 1 to a memory region's reference count

Parameters

MemoryRegion *mr
the MemoryRegion

Description

Whenever memory regions are accessed outside the BQL, they need to be preserved against hot-unplug. MemoryRegions actually do not have their own reference count; they piggyback on a QOM object, their "owner". This function adds a reference to the owner.

All MemoryRegions must have an owner if they can disappear, even if the device they belong to operates exclusively under the BQL. This is because the region could be returned at any time by memory_region_find, and this is usually under guest control.

void **memory_region_unref**(MemoryRegion *mr)

Remove 1 to a memory region's reference count

Parameters

MemoryRegion *mr
the MemoryRegion

Description

Whenever memory regions are accessed outside the BQL, they need to be preserved against hot-unplug. MemoryRegions actually do not have their own reference count; they piggyback on a QOM object, their "owner". This function removes a reference to the owner and possibly destroys it.

void **memory_region_init_io**(MemoryRegion *mr, *Object* *owner, const MemoryRegionOps *ops, void *opaque, const char *name, uint64_t size)

Initialize an I/O memory region.

Parameters

MemoryRegion *mr
the MemoryRegion to be initialized.

Object *owner
the object that tracks the region's reference count

const MemoryRegionOps *ops
a structure containing read and write callbacks to be used when I/O is performed on the region.

void *opaque
passed to the read and write callbacks of the **ops** structure.

const char *name
used for debugging; not visible to the user or ABI

uint64_t size
size of the region.

Description

Accesses into the region will cause the callbacks in **ops** to be called. if **size** is nonzero, subregions will be clipped to **size**.

bool **memory_region_init_ram_nomigrate**(MemoryRegion *mr, *Object* *owner, const char *name, uint64_t size, Error **errp)

Initialize RAM memory region. Accesses into the region will modify memory directly.

Parameters

MemoryRegion *mr
the MemoryRegion to be initialized.

Object *owner
the object that tracks the region's reference count

const char *name
Region name, becomes part of RAMBlock name used in migration stream must be unique within any device

uint64_t size
size of the region.

Error **errp
pointer to Error*, to store an error if it happens.

Description

Note that this function does not do anything to cause the data in the RAM memory region to be migrated; that is the responsibility of the caller.

Return

true on success, else false setting **errp** with error.

bool **memory_region_init_ram_flags_nomigrate**(MemoryRegion *mr, *Object* *owner, const char *name, uint64_t size, uint32_t ram_flags, Error **errp)

Initialize RAM memory region. Accesses into the region will modify memory directly.

Parameters

`MemoryRegion *mr`
the `MemoryRegion` to be initialized.

`Object *owner`
the object that tracks the region's reference count

`const char *name`
Region name, becomes part of RAMBlock name used in migration stream must be unique within any device

`uint64_t size`
size of the region.

`uint32_t ram_flags`
RamBlock flags. Supported flags: RAM_SHARED, RAM_NORESERVE, RAM_GUEST_MEMFD.

`Error **errp`
pointer to Error*, to store an error if it happens.

Description

Note that this function does not do anything to cause the data in the RAM memory region to be migrated; that is the responsibility of the caller.

Return

true on success, else false setting **errp** with error.

bool **memory_region_init_resizeable_ram**(MemoryRegion *mr, *Object* *owner, const char *name, uint64_t size, uint64_t max_size, void (*resized)(const char*, uint64_t length, void *host), Error **errp)

Initialize memory region with resizable RAM. Accesses into the region will modify memory directly. Only an initial portion of this RAM is actually used. Changing the size while migrating can result in the migration being canceled.

Parameters

`MemoryRegion *mr`
the `MemoryRegion` to be initialized.

`Object *owner`
the object that tracks the region's reference count

`const char *name`
Region name, becomes part of RAMBlock name used in migration stream must be unique within any device

`uint64_t size`
used size of the region.

uint64_t max_size
: max size of the region.

void (*resized)(const char*, uint64_t length, void *host)
: callback to notify owner about used size change.

Error **errp
: pointer to Error*, to store an error if it happens.

Description

Note that this function does not do anything to cause the data in the RAM memory region to be migrated; that is the responsibility of the caller.

Return

true on success, else false setting **errp** with error.

bool **memory_region_init_ram_from_file**(MemoryRegion *mr, *Object* *owner, const char *name, uint64_t size, uint64_t align, uint32_t ram_flags, const char *path, ram_addr_t offset, Error **errp)

Initialize RAM memory region with a mmap-ed backend.

Parameters

MemoryRegion *mr
: the `MemoryRegion` to be initialized.

Object *owner
: the object that tracks the region's reference count

const char *name
: Region name, becomes part of RAMBlock name used in migration stream must be unique within any device

uint64_t size
: size of the region.

uint64_t align
: alignment of the region base address; if 0, the default alignment (getpagesize()) will be used.

uint32_t ram_flags
: RamBlock flags. Supported flags: RAM_SHARED, RAM_PMEM, RAM_NORESERVE, RAM_PROTECTED, RAM_NAMED_FILE, RAM_READONLY, RAM_READONLY_FD, RAM_GUEST_MEMFD

const char *path
: the path in which to allocate the RAM.

ram_addr_t offset
: offset within the file referenced by path

Error **errp
: pointer to Error*, to store an error if it happens.

Description

Note that this function does not do anything to cause the data in the RAM memory region to be migrated; that is the responsibility of the caller.

Return

true on success, else false setting **errp** with error.

bool **memory_region_init_ram_from_fd**(MemoryRegion *mr, *Object* *owner, const char *name, uint64_t size, uint32_t ram_flags, int fd, ram_addr_t offset, Error **errp)

Initialize RAM memory region with a mmap-ed backend.

Parameters

MemoryRegion *mr
the MemoryRegion to be initialized.

Object *owner
the object that tracks the region's reference count

const char *name
the name of the region.

uint64_t size
size of the region.

uint32_t ram_flags
RamBlock flags. Supported flags: RAM_SHARED, RAM_PMEM, RAM_NORESERVE, RAM_PROTECTED, RAM_NAMED_FILE, RAM_READONLY, RAM_READONLY_FD, RAM_GUEST_MEMFD

int fd
the fd to mmap.

ram_addr_t offset
offset within the file referenced by fd

Error **errp
pointer to Error*, to store an error if it happens.

Description

Note that this function does not do anything to cause the data in the RAM memory region to be migrated; that is the responsibility of the caller.

Return

true on success, else false setting **errp** with error.

void **memory_region_init_ram_ptr**(MemoryRegion *mr, *Object* *owner, const char *name, uint64_t size, void *ptr)

Initialize RAM memory region from a user-provided pointer. Accesses into the region will modify memory directly.

Parameters

MemoryRegion *mr
the MemoryRegion to be initialized.

Object *owner
the object that tracks the region's reference count

const char *name
Region name, becomes part of RAMBlock name used in migration stream must be unique within any device

uint64_t size
size of the region.

void *ptr
memory to be mapped; must contain at least **size** bytes.

Description

Note that this function does not do anything to cause the data in the RAM memory region to be migrated; that is the responsibility of the caller.

void **memory_region_init_ram_device_ptr**(MemoryRegion *mr, *Object* *owner, const char *name, uint64_t size, void *ptr)

Initialize RAM device memory region from a user-provided pointer.

Parameters

MemoryRegion *mr
the MemoryRegion to be initialized.

Object *owner
the object that tracks the region's reference count

const char *name
the name of the region.

uint64_t size
size of the region.

void *ptr
memory to be mapped; must contain at least **size** bytes.

Description

A RAM device represents a mapping to a physical device, such as to a PCI MMIO BAR of an vfio-pci assigned device. The memory region may be mapped into the VM address space and access to the region will modify memory directly. However, the memory region should not be included in a memory dump (device may not be enabled/mapped at the time of the dump), and operations incompatible with manipulating MMIO should be avoided. Replaces skip_dump flag.

Note that this function does not do anything to cause the data in the RAM memory region to be migrated; that is the responsibility of the caller. (For RAM device memory regions, migrating the contents rarely makes sense.)

void **memory_region_init_alias**(MemoryRegion *mr, *Object* *owner, const char *name, MemoryRegion *orig, hwaddr offset, uint64_t size)

Initialize a memory region that aliases all or a part of another memory region.

Parameters

MemoryRegion *mr
the MemoryRegion to be initialized.

Object *owner
the object that tracks the region's reference count

const char *name
used for debugging; not visible to the user or ABI

MemoryRegion *orig
the region to be referenced; **mr** will be equivalent to **orig** between **offset** and **offset** + **size** - 1.

hwaddr offset
start of the section in **orig** to be referenced.

uint64_t size
size of the region.

bool **memory_region_init_rom_nomigrate**(MemoryRegion *mr, *Object* *owner, const char *name, uint64_t size, Error **errp)

Initialize a ROM memory region.

Parameters

MemoryRegion *mr
the MemoryRegion to be initialized.

Object *owner
the object that tracks the region's reference count

const char *name
Region name, becomes part of RAMBlock name used in migration stream must be unique within any device

uint64_t size
size of the region.

Error **errp
pointer to Error*, to store an error if it happens.

Description

This has the same effect as calling memory_region_init_ram_nomigrate() and then marking the resulting region read-only with memory_region_set_readonly().

Note that this function does not do anything to cause the data in the RAM side of the memory region to be migrated; that is the responsibility of the caller.

Return

true on success, else false setting **errp** with error.

bool **memory_region_init_rom_device_nomigrate**(MemoryRegion *mr, *Object* *owner, const MemoryRegionOps *ops, void *opaque, const char *name, uint64_t size, Error **errp)

Initialize a ROM memory region. Writes are handled via callbacks.

Parameters

MemoryRegion *mr
the MemoryRegion to be initialized.

Object *owner
the object that tracks the region's reference count

const MemoryRegionOps *ops
callbacks for write access handling (must not be NULL).

void *opaque
passed to the read and write callbacks of the **ops** structure.

const char *name
Region name, becomes part of RAMBlock name used in migration stream must be unique within any device

uint64_t size
size of the region.

Error **errp
pointer to Error*, to store an error if it happens.

Description

Note that this function does not do anything to cause the data in the RAM side of the memory region to be migrated; that is the responsibility of the caller.

Return

true on success, else false setting **errp** with error.

void **memory_region_init_iommu**(void *_iommu_mr, size_t instance_size, const char *mrtypename, *Object* *owner, const char *name, uint64_t size)

Initialize a memory region of a custom type that translates addresses

Parameters

void *_iommu_mr
the `IOMMUMemoryRegion` to be initialized

size_t instance_size
the IOMMUMemoryRegion subclass instance size

const char *mrtypename
the type name of the `IOMMUMemoryRegion`

Object *owner
the object that tracks the region's reference count

const char *name
used for debugging; not visible to the user or ABI

uint64_t size
size of the region.

Description

An IOMMU region translates addresses and forwards accesses to a target memory region.

The IOMMU implementation must define a subclass of TYPE_IOMMU_MEMORY_REGION. **_iommu_mr** should be a pointer to enough memory for an instance of that subclass, **instance_size** is the size of that subclass, and **mrtypename** is its name. This function will initialize **_iommu_mr** as an instance of the subclass, and its methods will then be called to handle accesses to the memory region. See the documentation of `IOMMUMemoryRegionClass` for further details.

bool **memory_region_init_ram**(MemoryRegion *mr, *Object* *owner, const char *name, uint64_t size, Error **errp)

Initialize RAM memory region. Accesses into the region will modify memory directly.

Parameters

MemoryRegion *mr
the `MemoryRegion` to be initialized

Object *owner
the object that tracks the region's reference count (must be TYPE_DEVICE or a subclass of TYPE_DEVICE, or NULL)

const char *name
name of the memory region

uint64_t size
size of the region in bytes

Error **errp
pointer to Error*, to store an error if it happens.

Description

This function allocates RAM for a board model or device, and arranges for it to be migrated (by calling vmstate_register_ram() if **owner** is a DeviceState, or vmstate_register_ram_global() if **owner** is NULL).

TODO: Currently we restrict **owner** to being either NULL (for global RAM regions with no owner) or devices, so that we can give the RAM block a unique name for migration purposes. We should lift this restriction and allow arbitrary Objects. If you pass a non-NULL non-device **owner** then we will assert.

Return

true on success, else false setting **errp** with error.

bool **memory_region_init_rom**(MemoryRegion *mr, *Object* *owner, const char *name, uint64_t size, Error **errp)

Initialize a ROM memory region.

Parameters

`MemoryRegion *mr`
the `MemoryRegion` to be initialized.

`Object *owner`
the object that tracks the region's reference count

`const char *name`
Region name, becomes part of RAMBlock name used in migration stream must be unique within any device

`uint64_t size`
size of the region.

`Error **errp`
pointer to Error*, to store an error if it happens.

Description

This has the same effect as calling memory_region_init_ram() and then marking the resulting region read-only with memory_region_set_readonly(). This includes arranging for the contents to be migrated.

TODO: Currently we restrict **owner** to being either NULL (for global RAM regions with no owner) or devices, so that we can give the RAM block a unique name for migration purposes. We should lift this restriction and allow arbitrary Objects. If you pass a non-NULL non-device **owner** then we will assert.

Return

true on success, else false setting **errp** with error.

bool **memory_region_init_rom_device**(MemoryRegion *mr, *Object* *owner, const MemoryRegionOps *ops, void *opaque, const char *name, uint64_t size, Error **errp)

Initialize a ROM memory region. Writes are handled via callbacks.

Parameters

`MemoryRegion *mr`
the `MemoryRegion` to be initialized.

`Object *owner`
the object that tracks the region's reference count

`const MemoryRegionOps *ops`
callbacks for write access handling (must not be NULL).

`void *opaque`
passed to the read and write callbacks of the **ops** structure.

const char *name
Region name, becomes part of RAMBlock name used in migration stream must be unique within any device

uint64_t size
size of the region.

Error **errp
pointer to Error*, to store an error if it happens.

Description

This function initializes a memory region backed by RAM for reads and callbacks for writes, and arranges for the RAM backing to be migrated (by calling vmstate_register_ram() if **owner** is a DeviceState, or vmstate_register_ram_global() if **owner** is NULL).

TODO: Currently we restrict **owner** to being either NULL (for global RAM regions with no owner) or devices, so that we can give the RAM block a unique name for migration purposes. We should lift this restriction and allow arbitrary Objects. If you pass a non-NULL non-device **owner** then we will assert.

Return

true on success, else false setting **errp** with error.

Object ***memory_region_owner**(MemoryRegion *mr)
get a memory region's owner.

Parameters

MemoryRegion *mr
the memory region being queried.

uint64_t **memory_region_size**(MemoryRegion *mr)
get a memory region's size.

Parameters

MemoryRegion *mr
the memory region being queried.

bool **memory_region_is_ram**(MemoryRegion *mr)
check whether a memory region is random access

Parameters

MemoryRegion *mr
the memory region being queried

Description

Returns `true` if a memory region is random access.

bool **memory_region_is_ram_device**(MemoryRegion *mr)
check whether a memory region is a ram device

Parameters

MemoryRegion *mr
the memory region being queried

Description

Returns `true` if a memory region is a device backed ram region

bool **memory_region_is_romd**(MemoryRegion *mr)

check whether a memory region is in ROMD mode

Parameters

MemoryRegion *mr

the memory region being queried

Description

Returns `true` if a memory region is a ROM device and currently set to allow direct reads.

bool **memory_region_is_protected**(MemoryRegion *mr)

check whether a memory region is protected

Parameters

MemoryRegion *mr

the memory region being queried

Description

Returns `true` if a memory region is protected RAM and cannot be accessed via standard mechanisms, e.g. DMA.

bool **memory_region_has_guest_memfd**(MemoryRegion *mr)

check whether a memory region has guest_memfd associated

Parameters

MemoryRegion *mr

the memory region being queried

Description

Returns `true` if a memory region's ram_block has valid guest_memfd assigned.

IOMMUMemoryRegion ***memory_region_get_iommu**(MemoryRegion *mr)

check whether a memory region is an iommu

Parameters

MemoryRegion *mr

the memory region being queried

Description

Returns pointer to IOMMUMemoryRegion if a memory region is an iommu, otherwise NULL.

IOMMUMemoryRegionClass ***memory_region_get_iommu_class_nocheck**(IOMMUMemoryRegion *iommu_mr)

returns iommu memory region class if an iommu or NULL if not

Parameters

IOMMUMemoryRegion *iommu_mr

the memory region being queried

Description

Returns pointer to IOMMUMemoryRegionClass if a memory region is an iommu, otherwise NULL. This is fast path avoiding QOM checking, use with caution.

uint64_t **memory_region_iommu_get_min_page_size**(IOMMUMemoryRegion *iommu_mr)

get minimum supported page size for an iommu

Parameters

IOMMUMemoryRegion *iommu_mr
the memory region being queried

Description

Returns minimum supported page size for an iommu.

void **memory_region_notify_iommu**(IOMMUMemoryRegion *iommu_mr, int iommu_idx, IOMMUTLBEvent event)

notify a change in an IOMMU translation entry.

Parameters

IOMMUMemoryRegion *iommu_mr
the memory region that was changed

int iommu_idx
the IOMMU index for the translation table which has changed

IOMMUTLBEvent event
TLB event with the new entry in the IOMMU translation table. The entry replaces all old entries for the same virtual I/O address range.

Note

for any IOMMU implementation, an in-place mapping change should be notified with an UNMAP followed by a MAP.

void **memory_region_notify_iommu_one**(IOMMUNotifier *notifier, IOMMUTLBEvent *event)

notify a change in an IOMMU translation entry to a single notifier

Parameters

IOMMUNotifier *notifier
the notifier to be notified

IOMMUTLBEvent *event
TLB event with the new entry in the IOMMU translation table. The entry replaces all old entries for the same virtual I/O address range.

Description

This works just like memory_region_notify_iommu(), but it only notifies a specific notifier, not all of them.

void **memory_region_unmap_iommu_notifier_range**(IOMMUNotifier *notifier)

notify a unmap for an IOMMU translation that covers the range of a notifier

Parameters

IOMMUNotifier *notifier
the notifier to be notified

int **memory_region_register_iommu_notifier**(MemoryRegion *mr, IOMMUNotifier *n, Error **errp)

register a notifier for changes to IOMMU translation entries.

Parameters

MemoryRegion *mr
the memory region to observe

IOMMUNotifier *n
: the IOMMUNotifier to be added; the notify callback receives a pointer to an `IOMMUTLBEntry` as the opaque value; the pointer ceases to be valid on exit from the notifier.

Error **errp
: pointer to Error*, to store an error if it happens.

Description

Returns 0 on success, or a negative errno otherwise. In particular, -EINVAL indicates that at least one of the attributes of the notifier is not supported (flag/range) by the IOMMU memory region. In case of error the error object must be created.

void **memory_region_iommu_replay**(IOMMUMemoryRegion *iommu_mr, IOMMUNotifier *n)

replay existing IOMMU translations to a notifier with the minimum page granularity returned by mr->iommu_ops->get_page_size().

Parameters

IOMMUMemoryRegion *iommu_mr
: the memory region to observe

IOMMUNotifier *n
: the notifier to which to replay iommu mappings

Note

this is not related to record-and-replay functionality.

void **memory_region_unregister_iommu_notifier**(MemoryRegion *mr, IOMMUNotifier *n)

unregister a notifier for changes to IOMMU translation entries.

Parameters

MemoryRegion *mr
: the memory region which was observed and for which notity_stopped() needs to be called

IOMMUNotifier *n
: the notifier to be removed.

int **memory_region_iommu_get_attr**(IOMMUMemoryRegion *iommu_mr, enum IOMMUMemoryRegionAttr attr, void *data)

return an IOMMU attr if get_attr() is defined on the IOMMU.

Parameters

IOMMUMemoryRegion *iommu_mr
: the memory region

enum IOMMUMemoryRegionAttr attr
: the requested attribute

void *data
: a pointer to the requested attribute data

Description

Returns 0 on success, or a negative errno otherwise. In particular, -EINVAL indicates that the IOMMU does not support the requested attribute.

int **memory_region_iommu_attrs_to_index**(IOMMUMemoryRegion *iommu_mr, MemTxAttrs attrs)

return the IOMMU index to use for translations with the given memory transaction attributes.

Parameters

IOMMUMemoryRegion *iommu_mr
the memory region

MemTxAttrs attrs
the memory transaction attributes

int **memory_region_iommu_num_indexes**(IOMMUMemoryRegion *iommu_mr)

return the total number of IOMMU indexes that this IOMMU supports.

Parameters

IOMMUMemoryRegion *iommu_mr
the memory region

int **memory_region_iommu_set_page_size_mask**(IOMMUMemoryRegion *iommu_mr, uint64_t page_size_mask, Error **errp)

set the supported page sizes for a given IOMMU memory region

Parameters

IOMMUMemoryRegion *iommu_mr
IOMMU memory region

uint64_t page_size_mask
supported page size mask

Error **errp
pointer to Error*, to store an error if it happens.

int **memory_region_iommu_set_iova_ranges**(IOMMUMemoryRegion *iommu, GList *iova_ranges, Error **errp)

Set the usable IOVA ranges for a given IOMMU MR region

Parameters

IOMMUMemoryRegion *iommu
IOMMU memory region

GList *iova_ranges
list of ordered IOVA ranges (at least one range)

Error **errp
pointer to Error*, to store an error if it happens.

const char ***memory_region_name**(const MemoryRegion *mr)

get a memory region's name

Parameters

const MemoryRegion *mr
the memory region being queried

Description

Returns the string that was used to initialize the memory region.

bool **memory_region_is_logging**(MemoryRegion *mr, uint8_t client)

return whether a memory region is logging writes

Parameters

MemoryRegion *mr
the memory region being queried

uint8_t client
the client being queried

Description

Returns `true` if the memory region is logging writes for the given client

uint8_t **memory_region_get_dirty_log_mask**(MemoryRegion *mr)
return the clients for which a memory region is logging writes.

Parameters

MemoryRegion *mr
the memory region being queried

Description

Returns a bitmap of clients, in which the DIRTY_MEMORY_* constants are the bit indices.

bool **memory_region_is_rom**(MemoryRegion *mr)
check whether a memory region is ROM

Parameters

MemoryRegion *mr
the memory region being queried

Description

Returns `true` if a memory region is read-only memory.

bool **memory_region_is_nonvolatile**(MemoryRegion *mr)
check whether a memory region is non-volatile

Parameters

MemoryRegion *mr
the memory region being queried

Description

Returns `true` is a memory region is non-volatile memory.

int **memory_region_get_fd**(MemoryRegion *mr)
Get a file descriptor backing a RAM memory region.

Parameters

MemoryRegion *mr
the RAM or alias memory region being queried.

Description

Returns a file descriptor backing a file-based RAM memory region, or -1 if the region is not a file-based RAM memory region.

MemoryRegion ***memory_region_from_host**(void *ptr, ram_addr_t *offset)
Convert a pointer into a RAM memory region and an offset within it.

Parameters

void *ptr
the host pointer to be converted

ram_addr_t *offset
the offset within memory region

Description

Given a host pointer inside a RAM memory region (created with memory_region_init_ram() or memory_region_init_ram_ptr()), return the MemoryRegion and the offset within it.

Use with care; by the time this function returns, the returned pointer is not protected by RCU anymore. If the caller is not within an RCU critical section and does not hold the BQL, it must have other means of protecting the pointer, such as a reference to the region that includes the incoming ram_addr_t.

void *memory_region_get_ram_ptr(MemoryRegion *mr)

Get a pointer into a RAM memory region.

Parameters

`MemoryRegion *mr`
the memory region being queried.

Description

Returns a host pointer to a RAM memory region (created with memory_region_init_ram() or memory_region_init_ram_ptr()).

Use with care; by the time this function returns, the returned pointer is not protected by RCU anymore. If the caller is not within an RCU critical section and does not hold the BQL, it must have other means of protecting the pointer, such as a reference to the region that includes the incoming ram_addr_t.

void memory_region_msync(MemoryRegion *mr, hwaddr addr, hwaddr size)

Synchronize selected address range of a memory mapped region

Parameters

`MemoryRegion *mr`
the memory region to be msync

`hwaddr addr`
the initial address of the range to be sync

`hwaddr size`
the size of the range to be sync

void memory_region_writeback(MemoryRegion *mr, hwaddr addr, hwaddr size)

Trigger cache writeback for selected address range

Parameters

`MemoryRegion *mr`
the memory region to be updated

`hwaddr addr`
the initial address of the range to be written back

`hwaddr size`
the size of the range to be written back

void memory_region_set_log(MemoryRegion *mr, bool log, unsigned client)

Turn dirty logging on or off for a region.

Parameters

`MemoryRegion *mr`
the memory region being updated.

`bool log`
whether dirty logging is to be enabled or disabled.

unsigned client
the user of the logging information; DIRTY_MEMORY_VGA only.

Description

Turns dirty logging on or off for a specified client (display, migration). Only meaningful for RAM regions.

void **memory_region_set_dirty**(MemoryRegion *mr, hwaddr addr, hwaddr size)

Mark a range of bytes as dirty in a memory region.

Parameters

MemoryRegion *mr
the memory region being dirtied.

hwaddr addr
the address (relative to the start of the region) being dirtied.

hwaddr size
size of the range being dirtied.

Description

Marks a range of bytes as dirty, after it has been dirtied outside guest code.

void **memory_region_clear_dirty_bitmap**(MemoryRegion *mr, hwaddr start, hwaddr len)

clear dirty bitmap for memory range

Parameters

MemoryRegion *mr
the memory region to clear the dirty log upon

hwaddr start
start address offset within the memory region

hwaddr len
length of the memory region to clear dirty bitmap

Description

This function is called when the caller wants to clear the remote dirty bitmap of a memory range within the memory region. This can be used by e.g. KVM to manually clear dirty log when KVM_CAP_MANUAL_DIRTY_LOG_PROTECT is declared support by the host kernel.

DirtyBitmapSnapshot *__memory_region_snapshot_and_clear_dirty__(MemoryRegion *mr, hwaddr addr, hwaddr size, unsigned client)

Get a snapshot of the dirty bitmap and clear it.

Parameters

MemoryRegion *mr
the memory region being queried.

hwaddr addr
the address (relative to the start of the region) being queried.

hwaddr size
the size of the range being queried.

unsigned client
the user of the logging information; typically DIRTY_MEMORY_VGA.

Description

Creates a snapshot of the dirty bitmap, clears the dirty bitmap and returns the snapshot. The snapshot can then be used to query dirty status, using memory_region_snapshot_get_dirty. Snapshotting allows querying the same page multiple times, which is especially useful for display updates where the scanlines often are not page aligned.

The dirty bitmap region which gets copied into the snapshot (and cleared afterwards) can be larger than requested. The boundaries are rounded up/down so complete bitmap longs (covering 64 pages on 64bit hosts) can be copied over into the bitmap snapshot. Which isn't a problem for display updates as the extra pages are outside the visible area, and in case the visible area changes a full display redraw is due anyway. Should other use cases for this function emerge we might have to revisit this implementation detail.

Use g_free to release DirtyBitmapSnapshot.

bool **memory_region_snapshot_get_dirty**(MemoryRegion *mr, DirtyBitmapSnapshot *snap, hwaddr addr, hwaddr size)

Check whether a range of bytes is dirty in the specified dirty bitmap snapshot.

Parameters

MemoryRegion *mr
the memory region being queried.

DirtyBitmapSnapshot *snap
the dirty bitmap snapshot

hwaddr addr
the address (relative to the start of the region) being queried.

hwaddr size
the size of the range being queried.

void **memory_region_reset_dirty**(MemoryRegion *mr, hwaddr addr, hwaddr size, unsigned client)

Mark a range of pages as clean, for a specified client.

Parameters

MemoryRegion *mr
the region being updated.

hwaddr addr
the start of the subrange being cleaned.

hwaddr size
the size of the subrange being cleaned.

unsigned client
the user of the logging information; `DIRTY_MEMORY_MIGRATION` or `DIRTY_MEMORY_VGA`.

Description

Marks a range of pages as no longer dirty.

void **memory_region_flush_rom_device**(MemoryRegion *mr, hwaddr addr, hwaddr size)

Mark a range of pages dirty and invalidate TBs (for self-modifying code).

Parameters

MemoryRegion *mr
the region being flushed.

hwaddr addr
the start, relative to the start of the region, of the range being flushed.

hwaddr size
the size, in bytes, of the range being flushed.

Description

The MemoryRegionOps->write() callback of a ROM device must use this function to mark byte ranges that have been modified internally, such as by directly accessing the memory returned by memory_region_get_ram_ptr().

This function marks the range dirty and invalidates TBs so that TCG can detect self-modifying code.

void **memory_region_set_readonly**(MemoryRegion *mr, bool readonly)

Turn a memory region read-only (or read-write)

Parameters

MemoryRegion *mr
the region being updated.

bool readonly
whether rhe region is to be ROM or RAM.

Description

Allows a memory region to be marked as read-only (turning it into a ROM). only useful on RAM regions.

void **memory_region_set_nonvolatile**(MemoryRegion *mr, bool nonvolatile)

Turn a memory region non-volatile

Parameters

MemoryRegion *mr
the region being updated.

bool nonvolatile
whether rhe region is to be non-volatile.

Description

Allows a memory region to be marked as non-volatile. only useful on RAM regions.

void **memory_region_rom_device_set_romd**(MemoryRegion *mr, bool romd_mode)

enable/disable ROMD mode

Parameters

MemoryRegion *mr
the memory region to be updated

bool romd_mode
`true` to put the region into ROMD mode

Description

Allows a ROM device (initialized with memory_region_init_rom_device() to set to ROMD mode (default) or MMIO mode. When it is in ROMD mode, the device is mapped to guest memory and satisfies read access directly. When in MMIO mode, reads are forwarded to the `MemoryRegion.read` function. Writes are always handled by the `MemoryRegion.write` function.

void **memory_region_set_coalescing**(MemoryRegion *mr)

Enable memory coalescing for the region.

Parameters

MemoryRegion *mr
the memory region to be write coalesced

Description

Enabled writes to a region to be queued for later processing. MMIO ->write callbacks may be delayed until a non-coalesced MMIO is issued. Only useful for IO regions. Roughly similar to write-combining hardware.

void **memory_region_add_coalescing**(MemoryRegion *mr, hwaddr offset, uint64_t size)

Enable memory coalescing for a sub-range of a region.

Parameters

MemoryRegion *mr
the memory region to be updated.

hwaddr offset
the start of the range within the region to be coalesced.

uint64_t size
the size of the subrange to be coalesced.

Description

Like memory_region_set_coalescing(), but works on a sub-range of a region. Multiple calls can be issued coalesced disjoint ranges.

void **memory_region_clear_coalescing**(MemoryRegion *mr)

Disable MMIO coalescing for the region.

Parameters

MemoryRegion *mr
the memory region to be updated.

Description

Disables any coalescing caused by memory_region_set_coalescing() or memory_region_add_coalescing(). Roughly equivalent to uncacheble memory hardware.

void **memory_region_set_flush_coalesced**(MemoryRegion *mr)

Enforce memory coalescing flush before accesses.

Parameters

MemoryRegion *mr
the memory region to be updated.

Description

Ensure that pending coalesced MMIO request are flushed before the memory region is accessed. This property is automatically enabled for all regions passed to memory_region_set_coalescing() and memory_region_add_coalescing().

void **memory_region_clear_flush_coalesced**(MemoryRegion *mr)

Disable memory coalescing flush before accesses.

Parameters

MemoryRegion *mr
the memory region to be updated.

Description

Clear the automatic coalesced MMIO flushing enabled via memory_region_set_flush_coalesced. Note that this service has no effect on memory regions that have MMIO coalescing enabled for themselves. For them, automatic flushing will stop once coalescing is disabled.

void **memory_region_add_eventfd**(MemoryRegion *mr, hwaddr addr, unsigned size, bool match_data, uint64_t data, EventNotifier *e)

Request an eventfd to be triggered when a word is written to a location.

Parameters

MemoryRegion *mr
the memory region being updated.

hwaddr addr
the address within **mr** that is to be monitored

unsigned size
the size of the access to trigger the eventfd

bool match_data
whether to match against **data**, instead of just **addr**

uint64_t data
the data to match against the guest write

EventNotifier *e
event notifier to be triggered when **addr**, **size**, and **data** all match.

Description

Marks a word in an IO region (initialized with memory_region_init_io()) as a trigger for an eventfd event. The I/O callback will not be called. The caller must be prepared to handle failure (that is, take the required action if the callback _is_ called).

void **memory_region_del_eventfd**(MemoryRegion *mr, hwaddr addr, unsigned size, bool match_data, uint64_t data, EventNotifier *e)

Cancel an eventfd.

Parameters

MemoryRegion *mr
the memory region being updated.

hwaddr addr
the address within **mr** that is to be monitored

unsigned size
the size of the access to trigger the eventfd

bool match_data
whether to match against **data**, instead of just **addr**

uint64_t data
the data to match against the guest write

EventNotifier *e
event notifier to be triggered when **addr**, **size**, and **data** all match.

Description

Cancels an eventfd trigger requested by a previous memory_region_add_eventfd() call.

void **memory_region_add_subregion**(MemoryRegion *mr, hwaddr offset, MemoryRegion *subregion)

Add a subregion to a container.

Parameters

`MemoryRegion *mr`
: the region to contain the new subregion; must be a container initialized with memory_region_init().

`hwaddr offset`
: the offset relative to **mr** where **subregion** is added.

`MemoryRegion *subregion`
: the subregion to be added.

Description

Adds a subregion at **offset**. The subregion may not overlap with other subregions (except for those explicitly marked as overlapping). A region may only be added once as a subregion (unless removed with memory_region_del_subregion()); use memory_region_init_alias() if you want a region to be a subregion in multiple locations.

void **memory_region_add_subregion_overlap**(MemoryRegion *mr, hwaddr offset, MemoryRegion *subregion, int priority)

Add a subregion to a container with overlap.

Parameters

`MemoryRegion *mr`
: the region to contain the new subregion; must be a container initialized with memory_region_init().

`hwaddr offset`
: the offset relative to **mr** where **subregion** is added.

`MemoryRegion *subregion`
: the subregion to be added.

`int priority`
: used for resolving overlaps; highest priority wins.

Description

Adds a subregion at **offset**. The subregion may overlap with other subregions. Conflicts are resolved by having a higher **priority** hide a lower **priority**. Subregions without priority are taken as **priority** 0. A region may only be added once as a subregion (unless removed with memory_region_del_subregion()); use memory_region_init_alias() if you want a region to be a subregion in multiple locations.

ram_addr_t **memory_region_get_ram_addr**(MemoryRegion *mr)

Get the ram address associated with a memory region

Parameters

`MemoryRegion *mr`
: the region to be queried

void **memory_region_del_subregion**(MemoryRegion *mr, MemoryRegion *subregion)

Remove a subregion.

Parameters

`MemoryRegion *mr`
: the container to be updated.

`MemoryRegion *subregion`
: the region being removed; must be a current subregion of **mr**.

Description

Removes a subregion from its container.

bool **memory_region_present**(MemoryRegion *container, hwaddr addr)

checks if an address relative to a **container** translates into MemoryRegion within **container**

Parameters

MemoryRegion *container
a MemoryRegion within which **addr** is a relative address

hwaddr addr
the area within **container** to be searched

Description

Answer whether a MemoryRegion within **container** covers the address **addr**.

bool **memory_region_is_mapped**(MemoryRegion *mr)

returns true if MemoryRegion is mapped into another memory region, which does not necessarily imply that it is mapped into an address space.

Parameters

MemoryRegion *mr
a MemoryRegion which should be checked if it's mapped

RamDiscardManager ***memory_region_get_ram_discard_manager**(MemoryRegion *mr)

get the RamDiscardManager for a MemoryRegion

Parameters

MemoryRegion *mr
the MemoryRegion

Description

The RamDiscardManager cannot change while a memory region is mapped.

bool **memory_region_has_ram_discard_manager**(MemoryRegion *mr)

check whether a MemoryRegion has a RamDiscardManager assigned

Parameters

MemoryRegion *mr
the MemoryRegion

void **memory_region_set_ram_discard_manager**(MemoryRegion *mr, RamDiscardManager *rdm)

set the RamDiscardManager for a MemoryRegion

Parameters

MemoryRegion *mr
the MemoryRegion

RamDiscardManager *rdm
RamDiscardManager to set

Description

This function must not be called for a mapped MemoryRegion, a MemoryRegion that does not cover RAM, or a MemoryRegion that already has a RamDiscardManager assigned.

MemoryRegionSection **memory_region_find**(MemoryRegion *mr, hwaddr addr, uint64_t size)

translate an address/size relative to a MemoryRegion into a *MemoryRegionSection*.

Parameters

`MemoryRegion *mr`
: a MemoryRegion within which **addr** is a relative address

`hwaddr addr`
: start of the area within **as** to be searched

`uint64_t size`
: size of the area to be searched

Description

Locates the first `MemoryRegion` within **mr** that overlaps the range given by **addr** and **size**.

Returns a *MemoryRegionSection* that describes a contiguous overlap. It will have the following characteristics: - **size** = 0 iff no overlap was found - **mr** is non-`NULL` iff an overlap was found

Remember that in the return value the **offset_within_region** is relative to the returned region (in the .**mr** field), not to the **mr** argument.

Similarly, the .**offset_within_address_space** is relative to the address space that contains both regions, the passed and the returned one. However, in the special case where the **mr** argument has no container (and thus is the root of the address space), the following will hold: - **offset_within_address_space** >= **addr** - **offset_within_address_space** + .**size** <= **addr** + **size**

void **memory_global_dirty_log_sync**(bool last_stage)
: synchronize the dirty log for all memory

Parameters

`bool last_stage`
: whether this is the last stage of live migration

Description

Synchronizes the dirty page log for all address spaces.

void **memory_global_after_dirty_log_sync**(void)
: synchronize the dirty log for all memory

Parameters

`void`
: no arguments

Description

Synchronizes the vCPUs with a thread that is reading the dirty bitmap. This function must be called after the dirty log bitmap is cleared, and before dirty guest memory pages are read. If you are using `DirtyBitmapSnapshot`, memory_region_snapshot_and_clear_dirty() takes care of doing this.

void **memory_region_transaction_begin**(void)
: Start a transaction.

Parameters

`void`
: no arguments

Description

During a transaction, changes will be accumulated and made visible only when the transaction ends (is committed).

void **memory_region_transaction_commit**(void)

Commit a transaction and make changes visible to the guest.

Parameters

void
no arguments

void **memory_listener_register**(*MemoryListener* *listener, *AddressSpace* *filter)

register callbacks to be called when memory sections are mapped or unmapped into an address space

Parameters

MemoryListener *listener
an object containing the callbacks to be called

AddressSpace *filter
if non-NULL, only regions in this address space will be observed

void **memory_listener_unregister**(*MemoryListener* *listener)

undo the effect of memory_listener_register()

Parameters

MemoryListener *listener
an object containing the callbacks to be removed

bool **memory_global_dirty_log_start**(unsigned int flags, Error **errp)

begin dirty logging for all regions

Parameters

unsigned int flags
purpose of starting dirty log, migration or dirty rate

Error **errp
pointer to Error*, to store an error if it happens.

Return

true on success, else false setting **errp** with error.

void **memory_global_dirty_log_stop**(unsigned int flags)

end dirty logging for all regions

Parameters

unsigned int flags
purpose of stopping dirty log, migration or dirty rate

MemTxResult **memory_region_dispatch_read**(MemoryRegion *mr, hwaddr addr, uint64_t *pval, MemOp op, MemTxAttrs attrs)

perform a read directly to the specified MemoryRegion.

Parameters

MemoryRegion *mr
MemoryRegion to access

hwaddr addr
address within that region

uint64_t *pval
pointer to uint64_t which the data is written to

MemOp op
size, sign, and endianness of the memory operation

MemTxAttrs attrs
memory transaction attributes to use for the access

MemTxResult **memory_region_dispatch_write**(MemoryRegion *mr, hwaddr addr, uint64_t data, MemOp op, MemTxAttrs attrs)

perform a write directly to the specified MemoryRegion.

Parameters

MemoryRegion *mr
`MemoryRegion` to access

hwaddr addr
address within that region

uint64_t data
data to write

MemOp op
size, sign, and endianness of the memory operation

MemTxAttrs attrs
memory transaction attributes to use for the access

void **address_space_init**(*AddressSpace* *as, MemoryRegion *root, const char *name)

initializes an address space

Parameters

AddressSpace *as
an uninitialized *AddressSpace*

MemoryRegion *root
a `MemoryRegion` that routes addresses for the address space

const char *name
an address space name. The name is only used for debugging output.

void **address_space_destroy**(*AddressSpace* *as)

destroy an address space

Parameters

AddressSpace *as
address space to be destroyed

Description

Releases all resources associated with an address space. After an address space is destroyed, its root memory region (given by address_space_init()) may be destroyed as well.

void **address_space_remove_listeners**(*AddressSpace* *as)

unregister all listeners of an address space

Parameters

AddressSpace *as
an initialized *AddressSpace*

Description

Removes all callbacks previously registered with memory_listener_register() for **as**.

MemTxResult **address_space_rw**(*AddressSpace* *as, hwaddr addr, MemTxAttrs attrs, void *buf, hwaddr len, bool is_write)

read from or write to an address space.

Parameters

AddressSpace *as
AddressSpace to be accessed

hwaddr addr
address within that address space

MemTxAttrs attrs
memory transaction attributes

void *buf
buffer with the data transferred

hwaddr len
the number of bytes to read or write

bool is_write
indicates the transfer direction

Description

Return a MemTxResult indicating whether the operation succeeded or failed (eg unassigned memory, device rejected the transaction, IOMMU fault).

MemTxResult **address_space_write**(*AddressSpace* *as, hwaddr addr, MemTxAttrs attrs, const void *buf, hwaddr len)

write to address space.

Parameters

AddressSpace *as
AddressSpace to be accessed

hwaddr addr
address within that address space

MemTxAttrs attrs
memory transaction attributes

const void *buf
buffer with the data transferred

hwaddr len
the number of bytes to write

Description

Return a MemTxResult indicating whether the operation succeeded or failed (eg unassigned memory, device rejected the transaction, IOMMU fault).

MemTxResult **address_space_write_rom**(*AddressSpace* *as, hwaddr addr, MemTxAttrs attrs, const void *buf, hwaddr len)

write to address space, including ROM.

Parameters

AddressSpace *as
AddressSpace to be accessed

hwaddr addr
address within that address space

MemTxAttrs attrs
memory transaction attributes

const void *buf
buffer with the data transferred

hwaddr len
the number of bytes to write

Description

This function writes to the specified address space, but will write data to both ROM and RAM. This is used for non-guest writes like writes from the gdb debug stub or initial loading of ROM contents.

Note that portions of the write which attempt to write data to a device will be silently ignored – only real RAM and ROM will be written to.

Return a MemTxResult indicating whether the operation succeeded or failed (eg unassigned memory, device rejected the transaction, IOMMU fault).

void **address_space_cache_init_empty**(MemoryRegionCache *cache)

Initialize empty `MemoryRegionCache`

Parameters

MemoryRegionCache *cache
The `MemoryRegionCache` to operate on.

Description

Initializes `MemoryRegionCache` structure without memory region attached. Cache initialized this way can only be safely destroyed, but not used.

void **address_space_cache_invalidate**(MemoryRegionCache *cache, hwaddr addr, hwaddr access_len)

complete a write to a `MemoryRegionCache`

Parameters

MemoryRegionCache *cache
The `MemoryRegionCache` to operate on.

hwaddr addr
The first physical address that was written, relative to the address that was passed to **address_space_cache_init**.

hwaddr access_len
The number of bytes that were written starting at **addr**.

void **address_space_cache_destroy**(MemoryRegionCache *cache)

free a `MemoryRegionCache`

Parameters

MemoryRegionCache *cache
The `MemoryRegionCache` whose memory should be released.

MemTxResult **address_space_read**(*AddressSpace* *as, hwaddr addr, MemTxAttrs attrs, void *buf, hwaddr len)

read from an address space.

Parameters

AddressSpace *as
AddressSpace to be accessed

hwaddr addr
address within that address space

MemTxAttrs attrs
memory transaction attributes

void *buf
buffer with the data transferred

hwaddr len
length of the data transferred

Description

Return a MemTxResult indicating whether the operation succeeded or failed (eg unassigned memory, device rejected the transaction, IOMMU fault). Called within RCU critical section.

MemTxResult **address_space_read_cached**(MemoryRegionCache *cache, hwaddr addr, void *buf, hwaddr len)

read from a cached RAM region

Parameters

MemoryRegionCache *cache
Cached region to be addressed

hwaddr addr
address relative to the base of the RAM region

void *buf
buffer with the data transferred

hwaddr len
length of the data transferred

MemTxResult **address_space_write_cached**(MemoryRegionCache *cache, hwaddr addr, const void *buf, hwaddr len)

write to a cached RAM region

Parameters

MemoryRegionCache *cache
Cached region to be addressed

hwaddr addr
address relative to the base of the RAM region

const void *buf
buffer with the data transferred

hwaddr len
length of the data transferred

MemTxResult **address_space_set**(*AddressSpace* *as, hwaddr addr, uint8_t c, hwaddr len, MemTxAttrs attrs)

Fill address space with a constant byte.

Parameters

AddressSpace *as
AddressSpace to be accessed

hwaddr addr
address within that address space

uint8_t c
constant byte to fill the memory

hwaddr len
the number of bytes to fill with the constant byte

MemTxAttrs attrs
memory transaction attributes

Description

Return a MemTxResult indicating whether the operation succeeded or failed (eg unassigned memory, device rejected the transaction, IOMMU fault).

7.3.4 QEMU modules

module info annotation macros

`scripts/modinfo-collect.py` will collect module info, using the preprocessor and -DQEMU_MODINFO.

`scripts/modinfo-generate.py` will create a module meta-data database from the collected information so qemu knows about module dependencies and QOM objects implemented by modules.

See `*.modinfo` and `modinfo.c` in the build directory to check the script results.

module_obj

`module_obj (name)`

Parameters

name
QOM type.

Description

This module implements QOM type **name**.

module_dep

`module_dep (name)`

Parameters

name
module name

Description

This module depends on module **name**.

module_arch

`module_arch (name)`

Parameters

name
target architecture

Description

This module is for target architecture **arch**.

Note that target-dependent modules are tagged automatically, so this is only needed in case target-independent modules should be restricted. Use case example: the ccw bus is implemented by s390x only.

module_opts

module_opts (name)

Parameters

name
QemuOpts name

Description

This module registers QemuOpts **name**.

module_kconfig

module_kconfig (name)

Parameters

name
Kconfig requirement necessary to load the module

Description

This module requires a core module that should be implemented and enabled in Kconfig.

7.3.5 PCI subsystem

API Reference

struct **PCIIOMMUOps**
callbacks structure for specific IOMMU handlers of a PCIBus

Definition

```
struct PCIIOMMUOps {
  AddressSpace * (*get_address_space)(PCIBus *bus, void *opaque, int devfn);
};
```

Members

get_address_space
get the address space for a set of devices on a PCI bus.

Mandatory callback which returns a pointer to an *AddressSpace*

bus: the PCIBus being accessed.

opaque: the data passed to pci_setup_iommu().

devfn: device and function number

Description

Allows to modify the behavior of some IOMMU operations of the PCI framework for a set of devices on a PCI bus.

void **pci_setup_iommu**(PCIBus *bus, const *PCIIOMMUOps* *ops, void *opaque)
Initialize specific IOMMU handlers for a PCIBus

Parameters

PCIBus *bus
the PCIBus being updated.

const PCIIOMMUOps *ops
the *PCIIOMMUOps*

void *opaque
passed to callbacks of the **ops** structure.

Description

Let PCI host bridges define specific operations.

7.3.6 QEMU Object Model (QOM) API Reference

This is the complete API documentation for *The QEMU Object Model (QOM)*.

ObjectPropertyAccessor

Typedef:

Syntax

```
void ObjectPropertyAccessor (Object *obj, Visitor *v, const char *name, void
*opaque, Error **errp)
```

Parameters

Object *obj
the object that owns the property

Visitor *v
the visitor that contains the property data

const char *name
the name of the property

void *opaque
the object property opaque

Error **errp
a pointer to an Error that is filled if getting/setting fails.

Description

Called when trying to get/set a property.

ObjectPropertyResolve

Typedef:

Syntax

```
Object * ObjectPropertyResolve (Object *obj, void *opaque, const char *part)
```

Parameters

Object *obj
the object that owns the property

void *opaque
the opaque registered with the property

const char *part
the name of the property

Description

Resolves the *Object* corresponding to property **part**.

The returned object can also be used as a starting point to resolve a relative path starting with "**part**".

Return

If **path** is the path that led to **obj**, the function returns the *Object* corresponding to "**path/part**". If "**path/part**" is not a valid object path, it returns NULL.

`ObjectPropertyRelease`

Typedef:

Syntax

```
void ObjectPropertyRelease (Object *obj, const char *name, void *opaque)
```

Parameters

`Object *obj`
the object that owns the property

`const char *name`
the name of the property

`void *opaque`
the opaque registered with the property

Description

Called when a property is removed from a object.

`ObjectPropertyInit`

Typedef:

Syntax

```
void ObjectPropertyInit (Object *obj, ObjectProperty *prop)
```

Parameters

`Object *obj`
the object that owns the property

`ObjectProperty *prop`
the property to set

Description

Called when a property is initialized.

`ObjectUnparent`

Typedef:

Syntax

```
void ObjectUnparent (Object *obj)
```

Parameters

`Object *obj`
the object that is being removed from the composition tree

Description

Called when an object is being removed from the QOM composition tree. The function should remove any backlinks from children objects to **obj**.

`ObjectFree`

Typedef:

Syntax

```
void ObjectFree (void *obj)
```

Parameters

`void *obj`
the object being freed

Description

Called when an object's last reference is removed.

struct `ObjectClass`

Definition

```
struct ObjectClass {
};
```

Members

Description

The base for all classes. The only thing that *ObjectClass* contains is an integer type handle.

struct `Object`

Definition

```
struct Object {
};
```

Members

Description

The base for all objects. The first member of this object is a pointer to a *ObjectClass*. Since C guarantees that the first member of a structure always begins at byte 0 of that structure, as long as any sub-object places its parent as the first member, we can cast directly to a *Object*.

As a result, *Object* contains a reference to the objects type as its first member. This allows identification of the real type of the object at run time.

`DECLARE_INSTANCE_CHECKER`

```
DECLARE_INSTANCE_CHECKER (InstanceType, OBJ_NAME, TYPENAME)
```

Parameters

`InstanceType`
instance struct name

`OBJ_NAME`
the object name in uppercase with underscore separators

TYPENAME
type name

Description

Direct usage of this macro should be avoided, and the complete OBJECT_DECLARE_TYPE macro is recommended instead.

This macro will provide the instance type cast functions for a QOM type.

DECLARE_CLASS_CHECKERS

```
DECLARE_CLASS_CHECKERS (ClassType, OBJ_NAME, TYPENAME)
```

Parameters

ClassType
class struct name

OBJ_NAME
the object name in uppercase with underscore separators

TYPENAME
type name

Description

Direct usage of this macro should be avoided, and the complete OBJECT_DECLARE_TYPE macro is recommended instead.

This macro will provide the class type cast functions for a QOM type.

DECLARE_OBJ_CHECKERS

```
DECLARE_OBJ_CHECKERS (InstanceType, ClassType, OBJ_NAME, TYPENAME)
```

Parameters

InstanceType
instance struct name

ClassType
class struct name

OBJ_NAME
the object name in uppercase with underscore separators

TYPENAME
type name

Description

Direct usage of this macro should be avoided, and the complete OBJECT_DECLARE_TYPE macro is recommended instead.

This macro will provide the three standard type cast functions for a QOM type.

OBJECT_DECLARE_TYPE

```
OBJECT_DECLARE_TYPE (InstanceType, ClassType, MODULE_OBJ_NAME)
```

Parameters

InstanceType
instance struct name

ClassType
class struct name

MODULE_OBJ_NAME
the object name in uppercase with underscore separators

Description

This macro is typically used in a header file, and will:

- create the typedefs for the object and class structs
- register the type for use with g_autoptr
- provide three standard type cast functions

The object struct and class struct need to be declared manually.

OBJECT_DECLARE_SIMPLE_TYPE

```
OBJECT_DECLARE_SIMPLE_TYPE (InstanceType, MODULE_OBJ_NAME)
```

Parameters

InstanceType
instance struct name

MODULE_OBJ_NAME
the object name in uppercase with underscore separators

Description

This does the same as OBJECT_DECLARE_TYPE(), but with no class struct declared.

This macro should be used unless the class struct needs to have virtual methods declared.

DO_OBJECT_DEFINE_TYPE_EXTENDED

```
DO_OBJECT_DEFINE_TYPE_EXTENDED (ModuleObjName, module_obj_name, MODULE_OBJ_NAME,
PARENT_MODULE_OBJ_NAME, ABSTRACT, CLASS_SIZE, ...)
```

Parameters

ModuleObjName
the object name with initial caps

module_obj_name
the object name in lowercase with underscore separators

MODULE_OBJ_NAME
the object name in uppercase with underscore separators

PARENT_MODULE_OBJ_NAME
the parent object name in uppercase with underscore separators

ABSTRACT
boolean flag to indicate whether the object can be instantiated

CLASS_SIZE
size of the type's class

...
list of initializers for "InterfaceInfo" to declare implemented interfaces

Description

This is the base macro used to implement all the OBJECT_DEFINE_* macros. It should never be used directly in a source file.

OBJECT_DEFINE_TYPE_EXTENDED

```
OBJECT_DEFINE_TYPE_EXTENDED (ModuleObjName, module_obj_name, MODULE_OBJ_NAME,
PARENT_MODULE_OBJ_NAME, ABSTRACT, ...)
```

Parameters

ModuleObjName
the object name with initial caps

module_obj_name
the object name in lowercase with underscore separators

MODULE_OBJ_NAME
the object name in uppercase with underscore separators

PARENT_MODULE_OBJ_NAME
the parent object name in uppercase with underscore separators

ABSTRACT
boolean flag to indicate whether the object can be instantiated

...
list of initializers for "InterfaceInfo" to declare implemented interfaces

Description

This macro is typically used in a source file, and will:

- declare prototypes for _finalize, _class_init and _init methods
- declare the TypeInfo struct instance
- provide the constructor to register the type

After using this macro, implementations of the _finalize, _class_init, and _init methods need to be written. Any of these can be zero-line no-op impls if no special logic is required for a given type.

This macro should rarely be used, instead one of the more specialized macros is usually a better choice.

OBJECT_DEFINE_TYPE

```
OBJECT_DEFINE_TYPE (ModuleObjName, module_obj_name, MODULE_OBJ_NAME,
PARENT_MODULE_OBJ_NAME)
```

Parameters

ModuleObjName
the object name with initial caps

module_obj_name
the object name in lowercase with underscore separators

MODULE_OBJ_NAME
the object name in uppercase with underscore separators

PARENT_MODULE_OBJ_NAME
the parent object name in uppercase with underscore separators

Description

This is a specialization of OBJECT_DEFINE_TYPE_EXTENDED, which is suitable for the common case of a non-abstract type, without any interfaces.

OBJECT_DEFINE_TYPE_WITH_INTERFACES

```
OBJECT_DEFINE_TYPE_WITH_INTERFACES (ModuleObjName, module_obj_name, MODULE_OBJ_NAME,
PARENT_MODULE_OBJ_NAME, ...)
```

Parameters

ModuleObjName
the object name with initial caps

module_obj_name
the object name in lowercase with underscore separators

MODULE_OBJ_NAME
the object name in uppercase with underscore separators

PARENT_MODULE_OBJ_NAME
the parent object name in uppercase with underscore separators

...
list of initializers for "InterfaceInfo" to declare implemented interfaces

Description

This is a specialization of OBJECT_DEFINE_TYPE_EXTENDED, which is suitable for the common case of a non-abstract type, with one or more implemented interfaces.

Note when passing the list of interfaces, be sure to include the final NULL entry, e.g. { TYPE_USER_CREATABLE }, { NULL }

OBJECT_DEFINE_ABSTRACT_TYPE

```
OBJECT_DEFINE_ABSTRACT_TYPE (ModuleObjName, module_obj_name, MODULE_OBJ_NAME,
PARENT_MODULE_OBJ_NAME)
```

Parameters

ModuleObjName
the object name with initial caps

module_obj_name
the object name in lowercase with underscore separators

MODULE_OBJ_NAME
the object name in uppercase with underscore separators

PARENT_MODULE_OBJ_NAME
the parent object name in uppercase with underscore separators

Description

This is a specialization of OBJECT_DEFINE_TYPE_EXTENDED, which is suitable for defining an abstract type, without any interfaces.

OBJECT_DEFINE_SIMPLE_TYPE_WITH_INTERFACES

```
OBJECT_DEFINE_SIMPLE_TYPE_WITH_INTERFACES (ModuleObjName, module_obj_name,
MODULE_OBJ_NAME, PARENT_MODULE_OBJ_NAME, ...)
```

Parameters

ModuleObjName
the object name with initial caps

module_obj_name
the object name in lowercase with underscore separators

MODULE_OBJ_NAME
the object name in uppercase with underscore separators

PARENT_MODULE_OBJ_NAME
the parent object name in uppercase with underscore separators

...
variable arguments

Description

This is a variant of OBJECT_DEFINE_TYPE_EXTENDED, which is suitable for the case of a non-abstract type, with interfaces, and with no requirement for a class struct.

OBJECT_DEFINE_SIMPLE_TYPE

```
OBJECT_DEFINE_SIMPLE_TYPE (ModuleObjName, module_obj_name, MODULE_OBJ_NAME,
PARENT_MODULE_OBJ_NAME)
```

Parameters

ModuleObjName
the object name with initial caps

module_obj_name
the object name in lowercase with underscore separators

MODULE_OBJ_NAME
the object name in uppercase with underscore separators

PARENT_MODULE_OBJ_NAME
the parent object name in uppercase with underscore separators

Description

This is a variant of OBJECT_DEFINE_TYPE_EXTENDED, which is suitable for the common case of a non-abstract type, without any interfaces, and with no requirement for a class struct. If you declared your type with OBJECT_DECLARE_SIMPLE_TYPE then this is probably the right choice for defining it.

struct **TypeInfo**

Definition

```
struct TypeInfo {
  const char *name;
  const char *parent;
  size_t instance_size;
  size_t instance_align;
  void (*instance_init)(Object *obj);
  void (*instance_post_init)(Object *obj);
  void (*instance_finalize)(Object *obj);
  bool abstract;
  size_t class_size;
  void (*class_init)(ObjectClass *klass, void *data);
  void (*class_base_init)(ObjectClass *klass, void *data);
```

(continues on next page)

(continued from previous page)

```
  void *class_data;
  InterfaceInfo *interfaces;
};
```

Members

name
The name of the type.

parent
The name of the parent type.

instance_size
The size of the object (derivative of *Object*). If **instance_size** is 0, then the size of the object will be the size of the parent object.

instance_align
The required alignment of the object. If **instance_align** is 0, then normal malloc alignment is sufficient; if non-zero, then we must use qemu_memalign for allocation.

instance_init
This function is called to initialize an object. The parent class will have already been initialized so the type is only responsible for initializing its own members.

instance_post_init
This function is called to finish initialization of an object, after all **instance_init** functions were called.

instance_finalize
This function is called during object destruction. This is called before the parent **instance_finalize** function has been called. An object should only free the members that are unique to its type in this function.

abstract
If this field is true, then the class is considered abstract and cannot be directly instantiated.

class_size
The size of the class object (derivative of *ObjectClass*) for this object. If **class_size** is 0, then the size of the class will be assumed to be the size of the parent class. This allows a type to avoid implementing an explicit class type if they are not adding additional virtual functions.

class_init
This function is called after all parent class initialization has occurred to allow a class to set its default virtual method pointers. This is also the function to use to override virtual methods from a parent class.

class_base_init
This function is called for all base classes after all parent class initialization has occurred, but before the class itself is initialized. This is the function to use to undo the effects of memcpy from the parent class to the descendants.

class_data
Data to pass to the **class_init**, **class_base_init**. This can be useful when building dynamic classes.

interfaces
The list of interfaces associated with this type. This should point to a static array that's terminated with a zero filled element.

OBJECT

```
OBJECT (obj)
```

Parameters

obj
A derivative of *Object*

Description

Converts an object to a *Object*. Since all objects are Objects, this function will always succeed.

OBJECT_CLASS

```
OBJECT_CLASS (class)
```

Parameters

class
A derivative of *ObjectClass*.

Description

Converts a class to an *ObjectClass*. Since all objects are Objects, this function will always succeed.

OBJECT_CHECK

```
OBJECT_CHECK (type, obj, name)
```

Parameters

type
The C type to use for the return value.

obj
A derivative of **type** to cast.

name
The QOM typename of **type**

Description

A type safe version of **object_dynamic_cast_assert**. Typically each class will define a macro based on this type to perform type safe dynamic_casts to this object type.

If an invalid object is passed to this function, a run time assert will be generated.

OBJECT_CLASS_CHECK

```
OBJECT_CLASS_CHECK (class_type, class, name)
```

Parameters

class_type
The C type to use for the return value.

class
A derivative class of **class_type** to cast.

name
the QOM typename of **class_type**.

Description

A type safe version of **object_class_dynamic_cast_assert**. This macro is typically wrapped by each type to perform type safe casts of a class to a specific class type.

OBJECT_GET_CLASS

```
OBJECT_GET_CLASS (class, obj, name)
```

Parameters

class
The C type to use for the return value.

obj
The object to obtain the class for.

name
The QOM typename of **obj**.

Description

This function will return a specific class for a given object. Its generally used by each type to provide a type safe macro to get a specific class type from an object.

struct **InterfaceInfo**

Definition

```
struct InterfaceInfo {
  const char *type;
};
```

Members

type
The name of the interface.

Description

The information associated with an interface.

struct **InterfaceClass**

Definition

```
struct InterfaceClass {
  ObjectClass parent_class;
};
```

Members

parent_class
the base class

Description

The class for all interfaces. Subclasses of this class should only add virtual methods.

INTERFACE_CLASS

```
INTERFACE_CLASS (klass)
```

Parameters

klass
class to cast from

Return

An *InterfaceClass* or raise an error if cast is invalid

INTERFACE_CHECK

INTERFACE_CHECK (interface, obj, name)

Parameters

interface
the type to return

obj
the object to convert to an interface

name
the interface type name

Return

obj casted to **interface** if cast is valid, otherwise raise error.

Object ***object_new_with_class**(*ObjectClass* *klass)

Parameters

ObjectClass *klass
The class to instantiate.

Description

This function will initialize a new object using heap allocated memory. The returned object has a reference count of 1, and will be freed when the last reference is dropped.

Return

The newly allocated and instantiated object.

Object ***object_new**(const char *typename)

Parameters

const char *typename
The name of the type of the object to instantiate.

Description

This function will initialize a new object using heap allocated memory. The returned object has a reference count of 1, and will be freed when the last reference is dropped.

Return

The newly allocated and instantiated object.

Object ***object_new_with_props**(const char *typename, *Object* *parent, const char *id, Error **errp, ...)

Parameters

const char *typename
The name of the type of the object to instantiate.

Object *parent
the parent object

const char *id
The unique ID of the object

Error **errp
pointer to error object

...
: list of property names and values

Description

This function will initialize a new object using heap allocated memory. The returned object has a reference count of 1, and will be freed when the last reference is dropped.

The **id** parameter will be used when registering the object as a child of **parent** in the composition tree.

The variadic parameters are a list of pairs of (propname, propvalue) strings. The propname of `NULL` indicates the end of the property list. If the object implements the user creatable interface, the object will be marked complete once all the properties have been processed.

Listing 1: Creating an object with properties

```
Error *err = NULL;
Object *obj;

obj = object_new_with_props(TYPE_MEMORY_BACKEND_FILE,
                            object_get_objects_root(),
                            "hostmem0",
                            &err,
                            "share", "yes",
                            "mem-path", "/dev/shm/somefile",
                            "prealloc", "yes",
                            "size", "1048576",
                            NULL);

if (!obj) {
  error_reportf_err(err, "Cannot create memory backend: ");
}
```

The returned object will have one stable reference maintained for as long as it is present in the object hierarchy.

Return

The newly allocated, instantiated & initialized object.

Object ***object_new_with_propv**(const char *typename, *Object* *parent, const char *id, Error **errp, va_list vargs)

Parameters

const char *typename
: The name of the type of the object to instantiate.

Object *parent
: the parent object

const char *id
: The unique ID of the object

Error **errp
: pointer to error object

va_list vargs
: list of property names and values

Description

See object_new_with_props() for documentation.

bool **object_set_props**(*Object* *obj, Error **errp, ...)

Parameters

Object *obj
: the object instance to set properties on

Error **errp
: pointer to error object

...
: list of property names and values

Description

This function will set a list of properties on an existing object instance.

The variadic parameters are a list of pairs of (propname, propvalue) strings. The propname of `NULL` indicates the end of the property list.

Listing 2: Update an object's properties

```
Error *err = NULL;
Object *obj = ...get / create object...;

if (!object_set_props(obj,
                      &err,
                      "share", "yes",
                      "mem-path", "/dev/shm/somefile",
                      "prealloc", "yes",
                      "size", "1048576",
                      NULL)) {
  error_reportf_err(err, "Cannot set properties: ");
}
```

The returned object will have one stable reference maintained for as long as it is present in the object hierarchy.

Return

`true` on success, `false` on error.

bool **object_set_propv**(*Object* *obj, Error **errp, va_list vargs)

Parameters

Object *obj
: the object instance to set properties on

Error **errp
: pointer to error object

va_list vargs
: list of property names and values

Description

See object_set_props() for documentation.

Return

`true` on success, `false` on error.

void **object_initialize**(void *obj, size_t size, const char *typename)

Parameters

void *obj
A pointer to the memory to be used for the object.

size_t size
The maximum size available at **obj** for the object.

const char *typename
The name of the type of the object to instantiate.

Description

This function will initialize an object. The memory for the object should have already been allocated. The returned object has a reference count of 1, and will be finalized when the last reference is dropped.

bool **object_initialize_child_with_props**(*Object* *parentobj, const char *propname, void *childobj, size_t size, const char *type, Error **errp, ...)

Parameters

Object *parentobj
The parent object to add a property to

const char *propname
The name of the property

void *childobj
A pointer to the memory to be used for the object.

size_t size
The maximum size available at **childobj** for the object.

const char *type
The name of the type of the object to instantiate.

Error **errp
If an error occurs, a pointer to an area to store the error

...
list of property names and values

Description

This function will initialize an object. The memory for the object should have already been allocated. The object will then be added as child property to a parent with object_property_add_child() function. The returned object has a reference count of 1 (for the "child<...>" property from the parent), so the object will be finalized automatically when the parent gets removed.

The variadic parameters are a list of pairs of (propname, propvalue) strings. The propname of `NULL` indicates the end of the property list. If the object implements the user creatable interface, the object will be marked complete once all the properties have been processed.

Return

`true` on success, `false` on failure.

bool **object_initialize_child_with_propsv**(*Object* *parentobj, const char *propname, void *childobj, size_t size, const char *type, Error **errp, va_list vargs)

Parameters

Object *parentobj
The parent object to add a property to

const char *propname
The name of the property

void *childobj
A pointer to the memory to be used for the object.

size_t size
The maximum size available at **childobj** for the object.

const char *type
The name of the type of the object to instantiate.

Error **errp
If an error occurs, a pointer to an area to store the error

va_list vargs
list of property names and values

Description

See object_initialize_child() for documentation.

Return

`true` on success, `false` on failure.

object_initialize_child

`object_initialize_child (parent, propname, child, type)`

Parameters

parent
The parent object to add a property to

propname
The name of the property

child
A precisely typed pointer to the memory to be used for the object.

type
The name of the type of the object to instantiate.

Description

This is like:

```
object_initialize_child_with_props(parent, propname,
                                   child, sizeof(*child), type,
                                   &error_abort, NULL)
```

Object ***object_dynamic_cast**(*Object* *obj, const char *typename)

Parameters

Object *obj
The object to cast.

const char *typename
The **typename** to cast to.

Description

This function will determine if **obj** is-a **typename**. **obj** can refer to an object or an interface associated with an object.

Return

This function returns **obj** on success or `NULL` on failure.

Object ***object_dynamic_cast_assert**(*Object* *obj, const char *typename, const char *file, int line, const char *func)

Parameters

Object *obj
The object to cast.

const char *typename
The **typename** to cast to.

const char *file
Source code file where function was called

int line
Source code line where function was called

const char *func
Name of function where this function was called

Description

See object_dynamic_cast() for a description of the parameters of this function. The only difference in behavior is that this function asserts instead of returning `NULL` on failure if QOM cast debugging is enabled. This function is not meant to be called directly, but only through the wrapper macro OBJECT_CHECK.

ObjectClass ***object_get_class**(*Object* *obj)

Parameters

Object *obj
A derivative of *Object*

Return

The *ObjectClass* of the type associated with **obj**.

const char ***object_get_typename**(const *Object* *obj)

Parameters

const Object *obj
A derivative of *Object*.

Return

The QOM typename of **obj**.

Type **type_register_static**(const *TypeInfo* *info)

Parameters

const TypeInfo *info
The *TypeInfo* of the new type.

Description

info and all of the strings it points to should exist for the life time that the type is registered.

Return

the new Type.

Type **type_register**(const *TypeInfo* *info)

Parameters

const TypeInfo *info
The *TypeInfo* of the new type

Description

Unlike type_register_static(), this call does not require **info** or its string members to continue to exist after the call returns.

Return

the new Type.

void **type_register_static_array**(const *TypeInfo* *infos, int nr_infos)

Parameters

const TypeInfo *infos
The array of the new type *TypeInfo* structures.

int nr_infos
number of entries in **infos**

Description

infos and all of the strings it points to should exist for the life time that the type is registered.

DEFINE_TYPES

```
DEFINE_TYPES (type_array)
```

Parameters

type_array
The array containing *TypeInfo* structures to register

Description

type_array should be static constant that exists for the life time that the type is registered.

bool **type_print_class_properties**(const char *type)

Parameters

const char *type
a QOM class name

Description

Print the object's class properties to stdout or the monitor. Return whether an object was found.

void **object_set_properties_from_keyval**(*Object* *obj, const QDict *qdict, bool from_json, Error **errp)

Parameters

Object *obj
a QOM object

const QDict *qdict
a dictionary with the properties to be set

bool from_json
: true if leaf values of **qdict** are typed, false if they are strings

Error **errp
: pointer to error object

Description

For each key in the dictionary, parse the value string if needed, then set the corresponding property in **obj**.

ObjectClass ***object_class_dynamic_cast_assert**(*ObjectClass* *klass, const char *typename, const char *file, int line, const char *func)

Parameters

ObjectClass *klass
: The *ObjectClass* to attempt to cast.

const char *typename
: The QOM typename of the class to cast to.

const char *file
: Source code file where function was called

int line
: Source code line where function was called

const char *func
: Name of function where this function was called

Description

See object_class_dynamic_cast() for a description of the parameters of this function. The only difference in behavior is that this function asserts instead of returning NULL on failure if QOM cast debugging is enabled. This function is not meant to be called directly, but only through the wrapper macro OBJECT_CLASS_CHECK.

ObjectClass ***object_class_dynamic_cast**(*ObjectClass* *klass, const char *typename)

Parameters

ObjectClass *klass
: The *ObjectClass* to attempt to cast.

const char *typename
: The QOM typename of the class to cast to.

Return

If **typename** is a class, this function returns **klass** if **typename** is a subtype of **klass**, else returns NULL.

Description

If **typename** is an interface, this function returns the interface definition for **klass** if **klass** implements it unambiguously; NULL is returned if **klass** does not implement the interface or if multiple classes or interfaces on the hierarchy leading to **klass** implement it. (FIXME: perhaps this can be detected at type definition time?)

ObjectClass ***object_class_get_parent**(*ObjectClass* *klass)

Parameters

ObjectClass *klass
: The class to obtain the parent for.

Return

The parent for **klass** or NULL if none.

const char *object_class_get_name(*ObjectClass* *klass)

Parameters

ObjectClass *klass
The class to obtain the QOM typename for.

Return

The QOM typename for **klass**.

bool object_class_is_abstract(*ObjectClass* *klass)

Parameters

ObjectClass *klass
The class to obtain the abstractness for.

Return

`true` if **klass** is abstract, `false` otherwise.

ObjectClass *object_class_by_name(const char *typename)

Parameters

const char *typename
The QOM typename to obtain the class for.

Return

The class for **typename** or `NULL` if not found.

ObjectClass *module_object_class_by_name(const char *typename)

Parameters

const char *typename
The QOM typename to obtain the class for.

Description

For objects which might be provided by a module. Behaves like object_class_by_name, but additionally tries to load the module needed in case the class is not available.

Return

The class for **typename** or `NULL` if not found.

GSList *object_class_get_list(const char *implements_type, bool include_abstract)

Parameters

const char *implements_type
The type to filter for, including its derivatives.

bool include_abstract
Whether to include abstract classes.

Return

A singly-linked list of the classes in reverse hashtable order.

GSList *object_class_get_list_sorted(const char *implements_type, bool include_abstract)

Parameters

const char *implements_type
The type to filter for, including its derivatives.

bool include_abstract
Whether to include abstract classes.

Return

A singly-linked list of the classes in alphabetical case-insensitive order.

Object *object_ref(void *obj)

Parameters

void *obj
the object

Description

Increase the reference count of a object. A object cannot be freed as long as its reference count is greater than zero.

Return

obj

void object_unref(void *obj)

Parameters

void *obj
the object

Description

Decrease the reference count of a object. A object cannot be freed as long as its reference count is greater than zero.

ObjectProperty *object_property_try_add(*Object* *obj, const char *name, const char *type, *ObjectPropertyAccessor* *get, *ObjectPropertyAccessor* *set, *ObjectPropertyRelease* *release, void *opaque, Error **errp)

Parameters

Object *obj
the object to add a property to

const char *name
the name of the property. This can contain any character except for a forward slash. In general, you should use hyphens '-' instead of underscores '_' when naming properties.

const char *type
the type name of the property. This namespace is pretty loosely defined. Sub namespaces are constructed by using a prefix and then to angle brackets. For instance, the type 'virtio-net-pci' in the 'link' namespace would be 'link<virtio-net-pci>'.

ObjectPropertyAccessor *get
The getter to be called to read a property. If this is NULL, then the property cannot be read.

ObjectPropertyAccessor *set
the setter to be called to write a property. If this is NULL, then the property cannot be written.

ObjectPropertyRelease *release
called when the property is removed from the object. This is meant to allow a property to free its opaque upon object destruction. This may be NULL.

void *opaque
an opaque pointer to pass to the callbacks for the property

Error **errp
pointer to error object

Return

The ObjectProperty; this can be used to set the **resolve** callback for child and link properties.

ObjectProperty ***object_property_add**(*Object* *obj, const char *name, const char *type, *ObjectPropertyAccessor* *get, *ObjectPropertyAccessor* *set, *ObjectPropertyRelease* *release, void *opaque)

Same as object_property_try_add() with **errp** hardcoded to &error_abort.

Parameters

Object *obj
the object to add a property to

const char *name
the name of the property. This can contain any character except for a forward slash. In general, you should use hyphens '-' instead of underscores '_' when naming properties.

const char *type
the type name of the property. This namespace is pretty loosely defined. Sub namespaces are constructed by using a prefix and then to angle brackets. For instance, the type 'virtio-net-pci' in the 'link' namespace would be 'link<virtio-net-pci>'.

ObjectPropertyAccessor *get
The getter to be called to read a property. If this is NULL, then the property cannot be read.

ObjectPropertyAccessor *set
the setter to be called to write a property. If this is NULL, then the property cannot be written.

ObjectPropertyRelease *release
called when the property is removed from the object. This is meant to allow a property to free its opaque upon object destruction. This may be NULL.

void *opaque
an opaque pointer to pass to the callbacks for the property

void **object_property_set_default_bool**(ObjectProperty *prop, bool value)

Parameters

ObjectProperty *prop
the property to set

bool value
the value to be written to the property

Description

Set the property default value.

void **object_property_set_default_str**(ObjectProperty *prop, const char *value)

Parameters

ObjectProperty *prop
: the property to set

const char *value
: the value to be written to the property

Description

Set the property default value.

void **object_property_set_default_list**(ObjectProperty *prop)

Parameters

ObjectProperty *prop
: the property to set

Description

Set the property default value to be an empty list.

void **object_property_set_default_int**(ObjectProperty *prop, int64_t value)

Parameters

ObjectProperty *prop
: the property to set

int64_t value
: the value to be written to the property

Description

Set the property default value.

void **object_property_set_default_uint**(ObjectProperty *prop, uint64_t value)

Parameters

ObjectProperty *prop
: the property to set

uint64_t value
: the value to be written to the property

Description

Set the property default value.

ObjectProperty ***object_property_find**(*Object* *obj, const char *name)

Parameters

Object *obj
: the object

const char *name
: the name of the property

Description

Look up a property for an object.

Return its `ObjectProperty` if found, or NULL.

ObjectProperty *object_property_find_err(*Object* *obj, const char *name, Error **errp)

Parameters

Object *obj
the object

const char *name
the name of the property

Error **errp
returns an error if this function fails

Description

Look up a property for an object.

Return its `ObjectProperty` if found, or NULL.

ObjectProperty *object_class_property_find(*ObjectClass* *klass, const char *name)

Parameters

ObjectClass *klass
the object class

const char *name
the name of the property

Description

Look up a property for an object class.

Return its `ObjectProperty` if found, or NULL.

ObjectProperty *object_class_property_find_err(*ObjectClass* *klass, const char *name, Error **errp)

Parameters

ObjectClass *klass
the object class

const char *name
the name of the property

Error **errp
returns an error if this function fails

Description

Look up a property for an object class.

Return its `ObjectProperty` if found, or NULL.

void object_property_iter_init(ObjectPropertyIterator *iter, *Object* *obj)

Parameters

ObjectPropertyIterator *iter
the iterator instance

Object *obj
the object

Description

Initializes an iterator for traversing all properties registered against an object instance, its class and all parent classes.

It is forbidden to modify the property list while iterating, whether removing or adding properties.

Typical usage pattern would be

Listing 3: Using object property iterators

```
ObjectProperty *prop;
ObjectPropertyIterator iter;

object_property_iter_init(&iter, obj);
while ((prop = object_property_iter_next(&iter))) {
  ... do something with prop ...
}
```

void **object_class_property_iter_init**(ObjectPropertyIterator *iter, *ObjectClass* *klass)

Parameters

ObjectPropertyIterator *iter
: the iterator instance

ObjectClass *klass
: the class

Description

Initializes an iterator for traversing all properties registered against an object class and all parent classes.

It is forbidden to modify the property list while iterating, whether removing or adding properties.

This can be used on abstract classes as it does not create a temporary instance.

ObjectProperty ***object_property_iter_next**(ObjectPropertyIterator *iter)

Parameters

ObjectPropertyIterator *iter
: the iterator instance

Description

Return the next available property. If no further properties are available, a `NULL` value will be returned and the **iter** pointer should not be used again after this point without re-initializing it.

Return

the next property, or `NULL` when all properties have been traversed.

bool **object_property_get**(*Object* *obj, const char *name, Visitor *v, Error **errp)

Parameters

Object *obj
: the object

const char *name
: the name of the property

Visitor *v
: the visitor that will receive the property value. This should be an Output visitor and the data will be written with **name** as the name.

Error **errp
: returns an error if this function fails

Description

Reads a property from a object.

Return

`true` on success, `false` on failure.

bool **object_property_set_str**(*Object* *obj, const char *name, const char *value, Error **errp)

Parameters

Object *obj
: the object

const char *name
: the name of the property

const char *value
: the value to be written to the property

Error **errp
: returns an error if this function fails

Description

Writes a string value to a property.

Return

`true` on success, `false` on failure.

char ***object_property_get_str**(*Object* *obj, const char *name, Error **errp)

Parameters

Object *obj
: the object

const char *name
: the name of the property

Error **errp
: returns an error if this function fails

Return

the value of the property, converted to a C string, or NULL if an error occurs (including when the property value is not a string). The caller should free the string.

bool **object_property_set_link**(*Object* *obj, const char *name, *Object* *value, Error **errp)

Parameters

Object *obj
: the object

const char *name
: the name of the property

Object *value
: the value to be written to the property

Error **errp
returns an error if this function fails

Description

Writes an object's canonical path to a property.

If the link property was created with `OBJ_PROP_LINK_STRONG` bit, the old target object is unreferenced, and a reference is added to the new target object.

Return

`true` on success, `false` on failure.

Object ***object_property_get_link**(*Object* *obj, const char *name, Error **errp)

Parameters

Object *obj
the object

const char *name
the name of the property

Error **errp
returns an error if this function fails

Return

the value of the property, resolved from a path to an Object, or NULL if an error occurs (including when the property value is not a string or not a valid object path).

bool **object_property_set_bool**(*Object* *obj, const char *name, bool value, Error **errp)

Parameters

Object *obj
the object

const char *name
the name of the property

bool value
the value to be written to the property

Error **errp
returns an error if this function fails

Description

Writes a bool value to a property.

Return

`true` on success, `false` on failure.

bool **object_property_get_bool**(*Object* *obj, const char *name, Error **errp)

Parameters

Object *obj
the object

const char *name
the name of the property

Error **errp
returns an error if this function fails

Return

the value of the property, converted to a boolean, or false if an error occurs (including when the property value is not a bool).

bool **object_property_set_int**(*Object* *obj, const char *name, int64_t value, Error **errp)

Parameters

Object *obj
the object

const char *name
the name of the property

int64_t value
the value to be written to the property

Error **errp
returns an error if this function fails

Description

Writes an integer value to a property.

Return

`true` on success, `false` on failure.

int64_t **object_property_get_int**(*Object* *obj, const char *name, Error **errp)

Parameters

Object *obj
the object

const char *name
the name of the property

Error **errp
returns an error if this function fails

Return

the value of the property, converted to an integer, or -1 if an error occurs (including when the property value is not an integer).

bool **object_property_set_uint**(*Object* *obj, const char *name, uint64_t value, Error **errp)

Parameters

Object *obj
the object

const char *name
the name of the property

uint64_t value
the value to be written to the property

Error **errp
returns an error if this function fails

Description

Writes an unsigned integer value to a property.

Return

`true` on success, `false` on failure.

uint64_t **object_property_get_uint**(*Object* *obj, const char *name, Error **errp)

Parameters

Object *obj
the object

const char *name
the name of the property

Error **errp
returns an error if this function fails

Return

the value of the property, converted to an unsigned integer, or 0 an error occurs (including when the property value is not an integer).

int **object_property_get_enum**(*Object* *obj, const char *name, const char *typename, Error **errp)

Parameters

Object *obj
the object

const char *name
the name of the property

const char *typename
the name of the enum data type

Error **errp
returns an error if this function fails

Return

the value of the property, converted to an integer (which can't be negative), or -1 on error (including when the property value is not an enum).

bool **object_property_set**(*Object* *obj, const char *name, Visitor *v, Error **errp)

Parameters

Object *obj
the object

const char *name
the name of the property

Visitor *v
the visitor that will be used to write the property value. This should be an Input visitor and the data will be first read with **name** as the name and then written as the property value.

Error **errp
returns an error if this function fails

Description

Writes a property to a object.

Return

`true` on success, `false` on failure.

bool **object_property_parse**(*Object* *obj, const char *name, const char *string, Error **errp)

Parameters

Object *obj
: the object

const char *name
: the name of the property

const char *string
: the string that will be used to parse the property value.

Error **errp
: returns an error if this function fails

Description

Parses a string and writes the result into a property of an object.

Return

`true` on success, `false` on failure.

char ***object_property_print**(*Object* *obj, const char *name, bool human, Error **errp)

Parameters

Object *obj
: the object

const char *name
: the name of the property

bool human
: if true, print for human consumption

Error **errp
: returns an error if this function fails

Description

Returns a string representation of the value of the property. The caller shall free the string.

const char ***object_property_get_type**(*Object* *obj, const char *name, Error **errp)

Parameters

Object *obj
: the object

const char *name
: the name of the property

Error **errp
: returns an error if this function fails

Return

The type name of the property.

Object ***object_get_root**(void)

Parameters

void
no arguments

Return

the root object of the composition tree

Object ***object_get_objects_root**(void)

Parameters

void
no arguments

Description

Get the container object that holds user created object instances. This is the object at path "/objects"

Return

the user object container

Object ***object_get_internal_root**(void)

Parameters

void
no arguments

Description

Get the container object that holds internally used object instances. Any object which is put into this container must not be user visible, and it will not be exposed in the QOM tree.

Return

the internal object container

const char ***object_get_canonical_path_component**(const *Object* *obj)

Parameters

const Object *obj
the object

Return

The final component in the object's canonical path. The canonical path is the path within the composition tree starting from the root. `NULL` if the object doesn't have a parent (and thus a canonical path).

char ***object_get_canonical_path**(const *Object* *obj)

Parameters

const Object *obj
the object

Return

The canonical path for a object, newly allocated. This is the path within the composition tree starting from the root. Use g_free() to free it.

Object ***object_resolve_path**(const char *path, bool *ambiguous)

Parameters

const char *path
: the path to resolve

bool *ambiguous
: returns true if the path resolution failed because of an ambiguous match

Description

There are two types of supported paths–absolute paths and partial paths.

Absolute paths are derived from the root object and can follow child<> or link<> properties. Since they can follow link<> properties, they can be arbitrarily long. Absolute paths look like absolute filenames and are prefixed with a leading slash.

Partial paths look like relative filenames. They do not begin with a prefix. The matching rules for partial paths are subtle but designed to make specifying objects easy. At each level of the composition tree, the partial path is matched as an absolute path. The first match is not returned. At least two matches are searched for. A successful result is only returned if only one match is found. If more than one match is found, a flag is returned to indicate that the match was ambiguous.

Return

The matched object or NULL on path lookup failure.

Object ***object_resolve_path_type**(const char *path, const char *typename, bool *ambiguous)

Parameters

const char *path
: the path to resolve

const char *typename
: the type to look for.

bool *ambiguous
: returns true if the path resolution failed because of an ambiguous match

Description

This is similar to object_resolve_path. However, when looking for a partial path only matches that implement the given type are considered. This restricts the search and avoids spuriously flagging matches as ambiguous.

For both partial and absolute paths, the return value goes through a dynamic cast to **typename**. This is important if either the link, or the typename itself are of interface types.

Return

The matched object or NULL on path lookup failure.

Object ***object_resolve_type_unambiguous**(const char *typename, Error **errp)

Parameters

const char *typename
: the type to look for

Error **errp
pointer to error object

Description

Return the only object in the QOM tree of type **typename**. If no match or more than one match is found, an error is returned.

Return

The matched object or NULL on path lookup failure.

Object ***object_resolve_path_at**(*Object* *parent, const char *path)

Parameters

Object *parent
the object in which to resolve the path

const char *path
the path to resolve

Description

This is like object_resolve_path(), except paths not starting with a slash are relative to **parent**.

Return

The resolved object or NULL on path lookup failure.

Object ***object_resolve_path_component**(*Object* *parent, const char *part)

Parameters

Object *parent
the object in which to resolve the path

const char *part
the component to resolve.

Description

This is similar to object_resolve_path with an absolute path, but it only resolves one element (**part**) and takes the others from **parent**.

Return

The resolved object or NULL on path lookup failure.

ObjectProperty ***object_property_try_add_child**(*Object* *obj, const char *name, *Object* *child, Error **errp)

Parameters

Object *obj
the object to add a property to

const char *name
the name of the property

Object *child
the child object

Error **errp
pointer to error object

Description

Child properties form the composition tree. All objects need to be a child of another object. Objects can only be a child of one object.

There is no way for a child to determine what its parent is. It is not a bidirectional relationship. This is by design.

The value of a child property as a C string will be the child object's canonical path. It can be retrieved using object_property_get_str(). The child object itself can be retrieved using object_property_get_link().

Return

The newly added property on success, or `NULL` on failure.

ObjectProperty ***object_property_add_child**(*Object* *obj, const char *name, *Object* *child)

Parameters

Object *obj
: the object to add a property to

const char *name
: the name of the property

Object *child
: the child object

Description

Same as object_property_try_add_child() with **errp** hardcoded to &error_abort

void **object_property_allow_set_link**(const *Object* *obj, const char *name, *Object* *child, Error **errp)

Parameters

const Object *obj
: the object to add a property to

const char *name
: the name of the property

Object *child
: the child object

Error **errp
: pointer to error object

Description

The default implementation of the object_property_add_link() check() callback function. It allows the link property to be set and never returns an error.

ObjectProperty ***object_property_add_link**(*Object* *obj, const char *name, const char *type, *Object* **targetp, void (*check)(const *Object* *obj, const char *name, *Object* *val, Error **errp), ObjectPropertyLinkFlags flags)

Parameters

Object *obj
: the object to add a property to

const char *name
: the name of the property

const char *type
: the qobj type of the link

`Object **targetp`
: a pointer to where the link object reference is stored

`void (*check)(const Object *obj, const char *name, Object *val, Error **errp)`
: callback to veto setting or NULL if the property is read-only

`ObjectPropertyLinkFlags flags`
: additional options for the link

Description

Links establish relationships between objects. Links are unidirectional although two links can be combined to form a bidirectional relationship between objects.

Links form the graph in the object model.

The **check()** callback is invoked when object_property_set_link() is called and can raise an error to prevent the link being set. If **check** is NULL, the property is read-only and cannot be set.

Ownership of the pointer that **child** points to is transferred to the link property. The reference count for ***child** is managed by the property from after the function returns till the property is deleted with object_property_del(). If the **flags** `OBJ_PROP_LINK_STRONG` bit is set, the reference count is decremented when the property is deleted or modified.

Return

The newly added property on success, or `NULL` on failure.

ObjectProperty ***object_property_add_str**(*Object* *obj, const char *name, char *(*get)(*Object**, Error**), void (*set)(*Object**, const char*, Error**))

Parameters

`Object *obj`
: the object to add a property to

`const char *name`
: the name of the property

`char *(*get)(Object *, Error **)`
: the getter or NULL if the property is write-only. This function must return a string to be freed by g_free().

`void (*set)(Object *, const char *, Error **)`
: the setter or NULL if the property is read-only

Description

Add a string property using getters/setters. This function will add a property of type 'string'.

Return

The newly added property on success, or `NULL` on failure.

ObjectProperty ***object_property_add_bool**(*Object* *obj, const char *name, bool (*get)(*Object**, Error**), void (*set)(*Object**, bool, Error**))

Parameters

`Object *obj`
: the object to add a property to

`const char *name`
: the name of the property

`bool (*get)(Object *, Error **)`
: the getter or NULL if the property is write-only.

void (*set)(Object *, bool, Error **)
the setter or NULL if the property is read-only

Description

Add a bool property using getters/setters. This function will add a property of type 'bool'.

Return

The newly added property on success, or NULL on failure.

ObjectProperty ***object_property_add_enum**(*Object* *obj, const char *name, const char *typename, const QEnumLookup *lookup, int (*get)(*Object*, Error**), void (*set)(*Object*, int, Error**))

Parameters

Object *obj
the object to add a property to

const char *name
the name of the property

const char *typename
the name of the enum data type

const QEnumLookup *lookup
enum value namelookup table

int (*get)(Object *, Error **)
the getter or NULL if the property is write-only.

void (*set)(Object *, int, Error **)
the setter or NULL if the property is read-only

Description

Add an enum property using getters/setters. This function will add a property of type '**typename**'.

Return

The newly added property on success, or NULL on failure.

ObjectProperty ***object_property_add_tm**(*Object* *obj, const char *name, void (*get)(*Object*, struct tm*, Error**))

Parameters

Object *obj
the object to add a property to

const char *name
the name of the property

void (*get)(Object *, struct tm *, Error **)
the getter or NULL if the property is write-only.

Description

Add a read-only struct tm valued property using a getter function. This function will add a property of type 'struct tm'.

Return

The newly added property on success, or NULL on failure.

ObjectProperty *object_property_add_uint8_ptr(*Object* *obj, const char *name, const uint8_t *v, ObjectPropertyFlags flags)

Parameters

Object *obj
the object to add a property to

const char *name
the name of the property

const uint8_t *v
pointer to value

ObjectPropertyFlags flags
bitwise-or'd ObjectPropertyFlags

Description

Add an integer property in memory. This function will add a property of type 'uint8'.

Return

The newly added property on success, or NULL on failure.

ObjectProperty *object_property_add_uint16_ptr(*Object* *obj, const char *name, const uint16_t *v, ObjectPropertyFlags flags)

Parameters

Object *obj
the object to add a property to

const char *name
the name of the property

const uint16_t *v
pointer to value

ObjectPropertyFlags flags
bitwise-or'd ObjectPropertyFlags

Description

Add an integer property in memory. This function will add a property of type 'uint16'.

Return

The newly added property on success, or NULL on failure.

ObjectProperty *object_property_add_uint32_ptr(*Object* *obj, const char *name, const uint32_t *v, ObjectPropertyFlags flags)

Parameters

Object *obj
the object to add a property to

const char *name
the name of the property

const uint32_t *v
pointer to value

ObjectPropertyFlags flags
bitwise-or'd ObjectPropertyFlags

Description

Add an integer property in memory. This function will add a property of type 'uint32'.

Return

The newly added property on success, or NULL on failure.

ObjectProperty *object_property_add_uint64_ptr(*Object* *obj, const char *name, const uint64_t *v, ObjectPropertyFlags flags)

Parameters

Object *obj
the object to add a property to

const char *name
the name of the property

const uint64_t *v
pointer to value

ObjectPropertyFlags flags
bitwise-or'd ObjectPropertyFlags

Description

Add an integer property in memory. This function will add a property of type 'uint64'.

Return

The newly added property on success, or NULL on failure.

ObjectProperty *object_property_add_alias(*Object* *obj, const char *name, *Object* *target_obj, const char *target_name)

Parameters

Object *obj
the object to add a property to

const char *name
the name of the property

Object *target_obj
the object to forward property access to

const char *target_name
the name of the property on the forwarded object

Description

Add an alias for a property on an object. This function will add a property of the same type as the forwarded property.

The caller must ensure that **target_obj** stays alive as long as this property exists. In the case of a child object or an alias on the same object this will be the case. For aliases to other objects the caller is responsible for taking a reference.

Return

The newly added property on success, or NULL on failure.

ObjectProperty *object_property_add_const_link(*Object* *obj, const char *name, *Object* *target)

Parameters

Object *obj
the object to add a property to

const char *name
: the name of the property

Object *target
: the object to be referred by the link

Description

Add an unmodifiable link for a property on an object. This function will add a property of type link<TYPE> where TYPE is the type of **target**.

The caller must ensure that **target** stays alive as long as this property exists. In the case **target** is a child of **obj**, this will be the case. Otherwise, the caller is responsible for taking a reference.

Return

The newly added property on success, or NULL on failure.

void **object_property_set_description**(*Object* *obj, const char *name, const char *description)

Parameters

Object *obj
: the object owning the property

const char *name
: the name of the property

const char *description
: the description of the property on the object

Description

Set an object property's description.

Return

true on success, false on failure.

int **object_child_foreach**(*Object* *obj, int (*fn)(*Object* *child, void *opaque), void *opaque)

Parameters

Object *obj
: the object whose children will be navigated

int (*fn)(Object *child, void *opaque)
: the iterator function to be called

void *opaque
: an opaque value that will be passed to the iterator

Description

Call **fn** passing each child of **obj** and **opaque** to it, until **fn** returns non-zero.

It is forbidden to add or remove children from **obj** from the **fn** callback.

Return

The last value returned by **fn**, or 0 if there is no child.

int **object_child_foreach_recursive**(*Object* *obj, int (*fn)(*Object* *child, void *opaque), void *opaque)

Parameters

Object *obj
: the object whose children will be navigated

int (*fn)(Object *child, void *opaque)
: the iterator function to be called

void *opaque
: an opaque value that will be passed to the iterator

Description

Call **fn** passing each child of **obj** and **opaque** to it, until **fn** returns non-zero. Calls recursively, all child nodes of **obj** will also be passed all the way down to the leaf nodes of the tree. Depth first ordering.

It is forbidden to add or remove children from **obj** (or its child nodes) from the **fn** callback.

Return

The last value returned by **fn**, or 0 if there is no child.

Object ***container_get**(*Object* *root, const char *path)

Parameters

Object *root
: root of the #path, e.g., object_get_root()

const char *path
: path to the container

Description

Return a container object whose path is **path**. Create more containers along the path if necessary.

Return

the container object.

size_t **object_type_get_instance_size**(const char *typename)

Parameters

const char *typename
: Name of the Type whose instance_size is required

Description

Returns the instance_size of the given **typename**.

char ***object_property_help**(const char *name, const char *type, QObject *defval, const char *description)

Parameters

const char *name
: the name of the property

const char *type
: the type of the property

QObject *defval
: the default value

const char *description
: description of the property

Return

a user-friendly formatted string describing the property for help purposes.

7.3.7 QEMU Device (qdev) API Reference

The QEMU Device API

All modern devices should represented as a derived QOM class of TYPE_DEVICE. The device API introduces the additional methods of **realize** and **unrealize** to represent additional stages in a device objects life cycle.

Realization

Devices are constructed in two stages:

1) object instantiation via object_initialize() and

2) device realization via the *DeviceState.realized* property

The former may not fail (and must not abort or exit, since it is called during device introspection already), and the latter may return error information to the caller and must be re-entrant. Trivial field initializations should go into *TypeInfo.instance_init*. Operations depending on **props** static properties should go into **realize**. After successful realization, setting static properties will fail.

As an interim step, the *DeviceState.realized* property can also be set with qdev_realize(). In the future, devices will propagate this state change to their children and along busses they expose. The point in time will be deferred to machine creation, so that values set in **realize** will not be introspectable beforehand. Therefore devices must not create children during **realize**; they should initialize them via object_initialize() in their own *TypeInfo.instance_init* and forward the realization events appropriately.

Any type may override the **realize** and/or **unrealize** callbacks but needs to call the parent type's implementation if keeping their functionality is desired. Refer to QOM documentation for further discussion and examples.

Note: Since TYPE_DEVICE doesn't implement **realize** and **unrealize**, types derived directly from it need not call their parent's **realize** and **unrealize**. For other types consult the documentation and implementation of the respective parent types.

Hiding a device

To hide a device, a DeviceListener function hide_device() needs to be registered. It can be used to defer adding a device and therefore hide it from the guest. The handler registering to this DeviceListener can save the QOpts passed to it for re-using it later. It must return if it wants the device to be hidden or visible. When the handler function decides the device shall be visible it will be added with qdev_device_add() and realized as any other device. Otherwise qdev_device_add() will return early without adding the device. The guest will not see a "hidden" device until it was marked visible and qdev_device_add called again.

struct **DeviceClass**

The base class for all devices.

Definition

```
struct DeviceClass {
  unsigned long categories[BITS_TO_LONGS(DEVICE_CATEGORY_MAX)];
  const char *fw_name;
```

(continues on next page)

(continued from previous page)

```
  const char *desc;
  Property *props_;
  bool user_creatable;
  bool hotpluggable;
  DeviceReset reset;
  DeviceRealize realize;
  DeviceUnrealize unrealize;
  const VMStateDescription *vmsd;
  const char *bus_type;
};
```

Members

categories
device categories device belongs to

fw_name
name used to identify device to firmware interfaces

desc
human readable description of device

props_
properties associated with device, should only be assigned by using device_class_set_props(). The underscore ensures a compile-time error if someone attempts to assign dc->props directly.

user_creatable
Can user instantiate with -device / device_add?

All devices should support instantiation with device_add, and this flag should not exist. But we're not there, yet. Some devices fail to instantiate with cryptic error messages. Others instantiate, but don't work. Exposing users to such behavior would be cruel; clearing this flag will protect them. It should never be cleared without a comment explaining why it is cleared.

TODO remove once we're there

hotpluggable
indicates if *DeviceClass* is hotpluggable, available as readonly "hotpluggable" property of *DeviceState* instance

reset
deprecated device reset method pointer

Modern code should use the ResettableClass interface to implement a multi-phase reset.

TODO: remove once every reset callback is unused

realize
Callback function invoked when the *DeviceState*:realized property is changed to `true`.

unrealize
Callback function invoked when the *DeviceState*:realized property is changed to `false`.

vmsd
device state serialisation description for migration/save/restore

bus_type
bus type private: to qdev / bus.

struct **DeviceState**

common device state, accessed with qdev helpers

Definition

```
struct DeviceState {
  char *id;
  char *canonical_path;
  bool realized;
  bool pending_deleted_event;
  int64_t pending_deleted_expires_ms;
  QDict *opts;
  int hotplugged;
  bool allow_unplug_during_migration;
  BusState *parent_bus;
  NamedGPIOListHead gpios;
  NamedClockListHead clocks;
  BusStateHead child_bus;
  int num_child_bus;
  int instance_id_alias;
  int alias_required_for_version;
  ResettableState reset;
  GSList *unplug_blockers;
  MemReentrancyGuard mem_reentrancy_guard;
};
```

Members

id
global device id

canonical_path
canonical path of realized device in the QOM tree

realized
has device been realized?

pending_deleted_event
track pending deletion events during unplug

pending_deleted_expires_ms
optional timeout for deletion events

opts
QDict of options for the device

hotplugged
was device added after PHASE_MACHINE_READY?

allow_unplug_during_migration
can device be unplugged during migration

parent_bus
bus this device belongs to

gpios
QLIST of named GPIOs the device provides.

clocks
QLIST of named clocks the device provides.

child_bus
QLIST of child buses

num_child_bus
number of **child_bus** entries

instance_id_alias
device alias for handling legacy migration setups

alias_required_for_version
indicates **instance_id_alias** is needed for migration

reset
ResettableState for the device; handled by Resettable interface.

unplug_blockers
list of reasons to block unplugging of device

mem_reentrancy_guard
Is the device currently in mmio/pio/dma?

Used to prevent re-entrancy confusing things.

Description

This structure should not be accessed directly. We declare it here so that it can be embedded in individual device state structures.

struct **BusState**

Definition

```
struct BusState {
  DeviceState *parent;
  char *name;
  HotplugHandler *hotplug_handler;
  int max_index;
  bool realized;
  bool full;
  int num_children;
  BusChildHead children;
  BusStateEntry sibling;
  ResettableState reset;
};
```

Members

parent
parent Device

name
name of bus

hotplug_handler
link to a hotplug handler associated with bus.

max_index
max number of child buses

realized
is the bus itself realized?

full
is the bus full?

num_children
current number of child buses

children
an RCU protected QTAILQ, thus readers must use RCU to access it, and writers must hold the big qemu lock

sibling
next bus

reset
ResettableState for the bus; handled by Resettable interface.

type **GlobalProperty**
a global property type

Description

An error is fatal for non-hotplugged devices, when the global is applied.

DeviceState ***qdev_new**(const char *name)*
Create a device on the heap

Parameters

const char *name
device type to create (we assert() that this type exists)

Description

This only allocates the memory and initializes the device state structure, ready for the caller to set properties if they wish. The device still needs to be realized.

Return

a derived DeviceState object with a reference count of 1.

DeviceState ***qdev_try_new**(const char *name)*
Try to create a device on the heap

Parameters

const char *name
device type to create

Description

This is like qdev_new(), except it returns `NULL` when type **name** does not exist, rather than asserting.

Return

a derived DeviceState object with a reference count of 1 or NULL if type **name** does not exist.

bool **qdev_is_realized**(*DeviceState* *dev)
check if device is realized

Parameters

DeviceState *dev
The device to check.

Context

May be called outside big qemu lock.

Return

true if the device has been fully constructed, false otherwise.

bool **qdev_realize**(*DeviceState* *dev, *BusState* *bus, Error **errp)

Realize **dev**.

Parameters

DeviceState *dev
device to realize

BusState *bus
bus to plug it into (may be NULL)

Error **errp
pointer to error object

Description

"Realize" the device, i.e. perform the second phase of device initialization. **dev** must not be plugged into a bus already. If **bus**, plug **dev** into **bus**. This takes a reference to **dev**. If **dev** has no QOM parent, make one up, taking another reference.

If you created **dev** using qdev_new(), you probably want to use qdev_realize_and_unref() instead.

Return

true on success, else false setting **errp** with error

bool **qdev_realize_and_unref**(*DeviceState* *dev, *BusState* *bus, Error **errp)

Realize **dev** and drop a reference

Parameters

DeviceState *dev
device to realize

BusState *bus
bus to plug it into (may be NULL)

Error **errp
pointer to error object

Description

Realize **dev** and drop a reference. This is like qdev_realize(), except the caller must hold a (private) reference, which is dropped on return regardless of success or failure. Intended use:

```
dev = qdev_new();
[...]
qdev_realize_and_unref(dev, bus, errp);
```

Now **dev** can go away without further ado.

If you are embedding the device into some other QOM device and initialized it via some variant on object_initialize_child() then do not use this function, because that family of functions arrange for the only reference to the child device to be held by the parent via the child<> property, and so the reference-count-drop done here would be incorrect. For that use case you want qdev_realize().

Return

true on success, else false setting **errp** with error

void **qdev_unrealize**(*DeviceState* *dev)

Unrealize a device

Parameters

DeviceState *dev
device to unrealize

Description

This function will "unrealize" a device, which is the first phase of correctly destroying a device that has been realized. It will:

- unrealize any child buses by calling qbus_unrealize() (this will recursively unrealize any devices on those buses)
- call the unrealize method of **dev**

The device can then be freed by causing its reference count to go to zero.

Warning: most devices in QEMU do not expect to be unrealized. Only devices which are hot-unpluggable should be unrealized (as part of the unplugging process); all other devices are expected to last for the life of the simulation and should not be unrealized and freed.

HotplugHandler *qdev_get_hotplug_handler(*DeviceState* *dev)

Get handler responsible for device wiring

Parameters

DeviceState *dev
the device we want the HOTPLUG_HANDLER for.

Note

in case **dev** has a parent bus, it will be returned as handler unless machine handler overrides it.

Return

pointer to object that implements TYPE_HOTPLUG_HANDLER interface or NULL if there aren't any.

void **qdev_add_unplug_blocker**(*DeviceState* *dev, Error *reason)

Add an unplug blocker to a device

Parameters

DeviceState *dev
Device to be blocked from unplug

Error *reason
Reason for blocking

void **qdev_del_unplug_blocker**(*DeviceState* *dev, Error *reason)

Remove an unplug blocker from a device

Parameters

DeviceState *dev
Device to be unblocked

Error *reason
Pointer to the Error used with qdev_add_unplug_blocker. Used as a handle to lookup the blocker for deletion.

bool **qdev_unplug_blocked**(*DeviceState* *dev, Error **errp)

Confirm if a device is blocked from unplug

Parameters

DeviceState *dev
Device to be tested

Error **errp
The reasons why the device is blocked, if any

Return

true (also setting **errp**) if device is blocked from unplug, false otherwise

type **GpioPolarity**

Polarity of a GPIO line

Description

GPIO lines use either positive (active-high) logic, or negative (active-low) logic.

In active-high logic (`GPIO_POLARITY_ACTIVE_HIGH`), a pin is active when the voltage on the pin is high (relative to ground); whereas in active-low logic (`GPIO_POLARITY_ACTIVE_LOW`), a pin is active when the voltage on the pin is low (or grounded).

qemu_irq **qdev_get_gpio_in**(*DeviceState* *dev, int n)

Get one of a device's anonymous input GPIO lines

Parameters

DeviceState *dev
Device whose GPIO we want

int n
Number of the anonymous GPIO line (which must be in range)

Description

Returns the qemu_irq corresponding to an anonymous input GPIO line (which the device has set up with qdev_init_gpio_in()). The index **n** of the GPIO line must be valid (i.e. be at least 0 and less than the total number of anonymous input GPIOs the device has); this function will assert() if passed an invalid index.

This function is intended to be used by board code or SoC "container" device models to wire up the GPIO lines; usually the return value will be passed to qdev_connect_gpio_out() or a similar function to connect another device's output GPIO line to this input.

For named input GPIO lines, use qdev_get_gpio_in_named().

Return

qemu_irq corresponding to anonymous input GPIO line

qemu_irq **qdev_get_gpio_in_named**(*DeviceState* *dev, const char *name, int n)

Get one of a device's named input GPIO lines

Parameters

DeviceState *dev
Device whose GPIO we want

const char *name
Name of the input GPIO array

int n
Number of the GPIO line in that array (which must be in range)

Description

Returns the qemu_irq corresponding to a named input GPIO line (which the device has set up with qdev_init_gpio_in_named()). The **name** string must correspond to an input GPIO array which exists on the device, and the index **n** of the GPIO line must be valid (i.e. be at least 0 and less than the total number of input GPIOs in that array); this function will assert() if passed an invalid name or index.

For anonymous input GPIO lines, use qdev_get_gpio_in().

Return

qemu_irq corresponding to named input GPIO line

void **qdev_connect_gpio_out**(*DeviceState* *dev, int n, qemu_irq pin)

Connect one of a device's anonymous output GPIO lines

Parameters

DeviceState *dev
Device whose GPIO to connect

int n
Number of the anonymous output GPIO line (which must be in range)

qemu_irq pin
qemu_irq to connect the output line to

Description

This function connects an anonymous output GPIO line on a device up to an arbitrary qemu_irq, so that when the device asserts that output GPIO line, the qemu_irq's callback is invoked. The index **n** of the GPIO line must be valid (i.e. be at least 0 and less than the total number of anonymous output GPIOs the device has created with qdev_init_gpio_out()); otherwise this function will assert().

Outbound GPIO lines can be connected to any qemu_irq, but the common case is connecting them to another device's inbound GPIO line, using the qemu_irq returned by qdev_get_gpio_in() or qdev_get_gpio_in_named().

It is not valid to try to connect one outbound GPIO to multiple qemu_irqs at once, or to connect multiple outbound GPIOs to the same qemu_irq. (Warning: there is no assertion or other guard to catch this error: the model will just not do the right thing.) Instead, for fan-out you can use the TYPE_SPLIT_IRQ device: connect a device's outbound GPIO to the splitter's input, and connect each of the splitter's outputs to a different device. For fan-in you can use the TYPE_OR_IRQ device, which is a model of a logical OR gate with multiple inputs and one output.

For named output GPIO lines, use qdev_connect_gpio_out_named().

void **qdev_connect_gpio_out_named**(*DeviceState* *dev, const char *name, int n, qemu_irq input_pin)

Connect one of a device's named output GPIO lines

Parameters

DeviceState *dev
Device whose GPIO to connect

const char *name
Name of the output GPIO array

int n
Number of the anonymous output GPIO line (which must be in range)

qemu_irq input_pin
qemu_irq to connect the output line to

Description

This function connects an anonymous output GPIO line on a device up to an arbitrary qemu_irq, so that when the device asserts that output GPIO line, the qemu_irq's callback is invoked. The **name** string must correspond to an output GPIO array which exists on the device, and the index **n** of the GPIO line must be valid (i.e. be at least 0 and less than the total number of input GPIOs in that array); this function will assert() if passed an invalid name or index.

Outbound GPIO lines can be connected to any qemu_irq, but the common case is connecting them to another device's inbound GPIO line, using the qemu_irq returned by qdev_get_gpio_in() or qdev_get_gpio_in_named().

It is not valid to try to connect one outbound GPIO to multiple qemu_irqs at once, or to connect multiple outbound GPIOs to the same qemu_irq; see qdev_connect_gpio_out() for details.

For anonymous output GPIO lines, use qdev_connect_gpio_out().

qemu_irq **qdev_get_gpio_out_connector**(*DeviceState* *dev, const char *name, int n)

Get the qemu_irq connected to an output GPIO

Parameters

`DeviceState *dev`
Device whose output GPIO we are interested in

`const char *name`
Name of the output GPIO array

`int n`
Number of the output GPIO line within that array

Description

Returns whatever qemu_irq is currently connected to the specified output GPIO line of **dev**. This will be NULL if the output GPIO line has never been wired up to the anything. Note that the qemu_irq returned does not belong to **dev** – it will be the input GPIO or IRQ of whichever device the board code has connected up to **dev**'s output GPIO.

You probably don't need to use this function – it is used only by the platform-bus subsystem.

Return

qemu_irq associated with GPIO or NULL if un-wired.

qemu_irq **qdev_intercept_gpio_out**(*DeviceState* *dev, qemu_irq icpt, const char *name, int n)

Intercept an existing GPIO connection

Parameters

`DeviceState *dev`
Device to intercept the outbound GPIO line from

`qemu_irq icpt`
New qemu_irq to connect instead

`const char *name`
Name of the output GPIO array

`int n`
Number of the GPIO line in the array

Description

Note: This function is provided only for use by the qtest testing framework and is not suitable for use in non-testing parts of QEMU.

This function breaks an existing connection of an outbound GPIO line from **dev**, and replaces it with the new qemu_irq **icpt**, as if `qdev_connect_gpio_out_named(dev, icpt, name, n)` had been called. The previously connected qemu_irq is returned, so it can be restored by a second call to qdev_intercept_gpio_out() if desired.

Return

old disconnected qemu_irq if one existed

void **qdev_init_gpio_in**(*DeviceState* *dev, qemu_irq_handler handler, int n)

create an array of anonymous input GPIO lines

Parameters

`DeviceState *dev`
Device to create input GPIOs for

`qemu_irq_handler handler`
Function to call when GPIO line value is set

`int n`
Number of GPIO lines to create

Description

Devices should use functions in the qdev_init_gpio_in* family in their instance_init or realize methods to create any input GPIO lines they need. There is no functional difference between anonymous and named GPIO lines. Stylistically, named GPIOs are preferable (easier to understand at callsites) unless a device has exactly one uniform kind of GPIO input whose purpose is obvious. Note that input GPIO lines can serve as 'sinks' for IRQ lines.

See qdev_get_gpio_in() for how code that uses such a device can get hold of an input GPIO line to manipulate it.

void **qdev_init_gpio_out**(*DeviceState* *dev, qemu_irq *pins, int n)

create an array of anonymous output GPIO lines

Parameters

`DeviceState *dev`
Device to create output GPIOs for

`qemu_irq *pins`
Pointer to qemu_irq or qemu_irq array for the GPIO lines

`int n`
Number of GPIO lines to create

Description

Devices should use functions in the qdev_init_gpio_out* family in their instance_init or realize methods to create any output GPIO lines they need. There is no functional difference between anonymous and named GPIO lines. Stylistically, named GPIOs are preferable (easier to understand at callsites) unless a device has exactly one uniform kind of GPIO output whose purpose is obvious.

The **pins** argument should be a pointer to either a "qemu_irq" (if **n** == 1) or a "qemu_irq []" array (if **n** > 1) in the device's state structure. The device implementation can then raise and lower the GPIO line by calling qemu_set_irq(). (If anything is connected to the other end of the GPIO this will cause the handler function for that input GPIO to be called.)

See qdev_connect_gpio_out() for how code that uses such a device can connect to one of its output GPIO lines.

There is no need to release the **pins** allocated array because it will be automatically released when **dev** calls its instance_finalize() handler.

void **qdev_init_gpio_out_named**(*DeviceState* *dev, qemu_irq *pins, const char *name, int n)

create an array of named output GPIO lines

Parameters

DeviceState *dev
Device to create output GPIOs for

qemu_irq *pins
Pointer to qemu_irq or qemu_irq array for the GPIO lines

const char *name
Name to give this array of GPIO lines

int n
Number of GPIO lines to create

Description

Like qdev_init_gpio_out(), but creates an array of GPIO output lines with a name. Code using the device can then connect these GPIO lines using qdev_connect_gpio_out_named().

void **qdev_init_gpio_in_named_with_opaque**(*DeviceState* *dev, qemu_irq_handler handler, void *opaque, const char *name, int n)

create an array of input GPIO lines

Parameters

DeviceState *dev
Device to create input GPIOs for

qemu_irq_handler handler
Function to call when GPIO line value is set

void *opaque
Opaque data pointer to pass to **handler**

const char *name
Name of the GPIO input (must be unique for this device)

int n
Number of GPIO lines in this input set

void **qdev_init_gpio_in_named**(*DeviceState* *dev, qemu_irq_handler handler, const char *name, int n)

create an array of input GPIO lines

Parameters

DeviceState *dev
device to add array to

qemu_irq_handler handler
a &typedef qemu_irq_handler function to call when GPIO is set

const char *name
Name of the GPIO input (must be unique for this device)

int n
Number of GPIO lines in this input set

Description

Like qdev_init_gpio_in_named_with_opaque(), but the opaque pointer passed to the handler is **dev** (which is the most commonly desired behaviour).

void **qdev_pass_gpios**(*DeviceState* *dev, *DeviceState* *container, const char *name)

create GPIO lines on container which pass through to device

Parameters

DeviceState *dev
Device which has GPIO lines

DeviceState *container
Container device which needs to expose them

const char *name
Name of GPIO array to pass through (NULL for the anonymous GPIO array)

Description

In QEMU, complicated devices like SoCs are often modelled with a "container" QOM device which itself contains other QOM devices and which wires them up appropriately. This function allows the container to create GPIO arrays on itself which simply pass through to a GPIO array of one of its internal devices.

If **dev** has both input and output GPIOs named **name** then both will be passed through. It is not possible to pass a subset of the array with this function.

To users of the container device, the GPIO array created on **container** behaves exactly like any other.

void **device_cold_reset**(*DeviceState* *dev)

perform a recursive cold reset on a device

Parameters

DeviceState *dev
device to reset.

Description

Reset device **dev** and perform a recursive processing using the resettable interface. It triggers a RESET_TYPE_COLD.

void **bus_cold_reset**(*BusState* *bus)

perform a recursive cold reset on a bus

Parameters

BusState *bus
bus to reset

Description

Reset bus **bus** and perform a recursive processing using the resettable interface. It triggers a RESET_TYPE_COLD.

bool **device_is_in_reset**(*DeviceState* *dev)

check device reset state

Parameters

DeviceState *dev
device to check

Return

true if the device **dev** is currently being reset.

bool **bus_is_in_reset**(*BusState* *bus)

check bus reset state

Parameters

BusState *bus
bus to check

Return

true if the bus **bus** is currently being reset.

void **device_class_set_props**(*DeviceClass* *dc, Property *props)

add a set of properties to an device

Parameters

DeviceClass *dc
the parent DeviceClass all devices inherit

Property *props
an array of properties, terminate by DEFINE_PROP_END_OF_LIST()

Description

This will add a set of properties to the object. It will fault if you attempt to add an existing property defined by a parent class. To modify an inherited property you need to use????

void **device_class_set_parent_reset**(*DeviceClass* *dc, DeviceReset dev_reset, DeviceReset *parent_reset)

legacy set device reset handlers

Parameters

DeviceClass *dc
device class

DeviceReset dev_reset
function pointer to reset handler

DeviceReset *parent_reset
function pointer to parents reset handler

Description

Modern code should use the ResettableClass interface to implement a multi-phase reset instead.

TODO: remove the function when DeviceClass's reset method is not used anymore.

void **device_class_set_parent_realize**(*DeviceClass* *dc, DeviceRealize dev_realize, DeviceRealize *parent_realize)

set up for chaining realize fns

Parameters

DeviceClass *dc
The device class

DeviceRealize dev_realize
the device realize function

DeviceRealize *parent_realize
somewhere to save the parents realize function

Description

This is intended to be used when the new realize function will eventually call its parent realization function during creation. This requires storing the function call somewhere (usually in the instance structure) so you can eventually call dc->parent_realize(dev, errp)

void **device_class_set_parent_unrealize**(*DeviceClass* *dc, DeviceUnrealize dev_unrealize, DeviceUnrealize *parent_unrealize)

set up for chaining unrealize fns

Parameters

DeviceClass *dc
The device class

DeviceUnrealize dev_unrealize
the device realize function

DeviceUnrealize *parent_unrealize
somewhere to save the parents unrealize function

Description

This is intended to be used when the new unrealize function will eventually call its parent unrealization function during the unrealize phase. This requires storing the function call somewhere (usually in the instance structure) so you can eventually call dc->parent_unrealize(dev);

char ***qdev_get_human_name**(*DeviceState* *dev)

Return a human-readable name for a device

Parameters

DeviceState *dev
The device. Must be a valid and non-NULL pointer.

Description

Note: This function is intended for user friendly error messages.

Use g_free() to free it.

Return

A newly allocated string containing the device id if not null, else the object canonical path.

void **qbus_mark_full**(*BusState* *bus)

Mark this bus as full, so no more devices can be attached

Parameters

BusState *bus
Bus to mark as full

Description

By default, QEMU will allow devices to be plugged into a bus up to the bus class's device count limit. Calling this function marks a particular bus as full, so that no more devices can be plugged into it. In particular this means that the bus will not be considered as a candidate for plugging in devices created by the user on the commandline or via the monitor. If a machine has multiple buses of a given type, such as I2C, where some of those buses in the real hardware are used only for internal devices and some are exposed via expansion ports, you can use this function to mark the internal-only buses as full after you have created all their internal devices. Then user created devices will appear on the expansion-port bus where guest software expects them.

bool **qdev_should_hide_device**(const QDict *opts, bool from_json, Error **errp)

check if device should be hidden

Parameters

const QDict *opts
options QDict

bool from_json
true if **opts** entries are typed, false for all strings

Error **errp
pointer to error object

Description

When a device is added via qdev_device_add() this will be called.

Return

if the device should be added now or not.

7.3.8 QEMU UI subsystem

QEMU Clipboard

Introduction

The header `ui/clipboard.h` declares the qemu clipboard interface.

All qemu elements which want use the clipboard can register as clipboard peer. Subsequently they can set the clipboard content and get notifications for clipboard updates.

Typical users are user interfaces (gtk), remote access protocols (vnc) and devices talking to the guest (vdagent).

Even though the design allows different data types only plain text is supported for now.

enum **QemuClipboardType**

Constants

QEMU_CLIPBOARD_TYPE_TEXT
text/plain; charset=utf-8

QEMU_CLIPBOARD_TYPE__COUNT
type count.

enum **QemuClipboardSelection**

Constants

QEMU_CLIPBOARD_SELECTION_CLIPBOARD
clipboard (explitcit cut+paste).

QEMU_CLIPBOARD_SELECTION_PRIMARY
primary selection (select + middle mouse button).

QEMU_CLIPBOARD_SELECTION_SECONDARY
secondary selection (dunno).

QEMU_CLIPBOARD_SELECTION__COUNT
selection count.

struct **QemuClipboardPeer**

Definition

```
struct QemuClipboardPeer {
  const char *name;
  Notifier notifier;
  void (*request)(QemuClipboardInfo *info, QemuClipboardType type);
};
```

Members

name
peer name.

notifier
notifier for clipboard updates.

request
callback for clipboard data requests.

Description

Clipboard peer description.

enum **QemuClipboardNotifyType**

Constants

QEMU_CLIPBOARD_UPDATE_INFO
clipboard info update

QEMU_CLIPBOARD_RESET_SERIAL
reset clipboard serial

Description

Clipboard notify type.

struct **QemuClipboardNotify**

Definition

```
struct QemuClipboardNotify {
  QemuClipboardNotifyType type;
  union {
    QemuClipboardInfo *info;
  };
};
```

Members

type
the type of event.

{unnamed_union}
anonymous

info
a QemuClipboardInfo event.

Description

Clipboard notify data.

struct **QemuClipboardInfo**

Definition

```
struct QemuClipboardInfo {
  uint32_t refcount;
  QemuClipboardPeer *owner;
  QemuClipboardSelection selection;
  bool has_serial;
  uint32_t serial;
  struct {
    bool available;
    bool requested;
    size_t size;
    void *data;
  } types[QEMU_CLIPBOARD_TYPE__COUNT];
};
```

Members

refcount
reference counter.

owner
clipboard owner.

selection
clipboard selection.

has_serial
whether **serial** is available.

serial
the grab serial counter.

types
clipboard data array (one entry per type).

Description

Clipboard content data and metadata.

void **qemu_clipboard_peer_register**(*QemuClipboardPeer* *peer)

Parameters

QemuClipboardPeer *peer
peer information.

Description

Register clipboard peer. Registering is needed for both active (set+grab clipboard) and passive (watch clipboard for updates) interaction with the qemu clipboard.

void **qemu_clipboard_peer_unregister**(*QemuClipboardPeer* *peer)

Parameters

QemuClipboardPeer *peer
peer information.

Description

Unregister clipboard peer.

bool **qemu_clipboard_peer_owns**(*QemuClipboardPeer* *peer, *QemuClipboardSelection* selection)

Parameters

QemuClipboardPeer *peer
peer information.

QemuClipboardSelection selection
clipboard selection.

Description

Return TRUE if the peer owns the clipboard.

void **qemu_clipboard_peer_release**(*QemuClipboardPeer* *peer, *QemuClipboardSelection* selection)

Parameters

QemuClipboardPeer *peer
peer information.

QemuClipboardSelection selection
clipboard selection.

Description

If the peer owns the clipboard, release it.

QemuClipboardInfo ***qemu_clipboard_info**(*QemuClipboardSelection* selection)

Parameters

QemuClipboardSelection selection
clipboard selection.

Description

Return the current clipboard data & owner information.

bool **qemu_clipboard_check_serial**(*QemuClipboardInfo* *info, bool client)

Parameters

QemuClipboardInfo *info
clipboard info.

bool client
whether to check from the client context and priority.

Description

Return TRUE if the **info** has a higher serial than the current clipboard.

QemuClipboardInfo ***qemu_clipboard_info_new**(*QemuClipboardPeer* *owner, *QemuClipboardSelection* selection)

Parameters

QemuClipboardPeer *owner
clipboard owner.

QemuClipboardSelection selection
clipboard selection.

Description

Allocate a new QemuClipboardInfo and initialize it with the given **owner** and **selection**.

QemuClipboardInfo is a reference-counted struct. The new struct is returned with a reference already taken (i.e. reference count is one).

QemuClipboardInfo ***qemu_clipboard_info_ref**(*QemuClipboardInfo* *info)

Parameters

QemuClipboardInfo *info
clipboard info.

Description

Increase **info** reference count.

void **qemu_clipboard_info_unref**(*QemuClipboardInfo* *info)

Parameters

QemuClipboardInfo *info
clipboard info.

Description

Decrease **info** reference count. When the count goes down to zero free the **info** struct itself and all clipboard data.

void **qemu_clipboard_update**(*QemuClipboardInfo* *info)

Parameters

QemuClipboardInfo *info
clipboard info.

Description

Update the qemu clipboard. Notify all registered peers (including the clipboard owner) that the qemu clipboard has been updated.

This is used for both new completely clipboard content and for clipboard data updates in response to qemu_clipboard_request() calls.

void **qemu_clipboard_reset_serial**(void)

Parameters

void
no arguments

Description

Reset the clipboard serial.

void **qemu_clipboard_request**(*QemuClipboardInfo* *info, *QemuClipboardType* type)

Parameters

QemuClipboardInfo *info
clipboard info.

QemuClipboardType type
clipboard data type.

Description

Request clipboard content. Typically the clipboard owner only advertises the available data types and provides the actual data only on request.

void **qemu_clipboard_set_data**(*QemuClipboardPeer* *peer, *QemuClipboardInfo* *info, *QemuClipboardType* type, uint32_t size, const void *data, bool update)

Parameters

QemuClipboardPeer *peer
clipboard peer.

QemuClipboardInfo *info
clipboard info.

QemuClipboardType type
clipboard data type.

uint32_t size
data size.

const void *data
data blob.

bool update
notify peers about the update.

Description

Set clipboard content for the given **type**. This function will make a copy of the content data and store that.

7.3.9 zoned-storage

Zoned Block Devices (ZBDs) divide the LBA space into block regions called zones that are larger than the LBA size. They can only allow sequential writes, which can reduce write amplification in SSDs, and potentially lead to higher throughput and increased capacity. More details about ZBDs can be found at:

https://zonedstorage.io/docs/introduction/zoned-storage

1. Block layer APIs for zoned storage

QEMU block layer supports three zoned storage models: - BLK_Z_HM: The host-managed zoned model only allows sequential writes access to zones. It supports ZBD-specific I/O commands that can be used by a host to manage the zones of a device. - BLK_Z_HA: The host-aware zoned model allows random write operations in zones, making it backward compatible with regular block devices. - BLK_Z_NONE: The non-zoned model has no zones support. It includes both regular and drive-managed ZBD devices. ZBD-specific I/O commands are not supported.

The block device information resides inside BlockDriverState. QEMU uses BlockLimits struct(BlockDriverState::bl) that is continuously accessed by the block layer while processing I/O requests. A BlockBackend has a root pointer to a BlockDriverState graph(for example, raw format on top of file-posix). The zoned storage information can be propagated from the leaf BlockDriverState all the way up to the BlockBackend. If the zoned storage model in file-posix is set to BLK_Z_HM, then block drivers will declare support for zoned host device.

The block layer APIs support commands needed for zoned storage devices, including report zones, four zone operations, and zone append.

2. Emulating zoned storage controllers

When the BlockBackend's BlockLimits model reports a zoned storage device, users like the virtio-blk emulation or the qemu-io-cmds.c utility can use block layer APIs for zoned storage emulation or testing.

For example, to test zone_report on a null_blk device using qemu-io is:

```
$ path/to/qemu-io --image-opts -n driver=host_device,filename=/dev/nullb0 -c "zrp offset␣
↪nr_zones"
```

To expose the host's zoned block device through virtio-blk, the command line can be (includes the -device parameter):

```
-blockdev node-name=drive0,driver=host_device,filename=/dev/nullb0,cache.direct=on \
-device virtio-blk-pci,drive=drive0
```

Or only use the -drive parameter:

```
-driver driver=host_device,file=/dev/nullb0,if=virtio,cache.direct=on
```

Additionally, QEMU has several ways of supporting zoned storage, including: (1) Using virtio-scsi: –device scsi-block allows for the passing through of SCSI ZBC devices, enabling the attachment of ZBC or ZAC HDDs to QEMU. (2) PCI device pass-through: While NVMe ZNS emulation is available for testing purposes, it cannot yet pass through a zoned device from the host. To pass on the NVMe ZNS device to the guest, use VFIO PCI pass the entire NVMe PCI adapter through to the guest. Likewise, an HDD HBA can be passed on to QEMU all HDDs attached to the HBA.

7.4 Internal Subsystem Information

Details about QEMU's various subsystems including how to add features to them.

7.4.1 The QEMU Object Model (QOM)

The QEMU Object Model provides a framework for registering user creatable types and instantiating objects from those types. QOM provides the following features:

- System for dynamically registering types
- Support for single-inheritance of types
- Multiple inheritance of stateless interfaces
- Mapping internal members to publicly exposed properties

The root object class is TYPE_OBJECT which provides for the basic object methods.

The QOM tree

The QOM tree is a composition tree which represents all of the objects that make up a QEMU "machine". You can view this tree by running `info qom-tree` in the *QEMU Monitor*. It will contain both objects created by the machine itself as well those created due to user configuration.

Creating a QOM class

A simple minimal device implementation may look something like below:

Listing 4: Creating a minimal type

```
#include "qdev.h"

#define TYPE_MY_DEVICE "my-device"

// No new virtual functions: we can reuse the typedef for the
// superclass.
typedef DeviceClass MyDeviceClass;
typedef struct MyDevice
{
    DeviceState parent_obj;

    int reg0, reg1, reg2;
} MyDevice;

static const TypeInfo my_device_info = {
    .name = TYPE_MY_DEVICE,
    .parent = TYPE_DEVICE,
    .instance_size = sizeof(MyDevice),
};

static void my_device_register_types(void)
{
    type_register_static(&my_device_info);
}

type_init(my_device_register_types)
```

In the above example, we create a simple type that is described by #TypeInfo. #TypeInfo describes information about the type including what it inherits from, the instance and class size, and constructor/destructor hooks.

The TYPE_DEVICE class is the parent class for all modern devices implemented in QEMU and adds some specific methods to handle QEMU device model. This includes managing the lifetime of devices from creation through to when they become visible to the guest and eventually unrealized.

Alternatively several static types could be registered using helper macro DEFINE_TYPES()

```
static const TypeInfo device_types_info[] = {
    {
        .name = TYPE_MY_DEVICE_A,
        .parent = TYPE_DEVICE,
        .instance_size = sizeof(MyDeviceA),
    },
    {
        .name = TYPE_MY_DEVICE_B,
        .parent = TYPE_DEVICE,
        .instance_size = sizeof(MyDeviceB),
    },
};
```

(continues on next page)

(continued from previous page)

```
DEFINE_TYPES(device_types_info)
```

Every type has an #ObjectClass associated with it. #ObjectClass derivatives are instantiated dynamically but there is only ever one instance for any given type. The #ObjectClass typically holds a table of function pointers for the virtual methods implemented by this type.

Using object_new(), a new #Object derivative will be instantiated. You can cast an #Object to a subclass (or base-class) type using object_dynamic_cast(). You typically want to define macro wrappers around OBJECT_CHECK() and OBJECT_CLASS_CHECK() to make it easier to convert to a specific type:

Listing 5: Typecasting macros

```
#define MY_DEVICE_GET_CLASS(obj) \
   OBJECT_GET_CLASS(MyDeviceClass, obj, TYPE_MY_DEVICE)
#define MY_DEVICE_CLASS(klass) \
   OBJECT_CLASS_CHECK(MyDeviceClass, klass, TYPE_MY_DEVICE)
#define MY_DEVICE(obj) \
   OBJECT_CHECK(MyDevice, obj, TYPE_MY_DEVICE)
```

In case the ObjectClass implementation can be built as module a module_obj() line must be added to make sure qemu loads the module when the object is needed.

```
module_obj(TYPE_MY_DEVICE);
```

Class Initialization

Before an object is initialized, the class for the object must be initialized. There is only one class object for all instance objects that is created lazily.

Classes are initialized by first initializing any parent classes (if necessary). After the parent class object has initialized, it will be copied into the current class object and any additional storage in the class object is zero filled.

The effect of this is that classes automatically inherit any virtual function pointers that the parent class has already initialized. All other fields will be zero filled.

Once all of the parent classes have been initialized, #TypeInfo::class_init is called to let the class being instantiated provide default initialize for its virtual functions. Here is how the above example might be modified to introduce an overridden virtual function:

Listing 6: Overriding a virtual function

```
#include "qdev.h"

void my_device_class_init(ObjectClass *klass, void *class_data)
{
    DeviceClass *dc = DEVICE_CLASS(klass);
    dc->reset = my_device_reset;
}

static const TypeInfo my_device_info = {
    .name = TYPE_MY_DEVICE,
    .parent = TYPE_DEVICE,
    .instance_size = sizeof(MyDevice),
```

(continues on next page)

(continued from previous page)

```
    .class_init = my_device_class_init,
};
```

Introducing new virtual methods requires a class to define its own struct and to add a .class_size member to the #TypeInfo. Each method will also have a wrapper function to call it easily:

Listing 7: Defining an abstract class

```
#include "qdev.h"

typedef struct MyDeviceClass
{
    DeviceClass parent_class;

    void (*frobnicate) (MyDevice *obj);
} MyDeviceClass;

static const TypeInfo my_device_info = {
    .name = TYPE_MY_DEVICE,
    .parent = TYPE_DEVICE,
    .instance_size = sizeof(MyDevice),
    .abstract = true, // or set a default in my_device_class_init
    .class_size = sizeof(MyDeviceClass),
};

void my_device_frobnicate(MyDevice *obj)
{
    MyDeviceClass *klass = MY_DEVICE_GET_CLASS(obj);

    klass->frobnicate(obj);
}
```

Interfaces

Interfaces allow a limited form of multiple inheritance. Instances are similar to normal types except for the fact that are only defined by their classes and never carry any state. As a consequence, a pointer to an interface instance should always be of incomplete type in order to be sure it cannot be dereferenced. That is, you should define the 'typedef struct SomethingIf SomethingIf' so that you can pass around `SomethingIf *si` arguments, but not define a `struct SomethingIf { ... }`. The only things you can validly do with a `SomethingIf *` are to pass it as an argument to a method on its corresponding SomethingIfClass, or to dynamically cast it to an object that implements the interface.

Methods

A *method* is a function within the namespace scope of a class. It usually operates on the object instance by passing it as a strongly-typed first argument. If it does not operate on an object instance, it is dubbed *class method*.

Methods cannot be overloaded. That is, the #ObjectClass and method name uniquely identity the function to be called; the signature does not vary except for trailing varargs.

Methods are always *virtual*. Overriding a method in #TypeInfo.class_init of a subclass leads to any user of the class obtained via OBJECT_GET_CLASS() accessing the overridden function. The original function is not automatically invoked. It is the responsibility of the overriding class to determine whether and when to invoke the method being overridden.

To invoke the method being overridden, the preferred solution is to store the original value in the overriding class before overriding the method. This corresponds to `{super,base}.method(...)` in Java and C# respectively; this frees the overriding class from hardcoding its parent class, which someone might choose to change at some point.

Listing 8: Overriding a virtual method

```
typedef struct MyState MyState;

typedef void (*MyDoSomething)(MyState *obj);

typedef struct MyClass {
    ObjectClass parent_class;

    MyDoSomething do_something;
} MyClass;

static void my_do_something(MyState *obj)
{
    // do something
}

static void my_class_init(ObjectClass *oc, void *data)
{
    MyClass *mc = MY_CLASS(oc);

    mc->do_something = my_do_something;
}

static const TypeInfo my_type_info = {
    .name = TYPE_MY,
    .parent = TYPE_OBJECT,
    .instance_size = sizeof(MyState),
    .class_size = sizeof(MyClass),
    .class_init = my_class_init,
};

typedef struct DerivedClass {
    MyClass parent_class;

    MyDoSomething parent_do_something;
} DerivedClass;
```

(continues on next page)

(continued from previous page)

```
static void derived_do_something(MyState *obj)
{
    DerivedClass *dc = DERIVED_GET_CLASS(obj);

    // do something here
    dc->parent_do_something(obj);
    // do something else here
}

static void derived_class_init(ObjectClass *oc, void *data)
{
    MyClass *mc = MY_CLASS(oc);
    DerivedClass *dc = DERIVED_CLASS(oc);

    dc->parent_do_something = mc->do_something;
    mc->do_something = derived_do_something;
}

static const TypeInfo derived_type_info = {
    .name = TYPE_DERIVED,
    .parent = TYPE_MY,
    .class_size = sizeof(DerivedClass),
    .class_init = derived_class_init,
};
```

Alternatively, object_class_by_name() can be used to obtain the class and its non-overridden methods for a specific type. This would correspond to `MyClass::method(...)` in C++.

One example of such methods is `DeviceClass.reset`. More examples can be found at *Device Life-cycle*.

Standard type declaration and definition macros

A lot of the code outlined above follows a standard pattern and naming convention. To reduce the amount of boilerplate code that needs to be written for a new type there are two sets of macros to generate the common parts in a standard format.

A type is declared using the OBJECT_DECLARE macro family. In types which do not require any virtual functions in the class, the OBJECT_DECLARE_SIMPLE_TYPE macro is suitable, and is commonly placed in the header file:

Listing 9: Declaring a simple type

```
OBJECT_DECLARE_SIMPLE_TYPE(MyDevice, MY_DEVICE)
```

This is equivalent to the following:

Listing 10: Expansion from declaring a simple type

```
typedef struct MyDevice MyDevice;
typedef struct MyDeviceClass MyDeviceClass;

G_DEFINE_AUTOPTR_CLEANUP_FUNC(MyDeviceClass, object_unref)

#define MY_DEVICE_GET_CLASS(void *obj) \
```

(continues on next page)

(continued from previous page)

```
        OBJECT_GET_CLASS(MyDeviceClass, obj, TYPE_MY_DEVICE)
#define MY_DEVICE_CLASS(void *klass) \
        OBJECT_CLASS_CHECK(MyDeviceClass, klass, TYPE_MY_DEVICE)
#define MY_DEVICE(void *obj)
        OBJECT_CHECK(MyDevice, obj, TYPE_MY_DEVICE)

struct MyDeviceClass {
    DeviceClass parent_class;
};
```

The 'struct MyDevice' needs to be declared separately. If the type requires virtual functions to be declared in the class struct, then the alternative OBJECT_DECLARE_TYPE() macro can be used. This does the same as OBJECT_DECLARE_SIMPLE_TYPE(), but without the 'struct MyDeviceClass' definition.

To implement the type, the OBJECT_DEFINE macro family is available. For the simplest case of a leaf class which doesn't need any of its own virtual functions (i.e. which was declared with OBJECT_DECLARE_SIMPLE_TYPE) the OBJECT_DEFINE_SIMPLE_TYPE macro is suitable:

Listing 11: Defining a simple type

```
OBJECT_DEFINE_SIMPLE_TYPE(MyDevice, my_device, MY_DEVICE, DEVICE)
```

This is equivalent to the following:

Listing 12: Expansion from defining a simple type

```
static void my_device_finalize(Object *obj);
static void my_device_class_init(ObjectClass *oc, void *data);
static void my_device_init(Object *obj);

static const TypeInfo my_device_info = {
    .parent = TYPE_DEVICE,
    .name = TYPE_MY_DEVICE,
    .instance_size = sizeof(MyDevice),
    .instance_init = my_device_init,
    .instance_finalize = my_device_finalize,
    .class_init = my_device_class_init,
};

static void
my_device_register_types(void)
{
    type_register_static(&my_device_info);
}
type_init(my_device_register_types);
```

This is sufficient to get the type registered with the type system, and the three standard methods now need to be implemented along with any other logic required for the type.

If the class needs its own virtual methods, or has some other per-class state it needs to store in its own class struct, then you can use the OBJECT_DEFINE_TYPE macro. This does the same thing as OBJECT_DEFINE_SIMPLE_TYPE, but it also sets the class_size of the type to the size of the class struct.

Listing 13: Defining a type which needs a class struct

```
OBJECT_DEFINE_TYPE(MyDevice, my_device, MY_DEVICE, DEVICE)
```

If the type needs to implement one or more interfaces, then the OBJECT_DEFINE_SIMPLE_TYPE_WITH_INTERFACES() and OBJECT_DEFINE_TYPE_WITH_INTERFACES() macros can be used instead. These accept an array of interface type names. The difference between them is that the former is for simple leaf classes that don't need a class struct, and the latter is for when you will be defining a class struct.

Listing 14: Defining a simple type implementing interfaces

```
OBJECT_DEFINE_SIMPLE_TYPE_WITH_INTERFACES(MyDevice, my_device,
                                          MY_DEVICE, DEVICE,
                                          { TYPE_USER_CREATABLE },
                                          { NULL })
```

Listing 15: Defining a type implementing interfaces

```
OBJECT_DEFINE_TYPE_WITH_INTERFACES(MyDevice, my_device,
                                   MY_DEVICE, DEVICE,
                                   { TYPE_USER_CREATABLE },
                                   { NULL })
```

If the type is not intended to be instantiated, then the OBJECT_DEFINE_ABSTRACT_TYPE() macro can be used instead:

Listing 16: Defining a simple abstract type

```
OBJECT_DEFINE_ABSTRACT_TYPE(MyDevice, my_device,
                            MY_DEVICE, DEVICE)
```

Device Life-cycle

As class initialisation cannot fail devices have an two additional methods to handle the creation of dynamic devices. The `realize` function is called with `Error **` pointer which should be set if the device cannot complete its setup. Otherwise on successful completion of the `realize` method the device object is added to the QOM tree and made visible to the guest.

The reverse function is `unrealize` and should be were clean-up code lives to tidy up after the system is done with the device.

All devices can be instantiated by C code, however only some can created dynamically via the command line or monitor.

Likewise only some can be unplugged after creation and need an explicit `unrealize` implementation. This is determined by the `user_creatable` variable in the root `DeviceClass` structure. Devices can only be unplugged if their `parent_bus` has a registered `HotplugHandler`.

API Reference

See the *QOM API* and *QDEV API* documents for the complete API description.

7.4.2 Atomic operations in QEMU

CPUs perform independent memory operations effectively in random order. but this can be a problem for CPU-CPU interaction (including interactions between QEMU and the guest). Multi-threaded programs use various tools to instruct the compiler and the CPU to restrict the order to something that is consistent with the expectations of the programmer.

The most basic tool is locking. Mutexes, condition variables and semaphores are used in QEMU, and should be the default approach to synchronization. Anything else is considerably harder, but it's also justified more often than one would like; the most performance-critical parts of QEMU in particular require a very low level approach to concurrency, involving memory barriers and atomic operations. The semantics of concurrent memory accesses are governed by the C11 memory model.

QEMU provides a header, `qemu/atomic.h`, which wraps C11 atomics to provide better portability and a less verbose syntax. `qemu/atomic.h` provides macros that fall in three camps:

- compiler barriers: `barrier()`;
- weak atomic access and manual memory barriers: `qatomic_read()`, `qatomic_set()`, `smp_rmb()`, `smp_wmb()`, `smp_mb()`, `smp_mb_acquire()`, `smp_mb_release()`, `smp_read_barrier_depends()`, `smp_mb__before_rmw()`, `smp_mb__after_rmw()`;
- sequentially consistent atomic access: everything else.

In general, use of `qemu/atomic.h` should be wrapped with more easily used data structures (e.g. the lock-free singly-linked list operations `QSLIST_INSERT_HEAD_ATOMIC` and `QSLIST_MOVE_ATOMIC`) or synchronization primitives (such as RCU, `QemuEvent` or `QemuLockCnt`). Bare use of atomic operations and memory barriers should be limited to inter-thread checking of flags and documented thoroughly.

Compiler memory barrier

`barrier()` prevents the compiler from moving the memory accesses on either side of it to the other side. The compiler barrier has no direct effect on the CPU, which may then reorder things however it wishes.

`barrier()` is mostly used within `qemu/atomic.h` itself. On some architectures, CPU guarantees are strong enough that blocking compiler optimizations already ensures the correct order of execution. In this case, `qemu/atomic.h` will reduce stronger memory barriers to simple compiler barriers.

Still, `barrier()` can be useful when writing code that can be interrupted by signal handlers.

Sequentially consistent atomic access

Most of the operations in the `qemu/atomic.h` header ensure *sequential consistency*, where "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program".

`qemu/atomic.h` provides the following set of atomic read-modify-write operations:

```
void qatomic_inc(ptr)
void qatomic_dec(ptr)
void qatomic_add(ptr, val)
void qatomic_sub(ptr, val)
```

(continues on next page)

(continued from previous page)

```
void qatomic_and(ptr, val)
void qatomic_or(ptr, val)

typeof(*ptr) qatomic_fetch_inc(ptr)
typeof(*ptr) qatomic_fetch_dec(ptr)
typeof(*ptr) qatomic_fetch_add(ptr, val)
typeof(*ptr) qatomic_fetch_sub(ptr, val)
typeof(*ptr) qatomic_fetch_and(ptr, val)
typeof(*ptr) qatomic_fetch_or(ptr, val)
typeof(*ptr) qatomic_fetch_xor(ptr, val)
typeof(*ptr) qatomic_fetch_inc_nonzero(ptr)
typeof(*ptr) qatomic_xchg(ptr, val)
typeof(*ptr) qatomic_cmpxchg(ptr, old, new)
```

all of which return the old value of `*ptr`. These operations are polymorphic; they operate on any type that is as wide as a pointer or smaller.

Similar operations return the new value of `*ptr`:

```
typeof(*ptr) qatomic_inc_fetch(ptr)
typeof(*ptr) qatomic_dec_fetch(ptr)
typeof(*ptr) qatomic_add_fetch(ptr, val)
typeof(*ptr) qatomic_sub_fetch(ptr, val)
typeof(*ptr) qatomic_and_fetch(ptr, val)
typeof(*ptr) qatomic_or_fetch(ptr, val)
typeof(*ptr) qatomic_xor_fetch(ptr, val)
```

`qemu/atomic.h` also provides an optimized shortcut for `qatomic_set` followed by `smp_mb`:

```
void        qatomic_set_mb(ptr, val)
```

Weak atomic access and manual memory barriers

Compared to sequentially consistent atomic access, programming with weaker consistency models can be considerably more complicated. The only guarantees that you can rely upon in this case are:

- atomic accesses will not cause data races (and hence undefined behavior); ordinary accesses instead cause data races if they are concurrent with other accesses of which at least one is a write. In order to ensure this, the compiler will not optimize accesses out of existence, create unsolicited accesses, or perform other similar optimizations.
- acquire operations will appear to happen, with respect to the other components of the system, before all the LOAD or STORE operations specified afterwards.
- release operations will appear to happen, with respect to the other components of the system, after all the LOAD or STORE operations specified before.
- release operations will *synchronize with* acquire operations; see *Acquire/release pairing and the synchronizes-with relation* for a detailed explanation.

When using this model, variables are accessed with:

- `qatomic_read()` and `qatomic_set()`; these prevent the compiler from optimizing accesses out of existence and creating unsolicited accesses, but do not otherwise impose any ordering on loads and stores: both the compiler and the processor are free to reorder them.

- `qatomic_load_acquire()`, which guarantees the LOAD to appear to happen, with respect to the other components of the system, before all the LOAD or STORE operations specified afterwards. Operations coming before `qatomic_load_acquire()` can still be reordered after it.
- `qatomic_store_release()`, which guarantees the STORE to appear to happen, with respect to the other components of the system, after all the LOAD or STORE operations specified before. Operations coming after `qatomic_store_release()` can still be reordered before it.

Restrictions to the ordering of accesses can also be specified using the memory barrier macros: `smp_rmb()`, `smp_wmb()`, `smp_mb()`, `smp_mb_acquire()`, `smp_mb_release()`, `smp_read_barrier_depends()`.

Memory barriers control the order of references to shared memory. They come in six kinds:

- `smp_rmb()` guarantees that all the LOAD operations specified before the barrier will appear to happen before all the LOAD operations specified after the barrier with respect to the other components of the system.

  In other words, `smp_rmb()` puts a partial ordering on loads, but is not required to have any effect on stores.

- `smp_wmb()` guarantees that all the STORE operations specified before the barrier will appear to happen before all the STORE operations specified after the barrier with respect to the other components of the system.

  In other words, `smp_wmb()` puts a partial ordering on stores, but is not required to have any effect on loads.

- `smp_mb_acquire()` guarantees that all the LOAD operations specified before the barrier will appear to happen before all the LOAD or STORE operations specified after the barrier with respect to the other components of the system.

- `smp_mb_release()` guarantees that all the STORE operations specified *after* the barrier will appear to happen after all the LOAD or STORE operations specified *before* the barrier with respect to the other components of the system.

- `smp_mb()` guarantees that all the LOAD and STORE operations specified before the barrier will appear to happen before all the LOAD and STORE operations specified after the barrier with respect to the other components of the system.

  `smp_mb()` puts a partial ordering on both loads and stores. It is stronger than both a read and a write memory barrier; it implies both `smp_mb_acquire()` and `smp_mb_release()`, but it also prevents STOREs coming before the barrier from overtaking LOADs coming after the barrier and vice versa.

- `smp_read_barrier_depends()` is a weaker kind of read barrier. On most processors, whenever two loads are performed such that the second depends on the result of the first (e.g., the first load retrieves the address to which the second load will be directed), the processor will guarantee that the first LOAD will appear to happen before the second with respect to the other components of the system. Therefore, unlike `smp_rmb()` or `qatomic_load_acquire()`, `smp_read_barrier_depends()` can be just a compiler barrier on weakly-ordered architectures such as Arm or PPC[#]_.

  Note that the first load really has to have a _data_ dependency and not a control dependency. If the address for the second load is dependent on the first load, but the dependency is through a conditional rather than actually loading the address itself, then it's a _control_ dependency and a full read barrier or better is required.

Memory barriers and `qatomic_load_acquire`/`qatomic_store_release` are mostly used when a data structure has one thread that is always a writer and one thread that is always a reader:

thread 1	thread 2
`qatomic_store_release(&a, x);` `qatomic_store_release(&b, y);`	`y = qatomic_load_acquire(&b);` `x = qatomic_load_acquire(&a);`

In this case, correctness is easy to check for using the "pairing" trick that is explained below.

Sometimes, a thread is accessing many variables that are otherwise unrelated to each other (for example because, apart from the current thread, exactly one other thread will read or write each of these variables). In this case, it is possible to "hoist" the barriers outside a loop. For example:

before	after

`n = 0;`
`for (i = 0; i < 10; i++)`
`  n += qatomic_load_acquire(&a[i]);`
	`n = 0;`
	`for (i = 0; i < 10; i++)`
	`  n += qatomic_read(&a[i]);`
	`smp_mb_acquire();`

`for (i = 0; i < 10; i++)`
`  qatomic_store_release(&a[i],␣`
`↪false);`
	`smp_mb_release();`
	`for (i = 0; i < 10; i++)`
	`  qatomic_set(&a[i], false);`

Splitting a loop can also be useful to reduce the number of barriers:

before	after

`n = 0;`
`for (i = 0; i < 10; i++) {`
`  qatomic_store_release(&a[i],␣`
`↪false);`
`  smp_mb();`
`  n += qatomic_read(&b[i]);`
`}`
	`smp_mb_release();`
	`for (i = 0; i < 10; i++)`
	`  qatomic_set(&a[i], false);`
	`smb_mb();`
	`n = 0;`
	`for (i = 0; i < 10; i++)`
	`  n += qatomic_read(&b[i]);`

In this case, a `smp_mb_release()` is also replaced with a (possibly cheaper, and clearer as well) `smp_wmb()`:

before	after

`for (i = 0; i < 10; i++) {`
`  qatomic_store_release(&a[i],␣`
`↪false);`
`  qatomic_store_release(&b[i],␣`
`↪false);`
`}`
	`smp_mb_release();`
	`for (i = 0; i < 10; i++)`
	`  qatomic_set(&a[i], false);`
	`smb_wmb();`
	`for (i = 0; i < 10; i++)`
	`  qatomic_set(&b[i], false);`

Acquire/release pairing and the *synchronizes-with* relation

Atomic operations other than `qatomic_set()` and `qatomic_read()` have either *acquire* or *release* semantics[2]. This has two effects:

- within a thread, they are ordered either before subsequent operations (for acquire) or after previous operations (for release).
- if a release operation in one thread *synchronizes with* an acquire operation in another thread, the ordering constraints propagates from the first to the second thread. That is, everything before the release operation in the first thread is guaranteed to *happen before* everything after the acquire operation in the second thread.

The concept of acquire and release semantics is not exclusive to atomic operations; almost all higher-level synchronization primitives also have acquire or release semantics. For example:

- `pthread_mutex_lock` has acquire semantics, `pthread_mutex_unlock` has release semantics and synchronizes with a `pthread_mutex_lock` for the same mutex.
- `pthread_cond_signal` and `pthread_cond_broadcast` have release semantics; `pthread_cond_wait` has both release semantics (synchronizing with `pthread_mutex_lock`) and acquire semantics (synchronizing with `pthread_mutex_unlock` and signaling of the condition variable).
- `pthread_create` has release semantics and synchronizes with the start of the new thread; `pthread_join` has acquire semantics and synchronizes with the exiting of the thread.
- `qemu_event_set` has release semantics, `qemu_event_wait` has acquire semantics.

For example, in the following example there are no atomic accesses, but still thread 2 is relying on the *synchronizes-with* relation between `pthread_exit` (release) and `pthread_join` (acquire):

thread 1	thread 2
`*a = 1;` `pthread_exit(a);`	`pthread_join(thread1, &a);` `x = *a;`

Synchronization between threads basically descends from this pairing of a release operation and an acquire operation. Therefore, atomic operations other than `qatomic_set()` and `qatomic_read()` will almost always be paired with another operation of the opposite kind: an acquire operation will pair with a release operation and vice versa. This rule of thumb is extremely useful; in the case of QEMU, however, note that the other operation may actually be in a driver that runs in the guest!

`smp_read_barrier_depends()`, `smp_rmb()`, `smp_mb_acquire()`, `qatomic_load_acquire()` and `qatomic_rcu_read()` all count as acquire operations. `smp_wmb()`, `smp_mb_release()`, `qatomic_store_release()` and `qatomic_rcu_set()` all count as release operations. `smp_mb()` counts as both acquire and release, therefore it can pair with any other atomic operation. Here is an example:

thread 1	thread 2

`qatomic_set(&a, 1);`
`smp_wmb();`
`qatomic_set(&b, 2);`
	`x = qatomic_read(&b);`
	`smp_rmb();`
	`y = qatomic_read(&a);`

[2] Read-modify-write operations can have both—acquire applies to the read part, and release to the write.

Note that a load-store pair only counts if the two operations access the same variable: that is, a store-release on a variable `x` *synchronizes with* a load-acquire on a variable `x`, while a release barrier synchronizes with any acquire operation. The following example shows correct synchronization:

thread 1	thread 2
`qatomic_set(&a, 1);` `qatomic_store_release(&b, 2);`	`x = qatomic_load_acquire(&b);` `y = qatomic_read(&a);`

Acquire and release semantics of higher-level primitives can also be relied upon for the purpose of establishing the *synchronizes with* relation.

Note that the "writing" thread is accessing the variables in the opposite order as the "reading" thread. This is expected: stores before a release operation will normally match the loads after the acquire operation, and vice versa. In fact, this happened already in the `pthread_exit/pthread_join` example above.

Finally, this more complex example has more than two accesses and data dependency barriers. It also does not use atomic accesses whenever there cannot be a data race:

thread 1	thread 2

`b[2] = 1;`
`smp_wmb();`
`x->i = 2;`
`smp_wmb();`
`qatomic_set(&a, x);`
	`x = qatomic_read(&a);`
	`smp_read_barrier_depends();`
	`y = x->i;`
	`smp_read_barrier_depends();`
	`z = b[y];`

Comparison with Linux kernel primitives

Here is a list of differences between Linux kernel atomic operations and memory barriers, and the equivalents in QEMU:

- atomic operations in Linux are always on a 32-bit int type and use a boxed `atomic_t` type; atomic operations in QEMU are polymorphic and use normal C types.
- Originally, `atomic_read` and `atomic_set` in Linux gave no guarantee at all. Linux 4.1 updated them to implement volatile semantics via `ACCESS_ONCE` (or the more recent `READ/WRITE_ONCE`).

  QEMU's `qatomic_read` and `qatomic_set` implement C11 atomic relaxed semantics if the compiler supports it, and volatile semantics otherwise. Both semantics prevent the compiler from doing certain transformations; the difference is that atomic accesses are guaranteed to be atomic, while volatile accesses aren't. Thus, in the volatile case we just cross our fingers hoping that the compiler will generate atomic accesses, since we assume the variables passed are machine-word sized and properly aligned.

  No barriers are implied by `qatomic_read` and `qatomic_set` in either Linux or QEMU.
- atomic read-modify-write operations in Linux are of three kinds:

`atomic_OP`	returns void
`atomic_OP_return`	returns new value of the variable
`atomic_fetch_OP`	returns the old value of the variable
`atomic_cmpxchg`	returns the old value of the variable

In QEMU, the second kind is named `atomic_OP_fetch`.

- different atomic read-modify-write operations in Linux imply a different set of memory barriers. In QEMU, all of them enforce sequential consistency: there is a single order in which the program sees them happen.
- however, according to the C11 memory model that QEMU uses, this order does not propagate to other memory accesses on either side of the read-modify-write operation. As far as those are concerned, the operation consist of just a load-acquire followed by a store-release. Stores that precede the RMW operation, and loads that follow it, can still be reordered and will happen *in the middle* of the read-modify-write operation!

  Therefore, the following example is correct in Linux but not in QEMU:

Linux (correct)	QEMU (incorrect)
`a = atomic_fetch_add(&x, 2);` `b = READ_ONCE(&y);`	`a = qatomic_fetch_add(&x, 2);` `b = qatomic_read(&y);`

because the read of `y` can be moved (by either the processor or the compiler) before the write of `x`.

Fixing this requires a full memory barrier between the write of `x` and the read of `y`. QEMU provides `smp_mb__before_rmw()` and `smp_mb__after_rmw()`; they act both as an optimization, avoiding the memory barrier on processors where it is unnecessary, and as a clarification of this corner case of the C11 memory model:

QEMU (correct)

`a = qatomic_fetch_add(&x, 2);`
`smp_mb__after_rmw();`
`b = qatomic_read(&y);`

In the common case where only one thread writes `x`, it is also possible to write it like this:

QEMU (correct)

`a = qatomic_read(&x);`
`qatomic_set_mb(&x, a + 2);`
`b = qatomic_read(&y);`

Sources

- `Documentation/memory-barriers.txt` from the Linux kernel

7.4.3 block-coroutine-wrapper

A lot of functions in QEMU block layer (see block/*) can only be called in coroutine context. Such functions are normally marked by the coroutine_fn specifier. Still, sometimes we need to call them from non-coroutine context; for this we need to start a coroutine, run the needed function from it and wait for the coroutine to finish in a BDRV_POLL_WHILE() loop. To run a coroutine we need a function with one void* argument. So for each coroutine_fn function which needs a non-coroutine interface, we should define a structure to pack the parameters, define a separate function to unpack the parameters and call the original function and finally define a new interface function with same list of arguments as original one, which will pack the parameters into a struct, create a coroutine, run it and wait in BDRV_POLL_WHILE() loop. It's boring to create such wrappers by hand, so we have a script to generate them.

Usage

Assume we have defined the coroutine_fn function bdrv_co_foo(<some args>) and need a non-coroutine interface for it, called bdrv_foo(<same args>). In this case the script can help. To trigger the generation:

1. You need bdrv_foo declaration somewhere (for example, in block/coroutines.h) with the co_wrapper mark, like this:

```
int co_wrapper bdrv_foo(<some args>);
```

2. You need to feed this declaration to block-coroutine-wrapper script. For this, add the .h (or .c) file with the declaration to the input: files(...) list of block_gen_c target declaration in block/meson.build

You are done. During the build, coroutine wrappers will be generated in <BUILD_DIR>/block/block-gen.c.

Links

1. The script location is scripts/block-coroutine-wrapper.py.
2. Generic place for private co_wrapper declarations is block/coroutines.h, for public declarations: include/block/block.h
3. The core API of generated coroutine wrappers is placed in (not generated) block/block-gen.h

7.4.4 Modelling a clock tree in QEMU

What are clocks?

Clocks are QOM objects developed for the purpose of modelling the distribution of clocks in QEMU.

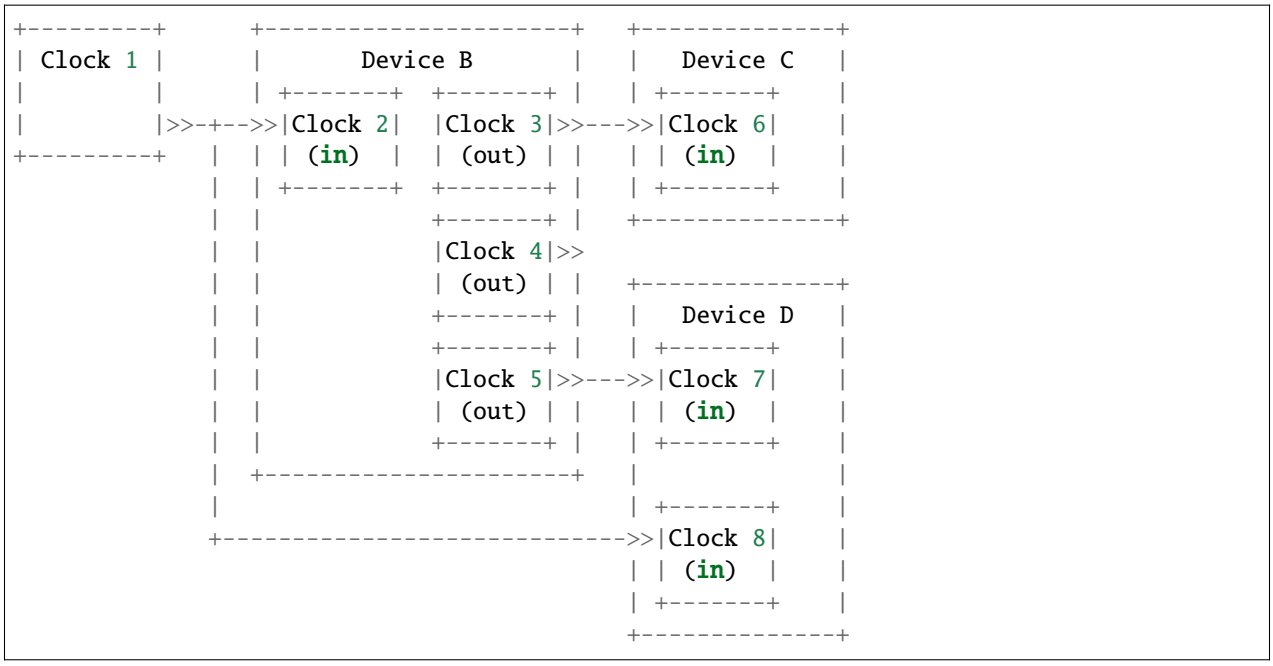
They allow us to model the clock distribution of a platform and detect configuration errors in the clock tree such as badly configured PLL, clock source selection or disabled clock.

The object is *Clock* and its QOM name is clock (in C code, the macro TYPE_CLOCK).

Clocks are typically used with devices where they are used to model inputs and outputs. They are created in a similar way to GPIOs. Inputs and outputs of different devices can be connected together.

In these cases a Clock object is a child of a Device object, but this is not a requirement. Clocks can be independent of devices. For example it is possible to create a clock outside of any device to model the main clock source of a machine.

Here is an example of clocks:



Clocks are defined in the `include/hw/clock.h` header and device related functions are defined in the `include/hw/qdev-clock.h` header.

The clock state

The state of a clock is its period; it is stored as an integer representing it in units of 2^{-32} ns. The special value of 0 is used to represent the clock being inactive or gated. The clocks do not model the signal itself (pin toggling) or other properties such as the duty cycle.

All clocks contain this state: outputs as well as inputs. This allows the current period of a clock to be fetched at any time. When a clock is updated, the value is immediately propagated to all connected clocks in the tree.

To ease interaction with clocks, helpers with a unit suffix are defined for every clock state setter or getter. The suffixes are:

- `_ns` for handling periods in nanoseconds
- `_hz` for handling frequencies in hertz

The 0 period value is converted to 0 in hertz and vice versa. 0 always means that the clock is disabled.

Adding a new clock

Adding clocks to a device must be done during the init method of the Device instance.

To add an input clock to a device, the function `qdev_init_clock_in()` must be used. It takes the name, a callback, an opaque parameter for the callback and a mask of events when the callback should be called (this will be explained in a following section). Output is simpler; only the name is required. Typically:

```
qdev_init_clock_in(DEVICE(dev), "clk_in", clk_in_callback, dev, ClockUpdate);
qdev_init_clock_out(DEVICE(dev), "clk_out");
```

Both functions return the created Clock pointer, which should be saved in the device's state structure for further use.

These objects will be automatically deleted by the QOM reference mechanism.

Note that it is possible to create a static array describing clock inputs and outputs. The function `qdev_init_clocks()` must be called with the array as parameter to initialize the clocks: it has the same behaviour as calling the `qdev_init_clock_in/out()` for each clock in the array. To ease the array construction, some macros are defined in `include/hw/qdev-clock.h`. As an example, the following creates 2 clocks to a device: one input and one output.

```
/* device structure containing pointers to the clock objects */
typedef struct MyDeviceState {
    DeviceState parent_obj;
    Clock *clk_in;
    Clock *clk_out;
} MyDeviceState;

/*
 * callback for the input clock (see "Callback on input clock
 * change" section below for more information).
 */
static void clk_in_callback(void *opaque, ClockEvent event);

/*
 * static array describing clocks:
 * + a clock input named "clk_in", whose pointer is stored in
 *   the clk_in field of a MyDeviceState structure with callback
 *   clk_in_callback.
 * + a clock output named "clk_out" whose pointer is stored in
 *   the clk_out field of a MyDeviceState structure.
 */
static const ClockPortInitArray mydev_clocks = {
    QDEV_CLOCK_IN(MyDeviceState, clk_in, clk_in_callback, ClockUpdate),
    QDEV_CLOCK_OUT(MyDeviceState, clk_out),
    QDEV_CLOCK_END
};

/* device initialization function */
static void mydev_init(Object *obj)
{
    /* cast to MyDeviceState */
    MyDeviceState *mydev = MYDEVICE(obj);
    /* create and fill the pointer fields in the MyDeviceState */
    qdev_init_clocks(mydev, mydev_clocks);
    [...]
}
```

An alternative way to create a clock is to simply call `object_new(TYPE_CLOCK)`. In that case the clock will neither be an input nor an output of a device. After the whole QOM hierarchy of the clock has been set `clock_setup_canonical_path()` should be called.

At creation, the period of the clock is 0: the clock is disabled. You can change it using `clock_set_ns()` or `clock_set_hz()`.

Note that if you are creating a clock with a fixed period which will never change (for example the main clock source of a board), then you'll have nothing else to do. This value will be propagated to other clocks when connecting the clocks together and devices will fetch the right value during the first reset.

Clock callbacks

You can give a clock a callback function in several ways:

- by passing it as an argument to `qdev_init_clock_in()`
- as an argument to the `QDEV_CLOCK_IN()` macro initializing an array to be passed to `qdev_init_clocks()`
- by directly calling the `clock_set_callback()` function

The callback function must be of this type:

```
typedef void ClockCallback(void *opaque, ClockEvent event);
```

The `opaque` argument is the pointer passed to `qdev_init_clock_in()` or `clock_set_callback()`; for `qdev_init_clocks()` it is the `dev` device pointer.

The `event` argument specifies why the callback has been called. When you register the callback you specify a mask of ClockEvent values that you are interested in. The callback will only be called for those events.

The events currently supported are:

- `ClockPreUpdate` : called when the input clock's period is about to update. This is useful if the device needs to do some action for which it needs to know the old value of the clock period. During this callback, Clock API functions like `clock_get()` or `clock_ticks_to_ns()` will use the old period.
- `ClockUpdate` : called after the input clock's period has changed. During this callback, Clock API functions like `clock_ticks_to_ns()` will use the new period.

Note that a clock only has one callback: it is not possible to register different functions for different events. You must register a single callback which listens for all of the events you are interested in, and use the `event` argument to identify which event has happened.

Retrieving clocks from a device

`qdev_get_clock_in()` and `dev_get_clock_out()` are available to get the clock inputs or outputs of a device. For example:

```
Clock *clk = qdev_get_clock_in(DEVICE(mydev), "clk_in");
```

or:

```
Clock *clk = qdev_get_clock_out(DEVICE(mydev), "clk_out");
```

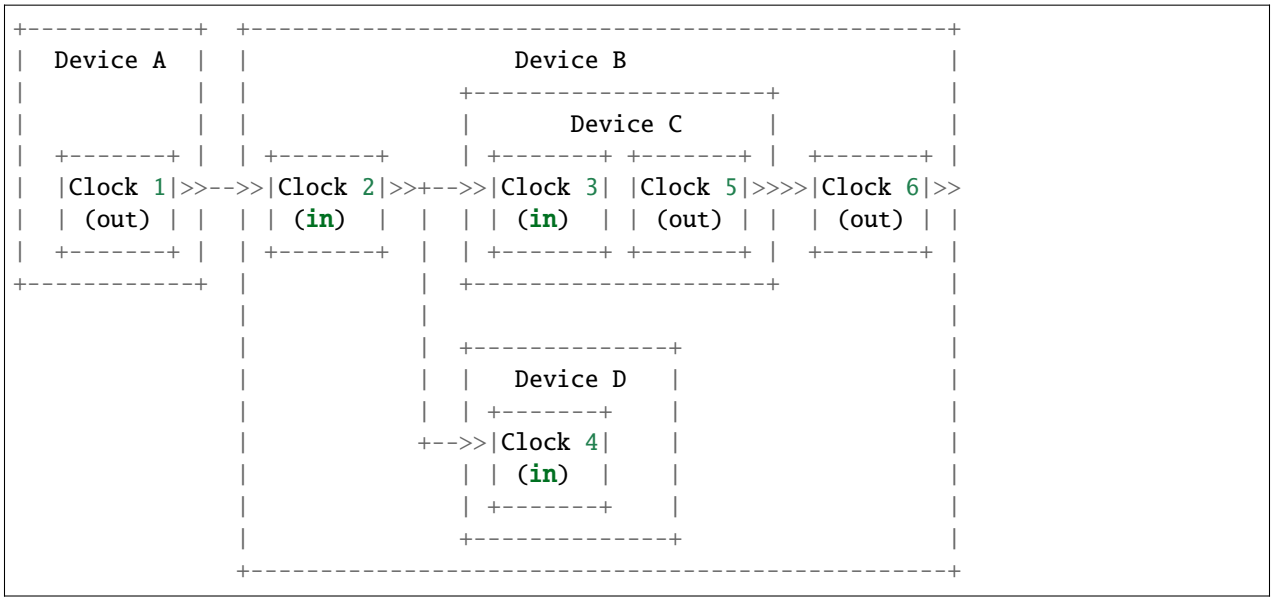
Connecting two clocks together

To connect two clocks together, use the `clock_set_source()` function. Given two clocks `clk1`, and `clk2`, `clock_set_source(clk2, clk1);` configures `clk2` to follow the `clk1` period changes. Every time `clk1` is updated, `clk2` will be updated too.

When connecting clock between devices, prefer using the `qdev_connect_clock_in()` function to set the source of an input device clock. For example, to connect the input clock `clk2` of `devB` to the output clock `clk1` of `devA`, do:

```
qdev_connect_clock_in(devB, "clk2", qdev_get_clock_out(devA, "clk1"))
```

We used `qdev_get_clock_out()` above, but any clock can drive an input clock, even another input clock. The following diagram shows some examples of connections. Note also that a clock can drive several other clocks.



In the above example, when *Clock 1* is updated by *Device A*, three clocks get the new clock period value: *Clock 2*, *Clock 3* and *Clock 4*.

It is not possible to disconnect a clock or to change the clock connection after it is connected.

Clock multiplier and divider settings

By default, when clocks are connected together, the child clocks run with the same period as their source (parent) clock. The Clock API supports a built-in period multiplier/divider mechanism so you can configure a clock to make its children run at a different period from its own. If you call the `clock_set_mul_div()` function you can specify the clock's multiplier and divider values. The children of that clock will all run with a period of `parent_period * multiplier / divider`. For instance, if the clock has a frequency of 8MHz and you set its multiplier to 2 and its divider to 3, the child clocks will run at 12MHz.

You can change the multiplier and divider of a clock at runtime, so you can use this to model clock controller devices which have guest-programmable frequency multipliers or dividers.

Similarly to `clock_set()`, `clock_set_mul_div()` returns `true` if the clock state was modified; that is, if the multiplier or the diviser or both were changed by the call.

Note that `clock_set_mul_div()` does not automatically call `clock_propagate()`. If you make a runtime change to the multiplier or divider you must call clock_propagate() yourself.

Unconnected input clocks

A newly created input clock is disabled (period of 0). This means the clock will be considered as disabled until the period is updated. If the clock remains unconnected it will always keep its initial value of 0. If this is not the desired behaviour, `clock_set()`, `clock_set_ns()` or `clock_set_hz()` should be called on the Clock object during device instance init. For example:

```
clk = qdev_init_clock_in(DEVICE(dev), "clk-in", clk_in_callback,
                         dev, ClockUpdate);
/* set initial value to 10ns / 100MHz */
clock_set_ns(clk, 10);
```

To enforce that the clock is wired up by the board code, you can call `clock_has_source()` in your device's realize method:

```
if (!clock_has_source(s->clk)) {
    error_setg(errp, "MyDevice: clk input must be connected");
    return;
}
```

Note that this only checks that the clock has been wired up; it is still possible that the output clock connected to it is disabled or has not yet been configured, in which case the period will be zero. You should use the clock callback to find out when the clock period changes.

Fetching clock frequency/period

To get the current state of a clock, use the functions `clock_get()` or `clock_get_hz()`.

`clock_get()` returns the period of the clock in its fully precise internal representation, as an unsigned 64-bit integer in units of 2^-32 nanoseconds. (For many purposes `clock_ticks_to_ns()` will be more convenient; see the section below on expiry deadlines.)

`clock_get_hz()` returns the frequency of the clock, rounded to the next lowest integer. This implies some inaccuracy due to the rounding, so be cautious about using it in calculations.

It is also possible to register a callback on clock frequency changes. Here is an example, which assumes that `clock_callback` has been specified as the callback for the `ClockUpdate` event:

```
void clock_callback(void *opaque, ClockEvent event) {
    MyDeviceState *s = (MyDeviceState *) opaque;
    /*
     * 'opaque' is the argument passed to qdev_init_clock_in();
     * usually this will be the device state pointer.
     */

    /* do something with the new period */
    fprintf(stdout, "device new period is %" PRIu64 "* 2^-32 ns\n",
                    clock_get(dev->my_clk_input));
}
```

If you are only interested in the frequency for displaying it to humans (for instance in debugging), use `clock_display_freq()`, which returns a prettified string-representation, e.g. "33.3 MHz". The caller must free the string with g_free() after use.

Calculating expiry deadlines

A commonly required operation for a clock is to calculate how long it will take for the clock to tick N times; this can then be used to set a timer expiry deadline. Use the function `clock_ticks_to_ns()`, which takes an unsigned 64-bit count of ticks and returns the length of time in nanoseconds required for the clock to tick that many times.

It is important not to try to calculate expiry deadlines using a shortcut like multiplying a "period of clock in nanoseconds" value by the tick count, because clocks can have periods which are not a whole number of nanoseconds, and the accumulated error in the multiplication can be significant.

For a clock with a very long period and a large number of ticks, the result of this function could in theory be too large to fit in a 64-bit value. To avoid overflow in this case, `clock_ticks_to_ns()` saturates the result to INT64_MAX (because this is the largest valid input to the QEMUTimer APIs). Since INT64_MAX nanoseconds is almost 300 years,

anything with an expiry later than that is in the "will never happen" category. Callers of `clock_ticks_to_ns()` should therefore generally not special-case the possibility of a saturated result but just allow the timer to be set to that far-future value. (If you are performing further calculations on the returned value rather than simply passing it to a QEMUTimer function like `timer_mod_ns()` then you should be careful to avoid overflow in those calculations, of course.)

Obtaining tick counts

For calculations where you need to know the number of ticks in a given duration, use `clock_ns_to_ticks()`. This function handles possible non-whole-number-of-nanoseconds periods and avoids potential rounding errors. It will return '0' if the clock is stopped (i.e. it has period zero). If the inputs imply a tick count that overflows a 64-bit value (a very long duration for a clock with a very short period) the output value is truncated, so effectively the 64-bit output wraps around.

Changing a clock period

A device can change its outputs using the `clock_update()`, `clock_update_ns()` or `clock_update_hz()` function. It will trigger updates on every connected input.

For example, let's say that we have an output clock *clkout* and we have a pointer to it in the device state because we did the following in init phase:

```
dev->clkout = qdev_init_clock_out(DEVICE(dev), "clkout");
```

Then at any time (apart from the cases listed below), it is possible to change the clock value by doing:

```
clock_update_hz(dev->clkout, 1000 * 1000 * 1000); /* 1GHz */
```

Because updating a clock may trigger any side effects through connected clocks and their callbacks, this operation must be done while holding the qemu io lock.

For the same reason, one can update clocks only when it is allowed to have side effects on other objects. In consequence, it is forbidden:

- during migration,
- and in the enter phase of reset.

Note that calling `clock_update[_ns|_hz]()` is equivalent to calling `clock_set[_ns|_hz]()` (with the same arguments) then `clock_propagate()` on the clock. Thus, setting the clock value can be separated from triggering the side-effects. This is often required to factorize code to handle reset and migration in devices.

Aliasing clocks

Sometimes, one needs to forward, or inherit, a clock from another device. Typically, when doing device composition, a device might expose a sub-device's clock without interfering with it. The function `qdev_alias_clock()` can be used to achieve this behaviour. Note that it is possible to expose the clock under a different name. `qdev_alias_clock()` works for both input and output clocks.

For example, if device B is a child of device A, `device_a_instance_init()` may do something like this:

```
void device_a_instance_init(Object *obj)
{
    AState *A = DEVICE_A(obj);
    BState *B;
    /* create object B as child of A */
```

(continues on next page)

(continued from previous page)

```
    [...]
    qdev_alias_clock(B, "clk", A, "b_clk");
    /*
     * Now A has a clock "b_clk" which is an alias to
     * the clock "clk" of its child B.
     */
}
```

This function does not return any clock object. The new clock has the same direction (input or output) as the original one. This function only adds a link to the existing clock. In the above example, object B remains the only object allowed to use the clock and device A must not try to change the clock period or set a callback to the clock. This diagram describes the example with an input clock:

```
+--------------------------+
|        Device A          |
|         +--------------+ |
|         |   Device B   | |
|         | +-------+    | |
>>"b_clk">>>| "clk" |    | |
|  (in)   | |  (in) |    | |
|         | +-------+    | |
|         +--------------+ |
+--------------------------+
```

Migration

Clock state is not migrated automatically. Every device must handle its clock migration. Alias clocks must not be migrated.

To ensure clock states are restored correctly during migration, there are two solutions.

Clock states can be migrated by adding an entry into the device vmstate description. You should use the `VMSTATE_CLOCK` macro for this. This is typically used to migrate an input clock state. For example:

```
MyDeviceState {
    DeviceState parent_obj;
    [...] /* some fields */
    Clock *clk;
};

VMStateDescription my_device_vmstate = {
    .name = "my_device",
    .fields = (const VMStateField[]) {
        [...], /* other migrated fields */
        VMSTATE_CLOCK(clk, MyDeviceState),
        VMSTATE_END_OF_LIST()
    }
};
```

The second solution is to restore the clock state using information already at our disposal. This can be used to restore output clock states using the device state. The functions `clock_set[_ns|_hz]()` can be used during the `post_load()` migration callback.

When adding clock support to an existing device, if you care about migration compatibility you will need to be careful, as simply adding a `VMSTATE_CLOCK()` line will break compatibility. Instead, you can put the `VMSTATE_CLOCK()` line into a vmstate subsection with a suitable `needed` function, and use `clock_set()` in a `pre_load()` function to set the default value that will be used if the source virtual machine in the migration does not send the clock state.

Care should be taken not to use `clock_update[_ns|_hz]()` or `clock_propagate()` during the whole migration procedure because it will trigger side effects to other devices in an unknown state.

7.4.5 eBPF RSS virtio-net support

RSS(Receive Side Scaling) is used to distribute network packets to guest virtqueues by calculating packet hash. Usually every queue is processed then by a specific guest CPU core.

For now there are 2 RSS implementations in qemu: - 'in-qemu' RSS (functions if qemu receives network packets, i.e. vhost=off) - eBPF RSS (can function with also with vhost=on)

eBPF support (CONFIG_EBPF) is enabled by 'configure' script. To enable eBPF RSS support use './configure –enable-bpf'.

If steering BPF is not set for kernel's TUN module, the TUN uses automatic selection of rx virtqueue based on lookup table built according to calculated symmetric hash of transmitted packets. If steering BPF is set for TUN the BPF code calculates the hash of packet header and returns the virtqueue number to place the packet to.

Simplified decision formula:

```
queue_index = indirection_table[hash(<packet data>)%<indirection_table size>]
```

Not for all packets, the hash can/should be calculated.

Note: currently, eBPF RSS does not support hash reporting.

eBPF RSS turned on by different combinations of vhost-net, vitrio-net and tap configurations:

- eBPF is used:

  tap,vhost=off & virtio-net-pci,rss=on,hash=off

- eBPF is used:

  tap,vhost=on & virtio-net-pci,rss=on,hash=off

- 'in-qemu' RSS is used:

  tap,vhost=off & virtio-net-pci,rss=on,hash=on

- eBPF is used, hash population feature is not reported to the guest:

  tap,vhost=on & virtio-net-pci,rss=on,hash=on

If CONFIG_EBPF is not set then only 'in-qemu' RSS is supported. Also 'in-qemu' RSS, as a fallback, is used if the eBPF program failed to load or set to TUN.

RSS eBPF program

RSS program located in ebpf/rss.bpf.skeleton.h generated by bpftool. So the program is part of the qemu binary. Initially, the eBPF program was compiled by clang and source code located at tools/ebpf/rss.bpf.c. Prerequisites to recompile the eBPF program (regenerate ebpf/rss.bpf.skeleton.h):

> llvm, clang, kernel source tree, bpftool Adjust Makefile.ebpf to reflect the location of the kernel source tree
>
> $ cd tools/ebpf $ make -f Makefile.ebpf

Current eBPF RSS implementation uses 'bounded loops' with 'backward jump instructions' which present in the last kernels. Overall eBPF RSS works on kernels 5.8+.

eBPF RSS implementation

eBPF RSS loading functionality located in ebpf/ebpf_rss.c and ebpf/ebpf_rss.h.

The `struct EBPFRSSContext` structure that holds 4 file descriptors:

- ctx - pointer of the libbpf context.
- program_fd - file descriptor of the eBPF RSS program.
- map_configuration - file descriptor of the 'configuration' map. This map contains one element of 'struct EBPFRSSConfig'. This configuration determines eBPF program behavior.
- map_toeplitz_key - file descriptor of the 'Toeplitz key' map. One element of the 40byte key prepared for the hashing algorithm.
- map_indirections_table - 128 elements of queue indexes.

`struct EBPFRSSConfig` fields:

- redirect - "boolean" value, should the hash be calculated, on false - `default_queue` would be used as the final decision.
- populate_hash - for now, not used. eBPF RSS doesn't support hash reporting.
- hash_types - binary mask of different hash types. See `VIRTIO_NET_RSS_HASH_TYPE_*` defines. If for packet hash should not be calculated - `default_queue` would be used.
- indirections_len - length of the indirections table, maximum 128.
- default_queue - the queue index that used for packet that shouldn't be hashed. For some packets, the hash can't be calculated(g.e ARP).

Functions:

- `ebpf_rss_init()` - sets ctx to NULL, which indicates that EBPFRSSContext is not loaded.
- `ebpf_rss_load()` - creates 3 maps and loads eBPF program from the rss.bpf.skeleton.h. Returns 'true' on success. After that, program_fd can be used to set steering for TAP.
- `ebpf_rss_set_all()` - sets values for eBPF maps. `indirections_table` length is in EBPFRSSConfig. `toeplitz_key` is VIRTIO_NET_RSS_MAX_KEY_SIZE aka 40 bytes array.
- `ebpf_rss_unload()` - close all file descriptors and set ctx to NULL.

Simplified eBPF RSS workflow:

```
struct EBPFRSSConfig config;
config.redirect = 1;
config.hash_types = VIRTIO_NET_RSS_HASH_TYPE_UDPv4 | VIRTIO_NET_RSS_HASH_TYPE_TCPv4;
config.indirections_len = VIRTIO_NET_RSS_MAX_TABLE_LEN;
config.default_queue = 0;

uint16_t table[VIRTIO_NET_RSS_MAX_TABLE_LEN] = {...};
uint8_t key[VIRTIO_NET_RSS_MAX_KEY_SIZE] = {...};

struct EBPFRSSContext ctx;
ebpf_rss_init(&ctx);
ebpf_rss_load(&ctx);
ebpf_rss_set_all(&ctx, &config, table, key);
if (net_client->info->set_steering_ebpf != NULL) {
    net_client->info->set_steering_ebpf(net_client, ctx->program_fd);
}
...
ebpf_unload(&ctx);
```

NetClientState SetSteeringEBPF()

For now, `set_steering_ebpf()` method supported by Linux TAP NetClientState. The method requires an eBPF program file descriptor as an argument.

7.4.6 Migration

This is the main entry for QEMU migration documentations. It explains how QEMU live migration works.

Migration framework

QEMU has code to load/save the state of the guest that it is running. These are two complementary operations. Saving the state just does that, saves the state for each device that the guest is running. Restoring a guest is just the opposite operation: we need to load the state of each device.

For this to work, QEMU has to be launched with the same arguments the two times. I.e. it can only restore the state in one guest that has the same devices that the one it was saved (this last requirement can be relaxed a bit, but for now we can consider that configuration has to be exactly the same).

Once that we are able to save/restore a guest, a new functionality is requested: migration. This means that QEMU is able to start in one machine and being "migrated" to another machine. I.e. being moved to another machine.

Next was the "live migration" functionality. This is important because some guests run with a lot of state (specially RAM), and it can take a while to move all state from one machine to another. Live migration allows the guest to continue running while the state is transferred. Only while the last part of the state is transferred has the guest to be stopped. Typically the time that the guest is unresponsive during live migration is the low hundred of milliseconds (notice that this depends on a lot of things).

Contents

Transports

The migration stream is normally just a byte stream that can be passed over any transport.

- tcp migration: do the migration using tcp sockets
- unix migration: do the migration using unix sockets
- exec migration: do the migration using the stdin/stdout through a process.
- fd migration: do the migration using a file descriptor that is passed to QEMU. QEMU doesn't care how this file descriptor is opened.
- file migration: do the migration using a file that is passed to QEMU by path. A file offset option is supported to allow a management application to add its own metadata to the start of the file without QEMU interference. Note that QEMU does not flush cached file data/metadata at the end of migration.

In addition, support is included for migration using RDMA, which transports the page data using `RDMA`, where the hardware takes care of transporting the pages, and the load on the CPU is much lower. While the internals of RDMA migration are a bit different, this isn't really visible outside the RAM migration code.

All these migration protocols use the same infrastructure to save/restore state devices. This infrastructure is shared with the savevm/loadvm functionality.

Common infrastructure

The files, sockets or fd's that carry the migration stream are abstracted by the `QEMUFile` type (see `migration/qemu-file.h`). In most cases this is connected to a subtype of `QIOChannel` (see `io/`).

Saving the state of one device

For most devices, the state is saved in a single call to the migration infrastructure; these are *non-iterative* devices. The data for these devices is sent at the end of precopy migration, when the CPUs are paused. There are also *iterative* devices, which contain a very large amount of data (e.g. RAM or large tables). See the iterative device section below.

General advice for device developers

- The migration state saved should reflect the device being modelled rather than the way your implementation works. That way if you change the implementation later the migration stream will stay compatible. That model may include internal state that's not directly visible in a register.
- When saving a migration stream the device code may walk and check the state of the device. These checks might fail in various ways (e.g. discovering internal state is corrupt or that the guest has done something bad). Consider carefully before asserting/aborting at this point, since the normal response from users is that *migration broke their VM* since it had apparently been running fine until then. In these error cases, the device should log a message indicating the cause of error, and should consider putting the device into an error state, allowing the rest of the VM to continue execution.
- The migration might happen at an inconvenient point, e.g. right in the middle of the guest reprogramming the device, during guest reboot or shutdown or while the device is waiting for external IO. It's strongly preferred that migrations do not fail in this situation, since in the cloud environment migrations might happen automatically to VMs that the administrator doesn't directly control.
- If you do need to fail a migration, ensure that sufficient information is logged to identify what went wrong.
- The destination should treat an incoming migration stream as hostile (which we do to varying degrees in the existing code). Check that offsets into buffers and the like can't cause overruns. Fail the incoming migration in the case of a corrupted stream like this.
- Take care with internal device state or behaviour that might become migration version dependent. For example, the order of PCI capabilities is required to stay constant across migration. Another example would be that a special case handled by subsections (see below) might become much more common if a default behaviour is changed.
- The state of the source should not be changed or destroyed by the outgoing migration. Migrations timing out or being failed by higher levels of management, or failures of the destination host are not unusual, and in that case the VM is restarted on the source. Note that the management layer can validly revert the migration even though the QEMU level of migration has succeeded as long as it does it before starting execution on the destination.
- Buses and devices should be able to explicitly specify addresses when instantiated, and management tools should use those. For example, when hot adding USB devices it's important to specify the ports and addresses, since implicit ordering based on the command line order may be different on the destination. This can result in the device state being loaded into the wrong device.

VMState

Most device data can be described using the `VMSTATE` macros (mostly defined in `include/migration/vmstate.h`).

An example (from hw/input/pckbd.c)

```
static const VMStateDescription vmstate_kbd = {
    .name = "pckbd",
    .version_id = 3,
    .minimum_version_id = 3,
    .fields = (const VMStateField[]) {
        VMSTATE_UINT8(write_cmd, KBDState),
        VMSTATE_UINT8(status, KBDState),
        VMSTATE_UINT8(mode, KBDState),
        VMSTATE_UINT8(pending, KBDState),
        VMSTATE_END_OF_LIST()
    }
};
```

We are declaring the state with name "pckbd". The `version_id` is 3, and there are 4 uint8_t fields in the KBDState structure. We registered this `VMSTATEDescription` with one of the following functions. The first one will generate a device `instance_id` different for each registration. Use the second one if you already have an id that is different for each instance of the device:

```
vmstate_register_any(NULL, &vmstate_kbd, s);
vmstate_register(NULL, instance_id, &vmstate_kbd, s);
```

For devices that are `qdev` based, we can register the device in the class init function:

```
dc->vmsd = &vmstate_kbd_isa;
```

The VMState macros take care of ensuring that the device data section is formatted portably (normally big endian) and make some compile time checks against the types of the fields in the structures.

VMState macros can include other VMStateDescriptions to store substructures (see `VMSTATE_STRUCT_`), arrays (`VMSTATE_ARRAY_`) and variable length arrays (`VMSTATE_VARRAY_`). Various other macros exist for special cases.

Note that the format on the wire is still very raw; i.e. a VMSTATE_UINT32 ends up with a 4 byte bigendian representation on the wire; in the future it might be possible to use a more structured format.

Legacy way

This way is going to disappear as soon as all current users are ported to VMSTATE; although converting existing code can be tricky, and thus 'soon' is relative.

Each device has to register two functions, one to save the state and another to load the state back.

```
int register_savevm_live(const char *idstr,
                         int instance_id,
                         int version_id,
                         SaveVMHandlers *ops,
                         void *opaque);
```

Two functions in the `ops` structure are the `save_state` and `load_state` functions. Notice that `load_state` receives a version_id parameter to know what state format is receiving. `save_state` doesn't have a version_id parameter because it always uses the latest version.

Note that because the VMState macros still save the data in a raw format, in many cases it's possible to replace legacy code with a carefully constructed VMState description that matches the byte layout of the existing code.

Changing migration data structures

When we migrate a device, we save/load the state as a series of fields. Sometimes, due to bugs or new functionality, we need to change the state to store more/different information. Changing the migration state saved for a device can break migration compatibility unless care is taken to use the appropriate techniques. In general QEMU tries to maintain forward migration compatibility (i.e. migrating from QEMU n->n+1) and there are users who benefit from backward compatibility as well.

Subsections

The most common structure change is adding new data, e.g. when adding a newer form of device, or adding that state that you previously forgot to migrate. This is best solved using a subsection.

A subsection is "like" a device vmstate, but with a particularity, it has a Boolean function that tells if that values are needed to be sent or not. If this functions returns false, the subsection is not sent. Subsections have a unique name, that is looked for on the receiving side.

On the receiving side, if we found a subsection for a device that we don't understand, we just fail the migration. If we understand all the subsections, then we load the state with success. There's no check that a subsection is loaded, so a newer QEMU that knows about a subsection can (with care) load a stream from an older QEMU that didn't send the subsection.

If the new data is only needed in a rare case, then the subsection can be made conditional on that case and the migration will still succeed to older QEMUs in most cases. This is OK for data that's critical, but in some use cases it's preferred that the migration should succeed even with the data missing. To support this the subsection can be connected to a device property and from there to a versioned machine type.

The 'pre_load' and 'post_load' functions on subsections are only called if the subsection is loaded.

One important note is that the outer post_load() function is called "after" loading all subsections, because a newer subsection could change the same value that it uses. A flag, and the combination of outer pre_load and post_load can be used to detect whether a subsection was loaded, and to fall back on default behaviour when the subsection isn't present.

Example:

```
static bool ide_drive_pio_state_needed(void *opaque)
{
    IDEState *s = opaque;

    return ((s->status & DRQ_STAT) != 0)
        || (s->bus->error_status & BM_STATUS_PIO_RETRY);
}

const VMStateDescription vmstate_ide_drive_pio_state = {
    .name = "ide_drive/pio_state",
    .version_id = 1,
    .minimum_version_id = 1,
    .pre_save = ide_drive_pio_pre_save,
    .post_load = ide_drive_pio_post_load,
    .needed = ide_drive_pio_state_needed,
```

(continues on next page)

(continued from previous page)

```
    .fields = (const VMStateField[]) {
        VMSTATE_INT32(req_nb_sectors, IDEState),
        VMSTATE_VARRAY_INT32(io_buffer, IDEState, io_buffer_total_len, 1,
                             vmstate_info_uint8, uint8_t),
        VMSTATE_INT32(cur_io_buffer_offset, IDEState),
        VMSTATE_INT32(cur_io_buffer_len, IDEState),
        VMSTATE_UINT8(end_transfer_fn_idx, IDEState),
        VMSTATE_INT32(elementary_transfer_size, IDEState),
        VMSTATE_INT32(packet_transfer_size, IDEState),
        VMSTATE_END_OF_LIST()
    }
};

const VMStateDescription vmstate_ide_drive = {
    .name = "ide_drive",
    .version_id = 3,
    .minimum_version_id = 0,
    .post_load = ide_drive_post_load,
    .fields = (const VMStateField[]) {
        .... several fields ....
        VMSTATE_END_OF_LIST()
    },
    .subsections = (const VMStateDescription * const []) {
        &vmstate_ide_drive_pio_state,
        NULL
    }
};
```

Here we have a subsection for the pio state. We only need to save/send this state when we are in the middle of a pio operation (that is what `ide_drive_pio_state_needed()` checks). If DRQ_STAT is not enabled, the values on that fields are garbage and don't need to be sent.

Connecting subsections to properties

Using a condition function that checks a 'property' to determine whether to send a subsection allows backward migration compatibility when new subsections are added, especially when combined with versioned machine types.

For example:

a) Add a new property using `DEFINE_PROP_BOOL` - e.g. support-foo and default it to true.

b) Add an entry to the `hw_compat_` for the previous version that sets the property to false.

c) Add a static bool support_foo function that tests the property.

d) Add a subsection with a .needed set to the support_foo function

e) (potentially) Add an outer pre_load that sets up a default value for 'foo' to be used if the subsection isn't loaded.

Now that subsection will not be generated when using an older machine type and the migration stream will be accepted by older QEMU versions.

Not sending existing elements

Sometimes members of the VMState are no longer needed:

- removing them will break migration compatibility
- making them version dependent and bumping the version will break backward migration compatibility.

Adding a dummy field into the migration stream is normally the best way to preserve compatibility.

If the field really does need to be removed then:

a) Add a new property/compatibility/function in the same way for subsections above.

b) replace the VMSTATE macro with the _TEST version of the macro, e.g.:

`VMSTATE_UINT32(foo, barstruct)`

becomes

`VMSTATE_UINT32_TEST(foo, barstruct, pre_version_baz)`

Sometime in the future when we no longer care about the ancient versions these can be killed off. Note that for backward compatibility it's important to fill in the structure with data that the destination will understand.

Any difference in the predicates on the source and destination will end up with different fields being enabled and data being loaded into the wrong fields; for this reason conditional fields like this are very fragile.

Versions

Version numbers are intended for major incompatible changes to the migration of a device, and using them breaks backward-migration compatibility; in general most changes can be made by adding Subsections (see above) or _TEST macros (see above) which won't break compatibility.

Each version is associated with a series of fields saved. The `save_state` always saves the state as the newer version. But `load_state` sometimes is able to load state from an older version.

You can see that there are two version fields:

- `version_id`: the maximum version_id supported by VMState for that device.
- `minimum_version_id`: the minimum version_id that VMState is able to understand for that device.

VMState is able to read versions from minimum_version_id to version_id.

There are *_V* forms of many `VMSTATE_` macros to load fields for version dependent fields, e.g.

```
VMSTATE_UINT16_V(ip_id, Slirp, 2),
```

only loads that field for versions 2 and newer.

Saving state will always create a section with the 'version_id' value and thus can't be loaded by any older QEMU.

Massaging functions

Sometimes, it is not enough to be able to save the state directly from one structure, we need to fill the correct values there. One example is when we are using kvm. Before saving the cpu state, we need to ask kvm to copy to QEMU the state that it is using. And the opposite when we are loading the state, we need a way to tell kvm to load the state for the cpu that we have just loaded from the QEMUFile.

The functions to do that are inside a vmstate definition, and are called:

- `int (*pre_load)(void *opaque);`

  This function is called before we load the state of one device.

- `int (*post_load)(void *opaque, int version_id);`

  This function is called after we load the state of one device.

- `int (*pre_save)(void *opaque);`

  This function is called before we save the state of one device.

- `int (*post_save)(void *opaque);`

  This function is called after we save the state of one device (even upon failure, unless the call to pre_save returned an error).

Example: You can look at hpet.c, that uses the first three functions to massage the state that is transferred.

The `VMSTATE_WITH_TMP` macro may be useful when the migration data doesn't match the stored device data well; it allows an intermediate temporary structure to be populated with migration data and then transferred to the main structure.

If you use memory or portio_list API functions that update memory layout outside initialization (i.e., in response to a guest action), this is a strong indication that you need to call these functions in a `post_load` callback. Examples of such API functions are:

- memory_region_add_subregion()
- memory_region_del_subregion()
- memory_region_set_readonly()
- memory_region_set_nonvolatile()
- memory_region_set_enabled()
- memory_region_set_address()
- memory_region_set_alias_offset()
- portio_list_set_address()
- portio_list_set_enabled()

Iterative device migration

Some devices, such as RAM or certain platform devices, have large amounts of data that would mean that the CPUs would be paused for too long if they were sent in one section. For these devices an *iterative* approach is taken.

The iterative devices generally don't use VMState macros (although it may be possible in some cases) and instead use qemu_put_*/qemu_get_* macros to read/write data to the stream. Specialist versions exist for high bandwidth IO.

An iterative device must provide:

- A `save_setup` function that initialises the data structures and transmits a first section containing information on the device. In the case of RAM this transmits a list of RAMBlocks and sizes.
- A `load_setup` function that initialises the data structures on the destination.
- A `state_pending_exact` function that indicates how much more data we must save. The core migration code will use this to determine when to pause the CPUs and complete the migration.
- A `state_pending_estimate` function that indicates how much more data we must save. When the estimated amount is smaller than the threshold, we call `state_pending_exact`.
- A `save_live_iterate` function should send a chunk of data until the point that stream bandwidth limits tell it to stop. Each call generates one section.
- A `save_live_complete_precopy` function that must transmit the last section for the device containing any remaining data.
- A `load_state` function used to load sections generated by any of the save functions that generate sections.
- `cleanup` functions for both save and load that are called at the end of migration.

Note that the contents of the sections for iterative migration tend to be open-coded by the devices; care should be taken in parsing the results and structuring the stream to make them easy to validate.

Device ordering

There are cases in which the ordering of device loading matters; for example in some systems where a device may assert an interrupt during loading, if the interrupt controller is loaded later then it might lose the state.

Some ordering is implicitly provided by the order in which the machine definition creates devices, however this is somewhat fragile.

The `MigrationPriority` enum provides a means of explicitly enforcing ordering. Numerically higher priorities are loaded earlier. The priority is set by setting the `priority` field of the top level `VMStateDescription` for the device.

Stream structure

The stream tries to be word and endian agnostic, allowing migration between hosts of different characteristics running the same VM.

- Header
  - Magic
  - Version
  - VM configuration section
    - Machine type
    - Target page bits

- List of sections Each section contains a device, or one iteration of a device save.
  - section type
  - section id
  - ID string (First section of each device)
  - instance id (First section of each device)
  - version id (First section of each device)
  - <device data>
  - Footer mark
- EOF mark
- VM Description structure Consisting of a JSON description of the contents for analysis only

The `device data` in each section consists of the data produced by the code described above. For non-iterative devices they have a single section; iterative devices have an initial and last section and a set of parts in between. Note that there is very little checking by the common code of the integrity of the `device data` contents, that's up to the devices themselves. The `footer mark` provides a little bit of protection for the case where the receiving side reads more or less data than expected.

The `ID string` is normally unique, having been formed from a bus name and device address, PCI devices and storage devices hung off PCI controllers fit this pattern well. Some devices are fixed single instances (e.g. "pc-ram"). Others (especially either older devices or system devices which for some reason don't have a bus concept) make use of the `instance id` for otherwise identically named devices.

Return path

Only a unidirectional stream is required for normal migration, however a `return path` can be created when bidirectional communication is desired. This is primarily used by postcopy, but is also used to return a success flag to the source at the end of migration.

`qemu_file_get_return_path(QEMUFile* fwdpath)` gives the QEMUFile* for the return path.

Source side

Forward path - written by migration thread Return path - opened by main thread, read by return-path thread

Destination side

Forward path - read by main thread Return path - opened by main thread, written by main thread AND postcopy thread (protected by rp_mutex)

Migration features

Migration has plenty of features to support different use cases.

Postcopy

Contents

'Postcopy' migration is a way to deal with migrations that refuse to converge (or take too long to converge) its plus side is that there is an upper bound on the amount of migration traffic and time it takes, the down side is that during the postcopy phase, a failure of *either* side causes the guest to be lost.

In postcopy the destination CPUs are started before all the memory has been transferred, and accesses to pages that are yet to be transferred cause a fault that's translated by QEMU into a request to the source QEMU.

Postcopy can be combined with precopy (i.e. normal migration) so that if precopy doesn't finish in a given time the switch is made to postcopy.

Enabling postcopy

To enable postcopy, issue this command on the monitor (both source and destination) prior to the start of migration:

```
migrate_set_capability postcopy-ram on
```

The normal commands are then used to start a migration, which is still started in precopy mode. Issuing:

```
migrate_start_postcopy
```

will now cause the transition from precopy to postcopy. It can be issued immediately after migration is started or any time later on. Issuing it after the end of a migration is harmless.

Blocktime is a postcopy live migration metric, intended to show how long the vCPU was in state of interruptible sleep due to pagefault. That metric is calculated both for all vCPUs as overlapped value, and separately for each vCPU. These values are calculated on destination side. To enable postcopy blocktime calculation, enter following command on destination monitor:

```
migrate_set_capability postcopy-blocktime on
```

Postcopy blocktime can be retrieved by query-migrate qmp command. postcopy-blocktime value of qmp command will show overlapped blocking time for all vCPU, postcopy-vcpu-blocktime will show list of blocking time per vCPU.

Note: During the postcopy phase, the bandwidth limits set using `migrate_set_parameter` is ignored (to avoid delaying requested pages that the destination is waiting for).

Postcopy internals

State machine

Postcopy moves through a series of states (see postcopy_state) from ADVISE->DISCARD->LISTEN->RUNNING->END

- Advise

  Set at the start of migration if postcopy is enabled, even if it hasn't had the start command; here the destination checks that its OS has the support needed for postcopy, and performs setup to ensure the RAM mappings are suitable for later postcopy. The destination will fail early in migration at this point if the required OS support is not present. (Triggered by reception of POSTCOPY_ADVISE command)

- Discard

  Entered on receipt of the first 'discard' command; prior to the first Discard being performed, hugepages are switched off (using madvise) to ensure that no new huge pages are created during the postcopy phase, and to cause any huge pages that have discards on them to be broken.

- Listen

  The first command in the package, POSTCOPY_LISTEN, switches the destination state to Listen, and starts a new thread (the 'listen thread') which takes over the job of receiving pages off the migration stream, while the main thread carries on processing the blob. With this thread able to process page reception, the destination now 'sensitises' the RAM to detect any access to missing pages (on Linux using the 'userfault' system).

- Running

  POSTCOPY_RUN causes the destination to synchronise all state and start the CPUs and IO devices running. The main thread now finishes processing the migration package and now carries on as it would for normal precopy migration (although it can't do the cleanup it would do as it finishes a normal migration).

- Paused

  Postcopy can run into a paused state (normally on both sides when happens), where all threads will be temporarily halted mostly due to network errors. When reaching paused state, migration will make sure the qemu binary on both sides maintain the data without corrupting the VM. To continue the migration, the admin needs to fix the migration channel using the QMP command 'migrate-recover' on the destination node, then resume the migration using QMP command 'migrate' again on source node, with resume=true flag set.

- End

  The listen thread can now quit, and perform the cleanup of migration state, the migration is now complete.

Device transfer

Loading of device data may cause the device emulation to access guest RAM that may trigger faults that have to be resolved by the source, as such the migration stream has to be able to respond with page data *during* the device load, and hence the device data has to be read from the stream completely before the device load begins to free the stream up. This is achieved by 'packaging' the device data into a blob that's read in one go.

Source behaviour

Until postcopy is entered the migration stream is identical to normal precopy, except for the addition of a 'postcopy advise' command at the beginning, to tell the destination that postcopy might happen. When postcopy starts the source sends the page discard data and then forms the 'package' containing:

- Command: 'postcopy listen'
- The device state

  A series of sections, identical to the precopy streams device state stream containing everything except postcopiable devices (i.e. RAM)

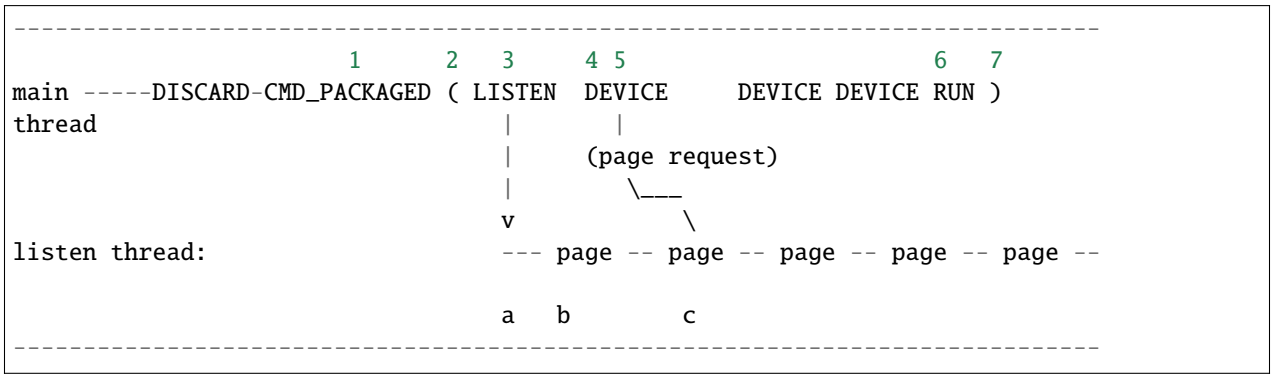
- Command: 'postcopy run'

The 'package' is sent as the data part of a Command: `CMD_PACKAGED`, and the contents are formatted in the same way as the main migration stream.

During postcopy the source scans the list of dirty pages and sends them to the destination without being requested (in much the same way as precopy), however when a page request is received from the destination, the dirty page scanning restarts from the requested location. This causes requested pages to be sent quickly, and also causes pages directly after the requested page to be sent quickly in the hope that those pages are likely to be used by the destination soon.

Destination behaviour

Initially the destination looks the same as precopy, with a single thread reading the migration stream; the 'postcopy advise' and 'discard' commands are processed to change the way RAM is managed, but don't affect the stream processing.



- On receipt of `CMD_PACKAGED` (1)

  All the data associated with the package - the (...) section in the diagram - is read into memory, and the main thread recurses into qemu_loadvm_state_main to process the contents of the package (2) which contains commands (3,6) and devices (4...)

- On receipt of 'postcopy listen' - 3 -(i.e. the 1st command in the package)

a new thread (a) is started that takes over servicing the migration stream, while the main thread carries on loading the package. It loads normal background page data (b) but if during a device load a fault happens (5) the returned page (c) is loaded by the listen thread allowing the main threads device load to carry on.

- The last thing in the `CMD_PACKAGED` is a 'RUN' command (6)

  letting the destination CPUs start running. At the end of the `CMD_PACKAGED` (7) the main thread returns to normal running behaviour and is no longer used by migration, while the listen thread carries on servicing page data until the end of migration.

Source side page bitmap

The 'migration bitmap' in postcopy is basically the same as in the precopy, where each of the bit to indicate that page is 'dirty' - i.e. needs sending. During the precopy phase this is updated as the CPU dirties pages, however during postcopy the CPUs are stopped and nothing should dirty anything any more. Instead, dirty bits are cleared when the relevant pages are sent during postcopy.

Postcopy features

Postcopy recovery

Comparing to precopy, postcopy is special on error handlings. When any error happens (in this case, mostly network errors), QEMU cannot easily fail a migration because VM data resides in both source and destination QEMU instances. On the other hand, when issue happens QEMU on both sides will go into a paused state. It'll need a recovery phase to continue a paused postcopy migration.

The recovery phase normally contains a few steps:

- When network issue occurs, both QEMU will go into PAUSED state
- When the network is recovered (or a new network is provided), the admin can setup the new channel for migration using QMP command 'migrate-recover' on destination node, preparing for a resume.
- On source host, the admin can continue the interrupted postcopy migration using QMP command 'migrate' with resume=true flag set.
- After the connection is re-established, QEMU will continue the postcopy migration on both sides.

During a paused postcopy migration, the VM can logically still continue running, and it will not be impacted from any page access to pages that were already migrated to destination VM before the interruption happens. However, if any of the missing pages got accessed on destination VM, the VM thread will be halted waiting for the page to be migrated, it means it can be halted until the recovery is complete.

The impact of accessing missing pages can be relevant to different configurations of the guest. For example, when with async page fault enabled, logically the guest can proactively schedule out the threads accessing missing pages.

Postcopy with hugepages

Postcopy now works with hugetlbfs backed memory:

a) The linux kernel on the destination must support userfault on hugepages.

b) The huge-page configuration on the source and destination VMs must be identical; i.e. RAMBlocks on both sides must use the same page size.

c) Note that `-mem-path /dev/hugepages` will fall back to allocating normal RAM if it doesn't have enough hugepages, triggering (b) to fail. Using `-mem-prealloc` enforces the allocation using hugepages.

d) Care should be taken with the size of hugepage used; postcopy with 2MB hugepages works well, however 1GB hugepages are likely to be problematic since it takes ~1 second to transfer a 1GB hugepage across a 10Gbps link, and until the full page is transferred the destination thread is blocked.

Postcopy with shared memory

Postcopy migration with shared memory needs explicit support from the other processes that share memory and from QEMU. There are restrictions on the type of memory that userfault can support shared.

The Linux kernel userfault support works on `/dev/shm` memory and on `hugetlbfs` (although the kernel doesn't provide an equivalent to `madvise(MADV_DONTNEED)` for hugetlbfs which may be a problem in some configurations).

The vhost-user code in QEMU supports clients that have Postcopy support, and the `vhost-user-bridge` (in `tests/`) and the DPDK package have changes to support postcopy.

The client needs to open a userfaultfd and register the areas of memory that it maps with userfault. The client must then pass the userfaultfd back to QEMU together with a mapping table that allows fault addresses in the clients address space to be converted back to RAMBlock/offsets. The client's userfaultfd is added to the postcopy fault-thread and page requests are made on behalf of the client by QEMU. QEMU performs 'wake' operations on the client's userfaultfd to allow it to continue after a page has arrived.

Note:

There are two future improvements that would be nice:

a) Some way to make QEMU ignorant of the addresses in the clients address space

b) Avoiding the need for QEMU to perform ufd-wake calls after the pages have arrived

---

Retro-fitting postcopy to existing clients is possible:

a) A mechanism is needed for the registration with userfault as above, and the registration needs to be coordinated with the phases of postcopy. In vhost-user extra messages are added to the existing control channel.

b) Any thread that can block due to guest memory accesses must be identified and the implication understood; for example if the guest memory access is made while holding a lock then all other threads waiting for that lock will also be blocked.

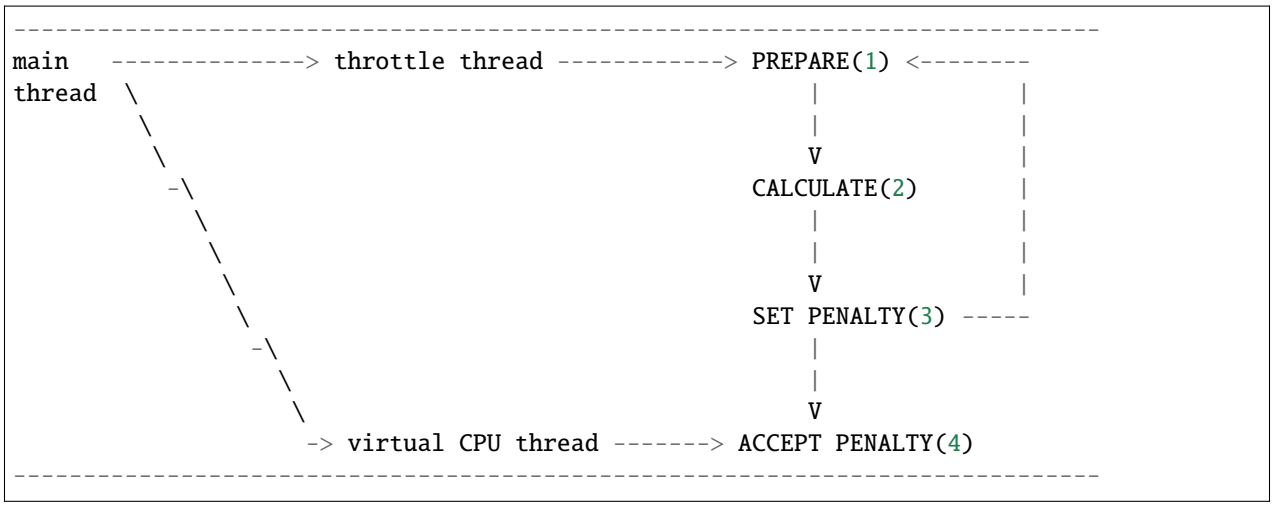
Postcopy preemption mode

Postcopy preempt is a new capability introduced in 8.0 QEMU release, it allows urgent pages (those got page fault requested from destination QEMU explicitly) to be sent in a separate preempt channel, rather than queued in the background migration channel. Anyone who cares about latencies of page faults during a postcopy migration should enable this feature. By default, it's not enabled.

Dirty limit

The dirty limit, short for dirty page rate upper limit, is a new capability introduced in the 8.1 QEMU release that uses a new algorithm based on the KVM dirty ring to throttle down the guest during live migration.

The algorithm framework is as follows:



When the qmp command qmp_set_vcpu_dirty_limit is called for the first time, the QEMU main thread starts the throttle thread. The throttle thread, once launched, executes the loop, which consists of three steps:

- PREPARE (1)

  The entire work of PREPARE (1) is preparation for the second stage, CALCULATE(2), as the name implies. It involves preparing the dirty page rate value and the corresponding upper limit of the VM: The dirty page rate is calculated via the KVM dirty ring mechanism, which tells QEMU how many dirty pages a virtual CPU has had since the last KVM_EXIT_DIRTY_RING_FULL exception; The dirty page rate upper limit is specified by caller, therefore fetch it directly.

- CALCULATE (2)

  Calculate a suitable sleep period for each virtual CPU, which will be used to determine the penalty for the target virtual CPU. The computation must be done carefully in order to reduce the dirty page rate progressively down to the upper limit without oscillation. To achieve this, two strategies are provided: the first is to add or subtract sleep time based on the ratio of the current dirty page rate to the limit, which is used when the current dirty page rate is far from the limit; the second is to add or subtract a fixed time when the current dirty page rate is close to the limit.

- SET PENALTY (3)

  Set the sleep time for each virtual CPU that should be penalized based on the results of the calculation supplied by step CALCULATE (2).

After completing the three above stages, the throttle thread loops back to step PREPARE (1) until the dirty limit is reached.

On the other hand, each virtual CPU thread reads the sleep duration and sleeps in the path of the KVM_EXIT_DIRTY_RING_FULL exception handler, that is ACCEPT PENALTY (4). Virtual CPUs tied with writing processes will obviously exit to the path and get penalized, whereas virtual CPUs involved with read processes will not.

In summary, thanks to the KVM dirty ring technology, the dirty limit algorithm will restrict virtual CPUs as needed to keep their dirty page rate inside the limit. This leads to more steady reading performance during live migration and can aid in improving large guest responsiveness.

VFIO device migration

Migration of virtual machine involves saving the state for each device that the guest is running on source host and restoring this saved state on the destination host. This document details how saving and restoring of VFIO devices is done in QEMU.

Migration of VFIO devices consists of two phases: the optional pre-copy phase, and the stop-and-copy phase. The pre-copy phase is iterative and allows to accommodate VFIO devices that have a large amount of data that needs to be transferred. The iterative pre-copy phase of migration allows for the guest to continue whilst the VFIO device state is transferred to the destination, this helps to reduce the total downtime of the VM. VFIO devices opt-in to pre-copy support by reporting the VFIO_MIGRATION_PRE_COPY flag in the VFIO_DEVICE_FEATURE_MIGRATION ioctl.

When pre-copy is supported, it's possible to further reduce downtime by enabling "switchover-ack" migration capability. VFIO migration uAPI defines "initial bytes" as part of its pre-copy data stream and recommends that the initial bytes are sent and loaded in the destination before stopping the source VM. Enabling this migration capability will guarantee that and thus, can potentially reduce downtime even further.

To support migration of multiple devices that might do P2P transactions between themselves, VFIO migration uAPI defines an intermediate P2P quiescent state. While in the P2P quiescent state, P2P DMA transactions cannot be initiated by the device, but the device can respond to incoming ones. Additionally, all outstanding P2P transactions are guaranteed to have been completed by the time the device enters this state.

All the devices that support P2P migration are first transitioned to the P2P quiescent state and only then are they stopped or started. This makes migration safe P2P-wise, since starting and stopping the devices is not done atomically for all the devices together.

Thus, multiple VFIO devices migration is allowed only if all the devices support P2P migration. Single VFIO device migration is allowed regardless of P2P migration support.

A detailed description of the UAPI for VFIO device migration can be found in the comment for the `vfio_device_mig_state` structure in the header file linux-headers/linux/vfio.h.

VFIO implements the device hooks for the iterative approach as follows:

- A `save_setup` function that sets up migration on the source.
- A `load_setup` function that sets the VFIO device on the destination in _RESUMING state.
- A `state_pending_estimate` function that reports an estimate of the remaining pre-copy data that the vendor driver has yet to save for the VFIO device.
- A `state_pending_exact` function that reads pending_bytes from the vendor driver, which indicates the amount of data that the vendor driver has yet to save for the VFIO device.
- An `is_active_iterate` function that indicates `save_live_iterate` is active only when the VFIO device is in pre-copy states.
- A `save_live_iterate` function that reads the VFIO device's data from the vendor driver during iterative pre-copy phase.

- A `switchover_ack_needed` function that checks if the VFIO device uses "switchover-ack" migration capability when this capability is enabled.
- A `save_state` function to save the device config space if it is present.
- A `save_live_complete_precopy` function that sets the VFIO device in _STOP_COPY state and iteratively copies the data for the VFIO device until the vendor driver indicates that no data remains.
- A `load_state` function that loads the config section and the data sections that are generated by the save functions above.
- `cleanup` functions for both save and load that perform any migration related cleanup.

The VFIO migration code uses a VM state change handler to change the VFIO device state when the VM state changes from running to not-running, and vice versa.

Similarly, a migration state change handler is used to trigger a transition of the VFIO device state when certain changes of the migration state occur. For example, the VFIO device state is transitioned back to _RUNNING in case a migration failed or was canceled.

System memory dirty pages tracking

A `log_global_start` and `log_global_stop` memory listener callback informs the VFIO dirty tracking module to start and stop dirty page tracking. A `log_sync` memory listener callback queries the dirty page bitmap from the dirty tracking module and marks system memory pages which were DMA-ed by the VFIO device as dirty. The dirty page bitmap is queried per container.

Currently there are two ways dirty page tracking can be done: (1) Device dirty tracking: In this method the device is responsible to log and report its DMAs. This method can be used only if the device is capable of tracking its DMAs. Discovering device capability, starting and stopping dirty tracking, and syncing the dirty bitmaps from the device are done using the DMA logging uAPI. More info about the uAPI can be found in the comments of the `vfio_device_feature_dma_logging_control` and `vfio_device_feature_dma_logging_report` structures in the header file linux-headers/linux/vfio.h.

(2) VFIO IOMMU module: In this method dirty tracking is done by IOMMU. However, there is currently no IOMMU support for dirty page tracking. For this reason, all pages are perpetually marked dirty, unless the device driver pins pages through external APIs in which case only those pinned pages are perpetually marked dirty.

If the above two methods are not supported, all pages are perpetually marked dirty by QEMU.

By default, dirty pages are tracked during pre-copy as well as stop-and-copy phase. So, a page marked as dirty will be copied to the destination in both phases. Copying dirty pages in pre-copy phase helps QEMU to predict if it can achieve its downtime tolerances. If QEMU during pre-copy phase keeps finding dirty pages continuously, then it understands that even in stop-and-copy phase, it is likely to find dirty pages and can predict the downtime accordingly.

QEMU also provides a per device opt-out option `pre-copy-dirty-page-tracking` which disables querying the dirty bitmap during pre-copy phase. If it is set to off, all dirty pages will be copied to the destination in stop-and-copy phase only.

System memory dirty pages tracking when vIOMMU is enabled

With vIOMMU, an IO virtual address range can get unmapped while in pre-copy phase of migration. In that case, the unmap ioctl returns any dirty pages in that range and QEMU reports corresponding guest physical pages dirty. During stop-and-copy phase, an IOMMU notifier is used to get a callback for mapped pages and then dirty pages bitmap is fetched from VFIO IOMMU modules for those mapped ranges. If device dirty tracking is enabled with vIOMMU, live migration will be blocked.

Flow of state changes during Live migration

Below is the state change flow during live migration for a VFIO device that supports both precopy and P2P migration. The flow for devices that don't support it is similar, except that the relevant states for precopy and P2P are skipped. The values in the parentheses represent the VM state, the migration state, and the VFIO device state, respectively.

Live migration save path

```
                        QEMU normal running state
                        (RUNNING, _NONE, _RUNNING)
                                    |
                    migrate_init spawns migration_thread
          Migration thread then calls each device's .save_setup()
                        (RUNNING, _SETUP, _PRE_COPY)
                                    |
                       (RUNNING, _ACTIVE, _PRE_COPY)
If device is active, get pending_bytes by .state_pending_{estimate,exact}()
     If total pending_bytes >= threshold_size, call .save_live_iterate()
              Data of VFIO device for pre-copy phase is copied
    Iterate till total pending bytes converge and are less than threshold
                                    |
     On migration completion, the vCPUs and the VFIO device are stopped
            The VFIO device is first put in P2P quiescent state
                  (FINISH_MIGRATE, _ACTIVE, _PRE_COPY_P2P)
                                    |
              Then the VFIO device is put in _STOP_COPY state
                   (FINISH_MIGRATE, _ACTIVE, _STOP_COPY)
       .save_live_complete_precopy() is called for each active device
    For the VFIO device, iterate in .save_live_complete_precopy() until
                             pending data is 0
                                    |
                    (POSTMIGRATE, _COMPLETED, _STOP_COPY)
          Migraton thread schedules cleanup bottom half and exits
                                    |
                         .save_cleanup() is called
                      (POSTMIGRATE, _COMPLETED, _STOP)
```

Live migration resume path

```
          Incoming migration calls .load_setup() for each device
                      (RESTORE_VM, _ACTIVE, _STOP)
                                  |
   For each device, .load_state() is called for that device section data
                     (RESTORE_VM, _ACTIVE, _RESUMING)
                                  |
At the end, .load_cleanup() is called for each device and vCPUs are started
            The VFIO device is first put in P2P quiescent state
                     (RUNNING, _ACTIVE, _RUNNING_P2P)
                                  |
                        (RUNNING, _NONE, _RUNNING)
```

Postcopy

Postcopy migration is currently not supported for VFIO devices.

Virtio device migration

Copyright 2015 IBM Corp.

This work is licensed under the terms of the GNU GPL, version 2 or later. See the COPYING file in the top-level directory.

Saving and restoring the state of virtio devices is a bit of a twisty maze, for several reasons:

- state is distributed between several parts:
  - virtio core, for common fields like features, number of queues, ...
  - virtio transport (pci, ccw, ...), for the different proxy devices and transport specific state (msix vectors, indicators, ...)
  - virtio device (net, blk, ...), for the different device types and their state (mac address, request queue, ...)
- most fields are saved via the stream interface; subsequently, subsections have been added to make cross-version migration possible

This file attempts to document the current procedure and point out some caveats.

Save state procedure

```
virtio core               virtio transport          virtio device
-----------               ----------------          -------------

                                                    save() function registered
                                                    via VMState wrapper on
                                                    device class
virtio_save()                                       <----------
             ------>      save_config()
                          - save proxy device
                          - save transport-specific
```

(continues on next page)

(continued from previous page)

```
                            device fields
- save common device
  fields
- save common virtqueue
  fields
              ------>       save_queue()
                            - save transport-specific
                              virtqueue fields
              ------>                                 save_device()
                                                      - save device-specific
                                                        fields
- save subsections
  - device endianness,
    if changed from
    default endianness
  - 64 bit features, if
    any high feature bit
    is set
  - virtio-1 virtqueue
    fields, if VERSION_1
    is set
```

Load state procedure

```
virtio core               virtio transport          virtio device
-----------               ----------------          -------------

                                                    load() function registered
                                                    via VMState wrapper on
                                                    device class
virtio_load()                                       <----------
              ------>     load_config()
                          - load proxy device
                          - load transport-specific
                            device fields
- load common device
  fields
- load common virtqueue
  fields
              ------>     load_queue()
                          - load transport-specific
                            virtqueue fields
- notify guest
              ------>                               load_device()
                                                    - load device-specific
                                                      fields
- load subsections
  - device endianness
  - 64 bit features
  - virtio-1 virtqueue
```

(continues on next page)

(continued from previous page)

```
    fields
- sanitize endianness
- sanitize features
- virtqueue index sanity
  check
                                                    - feature-dependent setup
```

Implications of this setup

Devices need to be careful in their state processing during load: The load_device() procedure is invoked by the core before subsections have been loaded. Any code that depends on information transmitted in subsections therefore has to be invoked in the device's load() function _after_ virtio_load() returned (like e.g. code depending on features).

Any extension of the state being migrated should be done in subsections added to the core for compatibility reasons. If transport or device specific state is added, core needs to invoke a callback from the new subsection.

Mapped-ram

Mapped-ram is a new stream format for the RAM section designed to supplement the existing `file:` migration and make it compatible with `multifd`. This enables parallel migration of a guest's RAM to a file.

The core of the feature is to ensure that RAM pages are mapped directly to offsets in the resulting migration file. This enables the `multifd` threads to write exclusively to those offsets even if the guest is constantly dirtying pages (i.e. live migration). Another benefit is that the resulting file will have a bounded size, since pages which are dirtied multiple times will always go to a fixed location in the file, rather than constantly being added to a sequential stream. Having the pages at fixed offsets also allows the usage of O_DIRECT for save/restore of the migration stream as the pages are ensured to be written respecting O_DIRECT alignment restrictions (direct-io support not yet implemented).

Usage

On both source and destination, enable the `multifd` and `mapped-ram` capabilities:

```
migrate_set_capability multifd on

migrate_set_capability mapped-ram on
```

Use a `file:` URL for migration:

```
migrate file:/path/to/migration/file
```

Mapped-ram migration is best done non-live, i.e. by stopping the VM on the source side before migrating.

Use-cases

The mapped-ram feature was designed for use cases where the migration stream will be directed to a file in the filesystem and not immediately restored on the destination VM[1]. These could be thought of as snapshots. We can further categorize them into live and non-live.

- Non-live snapshot

[1] While this same effect could be obtained with the usage of snapshots or the `file:` migration alone, mapped-ram provides a performance increase for VMs with larger RAM sizes (10s to 100s of GiBs), specially if the VM has been stopped beforehand.

If the use case requires a VM to be stopped before taking a snapshot, that's the ideal scenario for mapped-ram migration. Not having to track dirty pages, the migration will write the RAM pages to the disk as fast as it can.

Note: if a snapshot is taken of a running VM, but the VM will be stopped after the snapshot by the admin, then consider stopping it right before the snapshot to take benefit of the performance gains mentioned above.

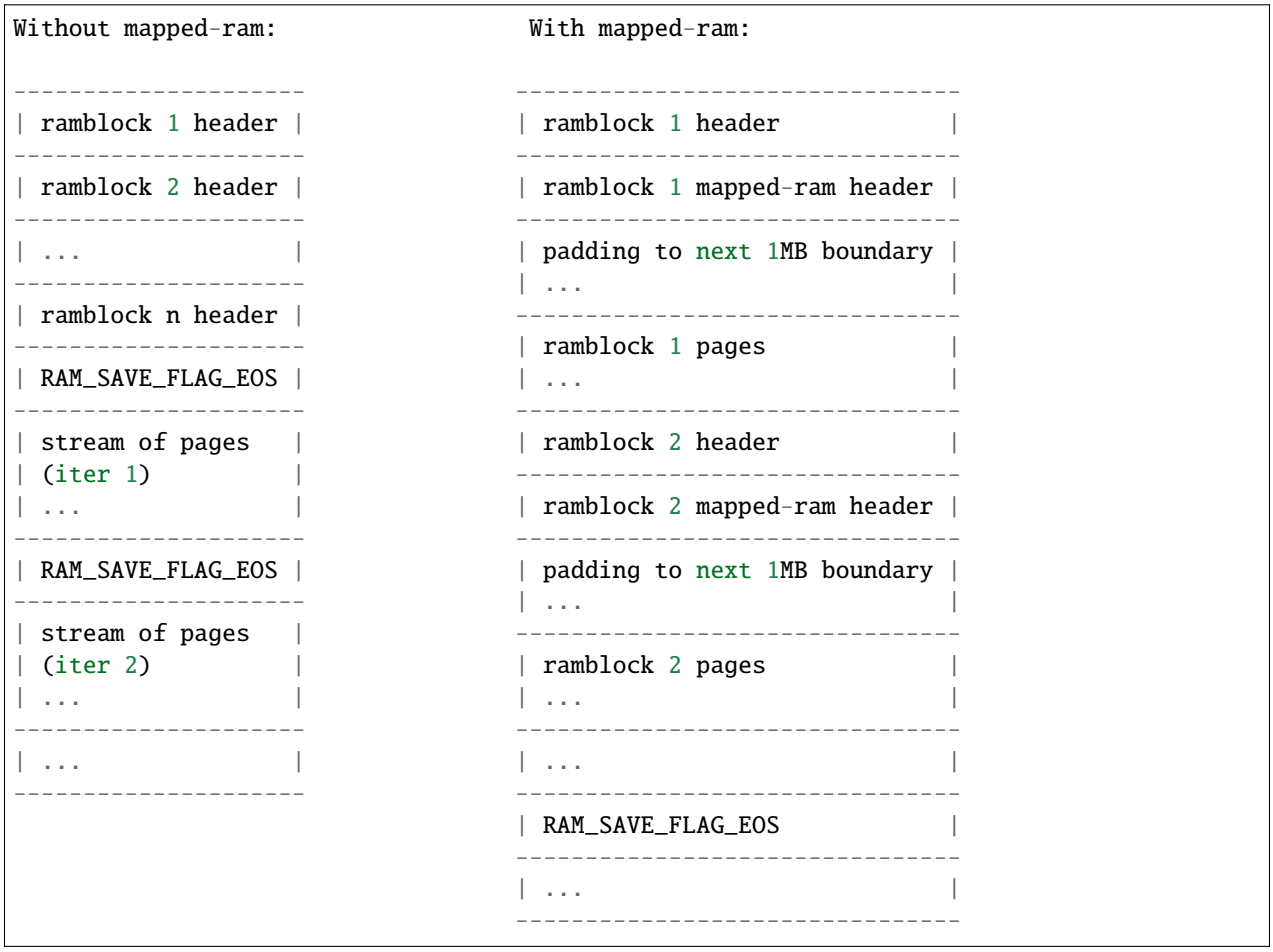
- Live snapshot

If the use case requires that the VM keeps running during and after the snapshot operation, then mapped-ram migration can still be used, but will be less performant. Other strategies such as background-snapshot should be evaluated as well. One benefit of mapped-ram in this scenario is portability since background-snapshot depends on async dirty tracking (KVM_GET_DIRTY_LOG) which is not supported outside of Linux.

RAM section format

Instead of having a sequential stream of pages that follow the RAMBlock headers, the dirty pages for a RAMBlock follow its header instead. This ensures that each RAM page has a fixed offset in the resulting migration file.

A bitmap is introduced to track which pages have been written in the migration file. Pages are written at a fixed location for every ramblock. Zero pages are ignored as they'd be zero in the destination migration as well.



where:

- ramblock header: the generic information for a ramblock, such as idstr, used_len, etc.

- ramblock mapped-ram header: the information added by this feature: bitmap of pages written, bitmap size and offset of pages in the migration file.

Restrictions

Since pages are written to their relative offsets and out of order (due to the memory dirtying patterns), streaming channels such as sockets are not supported. A seekable channel such as a file is required. This can be verified in the QIOChannel by the presence of the QIO_CHANNEL_FEATURE_SEEKABLE.

The improvements brought by this feature apply only to guest physical RAM. Other types of memory such as VRAM are migrated as part of device states.

CheckPoint and Restart (CPR)

CPR is the umbrella name for a set of migration modes in which the VM is migrated to a new QEMU instance on the same host. It is intended for use when the goal is to update host software components that run the VM, such as QEMU or even the host kernel. At this time, cpr-reboot is the only available mode.

Because QEMU is restarted on the same host, with access to the same local devices, CPR is allowed in certain cases where normal migration would be blocked. However, the user must not modify the contents of guest block devices between quitting old QEMU and starting new QEMU.

CPR unconditionally stops VM execution before memory is saved, and thus does not depend on any form of dirty page tracking.

cpr-reboot mode

In this mode, QEMU stops the VM, and writes VM state to the migration URI, which will typically be a file. After quitting QEMU, the user resumes by running QEMU with the `-incoming` option. Because the old and new QEMU instances are not active concurrently, the URI cannot be a type that streams data from one instance to the other.

Guest RAM can be saved in place if backed by shared memory, or can be copied to a file. The former is more efficient and is therefore preferred.

After state and memory are saved, the user may update userland host software before restarting QEMU and resuming the VM. Further, if the RAM is backed by persistent shared memory, such as a DAX device, then the user may reboot to a new host kernel before restarting QEMU.

This mode supports VFIO devices provided the user first puts the guest in the suspended runstate, such as by issuing the `guest-suspend-ram` command to the QEMU guest agent. The agent must be pre-installed in the guest, and the guest must support suspend to RAM. Beware that suspension can take a few seconds, so the user should poll to see the suspended state before proceeding with the CPR operation.

Usage

It is recommended that guest RAM be backed with some type of shared memory, such as `memory-backend-file, share=on`, and that the `x-ignore-shared` capability be set. This combination allows memory to be saved in place. Otherwise, after QEMU stops the VM, all guest RAM is copied to the migration URI.

Outgoing:

- Set the migration mode parameter to `cpr-reboot`.
- Set the `x-ignore-shared` capability if desired.

- Issue the `migrate` command. It is recommended the the URI be a `file` type, but one can use other types such as `exec`, provided the command captures all the data from the outgoing side, and provides all the data to the incoming side.
- Quit when QEMU reaches the postmigrate state.

Incoming:

- Start QEMU with the `-incoming defer` option.
- Set the migration mode parameter to `cpr-reboot`.
- Set the `x-ignore-shared` capability if desired.
- Issue the `migrate-incoming` command.
- If the VM was running when the outgoing `migrate` command was issued, then QEMU automatically resumes VM execution.

Example 1

```
# qemu-kvm -monitor stdio
-object memory-backend-file,id=ram0,size=4G,mem-path=/dev/dax0.0,align=2M,share=on -m 4G
...

(qemu) info status
VM status: running
(qemu) migrate_set_parameter mode cpr-reboot
(qemu) migrate_set_capability x-ignore-shared on
(qemu) migrate -d file:vm.state
(qemu) info status
VM status: paused (postmigrate)
(qemu) quit

### optionally update kernel and reboot
# systemctl kexec
kexec_core: Starting new kernel
...

# qemu-kvm ... -incoming defer
(qemu) info status
VM status: paused (inmigrate)
(qemu) migrate_set_parameter mode cpr-reboot
(qemu) migrate_set_capability x-ignore-shared on
(qemu) migrate_incoming file:vm.state
(qemu) info status
VM status: running
```

Example 2: VFIO

```
# qemu-kvm -monitor stdio
-object memory-backend-file,id=ram0,size=4G,mem-path=/dev/dax0.0,align=2M,share=on -m 4G
-device vfio-pci, ...
-chardev socket,id=qga0,path=qga.sock,server=on,wait=off
-device virtserialport,chardev=qga0,name=org.qemu.guest_agent.0
...

(qemu) info status
VM status: running

# echo '{"execute":"guest-suspend-ram"}' | ncat --send-only -U qga.sock

(qemu) info status
VM status: paused (suspended)
(qemu) migrate_set_parameter mode cpr-reboot
(qemu) migrate_set_capability x-ignore-shared on
(qemu) migrate -d file:vm.state
(qemu) info status
VM status: paused (postmigrate)
(qemu) quit

### optionally update kernel and reboot
# systemctl kexec
kexec_core: Starting new kernel
...

# qemu-kvm ... -incoming defer
(qemu) info status
VM status: paused (inmigrate)
(qemu) migrate_set_parameter mode cpr-reboot
(qemu) migrate_set_capability x-ignore-shared on
(qemu) migrate_incoming file:vm.state
(qemu) info status
VM status: paused (suspended)
(qemu) system_wakeup
(qemu) info status
VM status: running
```

Caveats

cpr-reboot mode may not be used with postcopy, background-snapshot, or COLO.

Backwards compatibility

How backwards compatibility works

When we do migration, we have two QEMU processes: the source and the target. There are two cases, they are the same version or they are different versions. The easy case is when they are the same version. The difficult one is when they are different versions.

There are two things that are different, but they have very similar names and sometimes get confused:

- QEMU version
- machine type version

Let's start with a practical example, we start with:

- qemu-system-x86_64 (v5.2), from now on qemu-5.2.
- qemu-system-x86_64 (v5.1), from now on qemu-5.1.

Related to this are the "latest" machine types defined on each of them:

- pc-q35-5.2 (newer one in qemu-5.2) from now on pc-5.2
- pc-q35-5.1 (newer one in qemu-5.1) from now on pc-5.1

First of all, migration is only supposed to work if you use the same machine type in both source and destination. The QEMU hardware configuration needs to be the same also on source and destination. Most aspects of the backend configuration can be changed at will, except for a few cases where the backend features influence frontend device feature exposure. But that is not relevant for this section.

I am going to list the number of combinations that we can have. Let's start with the trivial ones, QEMU is the same on source and destination:

1 - qemu-5.2 -M pc-5.2 -> migrates to -> qemu-5.2 -M pc-5.2

> This is the latest QEMU with the latest machine type. This have to work, and if it doesn't work it is a bug.

2 - qemu-5.1 -M pc-5.1 -> migrates to -> qemu-5.1 -M pc-5.1

> Exactly the same case than the previous one, but for 5.1. Nothing to see here either.

This are the easiest ones, we will not talk more about them in this section.

Now we start with the more interesting cases. Consider the case where we have the same QEMU version in both sides (qemu-5.2) but we are using the latest machine type for that version (pc-5.2) but one of an older QEMU version, in this case pc-5.1.

3 - qemu-5.2 -M pc-5.1 -> migrates to -> qemu-5.2 -M pc-5.1

> It needs to use the definition of pc-5.1 and the devices as they were configured on 5.1, but this should be easy in the sense that both sides are the same QEMU and both sides have exactly the same idea of what the pc-5.1 machine is.

4 - qemu-5.1 -M pc-5.2 -> migrates to -> qemu-5.1 -M pc-5.2

> This combination is not possible as the qemu-5.1 doesn't understand pc-5.2 machine type. So nothing to worry here.

Now it comes the interesting ones, when both QEMU processes are different. Notice also that the machine type needs to be pc-5.1, because we have the limitation than qemu-5.1 doesn't know pc-5.2. So the possible cases are:

5 - qemu-5.2 -M pc-5.1 -> migrates to -> qemu-5.1 -M pc-5.1

> This migration is known as newer to older. We need to make sure when we are developing 5.2 we need to take care about not to break migration to qemu-5.1. Notice that we can't make updates to qemu-5.1 to understand whatever qemu-5.2 decides to change, so it is in qemu-5.2 side to make the relevant changes.

6 - qemu-5.1 -M pc-5.1 -> migrates to -> qemu-5.2 -M pc-5.1

> This migration is known as older to newer. We need to make sure than we are able to receive migrations from qemu-5.1. The problem is similar to the previous one.

If qemu-5.1 and qemu-5.2 were the same, there will not be any compatibility problems. But the reason that we create qemu-5.2 is to get new features, devices, defaults, etc.

If we get a device that has a new feature, or change a default value, we have a problem when we try to migrate between different QEMU versions.

So we need a way to tell qemu-5.2 that when we are using machine type pc-5.1, it needs to **not** use the feature, to be able to migrate to real qemu-5.1.

And the equivalent part when migrating from qemu-5.1 to qemu-5.2. qemu-5.2 has to expect that it is not going to get data for the new feature, because qemu-5.1 doesn't know about it.

How do we tell QEMU about these device feature changes? In hw/core/machine.c:hw_compat_X_Y arrays.

If we change a default value, we need to put back the old value on that array. And the device, during initialization needs to look at that array to see what value it needs to get for that feature. And what are we going to put in that array, the value of a property.

To create a property for a device, we need to use one of the DEFINE_PROP_*() macros. See include/hw/qdev-properties.h to find the macros that exist. With it, we set the default value for that property, and that is what it is going to get in the latest released version. But if we want a different value for a previous version, we can change that in the hw_compat_X_Y arrays.

hw_compat_X_Y is an array of registers that have the format:

- name_device
- name_property
- value

Let's see a practical example.

In qemu-5.2 virtio-blk-device got multi queue support. This is a change that is not backward compatible. In qemu-5.1 it has one queue. In qemu-5.2 it has the same number of queues as the number of cpus in the system.

When we are doing migration, if we migrate from a device that has 4 queues to a device that have only one queue, we don't know where to put the extra information for the other 3 queues, and we fail migration.

Similar problem when we migrate from qemu-5.1 that has only one queue to qemu-5.2, we only sent information for one queue, but destination has 4, and we have 3 queues that are not properly initialized and anything can happen.

So, how can we address this problem. Easy, just convince qemu-5.2 that when it is running pc-5.1, it needs to set the number of queues for virtio-blk-devices to 1.

That way we fix the cases 5 and 6.

5 - qemu-5.2 -M pc-5.1 -> migrates to -> qemu-5.1 -M pc-5.1

> qemu-5.2 -M pc-5.1 sets number of queues to be 1. qemu-5.1 -M pc-5.1 expects number of queues to be 1.
>
> correct. migration works.

6 - qemu-5.1 -M pc-5.1 -> migrates to -> qemu-5.2 -M pc-5.1

qemu-5.1 -M pc-5.1 sets number of queues to be 1. qemu-5.2 -M pc-5.1 expects number of queues to be 1.

correct. migration works.

And now the other interesting case, case 3. In this case we have:

3 - qemu-5.2 -M pc-5.1 -> migrates to -> qemu-5.2 -M pc-5.1

Here we have the same QEMU in both sides. So it doesn't matter a lot if we have set the number of queues to 1 or not, because they are the same.

WRONG!

Think what happens if we do one of this double migrations:

A -> migrates -> B -> migrates -> C

where:

A: qemu-5.1 -M pc-5.1 B: qemu-5.2 -M pc-5.1 C: qemu-5.2 -M pc-5.1

migration A -> B is case 6, so number of queues needs to be 1.

migration B -> C is case 3, so we don't care. But actually we care because we haven't started the guest in qemu-5.2, it came migrated from qemu-5.1. So to be in the safe place, we need to always use number of queues 1 when we are using pc-5.1.

Now, how was this done in reality? The following commit shows how it was done:

```
commit 9445e1e15e66c19e42bea942ba810db28052cd05
Author: Stefan Hajnoczi <stefanha@redhat.com>
Date:   Tue Aug 18 15:33:47 2020 +0100

virtio-blk-pci: default num_queues to -smp N
```

The relevant parts for migration are:

```
@@ -1281,7 +1284,8 @@ static Property virtio_blk_properties[] = {
 #endif
     DEFINE_PROP_BIT("request-merging", VirtIOBlock, conf.request_merging, 0,
                     true),
-    DEFINE_PROP_UINT16("num-queues", VirtIOBlock, conf.num_queues, 1),
+    DEFINE_PROP_UINT16("num-queues", VirtIOBlock, conf.num_queues,
+                       VIRTIO_BLK_AUTO_NUM_QUEUES),
     DEFINE_PROP_UINT16("queue-size", VirtIOBlock, conf.queue_size, 256),
```

It changes the default value of num_queues. But it fishes it for old machine types to have the right value:

```
@@ -31,6 +31,7 @@
 GlobalProperty hw_compat_5_1[] = {
     ...
+    { "virtio-blk-device", "num-queues", "1"},
     ...
 };
```

A device with different features on both sides

Let's assume that we are using the same QEMU binary on both sides, just to make the things easier. But we have a device that has different features on both sides of the migration. That can be because the devices are different, because the kernel driver of both devices have different features, whatever.

How can we get this to work with migration. The way to do that is "theoretically" easy. You have to get the features that the device has in the source of the migration. The features that the device has on the target of the migration, you get the intersection of the features of both sides, and that is the way that you should launch QEMU.

Notice that this is not completely related to QEMU. The most important thing here is that this should be handled by the managing application that launches QEMU. If QEMU is configured correctly, the migration will succeed.

That said, actually doing it is complicated. Almost all devices are bad at being able to be launched with only some features enabled. With one big exception: cpus.

You can read the documentation for QEMU x86 cpu models here:

https://qemu-project.gitlab.io/qemu/system/qemu-cpu-models.html

See when they talk about migration they recommend that one chooses the newest cpu model that is supported for all cpus.

Let's say that we have:

Host A:

Device X has the feature Y

Host B:

Device X has not the feature Y

If we try to migrate without any care from host A to host B, it will fail because when migration tries to load the feature Y on destination, it will find that the hardware is not there.

Doing this would be the equivalent of doing with cpus:

Host A:

$ qemu-system-x86_64 -cpu host

Host B:

$ qemu-system-x86_64 -cpu host

When both hosts have different cpu features this is guaranteed to fail. Especially if Host B has less features than host A. If host A has less features than host B, sometimes it works. Important word of last sentence is "sometimes".

So, forgetting about cpu models and continuing with the -cpu host example, let's see that the differences of the cpus is that Host A and B have the following features:

Features: 'pcid' 'stibp' 'taa-no' Host A: X X Host B: X

And we want to migrate between them, the way configure both QEMU cpu will be:

Host A:

$ qemu-system-x86_64 -cpu host,pcid=off,stibp=off

Host B:

$ qemu-system-x86_64 -cpu host,taa-no=off

And you would be able to migrate between them. It is responsibility of the management application or of the user to make sure that the configuration is correct. QEMU doesn't know how to look at this kind of features in general.

Notice that we don't recommend to use -cpu host for migration. It is used in this example because it makes the example simpler.

Other devices have worse control about individual features. If they want to be able to migrate between hosts that show different features, the device needs a way to configure which ones it is going to use.

In this section we have considered that we are using the same QEMU binary in both sides of the migration. If we use different QEMU versions process, then we need to have into account all other differences and the examples become even more complicated.

How to mitigate when we have a backward compatibility error

We broke migration for old machine types continuously during development. But as soon as we find that there is a problem, we fix it. The problem is what happens when we detect after we have done a release that something has gone wrong.

Let see how it worked with one example.

After the release of qemu-8.0 we found a problem when doing migration of the machine type pc-7.2.

- $ qemu-7.2 -M pc-7.2 -> qemu-7.2 -M pc-7.2

  This migration works

- $ qemu-8.0 -M pc-7.2 -> qemu-8.0 -M pc-7.2

  This migration works

- $ qemu-8.0 -M pc-7.2 -> qemu-7.2 -M pc-7.2

  This migration fails

- $ qemu-7.2 -M pc-7.2 -> qemu-8.0 -M pc-7.2

  This migration fails

So clearly something fails when migration between qemu-7.2 and qemu-8.0 with machine type pc-7.2. The error messages, and git bisect pointed to this commit.

In qemu-8.0 we got this commit:

```
commit 010746ae1db7f52700cb2e2c46eb94f299cfa0d2
Author: Jonathan Cameron <Jonathan.Cameron@huawei.com>
Date:   Thu Mar 2 13:37:02 2023 +0000

hw/pci/aer: Implement PCI_ERR_UNCOR_MASK register
```

The relevant bits of the commit for our example are this ones:

```
--- a/hw/pci/pcie_aer.c
+++ b/hw/pci/pcie_aer.c
@@ -112,6 +112,10 @@ int pcie_aer_init(PCIDevice *dev,

     pci_set_long(dev->w1cmask + offset + PCI_ERR_UNCOR_STATUS,
                  PCI_ERR_UNC_SUPPORTED);
+    pci_set_long(dev->config + offset + PCI_ERR_UNCOR_MASK,
+                 PCI_ERR_UNC_MASK_DEFAULT);
+    pci_set_long(dev->wmask + offset + PCI_ERR_UNCOR_MASK,
+                 PCI_ERR_UNC_SUPPORTED);
```

(continues on next page)

(continued from previous page)

```
    pci_set_long(dev->config + offset + PCI_ERR_UNCOR_SEVER,
                 PCI_ERR_UNC_SEVERITY_DEFAULT);
```

The patch changes how we configure PCI space for AER. But QEMU fails when the PCI space configuration is different between source and destination.

The following commit shows how this got fixed:

```
commit 5ed3dabe57dd9f4c007404345e5f5bf0e347317f
Author: Leonardo Bras <leobras@redhat.com>
Date:   Tue May 2 21:27:02 2023 -0300

hw/pci: Disable PCI_ERR_UNCOR_MASK register for machine type < 8.0

[...]
```

The relevant parts of the fix in QEMU are as follow:

First, we create a new property for the device to be able to configure the old behaviour or the new behaviour:

```
diff --git a/hw/pci/pci.c b/hw/pci/pci.c
index 8a87ccc8b0..5153ad63d6 100644
--- a/hw/pci/pci.c
+++ b/hw/pci/pci.c
@@ -79,6 +79,8 @@ static Property pci_props[] = {
     DEFINE_PROP_STRING("failover_pair_id", PCIDevice,
                        failover_pair_id),
     DEFINE_PROP_UINT32("acpi-index",  PCIDevice, acpi_index, 0),
+    DEFINE_PROP_BIT("x-pcie-err-unc-mask", PCIDevice, cap_present,
+                    QEMU_PCIE_ERR_UNC_MASK_BITNR, true),
     DEFINE_PROP_END_OF_LIST()
 };
```

Notice that we enable the feature for new machine types.

Now we see how the fix is done. This is going to depend on what kind of breakage happens, but in this case it is quite simple:

```
diff --git a/hw/pci/pcie_aer.c b/hw/pci/pcie_aer.c
index 103667c368..374d593ead 100644
--- a/hw/pci/pcie_aer.c
+++ b/hw/pci/pcie_aer.c
@@ -112,10 +112,13 @@ int pcie_aer_init(PCIDevice *dev, uint8_t cap_ver,
uint16_t offset,

     pci_set_long(dev->w1cmask + offset + PCI_ERR_UNCOR_STATUS,
                  PCI_ERR_UNC_SUPPORTED);
-    pci_set_long(dev->config + offset + PCI_ERR_UNCOR_MASK,
-                 PCI_ERR_UNC_MASK_DEFAULT);
-    pci_set_long(dev->wmask + offset + PCI_ERR_UNCOR_MASK,
-                 PCI_ERR_UNC_SUPPORTED);
+
+    if (dev->cap_present & QEMU_PCIE_ERR_UNC_MASK) {
```

(continues on next page)

(continued from previous page)

```
+        pci_set_long(dev->config + offset + PCI_ERR_UNCOR_MASK,
+                     PCI_ERR_UNC_MASK_DEFAULT);
+        pci_set_long(dev->wmask + offset + PCI_ERR_UNCOR_MASK,
+                     PCI_ERR_UNC_SUPPORTED);
+    }

     pci_set_long(dev->config + offset + PCI_ERR_UNCOR_SEVER,
                  PCI_ERR_UNC_SEVERITY_DEFAULT);
```

I.e. If the property bit is enabled, we configure it as we did for qemu-8.0. If the property bit is not set, we configure it as it was in 7.2.

And now, everything that is missing is disabling the feature for old machine types:

```
diff --git a/hw/core/machine.c b/hw/core/machine.c
index 47a34841a5..07f763eb2e 100644
--- a/hw/core/machine.c
+++ b/hw/core/machine.c
@@ -48,6 +48,7 @@ GlobalProperty hw_compat_7_2[] = {
     { "e1000e", "migrate-timadj", "off" },
     { "virtio-mem", "x-early-migration", "false" },
     { "migration", "x-preempt-pre-7-2", "true" },
+    { TYPE_PCI_DEVICE, "x-pcie-err-unc-mask", "off" },
 };
 const size_t hw_compat_7_2_len = G_N_ELEMENTS(hw_compat_7_2);
```

And now, when qemu-8.0.1 is released with this fix, all combinations are going to work as supposed.

- $ qemu-7.2 -M pc-7.2 -> qemu-7.2 -M pc-7.2 (works)
- $ qemu-8.0.1 -M pc-7.2 -> qemu-8.0.1 -M pc-7.2 (works)
- $ qemu-8.0.1 -M pc-7.2 -> qemu-7.2 -M pc-7.2 (works)
- $ qemu-7.2 -M pc-7.2 -> qemu-8.0.1 -M pc-7.2 (works)

So the normality has been restored and everything is ok, no?

Not really, now our matrix is much bigger. We started with the easy cases, migration from the same version to the same version always works:

- $ qemu-7.2 -M pc-7.2 -> qemu-7.2 -M pc-7.2
- $ qemu-8.0 -M pc-7.2 -> qemu-8.0 -M pc-7.2
- $ qemu-8.0.1 -M pc-7.2 -> qemu-8.0.1 -M pc-7.2

Now the interesting ones. When the QEMU processes versions are different. For the 1st set, their fail and we can do nothing, both versions are released and we can't change anything.

- $ qemu-7.2 -M pc-7.2 -> qemu-8.0 -M pc-7.2
- $ qemu-8.0 -M pc-7.2 -> qemu-7.2 -M pc-7.2

This two are the ones that work. The whole point of making the change in qemu-8.0.1 release was to fix this issue:

- $ qemu-7.2 -M pc-7.2 -> qemu-8.0.1 -M pc-7.2
- $ qemu-8.0.1 -M pc-7.2 -> qemu-7.2 -M pc-7.2

But now we found that qemu-8.0 neither can migrate to qemu-7.2 not qemu-8.0.1.

- $ qemu-8.0 -M pc-7.2 -> qemu-8.0.1 -M pc-7.2
- $ qemu-8.0.1 -M pc-7.2 -> qemu-8.0 -M pc-7.2

So, if we start a pc-7.2 machine in qemu-8.0 we can't migrate it to anything except to qemu-8.0.

Can we do better?

Yeap. If we know that we are going to do this migration:

- $ qemu-8.0 -M pc-7.2 -> qemu-8.0.1 -M pc-7.2

We can launch the appropriate devices with:

```
--device...,x-pci-e-err-unc-mask=on
```

And now we can receive a migration from 8.0. And from now on, we can do that migration to new machine types if we remember to enable that property for pc-7.2. Notice that we need to remember, it is not enough to know that the source of the migration is qemu-8.0. Think of this example:

$ qemu-8.0 -M pc-7.2 -> qemu-8.0.1 -M pc-7.2 -> qemu-8.2 -M pc-7.2

In the second migration, the source is not qemu-8.0, but we still have that "problem" and have that property enabled. Notice that we need to continue having this mark/property until we have this machine rebooted. But it is not a normal reboot (that don't reload QEMU) we need the machine to poweroff/poweron on a fixed QEMU. And from now on we can use the proper real machine.

Best practices

Debugging

The migration stream can be analyzed thanks to `scripts/analyze-migration.py`.

Example usage:

```
$ qemu-system-x86_64 -display none -monitor stdio
(qemu) migrate "exec:cat > mig"
(qemu) q
$ ./scripts/analyze-migration.py -f mig
{
  "ram (3)": {
      "section sizes": {
          "pc.ram": "0x0000000008000000",
...
```

See also `analyze-migration.py -h` help for more options.

Firmware

Migration migrates the copies of RAM and ROM, and thus when running on the destination it includes the firmware from the source. Even after resetting a VM, the old firmware is used. Only once QEMU has been restarted is the new firmware in use.

- Changes in firmware size can cause changes in the required RAMBlock size to hold the firmware and thus migration can fail. In practice it's best to pad firmware images to convenient powers of 2 with plenty of space for growth.

- Care should be taken with device emulation code so that newer emulation code can work with older firmware to allow forward migration.
- Care should be taken with newer firmware so that backward migration to older systems with older device emulation code will work.

In some cases it may be best to tie specific firmware versions to specific versioned machine types to cut down on the combinations that will need support. This is also useful when newer versions of firmware outgrow the padding.

7.4.7 Multi-process QEMU

Note: This is the design document for multi-process QEMU. It does not necessarily reflect the status of the current implementation, which may lack features or be considerably different from what is described in this document. This document is still useful as a description of the goals and general direction of this feature.

Please refer to the following wiki for latest details: https://wiki.qemu.org/Features/MultiProcessQEMU

QEMU is often used as the hypervisor for virtual machines running in the Oracle cloud. Since one of the advantages of cloud computing is the ability to run many VMs from different tenants in the same cloud infrastructure, a guest that compromised its hypervisor could potentially use the hypervisor's access privileges to access data it is not authorized for.

QEMU can be susceptible to security attacks because it is a large, monolithic program that provides many features to the VMs it services. Many of these features can be configured out of QEMU, but even a reduced configuration QEMU has a large amount of code a guest can potentially attack. Separating QEMU reduces the attack surface by aiding to limit each component in the system to only access the resources that it needs to perform its job.

QEMU services

QEMU can be broadly described as providing three main services. One is a VM control point, where VMs can be created, migrated, re-configured, and destroyed. A second is to emulate the CPU instructions within the VM, often accelerated by HW virtualization features such as Intel's VT extensions. Finally, it provides IO services to the VM by emulating HW IO devices, such as disk and network devices.

A multi-process QEMU

A multi-process QEMU involves separating QEMU services into separate host processes. Each of these processes can be given only the privileges it needs to provide its service, e.g., a disk service could be given access only to the disk images it provides, and not be allowed to access other files, or any network devices. An attacker who compromised this service would not be able to use this exploit to access files or devices beyond what the disk service was given access to.

A QEMU control process would remain, but in multi-process mode, will have no direct interfaces to the VM. During VM execution, it would still provide the user interface to hot-plug devices or live migrate the VM.

A first step in creating a multi-process QEMU is to separate IO services from the main QEMU program, which would continue to provide CPU emulation. i.e., the control process would also be the CPU emulation process. In a later phase, CPU emulation could be separated from the control process.

Separating IO services

Separating IO services into individual host processes is a good place to begin for a couple of reasons. One is the sheer number of IO devices QEMU can emulate provides a large surface of interfaces which could potentially be exploited, and, indeed, have been a source of exploits in the past. Another is the modular nature of QEMU device emulation code provides interface points where the QEMU functions that perform device emulation can be separated from the QEMU functions that manage the emulation of guest CPU instructions. The devices emulated in the separate process are referred to as remote devices.

QEMU device emulation

QEMU uses an object oriented SW architecture for device emulation code. Configured objects are all compiled into the QEMU binary, then objects are instantiated by name when used by the guest VM. For example, the code to emulate a device named "foo" is always present in QEMU, but its instantiation code is only run when the device is included in the target VM. (e.g., via the QEMU command line as *-device foo*)

The object model is hierarchical, so device emulation code names its parent object (such as "pci-device" for a PCI device) and QEMU will instantiate a parent object before calling the device's instantiation code.

Current separation models

In order to separate the device emulation code from the CPU emulation code, the device object code must run in a different process. There are a couple of existing QEMU features that can run emulation code separately from the main QEMU process. These are examined below.

vhost user model

Virtio guest device drivers can be connected to vhost user applications in order to perform their IO operations. This model uses special virtio device drivers in the guest and vhost user device objects in QEMU, but once the QEMU vhost user code has configured the vhost user application, mission-mode IO is performed by the application. The vhost user application is a daemon process that can be contacted via a known UNIX domain socket.

vhost socket

As mentioned above, one of the tasks of the vhost device object within QEMU is to contact the vhost application and send it configuration information about this device instance. As part of the configuration process, the application can also be sent other file descriptors over the socket, which then can be used by the vhost user application in various ways, some of which are described below.

vhost MMIO store acceleration

VMs are often run using HW virtualization features via the KVM kernel driver. This driver allows QEMU to accelerate the emulation of guest CPU instructions by running the guest in a virtual HW mode. When the guest executes instructions that cannot be executed by virtual HW mode, execution returns to the KVM driver so it can inform QEMU to emulate the instructions in SW.

One of the events that can cause a return to QEMU is when a guest device driver accesses an IO location. QEMU then dispatches the memory operation to the corresponding QEMU device object. In the case of a vhost user device, the memory operation would need to be sent over a socket to the vhost application. This path is accelerated by the

QEMU virtio code by setting up an eventfd file descriptor that the vhost application can directly receive MMIO store notifications from the KVM driver, instead of needing them to be sent to the QEMU process first.

vhost interrupt acceleration

Another optimization used by the vhost application is the ability to directly inject interrupts into the VM via the KVM driver, again, bypassing the need to send the interrupt back to the QEMU process first. The QEMU virtio setup code configures the KVM driver with an eventfd that triggers the device interrupt in the guest when the eventfd is written. This irqfd file descriptor is then passed to the vhost user application program.

vhost access to guest memory

The vhost application is also allowed to directly access guest memory, instead of needing to send the data as messages to QEMU. This is also done with file descriptors sent to the vhost user application by QEMU. These descriptors can be passed to `mmap()` by the vhost application to map the guest address space into the vhost application.

IOMMUs introduce another level of complexity, since the address given to the guest virtio device to DMA to or from is not a guest physical address. This case is handled by having vhost code within QEMU register as a listener for IOMMU mapping changes. The vhost application maintains a cache of IOMMMU translations: sending translation requests back to QEMU on cache misses, and in turn receiving flush requests from QEMU when mappings are purged.

applicability to device separation

Much of the vhost model can be re-used by separated device emulation. In particular, the ideas of using a socket between QEMU and the device emulation application, using a file descriptor to inject interrupts into the VM via KVM, and allowing the application to `mmap()` the guest should be re used.

There are, however, some notable differences between how a vhost application works and the needs of separated device emulation. The most basic is that vhost uses custom virtio device drivers which always trigger IO with MMIO stores. A separated device emulation model must work with existing IO device models and guest device drivers. MMIO loads break vhost store acceleration since they are synchronous - guest progress cannot continue until the load has been emulated. By contrast, stores are asynchronous, the guest can continue after the store event has been sent to the vhost application.

Another difference is that in the vhost user model, a single daemon can support multiple QEMU instances. This is contrary to the security regime desired, in which the emulation application should only be allowed to access the files or devices the VM it's running on behalf of can access. #### qemu-io model

`qemu-io` is a test harness used to test changes to the QEMU block backend object code (e.g., the code that implements disk images for disk driver emulation). `qemu-io` is not a device emulation application per se, but it does compile the QEMU block objects into a separate binary from the main QEMU one. This could be useful for disk device emulation, since its emulation applications will need to include the QEMU block objects.

New separation model based on proxy objects

A different model based on proxy objects in the QEMU program communicating with remote emulation programs could provide separation while minimizing the changes needed to the device emulation code. The rest of this section is a discussion of how a proxy object model would work.

Remote emulation processes

The remote emulation process will run the QEMU object hierarchy without modification. The device emulation objects will be also be based on the QEMU code, because for anything but the simplest device, it would not be a tractable to re-implement both the object model and the many device backends that QEMU has.

The processes will communicate with the QEMU process over UNIX domain sockets. The processes can be executed either as standalone processes, or be executed by QEMU. In both cases, the host backends the emulation processes will provide are specified on its command line, as they would be for QEMU. For example:

```
disk-proc -blockdev driver=file,node-name=file0,filename=disk-file0  \
-blockdev driver=qcow2,node-name=drive0,file=file0
```

would indicate process *disk-proc* uses a qcow2 emulated disk named *file0* as its backend.

Emulation processes may emulate more than one guest controller. A common configuration might be to put all controllers of the same device class (e.g., disk, network, etc.) in a single process, so that all backends of the same type can be managed by a single QMP monitor.

communication with QEMU

The first argument to the remote emulation process will be a Unix domain socket that connects with the Proxy object. This is a required argument.

```
disk-proc <socket number> <backend list>
```

remote process QMP monitor

Remote emulation processes can be monitored via QMP, similar to QEMU itself. The QMP monitor socket is specified the same as for a QEMU process:

```
disk-proc -qmp unix:/tmp/disk-mon,server
```

can be monitored over the UNIX socket path */tmp/disk-mon*.

QEMU command line

Each remote device emulated in a remote process on the host is represented as a *-device* of type *pci-proxy-dev*. A socket sub-option to this option specifies the Unix socket that connects to the remote process. An *id* sub-option is required, and it should be the same id as used in the remote process.

```
qemu-system-x86_64 ... -device pci-proxy-dev,id=lsi0,socket=3
```

can be used to add a device emulated in a remote process

QEMU management of remote processes

QEMU is not aware of the type of type of the remote PCI device. It is a pass through device as far as QEMU is concerned.

communication with emulation process

primary channel

The primary channel (referred to as com in the code) is used to bootstrap the remote process. It is also used to pass on device-agnostic commands like reset.

per-device channels

Each remote device communicates with QEMU using a dedicated communication channel. The proxy object sets up this channel using the primary channel during its initialization.

QEMU device proxy objects

QEMU has an object model based on sub-classes inherited from the "object" super-class. The sub-classes that are of interest here are the "device" and "bus" sub-classes whose child sub-classes make up the device tree of a QEMU emulated system.

The proxy object model will use device proxy objects to replace the device emulation code within the QEMU process. These objects will live in the same place in the object and bus hierarchies as the objects they replace. i.e., the proxy object for an LSI SCSI controller will be a sub-class of the "pci-device" class, and will have the same PCI bus parent and the same SCSI bus child objects as the LSI controller object it replaces.

It is worth noting that the same proxy object is used to mediate with all types of remote PCI devices.

object initialization

The Proxy device objects are initialized in the exact same manner in which any other QEMU device would be initialized.

In addition, the Proxy objects perform the following two tasks: - Parses the "socket" sub option and connects to the remote process using this channel - Uses the "id" sub-option to connect to the emulated device on the separate process

class_init

The `class_init()` method of a proxy object will, in general behave similarly to the object it replaces, including setting any static properties and methods needed by the proxy.

instance_init / realize

The `instance_init()` and `realize()` functions would only need to perform tasks related to being a proxy, such are registering its own MMIO handlers, or creating a child bus that other proxy devices can be attached to later.

Other tasks will be device-specific. For example, PCI device objects will initialize the PCI config space in order to make a valid PCI device tree within the QEMU process.

address space registration

Most devices are driven by guest device driver accesses to IO addresses or ports. The QEMU device emulation code uses QEMU's memory region function calls (such as `memory_region_init_io()`) to add callback functions that QEMU will invoke when the guest accesses the device's areas of the IO address space. When a guest driver does access the device, the VM will exit HW virtualization mode and return to QEMU, which will then lookup and execute the corresponding callback function.

A proxy object would need to mirror the memory region calls the actual device emulator would perform in its initialization code, but with its own callbacks. When invoked by QEMU as a result of a guest IO operation, they will forward the operation to the device emulation process.

PCI config space

PCI devices also have a configuration space that can be accessed by the guest driver. Guest accesses to this space is not handled by the device emulation object, but by its PCI parent object. Much of this space is read-only, but certain registers (especially BAR and MSI-related ones) need to be propagated to the emulation process.

PCI parent proxy

One way to propagate guest PCI config accesses is to create a "pci-device-proxy" class that can serve as the parent of a PCI device proxy object. This class's parent would be "pci-device" and it would override the PCI parent's `config_read()` and `config_write()` methods with ones that forward these operations to the emulation program.

interrupt receipt

A proxy for a device that generates interrupts will need to create a socket to receive interrupt indications from the emulation process. An incoming interrupt indication would then be sent up to its bus parent to be injected into the guest. For example, a PCI device object may use `pci_set_irq()`.

live migration

The proxy will register to save and restore any *vmstate* it needs over a live migration event. The device proxy does not need to manage the remote device's *vmstate*; that will be handled by the remote process proxy (see below).

QEMU remote device operation

Generic device operations, such as DMA, will be performed by the remote process proxy by sending messages to the remote process.

DMA operations

DMA operations would be handled much like vhost applications do. One of the initial messages sent to the emulation process is a guest memory table. Each entry in this table consists of a file descriptor and size that the emulation process can `mmap()` to directly access guest memory, similar to `vhost_user_set_mem_table()`. Note guest memory must be backed by shared file-backed memory, for example, using *-object memory-backend-file,share=on* and setting that memory backend as RAM for the machine.

IOMMU operations

When the emulated system includes an IOMMU, the remote process proxy in QEMU will need to create a socket for IOMMU requests from the emulation process. It will handle those requests with an `address_space_get_iotlb_entry()` call. In order to handle IOMMU unmaps, the remote process proxy will also register as a listener on the device's DMA address space. When an IOMMU memory region is created within the DMA address space, an IOMMU notifier for unmaps will be added to the memory region that will forward unmaps to the emulation process over the IOMMU socket.

device hot-plug via QMP

An QMP "device_add" command can add a device emulated by a remote process. It will also have "rid" option to the command, just as the *-device* command line option does. The remote process may either be one started at QEMU startup, or be one added by the "add-process" QMP command described above. In either case, the remote process proxy will forward the new device's JSON description to the corresponding emulation process.

live migration

The remote process proxy will also register for live migration notifications with `vmstate_register()`. When called to save state, the proxy will send the remote process a secondary socket file descriptor to save the remote process's device *vmstate* over. The incoming byte stream length and data will be saved as the proxy's *vmstate*. When the proxy is resumed on its new host, this *vmstate* will be extracted, and a secondary socket file descriptor will be sent to the new remote process through which it receives the *vmstate* in order to restore the devices there.

device emulation in remote process

The parts of QEMU that the emulation program will need include the object model; the memory emulation objects; the device emulation objects of the targeted device, and any dependent devices; and, the device's backends. It will also need code to setup the machine environment, handle requests from the QEMU process, and route machine-level requests (such as interrupts or IOMMU mappings) back to the QEMU process.

initialization

The process initialization sequence will follow the same sequence followed by QEMU. It will first initialize the backend objects, then device emulation objects. The JSON descriptions sent by the QEMU process will drive which objects need to be created.

- address spaces

Before the device objects are created, the initial address spaces and memory regions must be configured with `memory_map_init()`. This creates a RAM memory region object (*system_memory*) and an IO memory region object (*system_io*).

- RAM

RAM memory region creation will follow how `pc_memory_init()` creates them, but must use `memory_region_init_ram_from_fd()` instead of `memory_region_allocate_system_memory()`. The file descriptors needed will be supplied by the guest memory table from above. Those RAM regions would then be added to the *system_memory* memory region with `memory_region_add_subregion()`.

- PCI

IO initialization will be driven by the JSON descriptions sent from the QEMU process. For a PCI device, a PCI bus will need to be created with `pci_root_bus_new()`, and a PCI memory region will need to be created and added to the *system_memory* memory region with `memory_region_add_subregion_overlap()`. The overlap version is required for architectures where PCI memory overlaps with RAM memory.

MMIO handling

The device emulation objects will use `memory_region_init_io()` to install their MMIO handlers, and `pci_register_bar()` to associate those handlers with a PCI BAR, as they do within QEMU currently.

In order to use `address_space_rw()` in the emulation process to handle MMIO requests from QEMU, the PCI physical addresses must be the same in the QEMU process and the device emulation process. In order to accomplish that, guest BAR programming must also be forwarded from QEMU to the emulation process.

interrupt injection

When device emulation wants to inject an interrupt into the VM, the request climbs the device's bus object hierarchy until the point where a bus object knows how to signal the interrupt to the guest. The details depend on the type of interrupt being raised.

- PCI pin interrupts

On x86 systems, there is an emulated IOAPIC object attached to the root PCI bus object, and the root PCI object forwards interrupt requests to it. The IOAPIC object, in turn, calls the KVM driver to inject the corresponding interrupt into the VM. The simplest way to handle this in an emulation process would be to setup the root PCI bus driver (via `pci_bus_irqs()`) to send a interrupt request back to the QEMU process, and have the device proxy object reflect it up the PCI tree there.

- PCI MSI/X interrupts

PCI MSI/X interrupts are implemented in HW as DMA writes to a CPU-specific PCI address. In QEMU on x86, a KVM APIC object receives these DMA writes, then calls into the KVM driver to inject the interrupt into the VM. A simple emulation process implementation would be to send the MSI DMA address from QEMU as a message at initialization, then install an address space handler at that address which forwards the MSI message back to QEMU.

DMA operations

When a emulation object wants to DMA into or out of guest memory, it first must use dma_memory_map() to convert the DMA address to a local virtual address. The emulation process memory region objects setup above will be used to translate the DMA address to a local virtual address the device emulation code can access.

IOMMU

When an IOMMU is in use in QEMU, DMA translation uses IOMMU memory regions to translate the DMA address to a guest physical address before that physical address can be translated to a local virtual address. The emulation process will need similar functionality.

- IOTLB cache

The emulation process will maintain a cache of recent IOMMU translations (the IOTLB). When the translate() callback of an IOMMU memory region is invoked, the IOTLB cache will be searched for an entry that will map the DMA address to a guest PA. On a cache miss, a message will be sent back to QEMU requesting the corresponding translation entry, which be both be used to return a guest address and be added to the cache.

- IOTLB purge

The IOMMU emulation will also need to act on unmap requests from QEMU. These happen when the guest IOMMU driver purges an entry from the guest's translation table.

live migration

When a remote process receives a live migration indication from QEMU, it will set up a channel using the received file descriptor with `qio_channel_socket_new_fd()`. This channel will be used to create a *QEMUfile* that can be passed to `qemu_save_device_state()` to send the process's device state back to QEMU. This method will be reversed on restore - the channel will be passed to `qemu_loadvm_state()` to restore the device state.

Accelerating device emulation

The messages that are required to be sent between QEMU and the emulation process can add considerable latency to IO operations. The optimizations described below attempt to ameliorate this effect by allowing the emulation process to communicate directly with the kernel KVM driver. The KVM file descriptors created would be passed to the emulation process via initialization messages, much like the guest memory table is done. #### MMIO acceleration

Vhost user applications can receive guest virtio driver stores directly from KVM. The issue with the eventfd mechanism used by vhost user is that it does not pass any data with the event indication, so it cannot handle guest loads or guest stores that carry store data. This concept could, however, be expanded to cover more cases.

The expanded idea would require a new type of KVM device: *KVM_DEV_TYPE_USER*. This device has two file descriptors: a master descriptor that QEMU can use for configuration, and a slave descriptor that the emulation process can use to receive MMIO notifications. QEMU would create both descriptors using the KVM driver, and pass the slave descriptor to the emulation process via an initialization message.

data structures

- guest physical range

The guest physical range structure describes the address range that a device will respond to. It includes the base and length of the range, as well as which bus the range resides on (e.g., on an x86machine, it can specify whether the range refers to memory or IO addresses).

A device can have multiple physical address ranges it responds to (e.g., a PCI device can have multiple BARs), so the structure will also include an enumerated identifier to specify which of the device's ranges is being referred to.

Name	Description
addr	range base address
len	range length
bus	addr type (memory or IO)
id	range ID (e.g., PCI BAR)

- MMIO request structure

This structure describes an MMIO operation. It includes which guest physical range the MMIO was within, the offset within that range, the MMIO type (e.g., load or store), and its length and data. It also includes a sequence number that can be used to reply to the MMIO, and the CPU that issued the MMIO.

Name	Description
rid	range MMIO is within
offset	offset within *rid*
type	e.g., load or store
len	MMIO length
data	store data
seq	sequence ID

- MMIO request queues

MMIO request queues are FIFO arrays of MMIO request structures. There are two queues: pending queue is for MMIOs that haven't been read by the emulation program, and the sent queue is for MMIOs that haven't been acknowledged. The main use of the second queue is to validate MMIO replies from the emulation program.

- scoreboard

Each CPU in the VM is emulated in QEMU by a separate thread, so multiple MMIOs may be waiting to be consumed by an emulation program and multiple threads may be waiting for MMIO replies. The scoreboard would contain a wait queue and sequence number for the per-CPU threads, allowing them to be individually woken when the MMIO reply is received from the emulation program. It also tracks the number of posted MMIO stores to the device that haven't been replied to, in order to satisfy the PCI constraint that a load to a device will not complete until all previous stores to that device have been completed.

- device shadow memory

Some MMIO loads do not have device side-effects. These MMIOs can be completed without sending a MMIO request to the emulation program if the emulation program shares a shadow image of the device's memory image with the KVM driver.

The emulation program will ask the KVM driver to allocate memory for the shadow image, and will then use `mmap()` to directly access it. The emulation program can control KVM access to the shadow image by sending KVM an access map telling it which areas of the image have no side-effects (and can be completed immediately), and which require a MMIO request to the emulation program. The access map can also inform the KVM drive which size accesses are allowed to the image.

master descriptor

The master descriptor is used by QEMU to configure the new KVM device. The descriptor would be returned by the KVM driver when QEMU issues a *KVM_CREATE_DEVICE* `ioctl()` with a *KVM_DEV_TYPE_USER* type.

KVM_DEV_TYPE_USER device ops

The *KVM_DEV_TYPE_USER* operations vector will be registered by a `kvm_register_device_ops()` call when the KVM system in initialized by `kvm_init()`. These device ops are called by the KVM driver when QEMU executes certain `ioctl()` operations on its KVM file descriptor. They include:

- create

This routine is called when QEMU issues a *KVM_CREATE_DEVICE* `ioctl()` on its per-VM file descriptor. It will allocate and initialize a KVM user device specific data structure, and assign the *kvm_device* private field to it.

- ioctl

This routine is invoked when QEMU issues an `ioctl()` on the master descriptor. The `ioctl()` commands supported are defined by the KVM device type. *KVM_DEV_TYPE_USER* ones will need several commands:

KVM_DEV_USER_SLAVE_FD creates the slave file descriptor that will be passed to the device emulation program. Only one slave can be created by each master descriptor. The file operations performed by this descriptor are described below.

The *KVM_DEV_USER_PA_RANGE* command configures a guest physical address range that the slave descriptor will receive MMIO notifications for. The range is specified by a guest physical range structure argument. For buses that assign addresses to devices dynamically, this command can be executed while the guest is running, such as the case when a guest changes a device's PCI BAR registers.

KVM_DEV_USER_PA_RANGE will use `kvm_io_bus_register_dev()` to register *kvm_io_device_ops* callbacks to be invoked when the guest performs a MMIO operation within the range. When a range is changed, `kvm_io_bus_unregister_dev()` is used to remove the previous instantiation.

KVM_DEV_USER_TIMEOUT will configure a timeout value that specifies how long KVM will wait for the emulation process to respond to a MMIO indication.

- destroy

This routine is called when the VM instance is destroyed. It will need to destroy the slave descriptor; and free any memory allocated by the driver, as well as the *kvm_device* structure itself.

slave descriptor

The slave descriptor will have its own file operations vector, which responds to system calls on the descriptor performed by the device emulation program.

- read

A read returns any pending MMIO requests from the KVM driver as MMIO request structures. Multiple structures can be returned if there are multiple MMIO operations pending. The MMIO requests are moved from the pending queue to the sent queue, and if there are threads waiting for space in the pending to add new MMIO operations, they will be woken here.

- write

A write also consists of a set of MMIO requests. They are compared to the MMIO requests in the sent queue. Matches are removed from the sent queue, and any threads waiting for the reply are woken. If a store is removed, then the number of posted stores in the per-CPU scoreboard is decremented. When the number is zero, and a non side-effect load was waiting for posted stores to complete, the load is continued.

- ioctl

There are several ioctl()s that can be performed on the slave descriptor.

A *KVM_DEV_USER_SHADOW_SIZE* `ioctl()` causes the KVM driver to allocate memory for the shadow image. This memory can later be `mmap()`ed by the emulation process to share the emulation's view of device memory with the KVM driver.

A *KVM_DEV_USER_SHADOW_CTRL* `ioctl()` controls access to the shadow image. It will send the KVM driver a shadow control map, which specifies which areas of the image can complete guest loads without sending the load request to the emulation program. It will also specify the size of load operations that are allowed.

- poll

An emulation program will use the `poll()` call with a *POLLIN* flag to determine if there are MMIO requests waiting to be read. It will return if the pending MMIO request queue is not empty.

- mmap

This call allows the emulation program to directly access the shadow image allocated by the KVM driver. As device emulation updates device memory, changes with no side-effects will be reflected in the shadow, and the KVM driver can satisfy guest loads from the shadow image without needing to wait for the emulation program.

kvm_io_device ops

Each KVM per-CPU thread can handle MMIO operation on behalf of the guest VM. KVM will use the MMIO's guest physical address to search for a matching *kvm_io_device* to see if the MMIO can be handled by the KVM driver instead of exiting back to QEMU. If a match is found, the corresponding callback will be invoked.

- read

This callback is invoked when the guest performs a load to the device. Loads with side-effects must be handled synchronously, with the KVM driver putting the QEMU thread to sleep waiting for the emulation process reply before re-starting the guest. Loads that do not have side-effects may be optimized by satisfying them from the shadow image, if there are no outstanding stores to the device by this CPU. PCI memory ordering demands that a load cannot complete before all older stores to the same device have been completed.

- write

Stores can be handled asynchronously unless the pending MMIO request queue is full. In this case, the QEMU thread must sleep waiting for space in the queue. Stores will increment the number of posted stores in the per-CPU scoreboard, in order to implement the PCI ordering constraint above.

interrupt acceleration

This performance optimization would work much like a vhost user application does, where the QEMU process sets up *eventfds* that cause the device's corresponding interrupt to be triggered by the KVM driver. These irq file descriptors are sent to the emulation process at initialization, and are used when the emulation code raises a device interrupt.

intx acceleration

Traditional PCI pin interrupts are level based, so, in addition to an irq file descriptor, a re-sampling file descriptor needs to be sent to the emulation program. This second file descriptor allows multiple devices sharing an irq to be notified when the interrupt has been acknowledged by the guest, so they can re-trigger the interrupt if their device has not de-asserted its interrupt.

intx irq descriptor

The irq descriptors are created by the proxy object `using event_notifier_init()` to create the irq and re-sampling *eventds*, and `kvm_vm_ioctl(KVM_IRQFD)` to bind them to an interrupt. The interrupt route can be found with `pci_device_route_intx_to_irq()`.

intx routing changes

Intx routing can be changed when the guest programs the APIC the device pin is connected to. The proxy object in QEMU will use `pci_device_set_intx_routing_notifier()` to be informed of any guest changes to the route. This handler will broadly follow the VFIO interrupt logic to change the route: de-assigning the existing irq descriptor from its route, then assigning it the new route. (see `vfio_intx_update()`)

MSI/X acceleration

MSI/X interrupts are sent as DMA transactions to the host. The interrupt data contains a vector that is programmed by the guest, A device may have multiple MSI interrupts associated with it, so multiple irq descriptors may need to be sent to the emulation program.

MSI/X irq descriptor

This case will also follow the VFIO example. For each MSI/X interrupt, an *eventfd* is created, a virtual interrupt is allocated by `kvm_irqchip_add_msi_route()`, and the virtual interrupt is bound to the eventfd with `kvm_irqchip_add_irqfd_notifier()`.

MSI/X config space changes

The guest may dynamically update several MSI-related tables in the device's PCI config space. These include per-MSI interrupt enables and vector data. Additionally, MSIX tables exist in device memory space, not config space. Much like the BAR case above, the proxy object must look at guest config space programming to keep the MSI interrupt state consistent between QEMU and the emulation program.

Disaggregated CPU emulation

After IO services have been disaggregated, a second phase would be to separate a process to handle CPU instruction emulation from the main QEMU control function. There are no object separation points for this code, so the first task would be to create one.

Host access controls

Separating QEMU relies on the host OS's access restriction mechanisms to enforce that the differing processes can only access the objects they are entitled to. There are a couple types of mechanisms usually provided by general purpose OSs.

Discretionary access control

Discretionary access control allows each user to control who can access their files. In Linux, this type of control is usually too coarse for QEMU separation, since it only provides three separate access controls: one for the same user ID, the second for users IDs with the same group ID, and the third for all other user IDs. Each device instance would need a separate user ID to provide access control, which is likely to be unwieldy for dynamically created VMs.

Mandatory access control

Mandatory access control allows the OS to add an additional set of controls on top of discretionary access for the OS to control. It also adds other attributes to processes and files such as types, roles, and categories, and can establish rules for how processes and files can interact.

Type enforcement

Type enforcement assigns a *type* attribute to processes and files, and allows rules to be written on what operations a process with a given type can perform on a file with a given type. QEMU separation could take advantage of type enforcement by running the emulation processes with different types, both from the main QEMU process, and from the emulation processes of different classes of devices.

For example, guest disk images and disk emulation processes could have types separate from the main QEMU process and non-disk emulation processes, and the type rules could prevent processes other than disk emulation ones from accessing guest disk images. Similarly, network emulation processes can have a type separate from the main QEMU process and non-network emulation process, and only that type can access the host tun/tap device used to provide guest networking.

Category enforcement

Category enforcement assigns a set of numbers within a given range to the process or file. The process is granted access to the file if the process's set is a superset of the file's set. This enforcement can be used to separate multiple instances of devices in the same class.

For example, if there are multiple disk devices provides to a guest, each device emulation process could be provisioned with a separate category. The different device emulation processes would not be able to access each other's backing disk images.

Alternatively, categories could be used in lieu of the type enforcement scheme described above. In this scenario, different categories would be used to prevent device emulation processes in different classes from accessing resources assigned to other classes.

7.4.8 Reset in QEMU: the Resettable interface

The reset of qemu objects is handled using the resettable interface declared in `include/hw/resettable.h`.

This interface allows objects to be grouped (on a tree basis); so that the whole group can be reset consistently. Each individual member object does not have to care about others; in particular, problems of order (which object is reset first) are addressed.

The main object types which implement this interface are DeviceClass and BusClass.

Triggering reset

This section documents the APIs which "users" of a resettable object should use to control it. All resettable control functions must be called while holding the BQL.

You can apply a reset to an object using `resettable_assert_reset()`. You need to call `resettable_release_reset()` to release the object from reset. To instantly reset an object, without keeping it in reset state, just call `resettable_reset()`. These functions take two parameters: a pointer to the object to reset and a reset type.

The Resettable interface handles reset types with an enum `ResetType`:

`RESET_TYPE_COLD`
: Cold reset is supported by every resettable object. In QEMU, it means we reset to the initial state corresponding to the start of QEMU; this might differ from what is a real hardware cold reset. It differs from other resets (like warm or bus resets) which may keep certain parts untouched.

`RESET_TYPE_SNAPSHOT_LOAD`
: This is called for a reset which is being done to put the system into a clean state prior to loading a snapshot. (This corresponds to a reset with `SHUTDOWN_CAUSE_SNAPSHOT_LOAD`.) Almost all devices should treat this the same as `RESET_TYPE_COLD`. The main exception is devices which have some non-deterministic state they want to reinitialize to a different value on each cold reset, such as RNG seed information, and which they must not reinitialize on a snapshot-load reset.

Devices which implement reset methods must treat any unknown `ResetType` as equivalent to `RESET_TYPE_COLD`; this will reduce the amount of existing code we need to change if we add more types in future.

Calling `resettable_reset()` is equivalent to calling `resettable_assert_reset()` then `resettable_release_reset()`. It is possible to interleave multiple calls to these three functions. There may be several reset sources/controllers of a given object. The interface handles everything and the different reset controllers do not need to know anything about each others. The object will leave reset state only when each other controllers end their reset operation. This point is handled internally by maintaining a count of in-progress resets; it is crucial to call `resettable_release_reset()` one time and only one time per `resettable_assert_reset()` call.

For now migration of a device or bus in reset is not supported. Care must be taken not to delay `resettable_release_reset()` after its `resettable_assert_reset()` counterpart.

Note that, since resettable is an interface, the API takes a simple Object as parameter. Still, it is a programming error to call a resettable function on a non-resettable object and it will trigger a run time assert error. Since most calls to resettable interface are done through base class functions, such an error is not likely to happen.

For Devices and Buses, the following helper functions exist:

- `device_cold_reset()`
- `bus_cold_reset()`

These are simple wrappers around resettable_reset() function; they only cast the Device or Bus into an Object and pass the cold reset type. When possible prefer to use these functions instead of `resettable_reset()`.

Device and bus functions co-exist because there can be semantic differences between resetting a bus and resetting the controller bridge which owns it. For example, consider a SCSI controller. Resetting the controller puts all its registers back to what reset state was as well as reset everything on the SCSI bus, whereas resetting just the SCSI bus only resets everything that's on it but not the controller.

Multi-phase mechanism

This section documents the internals of the resettable interface.

The resettable interface uses a multi-phase system to relieve objects and machines from reset ordering problems. To address this, the reset operation of an object is split into three well defined phases.

When resetting several objects (for example the whole machine at simulation startup), all first phases of all objects are executed, then all second phases and then all third phases.

The three phases are:

1. The **enter** phase is executed when the object enters reset. It resets only local state of the object; it must not do anything that has a side-effect on other objects, such as raising or lowering a qemu_irq line or reading or writing guest memory.
2. The **hold** phase is executed for entry into reset, once every object in the group which is being reset has had its *enter* phase executed. At this point devices can do actions that affect other objects.
3. The **exit** phase is executed when the object leaves the reset state. Actions affecting other objects are permitted.

As said in previous section, the interface maintains a count of reset. This count is used to ensure phases are executed only when required. *enter* and *hold* phases are executed only when asserting reset for the first time (if an object is already in reset state when calling `resettable_assert_reset()` or `resettable_reset()`, they are not executed). The *exit* phase is executed only when the last reset operation ends. Therefore the object does not need to care how many of reset controllers it has and how many of them have started a reset.

Handling reset in a resettable object

This section documents the APIs that an implementation of a resettable object must provide and what functions it has access to. It is intended for people who want to implement or convert a class which has the resettable interface; for example when specializing an existing device or bus.

Methods to implement

Three methods should be defined or left empty. Each method corresponds to a phase of the reset; they are name `phases.enter()`, `phases.hold()` and `phases.exit()`. They all take the object as parameter. The *enter* method also take the reset type as second parameter.

When extending an existing class, these methods may need to be extended too. The `resettable_class_set_parent_phases()` class function may be used to backup parent class methods.

Here follows an example to implement reset for a Device which sets an IO while in reset.

```
static void mydev_reset_enter(Object *obj, ResetType type)
{
    MyDevClass *myclass = MYDEV_GET_CLASS(obj);
    MyDevState *mydev = MYDEV(obj);
    /* call parent class enter phase */
    if (myclass->parent_phases.enter) {
```

(continues on next page)

(continued from previous page)

```
        myclass->parent_phases.enter(obj, type);
    }
    /* initialize local state only */
    mydev->var = 0;
}

static void mydev_reset_hold(Object *obj, ResetType type)
{
    MyDevClass *myclass = MYDEV_GET_CLASS(obj);
    MyDevState *mydev = MYDEV(obj);
    /* call parent class hold phase */
    if (myclass->parent_phases.hold) {
        myclass->parent_phases.hold(obj, type);
    }
    /* set an IO */
    qemu_set_irq(mydev->irq, 1);
}

static void mydev_reset_exit(Object *obj, ResetType type)
{
    MyDevClass *myclass = MYDEV_GET_CLASS(obj);
    MyDevState *mydev = MYDEV(obj);
    /* call parent class exit phase */
    if (myclass->parent_phases.exit) {
        myclass->parent_phases.exit(obj, type);
    }
    /* clear an IO */
    qemu_set_irq(mydev->irq, 0);
}

typedef struct MyDevClass {
    MyParentClass parent_class;
    /* to store eventual parent reset methods */
    ResettablePhases parent_phases;
} MyDevClass;

static void mydev_class_init(ObjectClass *class, void *data)
{
    MyDevClass *myclass = MYDEV_CLASS(class);
    ResettableClass *rc = RESETTABLE_CLASS(class);
    resettable_class_set_parent_phases(rc,
                                       mydev_reset_enter,
                                       mydev_reset_hold,
                                       mydev_reset_exit,
                                       &myclass->parent_phases);
}
```

In the above example, we override all three phases. It is possible to override only some of them by passing NULL instead of a function pointer to `resettable_class_set_parent_phases()`. For example, the following will only override the *enter* phase and leave *hold* and *exit* untouched:

```
resettable_class_set_parent_phases(rc, mydev_reset_enter, NULL, NULL,
```

(continues on next page)

(continued from previous page)

```
                                    &myclass->parent_phases);
```

This is equivalent to providing a trivial implementation of the hold and exit phases which does nothing but call the parent class's implementation of the phase.

Polling the reset state

Resettable interface provides the `resettable_is_in_reset()` function. This function returns true if the object parameter is currently under reset.

An object is under reset from the beginning of the *enter* phase (before either its children or its own enter method is called) to the *exit* phase. During *enter* and *hold* phase only, the function will return that the object is in reset. The state is changed after the *exit* is propagated to its children and just before calling the object's own *exit* method.

This function may be used if the object behavior has to be adapted while in reset state. For example if a device has an irq input, it will probably need to ignore it while in reset; then it can for example check the reset state at the beginning of the irq callback.

Note that until migration of the reset state is supported, an object should not be left in reset. So apart from being currently executing one of the reset phases, the only cases when this function will return true is if an external interaction (like changing an io) is made during *hold* or *exit* phase of another object in the same reset group.

Helpers `device_is_in_reset()` and `bus_is_in_reset()` are also provided for devices and buses and should be preferred.

Base class handling of reset

This section documents parts of the reset mechanism that you only need to know about if you are extending it to work with a new base class other than DeviceClass or BusClass, or maintaining the existing code in those classes. Most people can ignore it.

Methods to implement

There are two other methods that need to exist in a class implementing the interface: `get_state()` and `child_foreach()`.

`get_state()` is simple. *resettable* is an interface and, as a consequence, does not have any class state structure. But in order to factorize the code, we need one. This method must return a pointer to `ResettableState` structure. The structure must be allocated by the base class; preferably it should be located inside the object instance structure.

`child_foreach()` is more complex. It should execute the given callback on every reset child of the given resettable object. All children must be resettable too. Additional parameters (a reset type and an opaque pointer) must be passed to the callback too.

In `DeviceClass` and `BusClass` the `ResettableState` is located `DeviceState` and `BusState` structure. `child_foreach()` is implemented to follow the bus hierarchy; for a bus, it calls the function on every child device; for a device, it calls the function on every bus child. When we reset the main system bus, we reset the whole machine bus tree.

Changing a resettable parent

One thing which should be taken care of by the base class is handling reset hierarchy changes.

The reset hierarchy is supposed to be static and built during machine creation. But there are actually some exceptions. To cope with this, the resettable API provides `resettable_change_parent()`. This function allows to set, update or remove the parent of a resettable object after machine creation is done. As parameters, it takes the object being moved, the old parent if any and the new parent if any.

This function can be used at any time when not in a reset operation. During a reset operation it must be used only in *hold* phase. Using it in *enter* or *exit* phase is an error. Also it should not be used during machine creation, although it is harmless to do so: the function is a no-op as long as old and new parent are NULL or not in reset.

There is currently 2 cases where this function is used:

1. *device hotplug*; it means a new device is introduced on a live bus.
2. *hot bus change*; it means an existing live device is added, moved or removed in the bus hierarchy. At the moment, it occurs only in the raspi machines for changing the sdbus used by sd card.

Reset of the complete system

Reset of the complete system is a little complicated. The typical flow is:

1. Code which wishes to reset the entire system does so by calling `qemu_system_reset_request()`. This schedules a reset, but the reset will happen asynchronously after the function returns. That makes this safe to call from, for example, device models.
2. The function which is called to make the reset happen is `qemu_system_reset()`. Generally only core system code should call this directly.
3. `qemu_system_reset()` calls the `MachineClass::reset` method of the current machine, if it has one. That method must call `qemu_devices_reset()`. If the machine has no reset method, `qemu_system_reset()` calls `qemu_devices_reset()` directly.
4. `qemu_devices_reset()` performs a reset of the system, using the three-phase mechanism listed above. It resets all objects that were registered with it using `qemu_register_resettable()`. It also calls all the functions registered with it using `qemu_register_reset()`. Those functions are called during the "hold" phase of this reset.
5. The most important object that this reset resets is the 'sysbus' bus. The sysbus bus is the root of the qbus tree. This means that all devices on the sysbus are reset, and all their child buses, and all the devices on those child buses.
6. Devices which are not on the qbus tree are *not* automatically reset! (The most obvious example of this is CPU objects, but anything that directly inherits from `TYPE_OBJECT` or `TYPE_DEVICE` rather than from `TYPE_SYS_BUS_DEVICE` or some other plugs-into-a-bus type will be in this category.) You need to therefore arrange for these to be reset in some other way (e.g. using `qemu_register_resettable()` or `qemu_register_reset()`).

7.4.9 QAPI interface for S390 CPU topology

The following sections will explain the QAPI interface for S390 CPU topology with the help of exemplary output. For this, let's assume that QEMU has been started with the following command, defining 4 CPUs, where CPU[0] is defined by the -smp argument and will have default values:

```
qemu-system-s390x \
   -enable-kvm \
   -cpu z14,ctop=on \
   -smp 1,drawers=3,books=3,sockets=2,cores=2,maxcpus=36 \
   -device z14-s390x-cpu,core-id=19,entitlement=high \
   -device z14-s390x-cpu,core-id=11,entitlement=low \
   -device z14-s390x-cpu,core-id=12,entitlement=high \
  ...
```

Additions to query-cpus-fast

The command query-cpus-fast allows querying the topology tree and modifiers for all configured vCPUs.

```
{ "execute": "query-cpus-fast" }
{
 "return": [
   {
     "dedicated": false,
     "thread-id": 536993,
     "props": {
       "core-id": 0,
       "socket-id": 0,
       "drawer-id": 0,
       "book-id": 0
     },
     "cpu-state": "operating",
     "entitlement": "medium",
     "qom-path": "/machine/unattached/device[0]",
     "cpu-index": 0,
     "target": "s390x"
   },
   {
     "dedicated": false,
     "thread-id": 537003,
     "props": {
       "core-id": 19,
       "socket-id": 1,
       "drawer-id": 0,
       "book-id": 2
     },
     "cpu-state": "operating",
     "entitlement": "high",
     "qom-path": "/machine/peripheral-anon/device[0]",
     "cpu-index": 19,
     "target": "s390x"
   },
```

(continues on next page)

(continued from previous page)

```
   {
     "dedicated": false,
     "thread-id": 537004,
     "props": {
       "core-id": 11,
       "socket-id": 1,
       "drawer-id": 0,
       "book-id": 1
     },
     "cpu-state": "operating",
     "entitlement": "low",
     "qom-path": "/machine/peripheral-anon/device[1]",
     "cpu-index": 11,
     "target": "s390x"
   },
   {
     "dedicated": true,
     "thread-id": 537005,
     "props": {
       "core-id": 12,
       "socket-id": 0,
       "drawer-id": 3,
       "book-id": 2
     },
     "cpu-state": "operating",
     "entitlement": "high",
     "qom-path": "/machine/peripheral-anon/device[2]",
     "cpu-index": 12,
     "target": "s390x"
   }
 ]
}
```

QAPI command: set-cpu-topology

The command set-cpu-topology allows modifying the topology tree or the topology modifiers of a vCPU in the configuration.

```
{ "execute": "set-cpu-topology",
  "arguments": {
     "core-id": 11,
     "socket-id": 0,
     "book-id": 0,
     "drawer-id": 0,
     "entitlement": "low",
     "dedicated": false
  }
}
{"return": {}}
```

The core-id parameter is the only mandatory parameter and every unspecified parameter keeps its previous value.

QAPI event CPU_POLARIZATION_CHANGE

When a guest requests a modification of the polarization, QEMU sends a CPU_POLARIZATION_CHANGE event.

When requesting the change, the guest only specifies horizontal or vertical polarization. It is the job of the entity administrating QEMU to set the dedication and fine grained vertical entitlement in response to this event.

Note that a vertical polarized dedicated vCPU can only have a high entitlement, giving 6 possibilities for vCPU polarization:

- Horizontal
- Horizontal dedicated
- Vertical low
- Vertical medium
- Vertical high
- Vertical high dedicated

Example of the event received when the guest issues the CPU instruction Perform Topology Function PTF(0) to request an horizontal polarization:

```
{
  "timestamp": {
    "seconds": 1687870305,
    "microseconds": 566299
  },
  "event": "CPU_POLARIZATION_CHANGE",
  "data": {
    "polarization": "horizontal"
  }
}
```

QAPI query command: query-s390x-cpu-polarization

The query command query-s390x-cpu-polarization returns the current CPU polarization of the machine. In this case the guest previously issued a PTF(1) to request vertical polarization:

```
{ "execute": "query-s390x-cpu-polarization" }
{
    "return": {
      "polarization": "vertical"
    }
}
```

7.4.10 Booting from real channel-attached devices on s390x

s390 hardware IPL

The s390 hardware IPL process consists of the following steps.

1. A READ IPL ccw is constructed in memory location `0x0`. This ccw, by definition, reads the IPL1 record which is located on the disk at cylinder 0 track 0 record 1. Note that the chain flag is on in this ccw so when it is complete another ccw will be fetched and executed from memory location `0x08`.
2. Execute the Read IPL ccw at `0x00`, thereby reading IPL1 data into `0x00`. IPL1 data is 24 bytes in length and consists of the following pieces of information: `[psw][read ccw][tic ccw]`. When the machine executes the Read IPL ccw it read the 24-bytes of IPL1 to be read into memory starting at location `0x0`. Then the ccw program at `0x08` which consists of a read ccw and a tic ccw is automatically executed because of the chain flag from the original READ IPL ccw. The read ccw will read the IPL2 data into memory and the TIC (Transfer In Channel) will transfer control to the channel program contained in the IPL2 data. The TIC channel command is the equivalent of a branch/jump/goto instruction for channel programs.

   NOTE: The ccws in IPL1 are defined by the architecture to be format 0.
3. Execute IPL2. The TIC ccw instruction at the end of the IPL1 channel program will begin the execution of the IPL2 channel program. IPL2 is stage-2 of the boot process and will contain a larger channel program than IPL1. The point of IPL2 is to find and load either the operating system or a small program that loads the operating system from disk. At the end of this step all or some of the real operating system is loaded into memory and we are ready to hand control over to the guest operating system. At this point the guest operating system is entirely responsible for loading any more data it might need to function.

   NOTE: The IPL2 channel program might read data into memory location `0x0` thereby overwriting the IPL1 psw and channel program. This is ok as long as the data placed in location `0x0` contains a psw whose instruction address points to the guest operating system code to execute at the end of the IPL/boot process.

   NOTE: The ccws in IPL2 are defined by the architecture to be format 0.
4. Start executing the guest operating system. The psw that was loaded into memory location `0x0` as part of the ipl process should contain the needed flags for the operating system we have loaded. The psw's instruction address will point to the location in memory where we want to start executing the operating system. This psw is loaded (via LPSW instruction) causing control to be passed to the operating system code.

In a non-virtualized environment this process, handled entirely by the hardware, is kicked off by the user initiating a "Load" procedure from the hardware management console. This "Load" procedure crafts a special "Read IPL" ccw in memory location 0x0 that reads IPL1. It then executes this ccw thereby kicking off the reading of IPL1 data. Since the channel program from IPL1 will be written immediately after the special "Read IPL" ccw, the IPL1 channel program will be executed immediately (the special read ccw has the chaining bit turned on). The TIC at the end of the IPL1 channel program will cause the IPL2 channel program to be executed automatically. After this sequence completes the "Load" procedure then loads the psw from `0x0`.

How this all pertains to QEMU (and the kernel)

In theory we should merely have to do the following to IPL/boot a guest operating system from a DASD device:

1. Place a "Read IPL" ccw into memory location `0x0` with chaining bit on.
2. Execute channel program at `0x0`.
3. LPSW `0x0`.

However, our emulation of the machine's channel program logic within the kernel is missing one key feature that is required for this process to work: non-prefetch of ccw data.

When we start a channel program we pass the channel subsystem parameters via an ORB (Operation Request Block). One of those parameters is a prefetch bit. If the bit is on then the vfio-ccw kernel driver is allowed to read the entire channel program from guest memory before it starts executing it. This means that any channel commands that read additional channel commands will not work as expected because the newly read commands will only exist in guest memory and NOT within the kernel's channel subsystem memory. The kernel vfio-ccw driver currently requires this bit to be on for all channel programs. This is a problem because the IPL process consists of transferring control from the "Read IPL" ccw immediately to the IPL1 channel program that was read by "Read IPL".

Not being able to turn off prefetch will also prevent the TIC at the end of the IPL1 channel program from transferring control to the IPL2 channel program.

Lastly, in some cases (the zipl bootloader for example) the IPL2 program also transfers control to another channel program segment immediately after reading it from the disk. So we need to be able to handle this case.

What QEMU does

Since we are forced to live with prefetch we cannot use the very simple IPL procedure we defined in the preceding section. So we compensate by doing the following.

1. Place "Read IPL" ccw into memory location `0x0`, but turn off chaining bit.
2. Execute "Read IPL" at `0x0`.

   So now IPL1's psw is at `0x0` and IPL1's channel program is at `0x08`.
3. Write a custom channel program that will seek to the IPL2 record and then execute the READ and TIC ccws from IPL1. Normally the seek is not required because after reading the IPL1 record the disk is automatically positioned to read the very next record which will be IPL2. But since we are not reading both IPL1 and IPL2 as part of the same channel program we must manually set the position.
4. Grab the target address of the TIC instruction from the IPL1 channel program. This address is where the IPL2 channel program starts.

   Now IPL2 is loaded into memory somewhere, and we know the address.
5. Execute the IPL2 channel program at the address obtained in step #4.

   Because this channel program can be dynamic, we must use a special algorithm that detects a READ immediately followed by a TIC and breaks the ccw chain by turning off the chain bit in the READ ccw. When control is returned from the kernel/hardware to the QEMU bios code we immediately issue another start subchannel to execute the remaining TIC instruction. This causes the entire channel program (starting from the TIC) and all needed data to be refetched thereby stepping around the limitation that would otherwise prevent this channel program from executing properly.

   Now the operating system code is loaded somewhere in guest memory and the psw in memory location `0x0` will point to entry code for the guest operating system.
6. LPSW `0x0`

   LPSW transfers control to the guest operating system and we're done.

7.4.11 Tracing

Introduction

This document describes the tracing infrastructure in QEMU and how to use it for debugging, profiling, and observing execution.

Quickstart

Enable tracing of `memory_region_ops_read` and `memory_region_ops_write` events:

```
$ qemu --trace "memory_region_ops_*" ...
...
719585@1608130130.441188:memory_region_ops_read cpu 0 mr 0x562fdfbb3820 addr 0x3cc value␣
↪0x67 size 1
719585@1608130130.441190:memory_region_ops_write cpu 0 mr 0x562fdfbd2f00 addr 0x3d4␣
↪value 0x70e size 2
```

This output comes from the "log" trace backend that is enabled by default when `./configure --enable-trace-backends=BACKENDS` was not explicitly specified.

Multiple patterns can be specified by repeating the `--trace` option:

```
$ qemu --trace "kvm_*" --trace "virtio_*" ...
```

When patterns are used frequently it is more convenient to store them in a file to avoid long command-line options:

```
$ echo "memory_region_ops_*" >/tmp/events
$ echo "kvm_*" >>/tmp/events
$ qemu --trace events=/tmp/events ...
```

Trace events

Sub-directory setup

Each directory in the source tree can declare a set of trace events in a local "trace-events" file. All directories which contain "trace-events" files must be listed in the "trace_events_subdirs" variable in the top level meson.build file. During build, the "trace-events" file in each listed subdirectory will be processed by the "tracetool" script to generate code for the trace events.

The individual "trace-events" files are merged into a "trace-events-all" file, which is also installed into "/usr/share/qemu". This merged file is to be used by the "simpletrace.py" script to later analyse traces in the simpletrace data format.

The following files are automatically generated in <builddir>/trace/ during the build:

- trace-<subdir>.c - the trace event state declarations
- trace-<subdir>.h - the trace event enums and probe functions
- trace-dtrace-<subdir>.h - DTrace event probe specification
- trace-dtrace-<subdir>.dtrace - DTrace event probe helper declaration
- trace-dtrace-<subdir>.o - binary DTrace provider (generated by dtrace)
- trace-ust-<subdir>.h - UST event probe helper declarations

Here <subdir> is the sub-directory path with '/' replaced by '_'. For example, "accel/kvm" becomes "accel_kvm" and the final filename for "trace-<subdir>.c" becomes "trace-accel_kvm.c".

Source files in the source tree do not directly include generated files in "<builddir>/trace/". Instead they #include the local "trace.h" file, without any sub-directory path prefix. eg io/channel-buffer.c would do:

```
#include "trace.h"
```

The "io/trace.h" file must be created manually with an #include of the corresponding "trace/trace-<subdir>.h" file that will be generated in the builddir:

```
$ echo '#include "trace/trace-io.h"' >io/trace.h
```

While it is possible to include a trace.h file from outside a source file's own sub-directory, this is discouraged in general. It is strongly preferred that all events be declared directly in the sub-directory that uses them. The only exception is where there are some shared trace events defined in the top level directory trace-events file. The top level directory generates trace files with a filename prefix of "trace/trace-root" instead of just "trace". This is to avoid ambiguity between a trace.h in the current directory, vs the top level directory.

Using trace events

Trace events are invoked directly from source code like this:

```
#include "trace.h"  /* needed for trace event prototype */

void *qemu_vmalloc(size_t size)
{
    void *ptr;
    size_t align = QEMU_VMALLOC_ALIGN;

    if (size < align) {
        align = getpagesize();
    }
    ptr = qemu_memalign(align, size);
    trace_qemu_vmalloc(size, ptr);
    return ptr;
}
```

Declaring trace events

The "tracetool" script produces the trace.h header file which is included by every source file that uses trace events. Since many source files include trace.h, it uses a minimum of types and other header files included to keep the namespace clean and compile times and dependencies down.

Trace events should use types as follows:

- Use stdint.h types for fixed-size types. Most offsets and guest memory addresses are best represented with uint32_t or uint64_t. Use fixed-size types over primitive types whose size may change depending on the host (32-bit versus 64-bit) so trace events don't truncate values or break the build.
- Use void * for pointers to structs or for arrays. The trace.h header cannot include all user-defined struct declarations and it is therefore necessary to use void * for pointers to structs.
- For everything else, use primitive scalar types (char, int, long) with the appropriate signedness.

- Avoid floating point types (float and double) because SystemTap does not support them. In most cases it is possible to round to an integer type instead. This may require scaling the value first by multiplying it by 1000 or the like when digits after the decimal point need to be preserved.

Format strings should reflect the types defined in the trace event. Take special care to use PRId64 and PRIu64 for int64_t and uint64_t types, respectively. This ensures portability between 32- and 64-bit platforms. Format strings must not end with a newline character. It is the responsibility of backends to adapt line ending for proper logging.

Each event declaration will start with the event name, then its arguments, finally a format string for pretty-printing. For example:

```
qemu_vmalloc(size_t size, void *ptr) "size %zu ptr %p"
qemu_vfree(void *ptr) "ptr %p"
```

Hints for adding new trace events

1. Trace state changes in the code. Interesting points in the code usually involve a state change like starting, stopping, allocating, freeing. State changes are good trace events because they can be used to understand the execution of the system.
2. Trace guest operations. Guest I/O accesses like reading device registers are good trace events because they can be used to understand guest interactions.
3. Use correlator fields so the context of an individual line of trace output can be understood. For example, trace the pointer returned by malloc and used as an argument to free. This way mallocs and frees can be matched up. Trace events with no context are not very useful.
4. Name trace events after their function. If there are multiple trace events in one function, append a unique distinguisher at the end of the name.

Generic interface and monitor commands

You can programmatically query and control the state of trace events through a backend-agnostic interface provided by the header "trace/control.h".

Note that some of the backends do not provide an implementation for some parts of this interface, in which case QEMU will just print a warning (please refer to header "trace/control.h" to see which routines are backend-dependent).

The state of events can also be queried and modified through monitor commands:

- `info trace-events` View available trace events and their state. State 1 means enabled, state 0 means disabled.
- `trace-event NAME on|off` Enable/disable a given trace event or a group of events (using wildcards).

The "–trace events=<file>" command line argument can be used to enable the events listed in <file> from the very beginning of the program. This file must contain one event name per line.

If a line in the "–trace events=<file>" file begins with a '-', the trace event will be disabled instead of enabled. This is useful when a wildcard was used to enable an entire family of events but one noisy event needs to be disabled.

Wildcard matching is supported in both the monitor command "trace-event" and the events list file. That means you can enable/disable the events having a common prefix in a batch. For example, virtio-blk trace events could be enabled using the following monitor command:

```
trace-event virtio_blk_* on
```

Trace backends

The "tracetool" script automates tedious trace event code generation and also keeps the trace event declarations independent of the trace backend. The trace events are not tightly coupled to a specific trace backend, such as LTTng or SystemTap. Support for trace backends can be added by extending the "tracetool" script.

The trace backends are chosen at configure time:

```
./configure --enable-trace-backends=simple,dtrace
```

For a list of supported trace backends, try ./configure –help or see below. If multiple backends are enabled, the trace is sent to them all.

If no backends are explicitly selected, configure will default to the "log" backend.

The following subsections describe the supported trace backends.

Nop

The "nop" backend generates empty trace event functions so that the compiler can optimize out trace events completely. This imposes no performance penalty.

Note that regardless of the selected trace backend, events with the "disable" property will be generated with the "nop" backend.

Log

The "log" backend sends trace events directly to standard error. This effectively turns trace events into debug printfs.

This is the simplest backend and can be used together with existing code that uses DPRINTF().

The -msg timestamp=on|off command-line option controls whether or not to print the tid/timestamp prefix for each trace event.

Simpletrace

The "simple" backend writes binary trace logs to a file from a thread, making it lower overhead than the "log" backend. A Python API is available for writing offline trace file analysis scripts. It may not be as powerful as platform-specific or third-party trace backends but it is portable and has no special library dependencies.

Monitor commands

- `trace-file on|off|flush|set` <path> Enable/disable/flush the trace file or set the trace file name.

Analyzing trace files

The "simple" backend produces binary trace files that can be formatted with the simpletrace.py script. The script takes the "trace-events-all" file and the binary trace:

```
./scripts/simpletrace.py trace-events-all trace-12345
```

You must ensure that the same "trace-events-all" file was used to build QEMU, otherwise trace event declarations may have changed and output will not be consistent.

Ftrace

The "ftrace" backend writes trace data to ftrace marker. This effectively sends trace events to ftrace ring buffer, and you can compare qemu trace data and kernel(especially kvm.ko when using KVM) trace data.

if you use KVM, enable kvm events in ftrace:

```
# echo 1 > /sys/kernel/debug/tracing/events/kvm/enable
```

After running qemu by root user, you can get the trace:

```
# cat /sys/kernel/debug/tracing/trace
```

Restriction: "ftrace" backend is restricted to Linux only.

Syslog

The "syslog" backend sends trace events using the POSIX syslog API. The log is opened specifying the LOG_DAEMON facility and LOG_PID option (so events are tagged with the pid of the particular QEMU process that generated them). All events are logged at LOG_INFO level.

NOTE: syslog may squash duplicate consecutive trace events and apply rate limiting.

Restriction: "syslog" backend is restricted to POSIX compliant OS.

LTTng Userspace Tracer

The "ust" backend uses the LTTng Userspace Tracer library. There are no monitor commands built into QEMU, instead UST utilities should be used to list, enable/disable, and dump traces.

Package lttng-tools is required for userspace tracing. You must ensure that the current user belongs to the "tracing" group, or manually launch the lttng-sessiond daemon for the current user prior to running any instance of QEMU.

While running an instrumented QEMU, LTTng should be able to list all available events:

```
lttng list -u
```

Create tracing session:

```
lttng create mysession
```

Enable events:

```
lttng enable-event qemu:g_malloc -u
```

Where the events can either be a comma-separated list of events, or "-a" to enable all tracepoint events. Start and stop tracing as needed:

```
lttng start
lttng stop
```

View the trace:

```
lttng view
```

Destroy tracing session:

```
lttng destroy
```

Babeltrace can be used at any later time to view the trace:

```
babeltrace $HOME/lttng-traces/mysession-<date>-<time>
```

SystemTap

The "dtrace" backend uses DTrace sdt probes but has only been tested with SystemTap. When SystemTap support is detected a .stp file with wrapper probes is generated to make use in scripts more convenient. This step can also be performed manually after a build in order to change the binary name in the .stp probes:

```
scripts/tracetool.py --backends=dtrace --format=stap \
                     --binary path/to/qemu-binary \
                     --probe-prefix qemu.system.x86_64 \
                     --group=all \
                     trace-events-all \
                     qemu.stp
```

To facilitate simple usage of systemtap where there merely needs to be printf logging of certain probes, a helper script "qemu-trace-stap" is provided. Consult its manual page for guidance on its usage.

Trace event properties

Each event in the "trace-events-all" file can be prefixed with a space-separated list of zero or more of the following event properties.

"disable"

If a specific trace event is going to be invoked a huge number of times, this might have a noticeable performance impact even when the event is programmatically disabled.

In this case you should declare such event with the "disable" property. This will effectively disable the event at compile time (by using the "nop" backend), thus having no performance impact at all on regular builds (i.e., unless you edit the "trace-events-all" file).

In addition, there might be cases where relatively complex computations must be performed to generate values that are only used as arguments for a trace function. In these cases you can use 'trace_event_get_state_backends()' to guard

such computations, so they are skipped if the event has been either compile-time disabled or run-time disabled. If the event is compile-time disabled, this check will have no performance impact.

```
#include "trace.h"  /* needed for trace event prototype */

void *qemu_vmalloc(size_t size)
{
    void *ptr;
    size_t align = QEMU_VMALLOC_ALIGN;

    if (size < align) {
        align = getpagesize();
    }
    ptr = qemu_memalign(align, size);
    if (trace_event_get_state_backends(TRACE_QEMU_VMALLOC)) {
        void *complex;
        /* some complex computations to produce the 'complex' value */
        trace_qemu_vmalloc(size, ptr, complex);
    }
    return ptr;
}
```

7.4.12 IOMMUFD BACKEND usage with VFIO

(Same meaning for backend/container/BE)

With the introduction of iommufd, the Linux kernel provides a generic interface for user space drivers to propagate their DMA mappings to kernel for assigned devices. While the legacy kernel interface is group-centric, the new iommufd interface is device-centric, relying on device fd and iommufd.

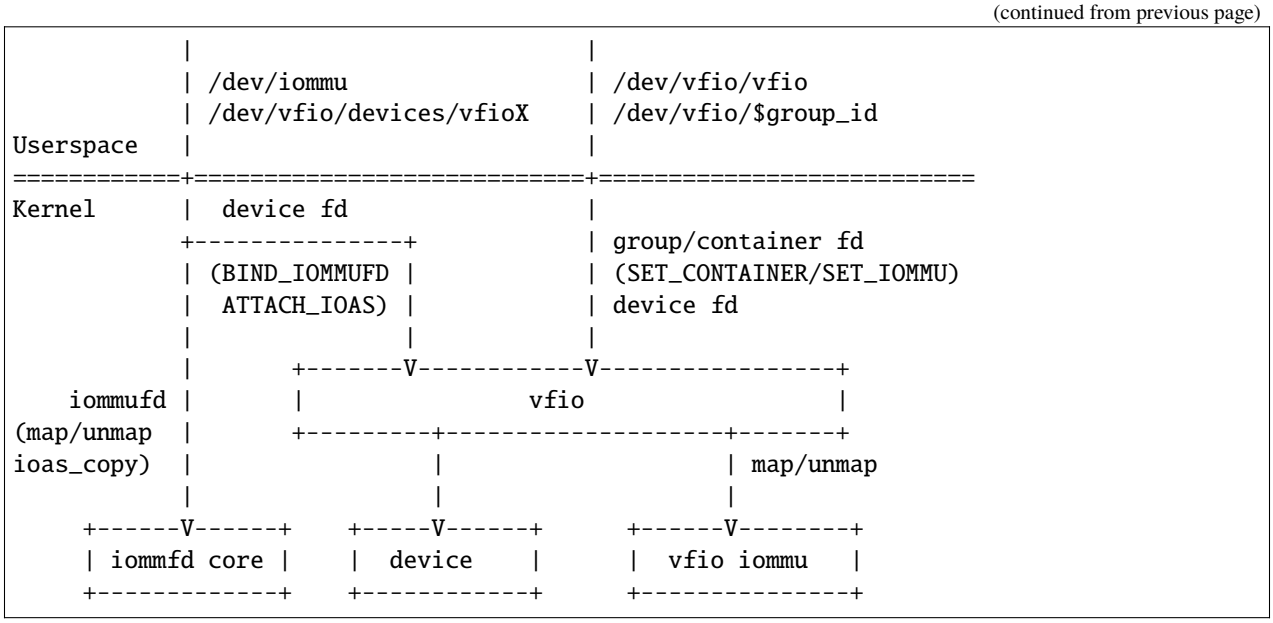
To support both interfaces in the QEMU VFIO device, introduce a base container to abstract the common part of VFIO legacy and iommufd container. So that the generic VFIO code can use either container.

The base container implements generic functions such as memory_listener and address space management whereas the derived container implements callbacks specific to either legacy or iommufd. Each container has its own way to setup secure context and dma management interface. The below diagram shows how it looks like with both containers.

```
                VFIO                           AddressSpace/Memory
    +-------+  +----------+  +-----+  +-----+
    |  pci  |  | platform |  |  ap |  | ccw |
    +---+---+  +----+-----+  +--+--+  +--+--+     +-----------------------+
        |           |           |        |        |   AddressSpace        |
        |           |           |        |        +-----------+-----------+
    +---V-----------V-----------V--------V----+               /
    |           VFIOAddressSpace              | <------------+
    |                  |                      |  MemoryListener
    |        VFIOContainerBase list           |
    +-------+----------------------------+----+
            |                            |
            |                            |
    +-------V------+            +--------V----------+
    |   iommufd    |            |    vfio legacy    |
    |  container   |            |     container     |
    +-------+------+            +---------+---------+
```

(continues on next page)

(continued from previous page)



- Secure Context setup
  - iommufd BE: uses device fd and iommufd to setup secure context (bind_iommufd, attach_ioas)
  - vfio legacy BE: uses group fd and container fd to setup secure context (set_container, set_iommu)
- Device access
  - iommufd BE: device fd is opened through `/dev/vfio/devices/vfioX`
  - vfio legacy BE: device fd is retrieved from group fd ioctl
- DMA Mapping flow
  1. VFIOAddressSpace receives MemoryRegion add/del via MemoryListener
  2. VFIO populates DMA map/unmap via the container BEs * iommufd BE: uses iommufd * vfio legacy BE: uses container fd

Example configuration

Step 1: configure the host device

It's exactly same as the VFIO device with legacy VFIO container.

Step 2: configure QEMU

Interactions with the `/dev/iommu` are abstracted by a new iommufd object (compiled in with the `CONFIG_IOMMUFD` option).

Any QEMU device (e.g. VFIO device) wishing to use `/dev/iommu` must be linked with an iommufd object. It gets a new optional property named iommufd which allows to pass an iommufd object. Take `vfio-pci` device for example:

```
-object iommufd,id=iommufd0
-device vfio-pci,host=0000:02:00.0,iommufd=iommufd0
```

Note the /dev/iommu and VFIO cdev can be externally opened by a management layer. In such a case the fd is passed, the fd supports a string naming the fd or a number, for example:

```
-object iommufd,id=iommufd0,fd=22
-device vfio-pci,iommufd=iommufd0,fd=23
```

If the fd property is not passed, the fd is opened by QEMU.

If no iommufd object is passed to the vfio-pci device, iommufd is not used and the user gets the behavior based on the legacy VFIO container:

```
-device vfio-pci,host=0000:02:00.0
```

Supported platform

Supports x86, ARM and s390x currently.

Caveats

Dirty page sync

Dirty page sync with iommufd backend is unsupported yet, live migration is disabled by default. But it can be force enabled like below, low efficient though.

```
-object iommufd,id=iommufd0
-device vfio-pci,host=0000:02:00.0,iommufd=iommufd0,enable-migration=on
```

P2P DMA

PCI p2p DMA is unsupported as IOMMUFD doesn't support mapping hardware PCI BAR region yet. Below warning shows for assigned PCI device, it's not a bug.

```
qemu-system-x86_64: warning: IOMMU_IOAS_MAP failed: Bad address, PCI BAR?
qemu-system-x86_64: vfio_container_dma_map(0x560cb6cb1620, 0xe000000021000, 0x3000,␣
↪0x7f32ed55c000) = -14 (Bad address)
```

FD passing with mdev

vfio-pci device checks sysfsdev property to decide if backend is a mdev. If FD passing is used, there is no way to know that and the mdev is treated like a real PCI device. There is an error as below if user wants to enable RAM discarding for mdev.

```
qemu-system-x86_64: -device vfio-pci,iommufd=iommufd0,x-balloon-allowed=on,fd=9: vfio␣
↪VFIO_FD9: x-balloon-allowed only potentially compatible with mdev devices
```

vfio-ap and vfio-ccw devices don't have same issue as their backend devices are always mdev and RAM discarding is force enabled.

7.4.13 How to write monitor commands

This document is a step-by-step guide on how to write new QMP commands using the QAPI framework and HMP commands.

This document doesn't discuss QMP protocol level details, nor does it dive into the QAPI framework implementation.

For an in-depth introduction to the QAPI framework, please refer to *How to use the QAPI code generator*. For the QMP protocol, see the *QEMU Machine Protocol Specification*.

New commands may be implemented in QMP only. New HMP commands should be implemented on top of QMP. The typical HMP command wraps around an equivalent QMP command, but HMP convenience commands built from QMP building blocks are also fine. The long term goal is to make all existing HMP commands conform to this, to fully isolate HMP from the internals of QEMU. Refer to the *Writing a debugging aid returning unstructured text* section for further guidance on commands that would have traditionally been HMP only.

Overview

Generally speaking, the following steps should be taken in order to write a new QMP command.

1. Define the command and any types it needs in the appropriate QAPI schema module.
2. Write the QMP command itself, which is a regular C function. Preferably, the command should be exported by some QEMU subsystem. But it can also be added to the monitor/qmp-cmds.c file
3. At this point the command can be tested under the QMP protocol
4. Write the HMP command equivalent. This is not required and should only be done if it does make sense to have the functionality in HMP. The HMP command is implemented in terms of the QMP command

The following sections will demonstrate each of the steps above. We will start very simple and get more complex as we progress.

Testing

For all the examples in the next sections, the test setup is the same and is shown here.

First, QEMU should be started like this:

```
# qemu-system-TARGET [...] \
    -chardev socket,id=qmp,port=4444,host=localhost,server=on \
    -mon chardev=qmp,mode=control,pretty=on
```

Then, in a different terminal:

```
$ telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
{
    "QMP": {
        "version": {
            "qemu": {
                "micro": 50,
                "minor": 2,
                "major": 8
            },
```

(continues on next page)

(continued from previous page)

```
            "package": ...
        },
        "capabilities": [
            "oob"
        ]
    }
}
```

The above output is the QMP server saying you're connected. The server is actually in capabilities negotiation mode. To enter in command mode type:

```
{ "execute": "qmp_capabilities" }
```

Then the server should respond:

```
{
    "return": {
    }
}
```

Which is QMP's way of saying "the latest command executed OK and didn't return any data". Now you're ready to enter the QMP example commands as explained in the following sections.

Writing a simple command: hello-world

That's the most simple QMP command that can be written. Usually, this kind of command carries some meaningful action in QEMU but here it will just print "Hello, world" to the standard output.

Our command will be called "hello-world". It takes no arguments, nor does it return any data.

The first step is defining the command in the appropriate QAPI schema module. We pick module qapi/misc.json, and add the following line at the bottom:

```
##
# @hello-world:
#
# Since: 9.0
##
{ 'command': 'hello-world' }
```

The "command" keyword defines a new QMP command. It instructs QAPI to generate any prototypes and the necessary code to marshal and unmarshal protocol data.

The next step is to write the "hello-world" implementation. As explained earlier, it's preferable for commands to live in QEMU subsystems. But "hello-world" doesn't pertain to any, so we put its implementation in monitor/qmp-cmds.c:

```
void qmp_hello_world(Error **errp)
{
    printf("Hello, world!\n");
}
```

There are a few things to be noticed:

1. QMP command implementation functions must be prefixed with "qmp_"

2. qmp_hello_world() returns void, this is in accordance with the fact that the command doesn't return any data
3. It takes an "Error **" argument. This is required. Later we will see how to return errors and take additional arguments. The Error argument should not be touched if the command doesn't return errors
4. We won't add the function's prototype. That's automatically done by QAPI
5. Printing to the terminal is discouraged for QMP commands, we do it here because it's the easiest way to demonstrate a QMP command

You're done. Now build QEMU, run it as suggested in the "Testing" section, and then type the following QMP command:

```
{ "execute": "hello-world" }
```

Then check the terminal running QEMU and look for the "Hello, world" string. If you don't see it then something went wrong.

Arguments

Let's add arguments to our "hello-world" command.

The first change we have to do is to modify the command specification in the schema file to the following:

```
##
# @hello-world:
#
# @message: message to be printed (default: "Hello, world!")
#
# @times: how many times to print the message (default: 1)
#
# Since: 9.0
##
{ 'command': 'hello-world',
  'data': { '*message': 'str', '*times': 'int' } }
```

Notice the new 'data' member in the schema. It specifies an argument 'message' of QAPI type 'str', and an argument 'times' of QAPI type 'int'. Also notice the asterisk, it's used to mark the argument optional.

Now, let's update our C implementation in monitor/qmp-cmds.c:

```
void qmp_hello_world(const char *message, bool has_times, int64_t times,
                     Error **errp)
{
    if (!message) {
        message = "Hello, world";
    }
    if (!has_times) {
        times = 1;
    }

    for (int i = 0; i < times; i++) {
        printf("%s\n", message);
    }
}
```

There are two important details to be noticed:

1. Optional arguments other than pointers are accompanied by a 'has_' boolean, which is set if the optional argument is present or false otherwise
2. The C implementation signature must follow the schema's argument ordering, which is defined by the "data" member

Time to test our new version of the "hello-world" command. Build QEMU, run it as described in the "Testing" section and then send two commands:

```
{ "execute": "hello-world" }
{
    "return": {
    }
}

{ "execute": "hello-world", "arguments": { "message": "We love QEMU" } }
{
    "return": {
    }
}
```

You should see "Hello, world" and "We love QEMU" in the terminal running QEMU, if you don't see these strings, then something went wrong.

Errors

QMP commands should use the error interface exported by the error.h header file. Basically, most errors are set by calling the error_setg() function.

Let's say we don't accept the string "message" to contain the word "love". If it does contain it, we want the "hello-world" command to return an error:

```
void qmp_hello_world(const char *message, Error **errp)
{
    if (message) {
        if (strstr(message, "love")) {
            error_setg(errp, "the word 'love' is not allowed");
            return;
        }
        printf("%s\n", message);
    } else {
        printf("Hello, world\n");
    }
}
```

The first argument to the error_setg() function is the Error pointer to pointer, which is passed to all QMP functions. The next argument is a human description of the error, this is a free-form printf-like string.

Let's test the example above. Build QEMU, run it as defined in the "Testing" section, and then issue the following command:

```
{ "execute": "hello-world", "arguments": { "message": "all you need is love" } }
```

The QMP server's response should be:

```
{
    "error": {
        "class": "GenericError",
        "desc": "the word 'love' is not allowed"
    }
}
```

Note that error_setg() produces a "GenericError" class. In general, all QMP errors should have that error class. There are two exceptions to this rule:

1. To support a management application's need to recognize a specific error for special handling
2. Backward compatibility

If the failure you want to report falls into one of the two cases above, use error_set() with a second argument of an ErrorClass value.

Implementing the HMP command

Now that the QMP command is in place, we can also make it available in the human monitor (HMP).

With the introduction of QAPI, HMP commands make QMP calls. Most of the time HMP commands are simple wrappers.

Here's the implementation of the "hello-world" HMP command:

```
void hmp_hello_world(Monitor *mon, const QDict *qdict)
{
    const char *message = qdict_get_try_str(qdict, "message");
    Error *err = NULL;

    qmp_hello_world(!!message, message, &err);
    if (hmp_handle_error(mon, err)) {
        return;
    }
}
```

Add it to monitor/hmp-cmds.c. Also, add its prototype to include/monitor/hmp.h.

There are four important points to be noticed:

1. The "mon" and "qdict" arguments are mandatory for all HMP functions. The former is the monitor object. The latter is how the monitor passes arguments entered by the user to the command implementation
2. We chose not to support the "times" argument in HMP
3. hmp_hello_world() performs error checking. In this example we just call hmp_handle_error() which prints a message to the user, but we could do more, like taking different actions depending on the error qmp_hello_world() returns
4. The "err" variable must be initialized to NULL before performing the QMP call

There's one last step to actually make the command available to monitor users, we should add it to the hmp-commands.hx file:

```
   {
        .name       = "hello-world",
```

(continues on next page)

(continued from previous page)

```
       .args_type  = "message:s?",
       .params     = "hello-world [message]",
       .help       = "Print message to the standard output",
       .cmd        = hmp_hello_world,
   },

SRST
``hello_world`` *message*
  Print message to the standard output
ERST
```

To test this you have to open a user monitor and issue the "hello-world" command. It might be instructive to check the command's documentation with HMP's "help" command.

Please check the "-monitor" command-line option to know how to open a user monitor.

Writing more complex commands

A QMP command is capable of returning any data QAPI supports like integers, strings, booleans, enumerations and user defined types.

In this section we will focus on user defined types. Please check the QAPI documentation for information about the other types.

Modelling data in QAPI

For a QMP command that to be considered stable and supported long term, there is a requirement returned data should be explicitly modelled using fine-grained QAPI types. As a general guide, a caller of the QMP command should never need to parse individual returned data fields. If a field appears to need parsing, then it should be split into separate fields corresponding to each distinct data item. This should be the common case for any new QMP command that is intended to be used by machines, as opposed to exclusively human operators.

Some QMP commands, however, are only intended as ad hoc debugging aids for human operators. While they may return large amounts of formatted data, it is not expected that machines will need to parse the result. The overhead of defining a fine grained QAPI type for the data may not be justified by the potential benefit. In such cases, it is permitted to have a command return a simple string that contains formatted data, however, it is mandatory for the command to be marked unstable. This indicates that the command is not guaranteed to be long term stable / liable to change in future and is not following QAPI design best practices. An example where this approach is taken is the QMP command "x-query-registers". This returns a formatted dump of the architecture specific CPU state. The way the data is formatted varies across QEMU targets, is liable to change over time, and is only intended to be consumed as an opaque string by machines. Refer to the *Writing a debugging aid returning unstructured text* section for an illustration.

User Defined Types

For this example we will write the query-option-roms command, which returns information about ROMs loaded into the option ROM space. For more information about it, please check the "-option-rom" command-line option.

For each option ROM, we want to return two pieces of information: the ROM image's file name, and its bootindex, if any. We need to create a new QAPI type for that, as shown below:

```
##
# @OptionRomInfo:
#
# @filename: option ROM image file name
#
# @bootindex: option ROM's bootindex
#
# Since: 9.0
##
{ 'struct': 'OptionRomInfo',
  'data': { 'filename': 'str', '*bootindex': 'int' } }
```

The "struct" keyword defines a new QAPI type. Its "data" member contains the type's members. In this example our members are "filename" and "bootindex". The latter is optional.

Now let's define the query-option-roms command:

```
##
# @query-option-roms:
#
# Query information on ROMs loaded into the option ROM space.
#
# Returns: OptionRomInfo
#
# Since: 9.0
##
{ 'command': 'query-option-roms',
  'returns': ['OptionRomInfo'] }
```

Notice the "returns" keyword. As its name suggests, it's used to define the data returned by a command.

Notice the syntax ['OptionRomInfo']". This should be read as "returns a list of OptionRomInfo".

It's time to implement the qmp_query_option_roms() function. Add to monitor/qmp-cmds.c:

```
OptionRomInfoList *qmp_query_option_roms(Error **errp)
{
    OptionRomInfoList *info_list = NULL;
    OptionRomInfoList **tailp = &info_list;
    OptionRomInfo *info;

    for (int i = 0; i < nb_option_roms; i++) {
        info = g_malloc0(sizeof(*info));
        info->filename = g_strdup(option_rom[i].name);
        info->has_bootindex = option_rom[i].bootindex >= 0;
        if (info->has_bootindex) {
            info->bootindex = option_rom[i].bootindex;
```

(continues on next page)

(continued from previous page)

```
        }
        QAPI_LIST_APPEND(tailp, info);
    }

    return info_list;
}
```

There are a number of things to be noticed:

1. Type OptionRomInfo is automatically generated by the QAPI framework, its members correspond to the type's specification in the schema file
2. Type OptionRomInfoList is also generated. It's a singly linked list.
3. As specified in the schema file, the function returns a OptionRomInfoList, and takes no arguments (besides the "errp" one, which is mandatory for all QMP functions)
4. The returned object is dynamically allocated
5. All strings are dynamically allocated. This is so because QAPI also generates a function to free its types and it cannot distinguish between dynamically or statically allocated strings
6. Remember that "bootindex" is optional? As a non-pointer optional member, it comes with a 'has_bootindex' member that needs to be set by the implementation, as shown above

Time to test the new command. Build QEMU, run it as described in the "Testing" section and try this:

```
{ "execute": "query-option-rom" }
{
    "return": [
        {
            "filename": "kvmvapic.bin"
        }
    ]
}
```

The HMP command

Here's the HMP counterpart of the query-option-roms command:

```
void hmp_info_option_roms(Monitor *mon, const QDict *qdict)
{
    Error *err = NULL;
    OptionRomInfoList *info_list, *tail;
    OptionRomInfo *info;

    info_list = qmp_query_option_roms(&err);
    if (hmp_handle_error(mon, err)) {
        return;
    }

    for (tail = info_list; tail; tail = tail->next) {
        info = tail->value;
        monitor_printf(mon, "%s", info->filename);
```

(continues on next page)

(continued from previous page)

```
        if (info->has_bootindex) {
            monitor_printf(mon, " %" PRId64, info->bootindex);
        }
        monitor_printf(mon, "\n");
    }

    qapi_free_OptionRomInfoList(info_list);
}
```

It's important to notice that hmp_info_option_roms() calls qapi_free_OptionRomInfoList() to free the data returned by qmp_query_option_roms(). For user defined types, QAPI will generate a qapi_free_QAPI_TYPE_NAME() function, and that's what you have to use to free the types you define and qapi_free_QAPI_TYPE_NAMEList() for list types (explained in the next section). If the QMP function returns a string, then you should g_free() to free it.

Also note that hmp_info_option_roms() performs error handling. That's not strictly required when you're sure the QMP function doesn't return errors; you could instead pass it &error_abort then.

Another important detail is that HMP's "info" commands go into hmp-commands-info.hx, not hmp-commands.hx. The entry for the "info option-roms" follows:

```
    {
        .name       = "option-roms",
        .args_type  = "",
        .params     = "",
        .help       = "show roms",
        .cmd        = hmp_info_option_roms,
    },
SRST
``info option-roms``
  Show the option ROMs.
ERST
```

To test this, run QEMU and type "info option-roms" in the user monitor.

Writing a debugging aid returning unstructured text

As discussed in section *Modelling data in QAPI*, it is required that commands expecting machine usage be using fine-grained QAPI data types. The exception to this rule applies when the command is solely intended as a debugging aid and allows for returning unstructured text, such as a query command that report aspects of QEMU's internal state that are useful only to human operators.

In this example we will consider the existing QMP command `x-query-roms` in qapi/machine.json. It has no parameters and returns a `HumanReadableText`:

```
##
# @x-query-roms:
#
# Query information on the registered ROMS
#
# Features:
#
# @unstable: This command is meant for debugging.
#
```

(continues on next page)

(continued from previous page)

```
# Returns: registered ROMs
#
# Since: 6.2
##
{ 'command': 'x-query-roms',
  'returns': 'HumanReadableText',
  'features': [ 'unstable' ] }
```

The `HumanReadableText` struct is defined in qapi/common.json as a struct with a string member. It is intended to be used for all commands that are returning unstructured text targeted at humans. These should all have feature 'unstable'. Note that the feature's documentation states why the command is unstable. We commonly use a `x-` command name prefix to make lack of stability obvious to human users.

Implementing the QMP command

The QMP implementation will typically involve creating a `GString` object and printing formatted data into it, like this:

```
HumanReadableText *qmp_x_query_roms(Error **errp)
{
    g_autoptr(GString) buf = g_string_new("");
    Rom *rom;

    QTAILQ_FOREACH(rom, &roms, next) {
       g_string_append_printf("%s size=0x%06zx name=\"%s\"\n",
                              memory_region_name(rom->mr),
                              rom->romsize,
                              rom->name);
    }

    return human_readable_text_from_str(buf);
}
```

The actual implementation emits more information. You can find it in hw/core/loader.c.

Implementing the HMP command

Now that the QMP command is in place, we can also make it available in the human monitor (HMP) as shown in previous examples. The HMP implementations will all look fairly similar, as all they need do is invoke the QMP command and then print the resulting text or error message. Here's an implementation of the "info roms" HMP command:

```
void hmp_info_roms(Monitor *mon, const QDict *qdict)
{
    Error err = NULL;
    g_autoptr(HumanReadableText) info = qmp_x_query_roms(&err);

    if (hmp_handle_error(mon, err)) {
        return;
    }
    monitor_puts(mon, info->human_readable_text);
}
```

Also, you have to add the function's prototype to the hmp.h file.

There's one last step to actually make the command available to monitor users, we should add it to the hmp-commands-info.hx file:

```
{
    .name       = "roms",
    .args_type  = "",
    .params     = "",
    .help       = "show roms",
    .cmd        = hmp_info_roms,
},
```

The case of writing a HMP info handler that calls a no-parameter QMP query command is quite common. To simplify the implementation there is a general purpose HMP info handler for this scenario. All that is required to expose a no-parameter QMP query command via HMP is to declare it using the '.cmd_info_hrt' field to point to the QMP handler, and leave the '.cmd' field NULL:

```
{
    .name         = "roms",
    .args_type    = "",
    .params       = "",
    .help         = "show roms",
    .cmd_info_hrt = qmp_x_query_roms,
},
```

This is how the actual HMP command is done.

7.4.14 Writing VirtIO backends for QEMU

This document attempts to outline the information a developer needs to know to write device emulations in QEMU. It is specifically focused on implementing VirtIO devices. For VirtIO the frontend is the driver running on the guest. The backend is the everything that QEMU needs to do to handle the emulation of the VirtIO device. This can be done entirely in QEMU, divided between QEMU and the kernel (vhost) or handled by a separate process which is configured by QEMU (vhost-user).

VirtIO Transports

VirtIO supports a number of different transports. While the details of the configuration and operation of the device will generally be the same QEMU represents them as different devices depending on the transport they use. For example -device virtio-foo represents the foo device using mmio and -device virtio-foo-pci is the same class of device using the PCI transport.

Using the QEMU Object Model (QOM)

Generally all devices in QEMU are super classes of `TYPE_DEVICE` however VirtIO devices should be based on `TYPE_VIRTIO_DEVICE` which itself is derived from the base class. For example:

```
static const TypeInfo virtio_blk_info = {
    .name = TYPE_VIRTIO_BLK,
    .parent = TYPE_VIRTIO_DEVICE,
    .instance_size = sizeof(VirtIOBlock),
    .instance_init = virtio_blk_instance_init,
    .class_init = virtio_blk_class_init,
};
```

The author may decide to have a more expansive class hierarchy to support multiple device types. For example the Virtio GPU device:

```
static const TypeInfo virtio_gpu_base_info = {
    .name = TYPE_VIRTIO_GPU_BASE,
    .parent = TYPE_VIRTIO_DEVICE,
    .instance_size = sizeof(VirtIOGPUBase),
    .class_size = sizeof(VirtIOGPUBaseClass),
    .class_init = virtio_gpu_base_class_init,
    .abstract = true
};

static const TypeInfo vhost_user_gpu_info = {
    .name = TYPE_VHOST_USER_GPU,
    .parent = TYPE_VIRTIO_GPU_BASE,
    .instance_size = sizeof(VhostUserGPU),
    .instance_init = vhost_user_gpu_instance_init,
    .instance_finalize = vhost_user_gpu_instance_finalize,
    .class_init = vhost_user_gpu_class_init,
};

static const TypeInfo virtio_gpu_info = {
    .name = TYPE_VIRTIO_GPU,
    .parent = TYPE_VIRTIO_GPU_BASE,
    .instance_size = sizeof(VirtIOGPU),
    .class_size = sizeof(VirtIOGPUClass),
    .class_init = virtio_gpu_class_init,
};
```

defines a base class for the VirtIO GPU and then specialises two versions, one for the internal implementation and the other for the vhost-user version.

VirtIOPCIProxy

[AJB: the following is supposition and welcomes more informed opinions]

Probably due to legacy from the pre-QOM days PCI VirtIO devices don't follow the normal hierarchy. Instead the a standalone object is based on the VirtIOPCIProxy class and the specific VirtIO instance is manually instantiated:

```
/*
 * virtio-blk-pci: This extends VirtioPCIProxy.
 */
#define TYPE_VIRTIO_BLK_PCI "virtio-blk-pci-base"
DECLARE_INSTANCE_CHECKER(VirtIOBlkPCI, VIRTIO_BLK_PCI,
                         TYPE_VIRTIO_BLK_PCI)

struct VirtIOBlkPCI {
    VirtIOPCIProxy parent_obj;
    VirtIOBlock vdev;
};

static Property virtio_blk_pci_properties[] = {
    DEFINE_PROP_UINT32("class", VirtIOPCIProxy, class_code, 0),
    DEFINE_PROP_BIT("ioeventfd", VirtIOPCIProxy, flags,
                    VIRTIO_PCI_FLAG_USE_IOEVENTFD_BIT, true),
    DEFINE_PROP_UINT32("vectors", VirtIOPCIProxy, nvectors,
                       DEV_NVECTORS_UNSPECIFIED),
    DEFINE_PROP_END_OF_LIST(),
};

static void virtio_blk_pci_realize(VirtIOPCIProxy *vpci_dev, Error **errp)
{
    VirtIOBlkPCI *dev = VIRTIO_BLK_PCI(vpci_dev);
    DeviceState *vdev = DEVICE(&dev->vdev);

    ...

    qdev_realize(vdev, BUS(&vpci_dev->bus), errp);
}

static void virtio_blk_pci_class_init(ObjectClass *klass, void *data)
{
    DeviceClass *dc = DEVICE_CLASS(klass);
    VirtioPCIClass *k = VIRTIO_PCI_CLASS(klass);
    PCIDeviceClass *pcidev_k = PCI_DEVICE_CLASS(klass);

    set_bit(DEVICE_CATEGORY_STORAGE, dc->categories);
    device_class_set_props(dc, virtio_blk_pci_properties);
    k->realize = virtio_blk_pci_realize;
    pcidev_k->vendor_id = PCI_VENDOR_ID_REDHAT_QUMRANET;
    pcidev_k->device_id = PCI_DEVICE_ID_VIRTIO_BLOCK;
    pcidev_k->revision = VIRTIO_PCI_ABI_VERSION;
    pcidev_k->class_id = PCI_CLASS_STORAGE_SCSI;
}
```

(continues on next page)

(continued from previous page)

```
static void virtio_blk_pci_instance_init(Object *obj)
{
    VirtIOBlkPCI *dev = VIRTIO_BLK_PCI(obj);

    virtio_instance_init_common(obj, &dev->vdev, sizeof(dev->vdev),
                                TYPE_VIRTIO_BLK);
    object_property_add_alias(obj, "bootindex", OBJECT(&dev->vdev),
                              "bootindex");
}

static const VirtioPCIDeviceTypeInfo virtio_blk_pci_info = {
    .base_name              = TYPE_VIRTIO_BLK_PCI,
    .generic_name           = "virtio-blk-pci",
    .transitional_name      = "virtio-blk-pci-transitional",
    .non_transitional_name  = "virtio-blk-pci-non-transitional",
    .instance_size = sizeof(VirtIOBlkPCI),
    .instance_init = virtio_blk_pci_instance_init,
    .class_init    = virtio_blk_pci_class_init,
};
```

Here you can see the instance_init has to manually instantiate the underlying `TYPE_VIRTIO_BLOCK` object and link an alias for one of it's properties to the PCI device.

Back End Implementations

There are a number of places where the implementation of the backend can be done:

- in QEMU itself
- in the host kernel (a.k.a vhost)
- in a separate process (a.k.a. vhost-user)

vhost_ops vs TYPE_VHOST_USER_BACKEND

There are two choices to how to implement vhost code. Most of the code which has to work with either vhost or vhost-user uses `vhost_dev_init()` to instantiate the appropriate backend. This means including a `struct vhost_dev` in the main object structure.

For vhost-user devices you also need to add code to track the initialisation of the `chardev` device used for the control socket between QEMU and the external vhost-user process.

If you only need to implement a vhost-user backed the other option is a use a QOM-ified version of vhost-user.

```
static void
vhost_user_gpu_instance_init(Object *obj)
{
    VhostUserGPU *g = VHOST_USER_GPU(obj);

    g->vhost = VHOST_USER_BACKEND(object_new(TYPE_VHOST_USER_BACKEND));
    object_property_add_alias(obj, "chardev",
                              OBJECT(g->vhost), "chardev");
}
```

(continues on next page)

(continued from previous page)

```
static const TypeInfo vhost_user_gpu_info = {
    .name = TYPE_VHOST_USER_GPU,
    .parent = TYPE_VIRTIO_GPU_BASE,
    .instance_size = sizeof(VhostUserGPU),
    .instance_init = vhost_user_gpu_instance_init,
    .instance_finalize = vhost_user_gpu_instance_finalize,
    .class_init = vhost_user_gpu_class_init,
};
```

Using it this way entails adding a `struct VhostUserBackend` to your core object structure and manually instantiating the backend. This sub-structure tracks both the `vhost_dev` and `CharDev` types needed for the connection. Instead of calling `vhost_dev_init` you would call `vhost_user_backend_dev_init` which does what is needed on your behalf.

7.5 TCG Emulation

Details about QEMU's Tiny Code Generator and the infrastructure associated with emulation. You do not need to worry about this if you are only implementing things for HW accelerated hypervisors.

7.5.1 Translator Internals

QEMU is a dynamic translator. When it first encounters a piece of code, it converts it to the host instruction set. Usually dynamic translators are very complicated and highly CPU dependent. QEMU uses some tricks which make it relatively easily portable and simple while achieving good performances.

QEMU's dynamic translation backend is called TCG, for "Tiny Code Generator". For more information, please take a look at *TCG Intermediate Representation*.

The following sections outline some notable features and implementation details of QEMU's dynamic translator.

CPU state optimisations

The target CPUs have many internal states which change the way they evaluate instructions. In order to achieve a good speed, the translation phase considers that some state information of the virtual CPU cannot change in it. The state is recorded in the Translation Block (TB). If the state changes (e.g. privilege level), a new TB will be generated and the previous TB won't be used anymore until the state matches the state recorded in the previous TB. The same idea can be applied to other aspects of the CPU state. For example, on x86, if the SS, DS and ES segments have a zero base, then the translator does not even generate an addition for the segment base.

Direct block chaining

After each translated basic block is executed, QEMU uses the simulated Program Counter (PC) and other CPU state information (such as the CS segment base value) to find the next basic block.

In its simplest, less optimized form, this is done by exiting from the current TB, going through the TB epilogue, and then back to the main loop. That's where QEMU looks for the next TB to execute, translating it from the guest architecture if it isn't already available in memory. Then QEMU proceeds to execute this next TB, starting at the prologue and then moving on to the translated instructions.

Exiting from the TB this way will cause the `cpu_exec_interrupt()` callback to be re-evaluated before executing additional instructions. It is mandatory to exit this way after any CPU state changes that may unmask interrupts.

In order to accelerate the cases where the TB for the new simulated PC is already available, QEMU has mechanisms that allow multiple TBs to be chained directly, without having to go back to the main loop as described above. These mechanisms are:

lookup_and_goto_ptr

Calling `tcg_gen_lookup_and_goto_ptr()` will emit a call to `helper_lookup_tb_ptr`. This helper will look for an existing TB that matches the current CPU state. If the destination TB is available its code address is returned, otherwise the address of the JIT epilogue is returned. The call to the helper is always followed by the tcg `goto_ptr` opcode, which branches to the returned address. In this way, we either branch to the next TB or return to the main loop.

goto_tb + exit_tb

The translation code usually implements branching by performing the following steps:

1. Call `tcg_gen_goto_tb()` passing a jump slot index (either 0 or 1) as a parameter.
2. Emit TCG instructions to update the CPU state with any information that has been assumed constant and is required by the main loop to correctly locate and execute the next TB. For most guests, this is just the PC of the branch destination, but others may store additional data. The information updated in this step must be inferable from both `cpu_get_tb_cpu_state()` and `cpu_restore_state()`.
3. Call `tcg_gen_exit_tb()` passing the address of the current TB and the jump slot index again.

Step 1, `tcg_gen_goto_tb()`, will emit a `goto_tb` TCG instruction that later on gets translated to a jump to an address associated with the specified jump slot. Initially, this is the address of step 2's instructions, which update the CPU state information. Step 3, `tcg_gen_exit_tb()`, exits from the current TB returning a tagged pointer composed of the last executed TB's address and the jump slot index.

The first time this whole sequence is executed, step 1 simply jumps to step 2. Then the CPU state information gets updated and we exit from the current TB. As a result, the behavior is very similar to the less optimized form described earlier in this section.

Next, the main loop looks for the next TB to execute using the current CPU state information (creating the TB if it wasn't already available) and, before starting to execute the new TB's instructions, patches the previously executed TB by associating one of its jump slots (the one specified in the call to `tcg_gen_exit_tb()`) with the address of the new TB.

The next time this previous TB is executed and we get to that same `goto_tb` step, it will already be patched (assuming the destination TB is still in memory) and will jump directly to the first instruction of the destination TB, without going back to the main loop.

For the `goto_tb + exit_tb` mechanism to be used, the following conditions need to be satisfied:

- The change in CPU state must be constant, e.g., a direct branch and not an indirect branch.

- The direct branch cannot cross a page boundary. Memory mappings may change, causing the code at the destination address to change.

Note that, on step 3 (`tcg_gen_exit_tb()`), in addition to the jump slot index, the address of the TB just executed is also returned. This address corresponds to the TB that will be patched; it may be different than the one that was directly executed from the main loop if the latter had already been chained to other TBs.

Self-modifying code and translated code invalidation

Self-modifying code is a special challenge in x86 emulation because no instruction cache invalidation is signaled by the application when code is modified.

User-mode emulation marks a host page as write-protected (if it is not already read-only) every time translated code is generated for a basic block. Then, if a write access is done to the page, Linux raises a SEGV signal. QEMU then invalidates all the translated code in the page and enables write accesses to the page. For system emulation, write protection is achieved through the software MMU.

Correct translated code invalidation is done efficiently by maintaining a linked list of every translated block contained in a given page. Other linked lists are also maintained to undo direct block chaining.

On RISC targets, correctly written software uses memory barriers and cache flushes, so some of the protection above would not be necessary. However, QEMU still requires that the generated code always matches the target instructions in memory in order to handle exceptions correctly.

Exception support

longjmp() is used when an exception such as division by zero is encountered.

The host SIGSEGV and SIGBUS signal handlers are used to get invalid memory accesses. QEMU keeps a map from host program counter to target program counter, and looks up where the exception happened based on the host program counter at the exception point.

On some targets, some bits of the virtual CPU's state are not flushed to the memory until the end of the translation block. This is done for internal emulation state that is rarely accessed directly by the program and/or changes very often throughout the execution of a translation block—this includes condition codes on x86, delay slots on SPARC, conditional execution on Arm, and so on. This state is stored for each target instruction, and looked up on exceptions.

MMU emulation

For system emulation QEMU uses a software MMU. In that mode, the MMU virtual to physical address translation is done at every memory access.

QEMU uses an address translation cache (TLB) to speed up the translation. In order to avoid flushing the translated code each time the MMU mappings change, all caches in QEMU are physically indexed. This means that each basic block is indexed with its physical address.

In order to avoid invalidating the basic block chain when MMU mappings change, chaining is only performed when the destination of the jump shares a page with the basic block that is performing the jump.

The MMU can also distinguish RAM and ROM memory areas from MMIO memory areas. Access is faster for RAM and ROM because the translation cache also hosts the offset between guest address and host memory. Accessing MMIO memory areas instead calls out to C code for device emulation. Finally, the MMU helps tracking dirty pages and pages pointed to by translation blocks.

Profiling JITted code

The Linux `perf` tool will treat all JITted code as a single block as unlike the main code it can't use debug information to link individual program counter samples with larger functions. To overcome this limitation you can use the `-perfmap` or the `-jitdump` option to generate map files. `-perfmap` is lightweight and produces only guest-host mappings. `-jitdump` additionally saves JITted code and guest debug information (if available); its output needs to be integrated with the `perf.data` file before the final report can be viewed.

```
perf record $QEMU -perfmap $REMAINING_ARGS
perf report

perf record -k 1 $QEMU -jitdump $REMAINING_ARGS
DEBUGINFOD_URLS= perf inject -j -i perf.data -o perf.data.jitted
perf report -i perf.data.jitted
```

Note that qemu-system generates mappings only for `-kernel` files in ELF format.

7.5.2 TCG Intermediate Representation

Introduction

TCG (Tiny Code Generator) began as a generic backend for a C compiler. It was simplified to be used in QEMU. It also has its roots in the QOP code generator written by Paul Brook.

Definitions

The TCG *target* is the architecture for which we generate the code. It is of course not the same as the "target" of QEMU which is the emulated architecture. As TCG started as a generic C backend used for cross compiling, the assumption was that TCG target might be different from the host, although this is never the case for QEMU.

In this document, we use *guest* to specify what architecture we are emulating; *target* always means the TCG target, the machine on which we are running QEMU.

An operation with *undefined behavior* may result in a crash.

An operation with *unspecified behavior* shall not crash. However, the result may be one of several possibilities so may be considered an *undefined result*.

Basic Blocks

A TCG *basic block* is a single entry, multiple exit region which corresponds to a list of instructions terminated by a label, or any branch instruction.

A TCG *extended basic block* is a single entry, multiple exit region which corresponds to a list of instructions terminated by a label or an unconditional branch. Specifically, an extended basic block is a sequence of basic blocks connected by the fall-through paths of zero or more conditional branch instructions.

Operations

TCG instructions or *ops* operate on TCG *variables*, both of which are strongly typed. Each instruction has a fixed number of output variable operands, input variable operands and constant operands. Vector instructions have a field specifying the element size within the vector. The notable exception is the call instruction which has a variable number of outputs and inputs.

In the textual form, output operands usually come first, followed by input operands, followed by constant operands. The output type is included in the instruction name. Constants are prefixed with a '$'.

```
add_i32 t0, t1, t2    /* (t0 <- t1 + t2) */
```

Variables

- `TEMP_FIXED`

  There is one TCG *fixed global* variable, `cpu_env`, which is live in all translation blocks, and holds a pointer to `CPUArchState`. This variable is held in a host cpu register at all times in all translation blocks.

- `TEMP_GLOBAL`

  A TCG *global* is a variable which is live in all translation blocks, and corresponds to memory location that is within `CPUArchState`. These may be specified as an offset from `cpu_env`, in which case they are called *direct globals*, or may be specified as an offset from a direct global, in which case they are called *indirect globals*. Even indirect globals should still reference memory within `CPUArchState`. All TCG globals are defined during `TCGCPUOps.initialize`, before any translation blocks are generated.

- `TEMP_CONST`

  A TCG *constant* is a variable which is live throughout the entire translation block, and contains a constant value. These variables are allocated on demand during translation and are hashed so that there is exactly one variable holding a given value.

- `TEMP_TB`

  A TCG *translation block temporary* is a variable which is live throughout the entire translation block, but dies on any exit. These temporaries are allocated explicitly during translation.

- `TEMP_EBB`

  A TCG *extended basic block temporary* is a variable which is live throughout an extended basic block, but dies on any exit. These temporaries are allocated explicitly during translation.

Types

- `TCG_TYPE_I32`

  A 32-bit integer.

- `TCG_TYPE_I64`

  A 64-bit integer. For 32-bit hosts, such variables are split into a pair of variables with `type=TCG_TYPE_I32` and `base_type=TCG_TYPE_I64`. The `temp_subindex` for each indicates where it falls within the host-endian representation.

- `TCG_TYPE_PTR`

  An alias for `TCG_TYPE_I32` or `TCG_TYPE_I64`, depending on the size of a pointer for the host.

- TCG_TYPE_REG

  An alias for TCG_TYPE_I32 or TCG_TYPE_I64, depending on the size of the integer registers for the host. This may be larger than TCG_TYPE_PTR depending on the host ABI.

- TCG_TYPE_I128

  A 128-bit integer. For all hosts, such variables are split into a number of variables with type=TCG_TYPE_REG and base_type=TCG_TYPE_I128. The temp_subindex for each indicates where it falls within the host-endian representation.

- TCG_TYPE_V64

  A 64-bit vector. This type is valid only if the TCG target sets TCG_TARGET_HAS_v64.

- TCG_TYPE_V128

  A 128-bit vector. This type is valid only if the TCG target sets TCG_TARGET_HAS_v128.

- TCG_TYPE_V256

  A 256-bit vector. This type is valid only if the TCG target sets TCG_TARGET_HAS_v256.

Helpers

Helpers are registered in a guest-specific helper.h, which is processed to generate tcg_gen_helper_* functions. With these functions it is possible to call a function taking i32, i64, i128 or pointer types.

By default, before calling a helper, all globals are stored at their canonical location. By default, the helper is allowed to modify the CPU state (including the state represented by tcg globals) or may raise an exception. This default can be overridden using the following function modifiers:

- TCG_CALL_NO_WRITE_GLOBALS

  The helper does not modify any globals, but may read them. Globals will be saved to their canonical location before calling helpers, but need not be reloaded afterwards.

- TCG_CALL_NO_READ_GLOBALS

  The helper does not read globals, either directly or via an exception. They will not be saved to their canonical locations before calling the helper. This implies TCG_CALL_NO_WRITE_GLOBALS.

- TCG_CALL_NO_SIDE_EFFECTS

  The call to the helper function may be removed if the return value is not used. This means that it may not modify any CPU state nor may it raise an exception.

Code Optimizations

When generating instructions, you can count on at least the following optimizations:

- Single instructions are simplified, e.g.

```
and_i32 t0, t0, $0xffffffff
```

  is suppressed.

- A liveness analysis is done at the basic block level. The information is used to suppress moves from a dead variable to another one. It is also used to remove instructions which compute dead results. The later is especially useful for condition code optimization in QEMU.

  In the following example:

```
add_i32 t0, t1, t2
add_i32 t0, t0, $1
mov_i32 t0, $1
```

only the last instruction is kept.

Instruction Reference

Function call

call <*ret*> <*params*> ptr	call function 'ptr' (pointer type) <*ret*> optional 32 bit or 64 bit return value <*params*> optional 32 bit or 64 bit parameters

Jumps/Labels

set_label $label	Define label 'label' at the current program point.
br $label	Jump to label.

brcond_i32/i64 *t0*, *t1*, *cond*, *label*	Conditional jump if *t0 cond t1* is true. *cond* can be:
	
	`TCG_COND_EQ`
	`TCG_COND_NE`
	`TCG_COND_LT /* signed */`
	`TCG_COND_GE /* signed */`
	`TCG_COND_LE /* signed */`
	`TCG_COND_GT /* signed */`
	`TCG_COND_LTU /* unsigned */`
	`TCG_COND_GEU /* unsigned */`
	`TCG_COND_LEU /* unsigned */`
	`TCG_COND_GTU /* unsigned */`
	`TCG_COND_TSTEQ /* t1 & t2 == 0 */`
	`TCG_COND_TSTNE /* t1 & t2 != 0 */`

Arithmetic

add_i32/i64 *t0*, *t1*, *t2*	*t0* = *t1* + *t2*
sub_i32/i64 *t0*, *t1*, *t2*	*t0* = *t1* - *t2*
neg_i32/i64 *t0*, *t1*	*t0* = *-t1* (two's complement)
mul_i32/i64 *t0*, *t1*, *t2*	*t0* = *t1* * *t2*
div_i32/i64 *t0*, *t1*, *t2*	*t0* = *t1* / *t2* (signed) Undefined behavior if division by zero or overflow.
divu_i32/i64 *t0*, *t1*, *t2*	*t0* = *t1* / *t2* (unsigned) Undefined behavior if division by zero.
rem_i32/i64 *t0*, *t1*, *t2*	*t0* = *t1* % *t2* (signed) Undefined behavior if division by zero or overflow.
remu_i32/i64 *t0*, *t1*, *t2*	*t0* = *t1* % *t2* (unsigned) Undefined behavior if division by zero.

Logical

and_i32/i64 *t0*, *t1*, *t2*	*t0* = *t1* & *t2*
or_i32/i64 *t0*, *t1*, *t2*	*t0* = *t1* | *t2*
xor_i32/i64 *t0*, *t1*, *t2*	*t0* = *t1* ^ *t2*
not_i32/i64 *t0*, *t1*	*t0* = ~*t1*
andc_i32/i64 *t0*, *t1*, *t2*	*t0* = *t1* & ~*t2*
eqv_i32/i64 *t0*, *t1*, *t2*	*t0* = ~(*t1* ^ *t2*), or equivalently, *t0* = *t1* ^ ~*t2*
nand_i32/i64 *t0*, *t1*, *t2*	*t0* = ~(*t1* & *t2*)
nor_i32/i64 *t0*, *t1*, *t2*	*t0* = ~(*t1* | *t2*)
orc_i32/i64 *t0*, *t1*, *t2*	*t0* = *t1* | ~*t2*
clz_i32/i64 *t0*, *t1*, *t2*	*t0* = *t1* ? clz(*t1*) : *t2*
ctz_i32/i64 *t0*, *t1*, *t2*	*t0* = *t1* ? ctz(*t1*) : *t2*
ctpop_i32/i64 *t0*, *t1*	*t0* = number of bits set in *t1* With *ctpop* short for "count population", matching the function name used in `include/qemu/host-utils.h`.

Shifts/Rotates

shl_i32/i64 *t0*, *t1*, *t2*	*t0* = *t1* << *t2* Unspecified behavior if *t2* < 0 or *t2* >= 32 (resp 64)
shr_i32/i64 *t0*, *t1*, *t2*	*t0* = *t1* >> *t2* (unsigned) Unspecified behavior if *t2* < 0 or *t2* >= 32 (resp 64)
sar_i32/i64 *t0*, *t1*, *t2*	*t0* = *t1* >> *t2* (signed) Unspecified behavior if *t2* < 0 or *t2* >= 32 (resp 64)
rotl_i32/i64 *t0*, *t1*, *t2*	Rotation of *t2* bits to the left Unspecified behavior if *t2* < 0 or *t2* >= 32 (resp 64)
rotr_i32/i64 *t0*, *t1*, *t2*	Rotation of *t2* bits to the right. Unspecified behavior if *t2* < 0 or *t2* >= 32 (resp 64)

Misc

mov_i32/i64 *t0*, *t1*	*t0* = *t1* Move *t1* to *t0* (both operands must have the same type).
ext8s_i32/i64 *t0*, *t1* ext8u_i32/i64 *t0*, *t1* ext16s_i32/i64 *t0*, *t1* ext16u_i32/i64 *t0*, *t1* ext32s_i64 *t0*, *t1* ext32u_i64 *t0*, *t1*	8, 16 or 32 bit sign/zero extension (both operands must have the same type)
bswap16_i32/i64 *t0*, *t1*, *flags*	16 bit byte swap on the low bits of a 32/64 bit input. If *flags* & `TCG_BSWAP_IZ`, then *t1* is known to be zero-extended from bit 15. If *flags* & `TCG_BSWAP_OZ`, then *t0* will be zero-extended from bit 15. If *flags* & `TCG_BSWAP_OS`, then *t0* will be sign-extended from bit 15. If neither `TCG_BSWAP_OZ` nor `TCG_BSWAP_OS` are set, then the bits of *t0* above bit 15 may contain any value.
bswap32_i64 *t0*, *t1*, *flags*	32 bit byte swap on a 64-bit value. The flags are the same as for bswap16, except they apply from bit 31 instead of bit 15.
bswap32_i32 *t0*, *t1*, *flags* bswap64_i64 *t0*, *t1*, *flags*	32/64 bit byte swap. The flags are ignored, but still present for consistency with the other bswap opcodes.
discard_i32/i64 *t0*	Indicate that the value of *t0* won't be used later. It is useful to force dead code elimination.
deposit_i32/i64 *dest*, *t1*, *t2*, *pos*, *len*	Deposit *t2* as a bitfield into *t1*, placing the result in *dest*. The bitfield is described by *pos*/*len*, which are immediate values: *len* - the length of the bitfield *pos* - the position of the first bit, counting from the LSB For example, "deposit_i32 dest, t1, t2, 8, 4" indicates a 4-bit field at bit 8. This operation would be equivalent to *dest* = (*t1* & ~0x0f00) | ((*t2* << 8) & 0x0f00)

Conditional moves

setcond_i32/i64 *dest*, *t1*, *t2*, *cond*	*dest* = (*t1* *cond* *t2*) Set *dest* to 1 if (*t1* *cond* *t2*) is true, otherwise set to 0.
negsetcond_i32/i64 *dest*, *t1*, *t2*, *cond*	*dest* = -(*t1* *cond* *t2*) Set *dest* to -1 if (*t1* *cond* *t2*) is true, otherwise set to 0.
movcond_i32/i64 *dest*, *c1*, *c2*, *v1*, *v2*, *cond*	*dest* = (*c1* *cond* *c2* ? *v1* : *v2*) Set *dest* to *v1* if (*c1* *cond* *c2*) is true, otherwise set to *v2*.

Type conversions

ext_i32_i64 *t0*, *t1*	Convert *t1* (32 bit) to *t0* (64 bit) and does sign extension
extu_i32_i64 *t0*, *t1*	Convert *t1* (32 bit) to *t0* (64 bit) and does zero extension
trunc_i64_i32 *t0*, *t1*	Truncate *t1* (64 bit) to *t0* (32 bit)
concat_i32_i64 *t0*, *t1*, *t2*	Construct *t0* (64-bit) taking the low half from *t1* (32 bit) and the high half from *t2* (32 bit).
concat32_i64 *t0*, *t1*, *t2*	Construct *t0* (64-bit) taking the low half from *t1* (64 bit) and the high half from *t2* (64 bit).

Load/Store

ld_i32/i64 *t0*, *t1*, *offset* ld8s_i32/i64 *t0*, *t1*, *offset* ld8u_i32/i64 *t0*, *t1*, *offset* ld16s_i32/i64 *t0*, *t1*, *offset* ld16u_i32/i64 *t0*, *t1*, *offset* ld32s_i64 t0, *t1*, *offset* ld32u_i64 t0, *t1*, *offset*	*t0* = read(*t1* + *offset*) Load 8, 16, 32 or 64 bits with or without sign extension from host memory. *offset* must be a constant.
st_i32/i64 *t0*, *t1*, *offset* st8_i32/i64 *t0*, *t1*, *offset* st16_i32/i64 *t0*, *t1*, *offset* st32_i64 *t0*, *t1*, *offset*	write(*t0*, *t1* + *offset*) Write 8, 16, 32 or 64 bits to host memory.

All this opcodes assume that the pointed host memory doesn't correspond to a global. In the latter case the behaviour is unpredictable.

Multiword arithmetic support

add2_i32/i64 *t0_low*, *t0_high*, *t1_low*, *t1_high*, *t2_low*, *t2_high* sub2_i32/i64 *t0_low*, *t0_high*, *t1_low*, *t1_high*, *t2_low*, *t2_high*	Similar to add/sub, except that the double-word inputs *t1* and *t2* are formed from two single-word arguments, and the double-word output *t0* is returned in two single-word outputs.
mulu2_i32/i64 *t0_low*, *t0_high*, *t1*, *t2*	Similar to mul, except two unsigned inputs *t1* and *t2* yielding the full double-word product *t0*. The latter is returned in two single-word outputs.
muls2_i32/i64 *t0_low*, *t0_high*, *t1*, *t2*	Similar to mulu2, except the two inputs *t1* and *t2* are signed.
mulsh_i32/i64 *t0*, *t1*, *t2* muluh_i32/i64 *t0*, *t1*, *t2*	Provide the high part of a signed or unsigned multiply, respectively. If mulu2/muls2 are not provided by the backend, the tcg-op generator can obtain the same results by emitting a pair of opcodes, mul + muluh/mulsh.

Memory Barrier support

mb <*$arg*>	Generate a target memory barrier instruction to ensure memory ordering as being enforced by a corresponding guest memory barrier instruction. The ordering enforced by the backend may be stricter than the ordering required by the guest. It cannot be weaker. This opcode takes a constant argument which is required to generate the appropriate barrier instruction. The backend should take care to emit the target barrier instruction only when necessary i.e., for SMP guests and when MTTCG is enabled. The guest translators should generate this opcode for all guest instructions which have ordering side effects. Please see *Atomic operations in QEMU* for more information on memory barriers.

64-bit guest on 32-bit host support

The following opcodes are internal to TCG. Thus they are to be implemented by 32-bit host code generators, but are not to be emitted by guest translators. They are emitted as needed by inline functions within `tcg-op.h`.

brcond2_i32 *t0_low*, *t0_high*, *t1_low*, *t1_high*, *cond*, *label*	Similar to brcond, except that the 64-bit values *t0* and *t1* are formed from two 32-bit arguments.
setcond2_i32 *dest*, *t1_low*, *t1_high*, *t2_low*, *t2_high*, *cond*	Similar to setcond, except that the 64-bit values *t1* and *t2* are formed from two 32-bit arguments. The result is a 32-bit value.

QEMU specific operations

exit_tb *t0*	Exit the current TB and return the value *t0* (word type).
goto_tb *index*	Exit the current TB and jump to the TB index *index* (constant) if the current TB was linked to this TB. Otherwise execute the next instructions. Only indices 0 and 1 are valid and tcg_gen_goto_tb may be issued at most once with each slot index per TB.
lookup_and_goto_ptr *tb_addr*	Look up a TB address *tb_addr* and jump to it if valid. If not valid, jump to the TCG epilogue to go back to the exec loop. This operation is optional. If the TCG backend does not implement the goto_ptr opcode, emitting this op is equivalent to emitting exit_tb(0).
qemu_ld_i32/i64/i128 *t0*, *t1*, *flags*, *memidx* qemu_st_i32/i64/i128 *t0*, *t1*, *flags*, *memidx* qemu_st8_i32 *t0*, *t1*, *flags*, *memidx*	Load data at the guest address *t1* into *t0*, or store data in *t0* at guest address *t1*. The _i32/_i64/_i128 size applies to the size of the input/output register *t0* only. The address *t1* is always sized according to the guest, and the width of the memory operation is controlled by *flags*. Both *t0* and *t1* may be split into little-endian ordered pairs of registers if dealing with 64-bit quantities on a 32-bit host, or 128-bit quantities on a 64-bit host. The *memidx* selects the qemu tlb index to use (e.g. user or kernel access). The flags are the MemOp bits, selecting the sign, width, and endianness of the memory access. For a 32-bit host, qemu_ld/st_i64 is guaranteed to only be used with a 64-bit memory access specified in *flags*. For qemu_ld/st_i128, these are only supported for a 64-bit host. For i386, qemu_st8_i32 is exactly like qemu_st_i32, except the size of the memory operation is known to be 8-bit. This allows the backend to provide a different set of register constraints.

Host vector operations

All of the vector ops have two parameters, TCGOP_VECL & TCGOP_VECE. The former specifies the length of the vector in log2 64-bit units; the latter specifies the length of the element (if applicable) in log2 8-bit units. E.g. VECL = 1 -> 64 << 1 -> v128, and VECE = 2 -> 1 << 2 -> i32.

mov_vec *v0*, *v1* ld_vec *v0*, *t1* st_vec *v0*, *t1*	Move, load and store.
dup_vec *v0*, *r1*	Duplicate the low N bits of *r1* into VECL/VECE copies across *v0*.
dupi_vec *v0*, *c*	Similarly, for a constant. Smaller values will be replicated to host register size by the expanders.
dup2_vec *v0*, *r1*, *r2*	Duplicate *r2*:*r1* into VECL/64 copies across *v0*. This opcode is only present for 32-bit hosts.
add_vec *v0*, *v1*, *v2*	*v0* = *v1* + *v2*, in elements across the vector.
sub_vec *v0*, *v1*, *v2*	Similarly, *v0* = *v1* - *v2*.
mul_vec *v0*, *v1*, *v2*	Similarly, *v0* = *v1* * *v2*.
neg_vec *v0*, *v1*	Similarly, *v0* = -*v1*.
abs_vec *v0*, *v1*	Similarly, *v0* = *v1* < 0 ? -*v1* : *v1*, in elements across the vector.
smin_vec *v0*, *v1*, *v2* umin_vec *v0*, *v1*, *v2*	Similarly, *v0* = MIN(*v1*, *v2*), for signed and unsigned element types.
smax_vec *v0*, *v1*, *v2* umax_vec *v0*, *v1*, *v2*	Similarly, *v0* = MAX(*v1*, *v2*), for signed and unsigned element types.
ssadd_vec *v0*, *v1*, *v2* sssub_vec *v0*, *v1*, *v2* usadd_vec *v0*, *v1*, *v2* ussub_vec *v0*, *v1*, *v2*	Signed and unsigned saturating addition and subtraction. If the true result is not representable within the element type, the element is set to the minimum or maximum value for the type.
and_vec *v0*, *v1*, *v2* or_vec *v0*, *v1*, *v2* xor_vec *v0*, *v1*, *v2* andc_vec *v0*, *v1*, *v2*	Similarly, logical operations with and without complement.

Note 1: Some shortcuts are defined when the last operand is known to be a constant (e.g. addi for add, movi for mov).

Note 2: When using TCG, the opcodes must never be generated directly as some of them may not be available as "real" opcodes. Always use the function tcg_gen_xxx(args).

Backend

`tcg-target.h` contains the target specific definitions. `tcg-target.c.inc` contains the target specific code; it is #included by `tcg/tcg.c`, rather than being a standalone C file.

Assumptions

The target word size (`TCG_TARGET_REG_BITS`) is expected to be 32 bit or 64 bit. It is expected that the pointer has the same size as the word.

On a 32 bit target, all 64 bit operations are converted to 32 bits. A few specific operations must be implemented to allow it (see add2_i32, sub2_i32, brcond2_i32).

On a 64 bit target, the values are transferred between 32 and 64-bit registers using the following ops:

- extrl_i64_i32
- extrh_i64_i32
- ext_i32_i64
- extu_i32_i64

They ensure that the values are correctly truncated or extended when moved from a 32-bit to a 64-bit register or vice-versa. Note that the extrl_i64_i32 and extrh_i64_i32 are optional ops. It is not necessary to implement them if all the following conditions are met:

- 64-bit registers can hold 32-bit values
- 32-bit values in a 64-bit register do not need to stay zero or sign extended
- all 32-bit TCG ops ignore the high part of 64-bit registers

Floating point operations are not supported in this version. A previous incarnation of the code generator had full support of them, but it is better to concentrate on integer operations first.

Constraints

GCC like constraints are used to define the constraints of every instruction. Memory constraints are not supported in this version. Aliases are specified in the input operands as for GCC.

The same register may be used for both an input and an output, even when they are not explicitly aliased. If an op expands to multiple target instructions then care must be taken to avoid clobbering input values. GCC style "early clobber" outputs are supported, with '`&`'.

A target can define specific register or constant constraints. If an operation uses a constant input constraint which does not allow all constants, it must also accept registers in order to have a fallback. The constraint '`i`' is defined generically to accept any constant. The constraint '`r`' is not defined generically, but is consistently used by each backend to indicate all registers.

The movi_i32 and movi_i64 operations must accept any constants.

The mov_i32 and mov_i64 operations must accept any registers of the same type.

The ld/st/sti instructions must accept signed 32 bit constant offsets. This can be implemented by reserving a specific register in which to compute the address if the offset is too big.

The ld/st instructions must accept any destination (ld) or source (st) register.

The sti instruction may fail if it cannot store the given constant.

Function call assumptions

- The only supported types for parameters and return value are: 32 and 64 bit integers and pointer.
- The stack grows downwards.
- The first N parameters are passed in registers.
- The next parameters are passed on the stack by storing them as words.
- Some registers are clobbered during the call.
- The function can return 0 or 1 value in registers. On a 32 bit target, functions must be able to return 2 values in registers for 64 bit return type.

Recommended coding rules for best performance

- Use globals to represent the parts of the QEMU CPU state which are often modified, e.g. the integer registers and the condition codes. TCG will be able to use host registers to store them.
- Don't hesitate to use helpers for complicated or seldom used guest instructions. There is little performance advantage in using TCG to implement guest instructions taking more than about twenty TCG instructions. Note that this rule of thumb is more applicable to helpers doing complex logic or arithmetic, where the C compiler has scope to do a good job of optimisation; it is less relevant where the instruction is mostly doing loads and stores, and in those cases inline TCG may still be faster for longer sequences.
- Use the 'discard' instruction if you know that TCG won't be able to prove that a given global is "dead" at a given program point. The x86 guest uses it to improve the condition codes optimisation.

7.5.3 Decodetree Specification

A *decodetree* is built from instruction *patterns*. A pattern may represent a single architectural instruction or a group of same, depending on what is convenient for further processing.

Each pattern has both *fixedbits* and *fixedmask*, the combination of which describes the condition under which the pattern is matched:

```
(insn & fixedmask) == fixedbits
```

Each pattern may have *fields*, which are extracted from the insn and passed along to the translator. Examples of such are registers, immediates, and sub-opcodes.

In support of patterns, one may declare *fields*, *argument sets*, and *formats*, each of which may be re-used to simplify further definitions.

Fields

Syntax:

```
field_def     := '%' identifier ( field )* ( !function=identifier )?
field         := unnamed_field | named_field
unnamed_field := number ':' ( 's' ) number
named_field   := identifier ':' ( 's' ) number
```

For *unnamed_field*, the first number is the least-significant bit position of the field and the second number is the length of the field. If the 's' is present, the field is considered signed.

A *named_field* refers to some other field in the instruction pattern or format. Regardless of the length of the other field where it is defined, it will be inserted into this field with the specified signedness and bit width.

Field definitions that involve loops (i.e. where a field is defined directly or indirectly in terms of itself) are errors.

A format can include fields that refer to named fields that are defined in the instruction pattern(s) that use the format. Conversely, an instruction pattern can include fields that refer to named fields that are defined in the format it uses. However you cannot currently do both at once (i.e. pattern P uses format F; F has a field A that refers to a named field B that is defined in P, and P has a field C that refers to a named field D that is defined in F).

If multiple `fields` are present, they are concatenated. In this way one can define disjoint fields.

If `!function` is specified, the concatenated result is passed through the named function, taking and returning an integral value.

One may use `!function` with zero `fields`. This case is called a *parameter*, and the named function is only passed the `DisasContext` and returns an integral value extracted from there.

A field with no `fields` and no `!function` is in error.

Field examples:

Input	Generated code
%disp 0:s16	sextract(i, 0, 16)
%imm9 16:6 10:3	extract(i, 16, 6) << 3 | extract(i, 10, 3)
%disp12 0:s1 1:1 2:10	**sextract(i, 0, 1) << 11 |** extract(i, 1, 1) << 10 | extract(i, 2, 10)
%shimm8 5:s8 13:1 !function=expand_shimm8	**expand_shimm8(sextract(i, 5, 8) << 1 |** extract(i, 13, 1))
%sz_imm 10:2 sz:3 !function=expand_sz_imm	**expand_sz_imm(extract(i, 10, 2) << 3 |** extract(a->sz, 0, 3))

Argument Sets

Syntax:

```
args_def    := '&' identifier ( args_elt )+ ( !extern )?
args_elt    := identifier (':' identifier)?
```

Each *args_elt* defines an argument within the argument set. If the form of the *args_elt* contains a colon, the first identifier is the argument name and the second identifier is the argument type. If the colon is missing, the argument type will be `int`.

Each argument set will be rendered as a C structure "arg_$name" with each of the fields being one of the member arguments.

If `!extern` is specified, the backing structure is assumed to have been already declared, typically via a second decoder.

Argument sets are useful when one wants to define helper functions for the translator functions that can perform operations on a common set of arguments. This can ensure, for instance, that the `AND` pattern and the `OR` pattern put their operands into the same named structure, so that a common `gen_logic_insn` may be able to handle the operations common between the two.

Argument set examples:

```
&reg3       ra rb rc
&loadstore  reg base offset
&longldst   reg base offset:int64_t
```

Formats

Syntax:

```
fmt_def      := '@' identifier ( fmt_elt )+
fmt_elt      := fixedbit_elt | field_elt | field_ref | args_ref
fixedbit_elt := [01.-]+
field_elt    := identifier ':' 's'? number
field_ref    := '%' identifier | identifier '=' '%' identifier
args_ref     := '&' identifier
```

Defining a format is a handy way to avoid replicating groups of fields across many instruction patterns.

A *fixedbit_elt* describes a contiguous sequence of bits that must be 1, 0, or don't care. The difference between '.' and '-' is that '.' means that the bit will be covered with a field or a final 0 or 1 from the pattern, and '-' means that the bit is really ignored by the cpu and will not be specified.

A *field_elt* describes a simple field only given a width; the position of the field is implied by its position with respect to other *fixedbit_elt* and *field_elt*.

If any *fixedbit_elt* or *field_elt* appear, then all bits must be defined. Padding with a *fixedbit_elt* of all '.' is an easy way to accomplish that.

A *field_ref* incorporates a field by reference. This is the only way to add a complex field to a format. A field may be renamed in the process via assignment to another identifier. This is intended to allow the same argument set be used with disjoint named fields.

A single *args_ref* may specify an argument set to use for the format. The set of fields in the format must be a subset of the arguments in the argument set. If an argument set is not specified, one will be inferred from the set of fields.

It is recommended, but not required, that all *field_ref* and *args_ref* appear at the end of the line, not interleaving with *fixedbit_elf* or *field_elt*.

Format examples:

```
@opr    ...... ra:5 rb:5 ... 0 ....... rc:5
@opi    ...... ra:5 lit:8    1 ....... rc:5
```

Patterns

Syntax:

```
pat_def      := identifier ( pat_elt )+
pat_elt      := fixedbit_elt | field_elt | field_ref | args_ref | fmt_ref | const_elt
fmt_ref      := '@' identifier
const_elt    := identifier '=' number
```

The *fixedbit_elt* and *field_elt* specifiers are unchanged from formats. A pattern that does not specify a named format will have one inferred from a referenced argument set (if present) and the set of fields.

A *const_elt* allows a argument to be set to a constant value. This may come in handy when fields overlap between patterns and one has to include the values in the *fixedbit_elt* instead.

The decoder will call a translator function for each pattern matched.

Pattern examples:

```
addl_r   010000 ..... ..... .... 0000000 ..... @opr
addl_i   010000 ..... ..... .... 0000000 ..... @opi
```

which will, in part, invoke:

```
trans_addl_r(ctx, &arg_opr, insn)
```

and:

```
trans_addl_i(ctx, &arg_opi, insn)
```

Pattern Groups

Syntax:

```
group            := overlap_group | no_overlap_group
overlap_group    := '{' ( pat_def | group )+ '}'
no_overlap_group := '[' ( pat_def | group )+ ']'
```

A *group* begins with a lone open-brace or open-bracket, with all subsequent lines indented two spaces, and ending with a lone close-brace or close-bracket. Groups may be nested, increasing the required indentation of the lines within the nested group to two spaces per nesting level.

Patterns within overlap groups are allowed to overlap. Conflicts are resolved by selecting the patterns in order. If all of the fixedbits for a pattern match, its translate function will be called. If the translate function returns false, then subsequent patterns within the group will be matched.

Patterns within no-overlap groups are not allowed to overlap, just the same as ungrouped patterns. Thus no-overlap groups are intended to be nested inside overlap groups.

The following example from PA-RISC shows specialization of the *or* instruction:

```
{
  {
    nop   000010 ----- ----- 0000 001001 0 00000
    copy  000010 00000 r1:5  0000 001001 0 rt:5
  }
  or      000010 rt2:5 r1:5  cf:4 001001 0 rt:5
}
```

When the *cf* field is zero, the instruction has no side effects, and may be specialized. When the *rt* field is zero, the output is discarded and so the instruction has no effect. When the *rt2* field is zero, the operation is `reg[r1] | 0` and so encodes the canonical register copy operation.

The output from the generator might look like:

```
switch (insn & 0xfc000fe0) {
case 0x08000240:
  /* 000010.. ........ ....0010 010..... */
  if ((insn & 0x0000f000) == 0x00000000) {
      /* 000010.. ........ 00000010 010..... */
      if ((insn & 0x0000001f) == 0x00000000) {
          /* 000010.. ........ 00000010 01000000 */
          extract_decode_Fmt_0(&u.f_decode0, insn);
          if (trans_nop(ctx, &u.f_decode0)) return true;
      }
      if ((insn & 0x03e00000) == 0x00000000) {
          /* 00001000 000..... 00000010 010..... */
          extract_decode_Fmt_1(&u.f_decode1, insn);
          if (trans_copy(ctx, &u.f_decode1)) return true;
      }
  }
  extract_decode_Fmt_2(&u.f_decode2, insn);
  if (trans_or(ctx, &u.f_decode2)) return true;
  return false;
}
```

7.5.4 Multi-threaded TCG

This document outlines the design for multi-threaded TCG (a.k.a MTTCG) system-mode emulation. user-mode emulation has always mirrored the thread structure of the translated executable although some of the changes done for MTTCG system emulation have improved the stability of linux-user emulation.

The original system-mode TCG implementation was single threaded and dealt with multiple CPUs with simple round-robin scheduling. This simplified a lot of things but became increasingly limited as systems being emulated gained additional cores and per-core performance gains for host systems started to level off.

vCPU Scheduling

We introduce a new running mode where each vCPU will run on its own user-space thread. This is enabled by default for all FE/BE combinations where the host memory model is able to accommodate the guest (TCG_GUEST_DEFAULT_MO & ~TCG_TARGET_DEFAULT_MO is zero) and the guest has had the required work done to support this safely (TARGET_SUPPORTS_MTTCG).

System emulation will fall back to the original round robin approach if:

- forced by –accel tcg,thread=single
- enabling –icount mode
- 64 bit guests on 32 bit hosts (TCG_OVERSIZED_GUEST)

In the general case of running translated code there should be no inter-vCPU dependencies and all vCPUs should be able to run at full speed. Synchronisation will only be required while accessing internal shared data structures or when the emulated architecture requires a coherent representation of the emulated machine state.

Shared Data Structures

Main Run Loop

Even when there is no code being generated there are a number of structures associated with the hot-path through the main run-loop. These are associated with looking up the next translation block to execute. These include:

> tb_jmp_cache (per-vCPU, cache of recent jumps) tb_ctx.htable (global hash table, phys address->tb lookup)

As TB linking only occurs when blocks are in the same page this code is critical to performance as looking up the next TB to execute is the most common reason to exit the generated code.

DESIGN REQUIREMENT: Make access to lookup structures safe with multiple reader/writer threads. Minimise any lock contention to do it.

The hot-path avoids using locks where possible. The tb_jmp_cache is updated with atomic accesses to ensure consistent results. The fall back QHT based hash table is also designed for lockless lookups. Locks are only taken when code generation is required or TranslationBlocks have their block-to-block jumps patched.

Global TCG State

User-mode emulation

We need to protect the entire code generation cycle including any post generation patching of the translated code. This also implies a shared translation buffer which contains code running on all cores. Any execution path that comes to the main run loop will need to hold a mutex for code generation. This also includes times when we need flush code or entries from any shared lookups/caches. Structures held on a per-vCPU basis won't need locking unless other vCPUs will need to modify them.

DESIGN REQUIREMENT: Add locking around all code generation and TB patching.

(Current solution)

Code generation is serialised with mmap_lock().

!User-mode emulation

Each vCPU has its own TCG context and associated TCG region, thereby requiring no locking during translation.

Translation Blocks

Currently the whole system shares a single code generation buffer which when full will force a flush of all translations and start from scratch again. Some operations also force a full flush of translations including:

- debugging operations (breakpoint insertion/removal)
- some CPU helper functions
- linux-user spawning its first thread
- operations related to TCG Plugins

This is done with the async_safe_run_on_cpu() mechanism to ensure all vCPUs are quiescent when changes are being made to shared global structures.

More granular translation invalidation events are typically due to a change of the state of a physical page:

- code modification (self modify code, patching code)
- page changes (new page mapping in linux-user mode)

While setting the invalid flag in a TranslationBlock will stop it being used when looked up in the hot-path there are a number of other book-keeping structures that need to be safely cleared.

Any TranslationBlocks which have been patched to jump directly to the now invalid blocks need the jump patches reversing so they will return to the C code.

There are a number of look-up caches that need to be properly updated including the:

- jump lookup cache
- the physical-to-tb lookup hash table
- the global page table

The global page table (l1_map) which provides a multi-level look-up for PageDesc structures which contain pointers to the start of a linked list of all Translation Blocks in that page (see page_next).

Both the jump patching and the page cache involve linked lists that the invalidated TranslationBlock needs to be removed from.

DESIGN REQUIREMENT: Safely handle invalidation of TBs

- safely patch/revert direct jumps
- remove central PageDesc lookup entries
- ensure lookup caches/hashes are safely updated

(Current solution)

The direct jump themselves are updated atomically by the TCG tb_set_jmp_target() code. Modification to the linked lists that allow searching for linked pages are done under the protection of tb->jmp_lock, where tb is the destination block of a jump. Each origin block keeps a pointer to its destinations so that the appropriate lock can be acquired before iterating over a jump list.

The global page table is a lockless radix tree; cmpxchg is used to atomically insert new elements.

The lookup caches are updated atomically and the lookup hash uses QHT which is designed for concurrent safe lookup.

Parallel code generation is supported. QHT is used at insertion time as the synchronization point across threads, thereby ensuring that we only keep track of a single TranslationBlock for each guest code block.

Memory maps and TLBs

The memory handling code is fairly critical to the speed of memory access in the emulated system. The SoftMMU code is designed so the hot-path can be handled entirely within translated code. This is handled with a per-vCPU TLB structure which once populated will allow a series of accesses to the page to occur without exiting the translated code. It is possible to set flags in the TLB address which will ensure the slow-path is taken for each access. This can be done to support:

- Memory regions (dividing up access to PIO, MMIO and RAM)
- Dirty page tracking (for code gen, SMC detection, migration and display)
- Virtual TLB (for translating guest address->real address)

When the TLB tables are updated by a vCPU thread other than their own we need to ensure it is done in a safe way so no inconsistent state is seen by the vCPU thread.

Some operations require updating a number of vCPUs TLBs at the same time in a synchronised manner.

DESIGN REQUIREMENTS:

- TLB Flush All/Page - can be across-vCPUs - cross vCPU TLB flush may need other vCPU brought to halt - change may need to be visible to the calling vCPU immediately
- TLB Flag Update - usually cross-vCPU - want change to be visible as soon as possible
- TLB Update (update a CPUTLBEntry, via tlb_set_page_with_attrs) - This is a per-vCPU table - by definition can't race - updated by its own thread when the slow-path is forced

(Current solution)

We have updated cputlb.c to defer operations when a cross-vCPU operation with async_run_on_cpu() which ensures each vCPU sees a coherent state when it next runs its work (in a few instructions time).

A new set up operations (tlb_flush_*_all_cpus) take an additional flag which when set will force synchronisation by setting the source vCPUs work as "safe work" and exiting the cpu run loop. This ensure by the time execution restarts all flush operations have completed.

TLB flag updates are all done atomically and are also protected by the corresponding page lock.

(Known limitation)

Not really a limitation but the wait mechanism is overly strict for some architectures which only need flushes completed by a barrier instruction. This could be a future optimisation.

Emulated hardware state

Currently thanks to KVM work any access to IO memory is automatically protected by the BQL (Big QEMU Lock). Any IO region that doesn't use the BQL is expected to do its own locking.

However IO memory isn't the only way emulated hardware state can be modified. Some architectures have model specific registers that trigger hardware emulation features. Generally any translation helper that needs to update more than a single vCPUs of state should take the BQL.

As the BQL, or global iothread mutex is shared across the system we push the use of the lock as far down into the TCG code as possible to minimise contention.

(Current solution)

MMIO access automatically serialises hardware emulation by way of the BQL. Currently Arm targets serialise all ARM_CP_IO register accesses and also defer the reset/startup of vCPUs to the vCPU context by way of async_run_on_cpu().

Updates to interrupt state are also protected by the BQL as they can often be cross vCPU.

Memory Consistency

Between emulated guests and host systems there are a range of memory consistency models. Even emulating weakly ordered systems on strongly ordered hosts needs to ensure things like store-after-load re-ordering can be prevented when the guest wants to.

Memory Barriers

Barriers (sometimes known as fences) provide a mechanism for software to enforce a particular ordering of memory operations from the point of view of external observers (e.g. another processor core). They can apply to any memory operations as well as just loads or stores.

The Linux kernel has an excellent write-up on the various forms of memory barrier and the guarantees they can provide.

Barriers are often wrapped around synchronisation primitives to provide explicit memory ordering semantics. However they can be used by themselves to provide safe lockless access by ensuring for example a change to a signal flag will only be visible once the changes to payload are.

DESIGN REQUIREMENT: Add a new tcg_memory_barrier op

This would enforce a strong load/store ordering so all loads/stores complete at the memory barrier. On single-core non-SMP strongly ordered backends this could become a NOP.

Aside from explicit standalone memory barrier instructions there are also implicit memory ordering semantics which comes with each guest memory access instruction. For example all x86 load/stores come with fairly strong guarantees of sequential consistency whereas Arm has special variants of load/store instructions that imply acquire/release semantics.

In the case of a strongly ordered guest architecture being emulated on a weakly ordered host the scope for a heavy performance impact is quite high.

DESIGN REQUIREMENTS: Be efficient with use of memory barriers

- host systems with stronger implied guarantees can skip some barriers
- merge consecutive barriers to the strongest one

(Current solution)

The system currently has a tcg_gen_mb() which will add memory barrier operations if code generation is being done in a parallel context. The tcg_optimize() function attempts to merge barriers up to their strongest form before any load/store operations. The solution was originally developed and tested for linux-user based systems. All backends have been converted to emit fences when required. So far the following front-ends have been updated to emit fences when required:

- target-i386
- target-arm
- target-aarch64
- target-alpha
- target-mips

Memory Control and Maintenance

This includes a class of instructions for controlling system cache behaviour. While QEMU doesn't model cache behaviour these instructions are often seen when code modification has taken place to ensure the changes take effect.

Synchronisation Primitives

There are two broad types of synchronisation primitives found in modern ISAs: atomic instructions and exclusive regions.

The first type offer a simple atomic instruction which will guarantee some sort of test and conditional store will be truly atomic w.r.t. other cores sharing access to the memory. The classic example is the x86 cmpxchg instruction.

The second type offer a pair of load/store instructions which offer a guarantee that a region of memory has not been touched between the load and store instructions. An example of this is Arm's ldrex/strex pair where the strex instruction will return a flag indicating a successful store only if no other CPU has accessed the memory region since the ldrex.

Traditionally TCG has generated a series of operations that work because they are within the context of a single translation block so will have completed before another CPU is scheduled. However with the ability to have multiple threads running to emulate multiple CPUs we will need to explicitly expose these semantics.

DESIGN REQUIREMENTS:

- Support classic atomic instructions
- Support load/store exclusive (or load link/store conditional) pairs
- Generic enough infrastructure to support all guest architectures

CURRENT OPEN QUESTIONS:

- How problematic is the ABA problem in general?

(Current solution)

The TCG provides a number of atomic helpers (tcg_gen_atomic_*) which can be used directly or combined to emulate other instructions like Arm's ldrex/strex instructions. While they are susceptible to the ABA problem so far common guests have not implemented patterns where this may be a problem - typically presenting a locking ABI which assumes cmpxchg like semantics.

The code also includes a fall-back for cases where multi-threaded TCG ops can't work (e.g. guest atomic width > host atomic width). In this case an EXCP_ATOMIC exit occurs and the instruction is emulated with an exclusive lock which ensures all emulation is serialised.

While the atomic helpers look good enough for now there may be a need to look at solutions that can more closely model the guest architectures semantics.

7.5.5 TCG Instruction Counting

TCG has long supported a feature known as icount which allows for instruction counting during execution. This should not be confused with cycle accurate emulation - QEMU does not attempt to emulate how long an instruction would take on real hardware. That is a job for other more detailed (and slower) tools that simulate the rest of a micro-architecture.

This feature is only available for system emulation and is incompatible with multi-threaded TCG. It can be used to better align execution time with wall-clock time so a "slow" device doesn't run too fast on modern hardware. It can also provides for a degree of deterministic execution and is an essential part of the record/replay support in QEMU.

Core Concepts

At its heart icount is simply a count of executed instructions which is stored in the TimersState of QEMU's timer sub-system. The number of executed instructions can then be used to calculate QEMU_CLOCK_VIRTUAL which represents the amount of elapsed time in the system since execution started. Depending on the icount mode this may either be a fixed number of ns per instruction or adjusted as execution continues to keep wall clock time and virtual time in sync.

To be able to calculate the number of executed instructions the translator starts by allocating a budget of instructions to be executed. The budget of instructions is limited by how long it will be until the next timer will expire. We store this budget as part of a vCPU icount_decr field which shared with the machinery for handling cpu_exit(). The whole field is checked at the start of every translated block and will cause a return to the outer loop to deal with whatever caused the exit.

In the case of icount, before the flag is checked we subtract the number of instructions the translation block would execute. If this would cause the instruction budget to go negative we exit the main loop and regenerate a new translation block with exactly the right number of instructions to take the budget to 0 meaning whatever timer was due to expire will expire exactly when we exit the main run loop.

Dealing with MMIO

While we can adjust the instruction budget for known events like timer expiry we cannot do the same for MMIO. Every load/store we execute might potentially trigger an I/O event, at which point we will need an up to date and accurate reading of the icount number.

To deal with this case, when an I/O access is made we:

- restore un-executed instructions to the icount budget
- re-compile a single[1] instruction block for the current PC
- exit the cpu loop and execute the re-compiled block

Other I/O operations

MMIO isn't the only type of operation for which we might need a correct and accurate clock. IO port instructions and accesses to system registers are the common examples here. These instructions have to be handled by the individual translators which have the knowledge of which operations are I/O operations.

When the translator is handling an instruction of this kind:

- **it must call gen_io_start() if icount is enabled, at some**
  point before the generation of the code which actually does the I/O, using a code fragment similar to:

```
if (tb_cflags(s->base.tb) & CF_USE_ICOUNT) {
    gen_io_start();
}
```

- it must end the TB immediately after this instruction

[1] sometimes two instructions if dealing with delay slots

7.5.6 QEMU TCG Plugins

QEMU TCG plugins provide a way for users to run experiments taking advantage of the total system control emulation can have over a guest. It provides a mechanism for plugins to subscribe to events during translation and execution and optionally callback into the plugin during these events. TCG plugins are unable to change the system state only monitor it passively. However they can do this down to an individual instruction granularity including potentially subscribing to all load and store operations.

Usage

Any QEMU binary with TCG support has plugins enabled by default. Earlier releases needed to be explicitly enabled with:

```
configure --enable-plugins
```

Once built a program can be run with multiple plugins loaded each with their own arguments:

```
$QEMU $OTHER_QEMU_ARGS \
    -plugin contrib/plugin/libhowvec.so,inline=on,count=hint \
    -plugin contrib/plugin/libhotblocks.so
```

Arguments are plugin specific and can be used to modify their behaviour. In this case the howvec plugin is being asked to use inline ops to count and break down the hint instructions by type.

Linux user-mode emulation also evaluates the environment variable `QEMU_PLUGIN`:

```
QEMU_PLUGIN="file=contrib/plugins/libhowvec.so,inline=on,count=hint" $QEMU
```

Writing plugins

API versioning

This is a new feature for QEMU and it does allow people to develop out-of-tree plugins that can be dynamically linked into a running QEMU process. However the project reserves the right to change or break the API should it need to do so. The best way to avoid this is to submit your plugin upstream so they can be updated if/when the API changes.

All plugins need to declare a symbol which exports the plugin API version they were built against. This can be done simply by:

```
QEMU_PLUGIN_EXPORT int qemu_plugin_version = QEMU_PLUGIN_VERSION;
```

The core code will refuse to load a plugin that doesn't export a `qemu_plugin_version` symbol or if plugin version is outside of QEMU's supported range of API versions.

Additionally the `qemu_info_t` structure which is passed to the `qemu_plugin_install` method of a plugin will detail the minimum and current API versions supported by QEMU. The API version will be incremented if new APIs are added. The minimum API version will be incremented if existing APIs are changed or removed.

Lifetime of the query handle

Each callback provides an opaque anonymous information handle which can usually be further queried to find out information about a translation, instruction or operation. The handles themselves are only valid during the lifetime of the callback so it is important that any information that is needed is extracted during the callback and saved by the plugin.

Plugin life cycle

First the plugin is loaded and the public qemu_plugin_install function is called. The plugin will then register callbacks for various plugin events. Generally plugins will register a handler for the *atexit* if they want to dump a summary of collected information once the program/system has finished running.

When a registered event occurs the plugin callback is invoked. The callbacks may provide additional information. In the case of a translation event the plugin has an option to enumerate the instructions in a block of instructions and optionally register callbacks to some or all instructions when they are executed.

There is also a facility to add an inline event where code to increment a counter can be directly inlined with the translation. Currently only a simple increment is supported. This is not atomic so can miss counts. If you want absolute precision you should use a callback which can then ensure atomicity itself.

Finally when QEMU exits all the registered *atexit* callbacks are invoked.

Exposure of QEMU internals

The plugin architecture actively avoids leaking implementation details about how QEMU's translation works to the plugins. While there are conceptions such as translation time and translation blocks the details are opaque to plugins. The plugin is able to query select details of instructions and system configuration only through the exported *qemu_plugin* functions.

However the following assumptions can be made:

Translation Blocks

All code will go through a translation phase although not all translations will be necessarily be executed. You need to instrument actual executions to track what is happening.

It is quite normal to see the same address translated multiple times. If you want to track the code in system emulation you should examine the underlying physical address (`qemu_plugin_insn_haddr`) to take into account the effects of virtual memory although if the system does paging this will change too.

Not all instructions in a block will always execute so if its important to track individual instruction execution you need to instrument them directly. However asynchronous interrupts will not change control flow mid-block.

Instructions

Instruction instrumentation runs before the instruction executes. You can be can be sure the instruction will be dispatched, but you can't be sure it will complete. Generally this will be because of a synchronous exception (e.g. SIGILL) triggered by the instruction attempting to execute. If you want to be sure you will need to instrument the next instruction as well. See the `execlog.c` plugin for examples of how to track this and finalise details after execution.

Memory Accesses

Memory callbacks are called after a successful load or store. Unsuccessful operations (i.e. faults) will not be visible to memory instrumentation although the execution side effects can be observed (e.g. entering a exception handler).

System Idle and Resume States

The `qemu_plugin_register_vcpu_idle_cb` and `qemu_plugin_register_vcpu_resume_cb` functions can be used to track when CPUs go into and return from sleep states when waiting for external I/O. Be aware though that these may occur less frequently than in real HW due to the inefficiencies of emulation giving less chance for the CPU to idle.

Internals

Locking

We have to ensure we cannot deadlock, particularly under MTTCG. For this we acquire a lock when called from plugin code. We also keep the list of callbacks under RCU so that we do not have to hold the lock when calling the callbacks. This is also for performance, since some callbacks (e.g. memory access callbacks) might be called very frequently.

- A consequence of this is that we keep our own list of CPUs, so that we do not have to worry about locking order wrt cpu_list_lock.
- Use a recursive lock, since we can get registration calls from callbacks.

As a result registering/unregistering callbacks is "slow", since it takes a lock. But this is very infrequent; we want performance when calling (or not calling) callbacks, not when registering them. Using RCU is great for this.

We support the uninstallation of a plugin at any time (e.g. from plugin callbacks). This allows plugins to remove themselves if they no longer want to instrument the code. This operation is asynchronous which means callbacks may still occur after the uninstall operation is requested. The plugin isn't completely uninstalled until the safe work has executed while all vCPUs are quiescent.

7.5.7 Example Plugins

There are a number of plugins included with QEMU and you are encouraged to contribute your own plugins plugins upstream. There is a `contrib/plugins` directory where they can go. There are also some basic plugins that are used to test and exercise the API during the `make check-tcg` target in `tests\plugins`.

- tests/plugins/empty.c

Purely a test plugin for measuring the overhead of the plugins system itself. Does no instrumentation.

- tests/plugins/bb.c

A very basic plugin which will measure execution in course terms as each basic block is executed. By default the results are shown once execution finishes:

```
$ qemu-aarch64 -plugin tests/plugin/libbb.so \
    -d plugin ./tests/tcg/aarch64-linux-user/sha1
SHA1=15dd99a1991e0b3826fede3deffc1feba42278e6
bb's: 2277338, insns: 158483046
```

Behaviour can be tweaked with the following arguments:

- inline=true|false

Use faster inline addition of a single counter. Not per-cpu and not thread safe.

- idle=true|false

Dump the current execution stats whenever the guest vCPU idles

- tests/plugins/insn.c

This is a basic instruction level instrumentation which can count the number of instructions executed on each core/thread:

```
$ qemu-aarch64 -plugin tests/plugin/libinsn.so \
    -d plugin ./tests/tcg/aarch64-linux-user/threadcount
Created 10 threads
Done
cpu 0 insns: 46765
cpu 1 insns: 3694
cpu 2 insns: 3694
cpu 3 insns: 2994
cpu 4 insns: 1497
cpu 5 insns: 1497
cpu 6 insns: 1497
cpu 7 insns: 1497
total insns: 63135
```

Behaviour can be tweaked with the following arguments:

- inline=true|false

Use faster inline addition of a single counter. Not per-cpu and not thread safe.

- sizes=true|false

Give a summary of the instruction sizes for the execution

- match=<string>

Only instrument instructions matching the string prefix. Will show some basic stats including how many instructions have executed since the last execution. For example:

```
$ qemu-aarch64 -plugin tests/plugin/libinsn.so,match=bl \
    -d plugin ./tests/tcg/aarch64-linux-user/sha512-vector
...
0x40069c, 'bl #0x4002b0', 10 hits, 1093 match hits, +1257 since last match, 98␣
↪avg insns/match
0x4006ac, 'bl #0x403690', 10 hits, 1094 match hits, +47 since last match, 98␣
↪avg insns/match
0x4037fc, 'bl #0x4002b0', 18 hits, 1095 match hits, +22 since last match, 98␣
↪avg insns/match
0x400720, 'bl #0x403690', 10 hits, 1096 match hits, +58 since last match, 98␣
```

(continues on next page)

(continued from previous page)

```
↪avg insns/match
0x4037fc, 'bl #0x4002b0', 19 hits, 1097 match hits, +22 since last match, 98␣
↪avg insns/match
0x400730, 'bl #0x403690', 10 hits, 1098 match hits, +33 since last match, 98␣
↪avg insns/match
0x4037ac, 'bl #0x4002b0', 12 hits, 1099 match hits, +20 since last match, 98␣
↪avg insns/match
...
```

For more detailed execution tracing see the `execlog` plugin for other options.

- tests/plugins/mem.c

Basic instruction level memory instrumentation:

```
$ qemu-aarch64 -plugin tests/plugin/libmem.so,inline=true \
    -d plugin ./tests/tcg/aarch64-linux-user/sha1
SHA1=15dd99a1991e0b3826fede3deffc1feba42278e6
inline mem accesses: 79525013
```

Behaviour can be tweaked with the following arguments:

- inline=true|false

Use faster inline addition of a single counter. Not per-cpu and not thread safe.

- callback=true|false

Use callbacks on each memory instrumentation.

- hwaddr=true|false

Count IO accesses (only for system emulation)

- tests/plugins/syscall.c

A basic syscall tracing plugin. This only works for user-mode. By default it will give a summary of syscall stats at the end of the run:

```
$ qemu-aarch64 -plugin tests/plugin/libsyscall \
    -d plugin ./tests/tcg/aarch64-linux-user/threadcount
Created 10 threads
Done
syscall no.  calls  errors
226          12     0
99           11     11
115          11     0
222          11     0
93           10     0
220          10     0
233          10     0
215          8      0
214          4      0
134          2      0
64           2      0
96           1      0
94           1      0
```

(continues on next page)

(continued from previous page)

```
80          1      0
261         1      0
78          1      0
160         1      0
135         1      0
```

- contrib/plugins/hotblocks.c

The hotblocks plugin allows you to examine the where hot paths of execution are in your program. Once the program has finished you will get a sorted list of blocks reporting the starting PC, translation count, number of instructions and execution count. This will work best with linux-user execution as system emulation tends to generate re-translations as blocks from different programs get swapped in and out of system memory.

If your program is single-threaded you can use the `inline` option for slightly faster (but not thread safe) counters.

Example:

```
$ qemu-aarch64 \
  -plugin contrib/plugins/libhotblocks.so -d plugin \
  ./tests/tcg/aarch64-linux-user/sha1
SHA1=15dd99a1991e0b3826fede3deffc1feba42278e6
collected 903 entries in the hash table
pc, tcount, icount, ecount
0x0000000041ed10, 1, 5, 66087
0x000000004002b0, 1, 4, 66087
...
```

- contrib/plugins/hotpages.c

Similar to hotblocks but this time tracks memory accesses:

```
$ qemu-aarch64 \
  -plugin contrib/plugins/libhotpages.so -d plugin \
  ./tests/tcg/aarch64-linux-user/sha1
SHA1=15dd99a1991e0b3826fede3deffc1feba42278e6
Addr, RCPUs, Reads, WCPUs, Writes
0x000055007fe000, 0x0001, 31747952, 0x0001, 8835161
0x000055007ff000, 0x0001, 29001054, 0x0001, 8780625
0x00005500800000, 0x0001, 687465, 0x0001, 335857
0x0000000048b000, 0x0001, 130594, 0x0001, 355
0x0000000048a000, 0x0001, 1826, 0x0001, 11
```

The hotpages plugin can be configured using the following arguments:

- sortby=reads|writes|address

  Log the data sorted by either the number of reads, the number of writes, or memory address. (Default: entries are sorted by the sum of reads and writes)

- io=on

  Track IO addresses. Only relevant to full system emulation. (Default: off)

- pagesize=N

  The page size used. (Default: N = 4096)

- contrib/plugins/howvec.c

This is an instruction classifier so can be used to count different types of instructions. It has a number of options to refine which get counted. You can give a value to the `count` argument for a class of instructions to break it down fully, so for example to see all the system registers accesses:

```
$ qemu-system-aarch64 $(QEMU_ARGS) \
  -append "root=/dev/sda2 systemd.unit=benchmark.service" \
  -smp 4 -plugin ./contrib/plugins/libhowvec.so,count=sreg -d plugin
```

which will lead to a sorted list after the class breakdown:

```
Instruction Classes:
Class:   UDEF                  not counted
Class:   SVE                   (68 hits)
Class:   PCrel addr            (47789483 hits)
Class:   Add/Sub (imm)         (192817388 hits)
Class:   Logical (imm)         (93852565 hits)
Class:   Move Wide (imm)       (76398116 hits)
Class:   Bitfield              (44706084 hits)
Class:   Extract               (5499257 hits)
Class:   Cond Branch (imm)     (147202932 hits)
Class:   Exception Gen         (193581 hits)
Class:     NOP                 not counted
Class:   Hints                 (6652291 hits)
Class:   Barriers              (8001661 hits)
Class:   PSTATE                (1801695 hits)
Class:   System Insn           (6385349 hits)
Class:   System Reg            counted individually
Class:   Branch (reg)          (69497127 hits)
Class:   Branch (imm)          (84393665 hits)
Class:   Cmp & Branch          (110929659 hits)
Class:   Tst & Branch          (44681442 hits)
Class:   AdvSimd ldstmult      (736 hits)
Class:   ldst excl             (9098783 hits)
Class:   Load Reg (lit)        (87189424 hits)
Class:   ldst noalloc pair     (3264433 hits)
Class:   ldst pair             (412526434 hits)
Class:   ldst reg (imm)        (314734576 hits)
Class: Loads & Stores          (2117774 hits)
Class: Data Proc Reg           (223519077 hits)
Class: Scalar FP               (31657954 hits)
Individual Instructions:
Instr: mrs x0, sp_el0          (2682661 hits)  (op=0xd5384100/  System Reg)
Instr: mrs x1, tpidr_el2       (1789339 hits)  (op=0xd53cd041/  System Reg)
Instr: mrs x2, tpidr_el2       (1513494 hits)  (op=0xd53cd042/  System Reg)
Instr: mrs x0, tpidr_el2       (1490823 hits)  (op=0xd53cd040/  System Reg)
Instr: mrs x1, sp_el0          (933793 hits)   (op=0xd5384101/  System Reg)
Instr: mrs x2, sp_el0          (699516 hits)   (op=0xd5384102/  System Reg)
Instr: mrs x4, tpidr_el2       (528437 hits)   (op=0xd53cd044/  System Reg)
Instr: mrs x30, ttbr1_el1      (480776 hits)   (op=0xd538203e/  System Reg)
Instr: msr ttbr1_el1, x30      (480713 hits)   (op=0xd518203e/  System Reg)
Instr: msr vbar_el1, x30       (480671 hits)   (op=0xd518c01e/  System Reg)
...
```

To find the argument shorthand for the class you need to examine the source code of the plugin at the moment, specif-

ically the *opt argument in the InsnClassExecCount tables.

- contrib/plugins/lockstep.c

This is a debugging tool for developers who want to find out when and where execution diverges after a subtle change to TCG code generation. It is not an exact science and results are likely to be mixed once asynchronous events are introduced. While the use of -icount can introduce determinism to the execution flow it doesn't always follow the translation sequence will be exactly the same. Typically this is caused by a timer firing to service the GUI causing a block to end early. However in some cases it has proved to be useful in pointing people at roughly where execution diverges. The only argument you need for the plugin is a path for the socket the two instances will communicate over:

```
$ qemu-system-sparc -monitor none -parallel none \
  -net none -M SS-20 -m 256 -kernel day11/zImage.elf \
  -plugin ./contrib/plugins/liblockstep.so,sockpath=lockstep-sparc.sock \
  -d plugin,nochain
```

which will eventually report:

```
qemu-system-sparc: warning: nic lance.0 has no peer
@ 0x000000ffd06678 vs 0x000000ffd001e0 (2/1 since last)
@ 0x000000ffd07d9c vs 0x000000ffd06678 (3/1 since last)
 insn_count @ 0x000000ffd07d9c (809900609) vs 0x000000ffd06678 (809900612)
  previously @ 0x000000ffd06678/10 (809900609 insns)
  previously @ 0x000000ffd001e0/4 (809900599 insns)
  previously @ 0x000000ffd080ac/2 (809900595 insns)
  previously @ 0x000000ffd08098/5 (809900593 insns)
  previously @ 0x000000ffd080c0/1 (809900588 insns)
```

- contrib/plugins/hwprofile.c

The hwprofile tool can only be used with system emulation and allows the user to see what hardware is accessed how often. It has a number of options:

  - track=read or track=write

  By default the plugin tracks both reads and writes. You can use one of these options to limit the tracking to just one class of accesses.

  - source

  Will include a detailed break down of what the guest PC that made the access was. Not compatible with the pattern option. Example output:

```
cirrus-low-memory @ 0xfffffd00000a0000
 pc:fffffc0000005cdc, 1, 256
 pc:fffffc0000005ce8, 1, 256
 pc:fffffc0000005cec, 1, 256
```

  - pattern

  Instead break down the accesses based on the offset into the HW region. This can be useful for seeing the most used registers of a device. Example output:

```
pci0-conf @ 0xfffffd01fe000000
  off:00000004, 1, 1
  off:00000010, 1, 3
  off:00000014, 1, 3
  off:00000018, 1, 2
```

(continues on next page)

(continued from previous page)

```
  off:0000001c, 1, 2
  off:00000020, 1, 2
  ...
```

- contrib/plugins/execlog.c

The execlog tool traces executed instructions with memory access. It can be used for debugging and security analysis purposes. Please be aware that this will generate a lot of output.

The plugin needs default argument:

```
$ qemu-system-arm $(QEMU_ARGS) \
  -plugin ./contrib/plugins/libexeclog.so -d plugin
```

which will output an execution trace following this structure:

```
# vCPU, vAddr, opcode, disassembly[, load/store, memory addr, device]...
0, 0xa12, 0xf8012400, "movs r4, #0"
0, 0xa14, 0xf87f42b4, "cmp r4, r6"
0, 0xa16, 0xd206, "bhs #0xa26"
0, 0xa18, 0xfff94803, "ldr r0, [pc, #0xc]", load, 0x00010a28, RAM
0, 0xa1a, 0xf989f000, "bl #0xd30"
0, 0xd30, 0xfff9b510, "push {r4, lr}", store, 0x20003ee0, RAM, store, 0x20003ee4, RAM
0, 0xd32, 0xf9893014, "adds r0, #0x14"
0, 0xd34, 0xf9c8f000, "bl #0x10c8"
0, 0x10c8, 0xfff96c43, "ldr r3, [r0, #0x44]", load, 0x200000e4, RAM
```

the output can be filtered to only track certain instructions or addresses using the `ifilter` or `afilter` options. You can stack the arguments if required:

```
$ qemu-system-arm $(QEMU_ARGS) \
  -plugin ./contrib/plugins/libexeclog.so,ifilter=st1w,afilter=0x40001808 -d plugin
```

This plugin can also dump registers when they change value. Specify the name of the registers with multiple `reg` options. You can also use glob style matching if you wish:

```
$ qemu-system-arm $(QEMU_ARGS) \
  -plugin ./contrib/plugins/libexeclog.so,reg=\*_el2,reg=sp -d plugin
```

Be aware that each additional register to check will slow down execution quite considerably. You can optimise the number of register checks done by using the rdisas option. This will only instrument instructions that mention the registers in question in disassembly. This is not foolproof as some instructions implicitly change instructions. You can use the ifilter to catch these cases:

$ qemu-system-arm $(QEMU_ARGS)
-plugin ./contrib/plugins/libexeclog.so,ifilter=msr,ifilter=blr,reg=x30,reg=*_el1,rdisas=on

- contrib/plugins/cache.c

Cache modelling plugin that measures the performance of a given L1 cache configuration, and optionally a unified L2 per-core cache when a given working set is run:

```
$ qemu-x86_64 -plugin ./contrib/plugins/libcache.so \
    -d plugin -D cache.log ./tests/tcg/x86_64-linux-user/float_convs
```

will report the following:

```
core #, data accesses, data misses, dmiss rate, insn accesses, insn misses, imiss rate
0       996695         508             0.0510%  2642799        18617           0.7044%

address, data misses, instruction
0x424f1e (_int_malloc), 109, movq %rax, 8(%rcx)
0x41f395 (_IO_default_xsputn), 49, movb %dl, (%rdi, %rax)
0x42584d (ptmalloc_init.part.0), 33, movaps %xmm0, (%rax)
0x454d48 (__tunables_init), 20, cmpb $0, (%r8)
...

address, fetch misses, instruction
0x4160a0 (__vfprintf_internal), 744, movl $1, %ebx
0x41f0a0 (_IO_setb), 744, endbr64
0x415882 (__vfprintf_internal), 744, movq %r12, %rdi
0x4268a0 (__malloc), 696, andq $0xfffffffffffffff0, %rax
...
```

The plugin has a number of arguments, all of them are optional:

- limit=N

Print top N icache and dcache thrashing instructions along with their address, number of misses, and its disassembly. (default: 32)

- icachesize=N
- iblksize=B
- iassoc=A

Instruction cache configuration arguments. They specify the cache size, block size, and associativity of the instruction cache, respectively. (default: N = 16384, B = 64, A = 8)

- dcachesize=N
- dblksize=B
- dassoc=A

Data cache configuration arguments. They specify the cache size, block size, and associativity of the data cache, respectively. (default: N = 16384, B = 64, A = 8)

- evict=POLICY

Sets the eviction policy to POLICY. Available policies are: `lru`, `fifo`, and `rand`. The plugin will use the specified policy for both instruction and data caches. (default: POLICY = `lru`)

- cores=N

Sets the number of cores for which we maintain separate icache and dcache. (default: for linux-user, N = 1, for full system emulation: N = cores available to guest)

- l2=on

Simulates a unified L2 cache (stores blocks for both instructions and data) using the default L2 configuration (cache size = 2MB, associativity = 16-way, block size = 64B).

- l2cachesize=N
- l2blksize=B
- l2assoc=A

L2 cache configuration arguments. They specify the cache size, block size, and associativity of the L2 cache, respectively. Setting any of the L2 configuration arguments implies `l2=on`. (default: N = 2097152 (2MB), B = 64, A = 16)

7.5.8 Plugin API

The following API is generated from the inline documentation in `include/qemu/qemu-plugin.h`. Please ensure any updates to the API include the full kernel-doc annotations.

type **qemu_plugin_id_t**

Unique plugin ID

struct **qemu_info_t**

system information for plugins

Definition

```
struct qemu_info_t {
  const char *target_name;
  struct {
    int min;
    int cur;
  } version;
  bool system_emulation;
  union {
    struct {
      int smp_vcpus;
      int max_vcpus;
    } system;
  };
};
```

Members

target_name
string describing architecture

version
minimum and current plugin API level

system_emulation
is this a full system emulation?

{unnamed_union}
anonymous

system
information relevant to system emulation

Description

This structure provides for some limited information about the system to allow the plugin to make decisions on how to proceed. For example it might only be suitable for running on some guest architectures or when under full system emulation.

int **qemu_plugin_install**(*qemu_plugin_id_t* id, const *qemu_info_t* *info, int argc, char **argv)

Install a plugin

Parameters

`qemu_plugin_id_t id`
: this plugin's opaque ID

`const qemu_info_t *info`
: a block describing some details about the guest

`int argc`
: number of arguments

`char **argv`
: array of arguments (**argc** elements)

Description

All plugins must export this symbol which is called when the plugin is first loaded. Calling qemu_plugin_uninstall() from this function is a bug.

Note

info is only live during the call. Copy any information we want to keep. **argv** remains valid throughout the lifetime of the loaded plugin.

Return

0 on successful loading, !0 for an error.

`qemu_plugin_simple_cb_t`

Typedef: simple callback

Syntax

```
void qemu_plugin_simple_cb_t (qemu_plugin_id_t id)
```

Parameters

`qemu_plugin_id_t id`
: the unique qemu_plugin_id_t

Description

This callback passes no information aside from the unique **id**.

`qemu_plugin_udata_cb_t`

Typedef: callback with user data

Syntax

```
void qemu_plugin_udata_cb_t (qemu_plugin_id_t id, void *userdata)
```

Parameters

`qemu_plugin_id_t id`
: the unique qemu_plugin_id_t

`void *userdata`
: a pointer to some user data supplied when the callback was registered.

`qemu_plugin_vcpu_simple_cb_t`

Typedef: vcpu callback

Syntax

```
void qemu_plugin_vcpu_simple_cb_t (qemu_plugin_id_t id, unsigned int
vcpu_index)
```

Parameters

qemu_plugin_id_t id
the unique qemu_plugin_id_t

unsigned int vcpu_index
the current vcpu context

qemu_plugin_vcpu_udata_cb_t

Typedef: vcpu callback

Syntax

```
void qemu_plugin_vcpu_udata_cb_t (unsigned int vcpu_index, void *userdata)
```

Parameters

unsigned int vcpu_index
the current vcpu context

void *userdata
a pointer to some user data supplied when the callback was registered.

void **qemu_plugin_uninstall**(*qemu_plugin_id_t* id, *qemu_plugin_simple_cb_t* cb)

Uninstall a plugin

Parameters

qemu_plugin_id_t id
this plugin's opaque ID

qemu_plugin_simple_cb_t cb
callback to be called once the plugin has been removed

Description

Do NOT assume that the plugin has been uninstalled once this function returns. Plugins are uninstalled asynchronously, and therefore the given plugin receives callbacks until **cb** is called.

Note

Calling this function from qemu_plugin_install() is a bug.

void **qemu_plugin_reset**(*qemu_plugin_id_t* id, *qemu_plugin_simple_cb_t* cb)

Reset a plugin

Parameters

qemu_plugin_id_t id
this plugin's opaque ID

qemu_plugin_simple_cb_t cb
callback to be called once the plugin has been reset

Description

Unregisters all callbacks for the plugin given by **id**.

Do NOT assume that the plugin has been reset once this function returns. Plugins are reset asynchronously, and therefore the given plugin receives callbacks until **cb** is called.

void **qemu_plugin_register_vcpu_init_cb**(*qemu_plugin_id_t* id, *qemu_plugin_vcpu_simple_cb_t* cb)

register a vCPU initialization callback

Parameters

qemu_plugin_id_t id
plugin ID

qemu_plugin_vcpu_simple_cb_t cb
callback function

Description

The **cb** function is called every time a vCPU is initialized.

See also: qemu_plugin_register_vcpu_exit_cb()

void **qemu_plugin_register_vcpu_exit_cb**(*qemu_plugin_id_t* id, *qemu_plugin_vcpu_simple_cb_t* cb)
register a vCPU exit callback

Parameters

qemu_plugin_id_t id
plugin ID

qemu_plugin_vcpu_simple_cb_t cb
callback function

Description

The **cb** function is called every time a vCPU exits.

See also: qemu_plugin_register_vcpu_init_cb()

void **qemu_plugin_register_vcpu_idle_cb**(*qemu_plugin_id_t* id, *qemu_plugin_vcpu_simple_cb_t* cb)
register a vCPU idle callback

Parameters

qemu_plugin_id_t id
plugin ID

qemu_plugin_vcpu_simple_cb_t cb
callback function

Description

The **cb** function is called every time a vCPU idles.

void **qemu_plugin_register_vcpu_resume_cb**(*qemu_plugin_id_t* id, *qemu_plugin_vcpu_simple_cb_t* cb)
register a vCPU resume callback

Parameters

qemu_plugin_id_t id
plugin ID

qemu_plugin_vcpu_simple_cb_t cb
callback function

Description

The **cb** function is called every time a vCPU resumes execution.

type **qemu_plugin_u64**
uint64_t member of an entry in a scoreboard

Description

This field allows to access a specific uint64_t member in one given entry, located at a specified offset. Inline operations expect this as entry.

enum **qemu_plugin_cb_flags**

type of callback

Constants

QEMU_PLUGIN_CB_NO_REGS
callback does not access the CPU's regs

QEMU_PLUGIN_CB_R_REGS
callback reads the CPU's regs

QEMU_PLUGIN_CB_RW_REGS
callback reads and writes the CPU's regs

Note

currently QEMU_PLUGIN_CB_RW_REGS is unused, plugins cannot change system register state.

enum **qemu_plugin_cond**

condition to enable callback

Constants

QEMU_PLUGIN_COND_NEVER
false

QEMU_PLUGIN_COND_ALWAYS
true

QEMU_PLUGIN_COND_EQ
is equal?

QEMU_PLUGIN_COND_NE
is not equal?

QEMU_PLUGIN_COND_LT
is less than?

QEMU_PLUGIN_COND_LE
is less than or equal?

QEMU_PLUGIN_COND_GT
is greater than?

QEMU_PLUGIN_COND_GE
is greater than or equal?

qemu_plugin_vcpu_tb_trans_cb_t

Typedef: translation callback

Syntax

```
void qemu_plugin_vcpu_tb_trans_cb_t (qemu_plugin_id_t id, struct qemu_plugin_tb
*tb)
```

Parameters

qemu_plugin_id_t id
unique plugin id

struct qemu_plugin_tb *tb
opaque handle used for querying and instrumenting a block.

void **qemu_plugin_register_vcpu_tb_trans_cb**(*qemu_plugin_id_t* id, *qemu_plugin_vcpu_tb_trans_cb_t* cb)

register a translate cb

Parameters

qemu_plugin_id_t id
plugin ID

qemu_plugin_vcpu_tb_trans_cb_t cb
callback function

Description

The **cb** function is called every time a translation occurs. The **cb** function is passed an opaque qemu_plugin_type which it can query for additional information including the list of translated instructions. At this point the plugin can register further callbacks to be triggered when the block or individual instruction executes.

void **qemu_plugin_register_vcpu_tb_exec_cb**(struct qemu_plugin_tb *tb, *qemu_plugin_vcpu_udata_cb_t* cb, enum *qemu_plugin_cb_flags* flags, void *userdata)

register execution callback

Parameters

struct qemu_plugin_tb *tb
the opaque qemu_plugin_tb handle for the translation

qemu_plugin_vcpu_udata_cb_t cb
callback function

enum qemu_plugin_cb_flags flags
does the plugin read or write the CPU's registers?

void *userdata
any plugin data to pass to the **cb**?

Description

The **cb** function is called every time a translated unit executes.

void **qemu_plugin_register_vcpu_tb_exec_cond_cb**(struct qemu_plugin_tb *tb, *qemu_plugin_vcpu_udata_cb_t* cb, enum *qemu_plugin_cb_flags* flags, enum *qemu_plugin_cond* cond, *qemu_plugin_u64* entry, uint64_t imm, void *userdata)

register conditional callback

Parameters

struct qemu_plugin_tb *tb
the opaque qemu_plugin_tb handle for the translation

qemu_plugin_vcpu_udata_cb_t cb
callback function

enum qemu_plugin_cb_flags flags
does the plugin read or write the CPU's registers?

enum qemu_plugin_cond cond
condition to enable callback

qemu_plugin_u64 entry
first operand for condition

uint64_t imm
second operand for condition

void *userdata
any plugin data to pass to the **cb**?

Description

The **cb** function is called when a translated unit executes if entry **cond** imm is true. If condition is QEMU_PLUGIN_COND_ALWAYS, condition is never interpreted and this function is equivalent to qemu_plugin_register_vcpu_tb_exec_cb. If condition QEMU_PLUGIN_COND_NEVER, condition is never interpreted and callback is never installed.

enum **qemu_plugin_op**
describes an inline op

Constants

QEMU_PLUGIN_INLINE_ADD_U64
add an immediate value uint64_t

QEMU_PLUGIN_INLINE_STORE_U64
store an immediate value uint64_t

void **qemu_plugin_register_vcpu_tb_exec_inline_per_vcpu**(struct qemu_plugin_tb *tb, enum *qemu_plugin_op* op, *qemu_plugin_u64* entry, uint64_t imm)
execution inline op

Parameters

struct qemu_plugin_tb *tb
the opaque qemu_plugin_tb handle for the translation

enum qemu_plugin_op op
the type of qemu_plugin_op (e.g. ADD_U64)

qemu_plugin_u64 entry
entry to run op

uint64_t imm
the op data (e.g. 1)

Description

Insert an inline op on a given scoreboard entry.

void **qemu_plugin_register_vcpu_insn_exec_cb**(struct qemu_plugin_insn *insn, *qemu_plugin_vcpu_udata_cb_t* cb, enum *qemu_plugin_cb_flags* flags, void *userdata)
register insn execution cb

Parameters

struct qemu_plugin_insn *insn
the opaque qemu_plugin_insn handle for an instruction

qemu_plugin_vcpu_udata_cb_t cb
callback function

enum qemu_plugin_cb_flags flags
does the plugin read or write the CPU's registers?

void *userdata
any plugin data to pass to the **cb**?

Description

The **cb** function is called every time an instruction is executed

void **qemu_plugin_register_vcpu_insn_exec_cond_cb**(struct qemu_plugin_insn *insn, *qemu_plugin_vcpu_udata_cb_t* cb, enum *qemu_plugin_cb_flags* flags, enum *qemu_plugin_cond* cond, *qemu_plugin_u64* entry, uint64_t imm, void *userdata)

conditional insn execution cb

Parameters

struct qemu_plugin_insn *insn
the opaque qemu_plugin_insn handle for an instruction

qemu_plugin_vcpu_udata_cb_t cb
callback function

enum qemu_plugin_cb_flags flags
does the plugin read or write the CPU's registers?

enum qemu_plugin_cond cond
condition to enable callback

qemu_plugin_u64 entry
first operand for condition

uint64_t imm
second operand for condition

void *userdata
any plugin data to pass to the **cb**?

Description

The **cb** function is called when an instruction executes if entry **cond** imm is true. If condition is QEMU_PLUGIN_COND_ALWAYS, condition is never interpreted and this function is equivalent to qemu_plugin_register_vcpu_insn_exec_cb. If condition QEMU_PLUGIN_COND_NEVER, condition is never interpreted and callback is never installed.

void **qemu_plugin_register_vcpu_insn_exec_inline_per_vcpu**(struct qemu_plugin_insn *insn, enum *qemu_plugin_op* op, *qemu_plugin_u64* entry, uint64_t imm)

insn exec inline op

Parameters

struct qemu_plugin_insn *insn
the opaque qemu_plugin_insn handle for an instruction

enum qemu_plugin_op op
the type of qemu_plugin_op (e.g. ADD_U64)

qemu_plugin_u64 entry
entry to run op

uint64_t imm
the op data (e.g. 1)

Description

Insert an inline op to every time an instruction executes.

size_t **qemu_plugin_tb_n_insns**(const struct qemu_plugin_tb *tb)

query helper for number of insns in TB

Parameters

const struct qemu_plugin_tb *tb
opaque handle to TB passed to callback

Return

number of instructions in this block

uint64_t **qemu_plugin_tb_vaddr**(const struct qemu_plugin_tb *tb)

query helper for vaddr of TB start

Parameters

const struct qemu_plugin_tb *tb
opaque handle to TB passed to callback

Return

virtual address of block start

struct qemu_plugin_insn ***qemu_plugin_tb_get_insn**(const struct qemu_plugin_tb *tb, size_t idx)

retrieve handle for instruction

Parameters

const struct qemu_plugin_tb *tb
opaque handle to TB passed to callback

size_t idx
instruction number, 0 indexed

Description

The returned handle can be used in follow up helper queries as well as when instrumenting an instruction. It is only valid for the lifetime of the callback.

Return

opaque handle to instruction

size_t **qemu_plugin_insn_data**(const struct qemu_plugin_insn *insn, void *dest, size_t len)

copy instruction data

Parameters

const struct qemu_plugin_insn *insn
opaque instruction handle from qemu_plugin_tb_get_insn()

void *dest
destination into which data is copied

size_t len
length of dest

Description

Returns the number of bytes copied, minimum of **len** and insn size.

size_t **qemu_plugin_insn_size**(const struct qemu_plugin_insn *insn)

return size of instruction

Parameters

const struct qemu_plugin_insn *insn
opaque instruction handle from qemu_plugin_tb_get_insn()

Return

size of instruction in bytes

uint64_t **qemu_plugin_insn_vaddr**(const struct qemu_plugin_insn *insn)

return vaddr of instruction

Parameters

const struct qemu_plugin_insn *insn
opaque instruction handle from qemu_plugin_tb_get_insn()

Return

virtual address of instruction

void ***qemu_plugin_insn_haddr**(const struct qemu_plugin_insn *insn)

return hardware addr of instruction

Parameters

const struct qemu_plugin_insn *insn
opaque instruction handle from qemu_plugin_tb_get_insn()

Return

hardware (physical) target address of instruction

type **qemu_plugin_meminfo_t**

opaque memory transaction handle

Description

This can be further queried using the qemu_plugin_mem_* query functions.

unsigned int **qemu_plugin_mem_size_shift**(*qemu_plugin_meminfo_t* info)

get size of access

Parameters

qemu_plugin_meminfo_t info
opaque memory transaction handle

Return

size of access in ^2 (0=byte, 1=16bit, 2=32bit etc...)

bool **qemu_plugin_mem_is_sign_extended**(*qemu_plugin_meminfo_t* info)

was the access sign extended

Parameters

qemu_plugin_meminfo_t info
opaque memory transaction handle

Return

true if it was, otherwise false

bool **qemu_plugin_mem_is_big_endian**(*qemu_plugin_meminfo_t* info)

was the access big endian

Parameters

qemu_plugin_meminfo_t info
opaque memory transaction handle

Return

true if it was, otherwise false

bool **qemu_plugin_mem_is_store**(*qemu_plugin_meminfo_t* info)

was the access a store

Parameters

qemu_plugin_meminfo_t info
opaque memory transaction handle

Return

true if it was, otherwise false

struct qemu_plugin_hwaddr ***qemu_plugin_get_hwaddr**(*qemu_plugin_meminfo_t* info, uint64_t vaddr)

return handle for memory operation

Parameters

qemu_plugin_meminfo_t info
opaque memory info structure

uint64_t vaddr
the virtual address of the memory operation

Description

For system emulation returns a qemu_plugin_hwaddr handle to query details about the actual physical address backing the virtual address. For linux-user guests it just returns NULL.

This handle is *only* valid for the duration of the callback. Any information about the handle should be recovered before the callback returns.

bool **qemu_plugin_hwaddr_is_io**(const struct qemu_plugin_hwaddr *haddr)

query whether memory operation is IO

Parameters

const struct qemu_plugin_hwaddr *haddr
address handle from qemu_plugin_get_hwaddr()

Description

Returns true if the handle's memory operation is to memory-mapped IO, or false if it is to RAM

uint64_t **qemu_plugin_hwaddr_phys_addr**(const struct qemu_plugin_hwaddr *haddr)

query physical address for memory operation

Parameters

const struct qemu_plugin_hwaddr *haddr
address handle from qemu_plugin_get_hwaddr()

Description

Returns the physical address associated with the memory operation

Note that the returned physical address may not be unique if you are dealing with multiple address spaces.

`qemu_plugin_vcpu_mem_cb_t`

Typedef: memory callback function type

Syntax

```
void qemu_plugin_vcpu_mem_cb_t (unsigned int vcpu_index, qemu_plugin_meminfo_t
info, uint64_t vaddr, void *userdata)
```

Parameters

`unsigned int vcpu_index`
the executing vCPU

`qemu_plugin_meminfo_t info`
an opaque handle for further queries about the memory

`uint64_t vaddr`
the virtual address of the transaction

`void *userdata`
any user data attached to the callback

void **qemu_plugin_register_vcpu_mem_cb**(struct qemu_plugin_insn *insn, *qemu_plugin_vcpu_mem_cb_t* cb, enum *qemu_plugin_cb_flags* flags, enum qemu_plugin_mem_rw rw, void *userdata)

register memory access callback

Parameters

`struct qemu_plugin_insn *insn`
handle for instruction to instrument

`qemu_plugin_vcpu_mem_cb_t cb`
callback of type qemu_plugin_vcpu_mem_cb_t

`enum qemu_plugin_cb_flags flags`
(currently unused) callback flags

`enum qemu_plugin_mem_rw rw`
monitor reads, writes or both

`void *userdata`
opaque pointer for userdata

Description

This registers a full callback for every memory access generated by an instruction. If the instruction doesn't access memory no callback will be made.

The callback reports the vCPU the access took place on, the virtual address of the access and a handle for further queries. The user can attach some userdata to the callback for additional purposes.

Other execution threads will continue to execute during the callback so the plugin is responsible for ensuring it doesn't get confused by making appropriate use of locking if required.

void **qemu_plugin_register_vcpu_mem_inline_per_vcpu**(struct qemu_plugin_insn *insn, enum qemu_plugin_mem_rw rw, enum *qemu_plugin_op* op, *qemu_plugin_u64* entry, uint64_t imm)

inline op for mem access

Parameters

struct qemu_plugin_insn *insn
handle for instruction to instrument

enum qemu_plugin_mem_rw rw
apply to reads, writes or both

enum qemu_plugin_op op
the op, of type qemu_plugin_op

qemu_plugin_u64 entry
entry to run op

uint64_t imm
immediate data for **op**

Description

This registers a inline op every memory access generated by the instruction.

char ***qemu_plugin_insn_disas**(const struct qemu_plugin_insn *insn)

return disassembly string for instruction

Parameters

const struct qemu_plugin_insn *insn
instruction reference

Description

Returns an allocated string containing the disassembly

const char ***qemu_plugin_insn_symbol**(const struct qemu_plugin_insn *insn)

best effort symbol lookup

Parameters

const struct qemu_plugin_insn *insn
instruction reference

Description

Return a static string referring to the symbol. This is dependent on the binary QEMU is running having provided a symbol table.

void **qemu_plugin_vcpu_for_each**(*qemu_plugin_id_t* id, *qemu_plugin_vcpu_simple_cb_t* cb)

iterate over the existing vCPU

Parameters

qemu_plugin_id_t id
plugin ID

qemu_plugin_vcpu_simple_cb_t cb
callback function

Description

The **cb** function is called once for each existing vCPU.

See also: qemu_plugin_register_vcpu_init_cb()

void **qemu_plugin_register_atexit_cb**(*qemu_plugin_id_t* id, *qemu_plugin_udata_cb_t* cb, void *userdata)

register exit callback

Parameters

qemu_plugin_id_t id
plugin ID

qemu_plugin_udata_cb_t cb
callback

void *userdata
user data for callback

Description

The **cb** function is called once execution has finished. Plugins should be able to free all their resources at this point much like after a reset/uninstall callback is called.

In user-mode it is possible a few un-instrumented instructions from child threads may run before the host kernel reaps the threads.

void **qemu_plugin_outs**(const char *string)

output string via QEMU's logging system

Parameters

const char *string
a string

bool **qemu_plugin_bool_parse**(const char *name, const char *val, bool *ret)

parses a boolean argument in the form of "<argname>=[on|yes|true|off|no|false]"

Parameters

const char *name
argument name, the part before the equals sign

const char *val
argument value, what's after the equals sign

bool *ret
output return value

Description

returns true if the combination **name**=**val** parses correctly to a boolean argument, and false otherwise

const char ***qemu_plugin_path_to_binary**(void)

path to binary file being executed

Parameters

void
no arguments

Description

Return a string representing the path to the binary. For user-mode this is the main executable. For system emulation we currently return NULL. The user should g_free() the string once no longer needed.

uint64_t **qemu_plugin_start_code**(void)

returns start of text segment

Parameters

void
no arguments

Description

Returns the nominal start address of the main text segment in user-mode. Currently returns 0 for system emulation.

uint64_t **qemu_plugin_end_code**(void)
returns end of text segment

Parameters

void
no arguments

Description

Returns the nominal end address of the main text segment in user-mode. Currently returns 0 for system emulation.

uint64_t **qemu_plugin_entry_code**(void)
returns start address for module

Parameters

void
no arguments

Description

Returns the nominal entry address of the main text segment in user-mode. Currently returns 0 for system emulation.

type **qemu_plugin_reg_descriptor**
register descriptions

GArray ***qemu_plugin_get_registers**(void)
return register list for current vCPU

Parameters

void
no arguments

Description

Returns a potentially empty GArray of qemu_plugin_reg_descriptor. Caller frees the array (but not the const strings).

Should be used from a qemu_plugin_register_vcpu_init_cb() callback after the vCPU is initialised, i.e. in the vCPU context.

int **qemu_plugin_read_register**(struct qemu_plugin_register *handle, GByteArray *buf)
read register for current vCPU

Parameters

struct qemu_plugin_register *handle
a **qemu_plugin_reg_handle** handle

GByteArray *buf
A GByteArray for the data owned by the plugin

Description

This function is only available in a context that register read access is explicitly requested via the QEMU_PLUGIN_CB_R_REGS flag.

Returns the size of the read register. The content of **buf** is in target byte order. On failure returns -1.

struct qemu_plugin_scoreboard ***qemu_plugin_scoreboard_new**(size_t element_size)

alloc a new scoreboard

Parameters

size_t element_size
size (in bytes) for one entry

Description

Returns a pointer to a new scoreboard. It must be freed using qemu_plugin_scoreboard_free.

void **qemu_plugin_scoreboard_free**(struct qemu_plugin_scoreboard *score)

free a scoreboard

Parameters

struct qemu_plugin_scoreboard *score
scoreboard to free

void ***qemu_plugin_scoreboard_find**(struct qemu_plugin_scoreboard *score, unsigned int vcpu_index)

get pointer to an entry of a scoreboard

Parameters

struct qemu_plugin_scoreboard *score
scoreboard to query

unsigned int vcpu_index
entry index

Description

Returns address of entry of a scoreboard matching a given vcpu_index. This address can be modified later if scoreboard is resized.

void **qemu_plugin_u64_add**(*qemu_plugin_u64* entry, unsigned int vcpu_index, uint64_t added)

add a value to a qemu_plugin_u64 for a given vcpu

Parameters

qemu_plugin_u64 entry
entry to query

unsigned int vcpu_index
entry index

uint64_t added
value to add

uint64_t **qemu_plugin_u64_get**(*qemu_plugin_u64* entry, unsigned int vcpu_index)

get value of a qemu_plugin_u64 for a given vcpu

Parameters

qemu_plugin_u64 entry
entry to query

unsigned int vcpu_index
entry index

void **qemu_plugin_u64_set**(*qemu_plugin_u64* entry, unsigned int vcpu_index, uint64_t val)

set value of a qemu_plugin_u64 for a given vcpu

Parameters

qemu_plugin_u64 entry
entry to query

unsigned int vcpu_index
entry index

uint64_t val
new value

uint64_t **qemu_plugin_u64_sum**(*qemu_plugin_u64* entry)

return sum of all vcpu entries in a scoreboard

Parameters

qemu_plugin_u64 entry
entry to sum

7.5.9 Execution Record/Replay

Core concepts

Record/replay functions are used for the deterministic replay of qemu execution. Execution recording writes a non-deterministic events log, which can be later used for replaying the execution anywhere and for unlimited number of times. Execution replaying reads the log and replays all non-deterministic events including external input, hardware clocks, and interrupts.

Several parts of QEMU include function calls to make event log recording and replaying. Devices' models that have non-deterministic input from external devices were changed to write every external event into the execution log immediately. E.g. network packets are written into the log when they arrive into the virtual network adapter.

All non-deterministic events are coming from these devices. But to replay them we need to know at which moments they occur. We specify these moments by counting the number of instructions executed between every pair of consecutive events.

Academic papers with description of deterministic replay implementation:

- Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging
- Don't panic: reverse debugging of kernel drivers

Modifications of qemu include:

- wrappers for clock and time functions to save their return values in the log
- saving different asynchronous events (e.g. system shutdown) into the log
- synchronization of the bottom halves execution
- synchronization of the threads from thread pool
- recording/replaying user input (mouse, keyboard, and microphone)
- adding internal checkpoints for cpu and io synchronization
- network filter for recording and replaying the packets
- block driver for making block layer deterministic
- serial port input record and replay
- recording of random numbers obtained from the external sources

Instruction counting

QEMU should work in icount mode to use record/replay feature. icount was designed to allow deterministic execution in absence of external inputs of the virtual machine. We also use icount to control the occurrence of the non-deterministic events. The number of instructions elapsed from the last event is written to the log while recording the execution. In replay mode we can predict when to inject that event using the instruction counter.

Locking and thread synchronisation

Previously the synchronisation of the main thread and the vCPU thread was ensured by the holding of the BQL. However the trend has been to reduce the time the BQL was held across the system including under TCG system emulation. As it is important that batches of events are kept in sequence (e.g. expiring timers and checkpoints in the main thread while instruction checkpoints are written by the vCPU thread) we need another lock to keep things in lock-step. This role is now handled by the replay_mutex_lock. It used to be held only for each event being written but now it is held for a whole execution period. This results in a deterministic ping-pong between the two main threads.

As the BQL is now a finer grained lock than the replay_lock it is almost certainly a bug, and a source of deadlocks, to take the replay_mutex_lock while the BQL is held. This is enforced by an assert. While the unlocks are usually in the reverse order, this is not necessary; you can drop the replay_lock while holding the BQL, without doing a more complicated unlock_iothread/replay_unlock/lock_iothread sequence.

Checkpoints

Replaying the execution of virtual machine is bound by sources of non-determinism. These are inputs from clock and peripheral devices, and QEMU thread scheduling. Thread scheduling affect on processing events from timers, asynchronous input-output, and bottom halves.

Invocations of timers are coupled with clock reads and changing the state of the virtual machine. Reads produce non-deterministic data taken from host clock. And VM state changes should preserve their order. Their relative order in replay mode must replicate the order of callbacks in record mode. To preserve this order we use checkpoints. When a specific clock is processed in record mode we save to the log special "checkpoint" event. Checkpoints here do not refer to virtual machine snapshots. They are just record/replay events used for synchronization.

QEMU in replay mode will try to invoke timers processing in random moment of time. That's why we do not process a group of timers until the checkpoint event will be read from the log. Such an event allows synchronizing CPU execution and timer events.

Two other checkpoints govern the "warping" of the virtual clock. While the virtual machine is idle, the virtual clock increments at 1 ns per *real time* nanosecond. This is done by setting up a timer (called the warp timer) on the virtual real time clock, so that the timer fires at the next deadline of the virtual clock; the virtual clock is then incremented (which is called "warping" the virtual clock) as soon as the timer fires or the CPUs need to go out of the idle state. Two functions are used for this purpose; because these actions change virtual machine state and must be deterministic, each of them creates a checkpoint. `icount_start_warp_timer` checks if the CPUs are idle and if so starts accounting real time to virtual clock. `icount_account_warp_timer` is called when the CPUs get an interrupt or when the warp timer fires, and it warps the virtual clock by the amount of real time that has passed since `icount_start_warp_timer`.

Virtual devices

Record/replay mechanism, that could be enabled through icount mode, expects the virtual devices to satisfy the following requirement: everything that affects the guest state during execution in icount mode should be deterministic.

Timers

Timers are used to execute callbacks from different subsystems of QEMU at the specified moments of time. There are several kinds of timers:

- Real time clock. Based on host time and used only for callbacks that do not change the virtual machine state. For this reason real time clock and timers does not affect deterministic replay at all.
- Virtual clock. These timers run only during the emulation. In icount mode virtual clock value is calculated using executed instructions counter. That is why it is completely deterministic and does not have to be recorded.
- Host clock. This clock is used by device models that simulate real time sources (e.g. real time clock chip). Host clock is the one of the sources of non-determinism. Host clock read operations should be logged to make the execution deterministic.
- Virtual real time clock. This clock is similar to real time clock but it is used only for increasing virtual clock while virtual machine is sleeping. Due to its nature it is also non-deterministic as the host clock and has to be logged too.

All virtual devices should use virtual clock for timers that change the guest state. Virtual clock is deterministic, therefore such timers are deterministic too.

Virtual devices can also use realtime clock for the events that do not change the guest state directly. When the clock ticking should depend on VM execution speed, use virtual clock with EXTERNAL attribute. It is not deterministic, but its speed depends on the guest execution. This clock is used by the virtual devices (e.g., slirp routing device) that lie outside the replayed guest.

Block devices

Block devices record/replay module (`blkreplay`) intercepts calls of bdrv coroutine functions at the top of block drivers stack.

All block completion operations are added to the queue in the coroutines. When the queue is flushed the information about processed requests is recorded to the log. In replay phase the queue is matched with events read from the log. Therefore block devices requests are processed deterministically.

Bottom halves

Bottom half callbacks, that affect the guest state, should be invoked through `replay_bh_schedule_event` or `replay_bh_schedule_oneshot_event` functions. Their invocations are saved in record mode and synchronized with the existing log in replay mode.

Disk I/O events are completely deterministic in our model, because in both record and replay modes we start virtual machine from the same disk state. But callbacks that virtual disk controller uses for reading and writing the disk may occur at different moments of time in record and replay modes.

Reading and writing requests are created by CPU thread of QEMU. Later these requests proceed to block layer which creates “bottom halves”. Bottom halves consist of callback and its parameters. They are processed when main loop locks the BQL. These locks are not synchronized with replaying process because main loop also processes the events that do not affect the virtual machine state (like user interaction with monitor).

That is why we had to implement saving and replaying bottom halves callbacks synchronously to the CPU execution. When the callback is about to execute it is added to the queue in the replay module. This queue is written to the log when its callbacks are executed. In replay mode callbacks are not processed until the corresponding event is read from the events log file.

Sometimes the block layer uses asynchronous callbacks for its internal purposes (like reading or writing VM snapshots or disk image cluster tables). In this case bottom halves are not marked as "replayable" and do not saved into the log.

Saving/restoring the VM state

All fields in the device state structure (including virtual timers) should be restored by loadvm to the same values they had before savevm.

Avoid accessing other devices' state, because the order of saving/restoring is not defined. It means that you should not call functions like `update_irq` in `post_load` callback. Save everything explicitly to avoid the dependencies that may make restoring the VM state non-deterministic.

Stopping the VM

Stopping the guest should not interfere with its state (with the exception of the network connections, that could be broken by the remote timeouts). VM can be stopped at any moment of replay by the user. Restarting the VM after that stop should not break the replay by the unneeded guest state change.

Replay log format

Record/replay log consists of the header and the sequence of execution events. The header includes 4-byte replay version id and 8-byte reserved field. Version is updated every time replay log format changes to prevent using replay log created by another build of qemu.

The sequence of the events describes virtual machine state changes. It includes all non-deterministic inputs of VM, synchronization marks and instruction counts used to correctly inject inputs at replay.

Synchronization marks (checkpoints) are used for synchronizing qemu threads that perform operations with virtual hardware. These operations may change system's state (e.g., change some register or generate interrupt) and therefore should execute synchronously with CPU thread.

Every event in the log includes 1-byte event id and optional arguments. When argument is an array, it is stored as 4-byte array length and corresponding number of bytes with data. Here is the list of events that are written into the log:

- EVENT_INSTRUCTION. Instructions executed since last event. Followed by:
  - 4-byte number of executed instructions.
- EVENT_INTERRUPT. Used to synchronize interrupt processing.
- EVENT_EXCEPTION. Used to synchronize exception handling.
- EVENT_ASYNC. This is a group of events. When such an event is generated, it is stored in the queue and processed in icount_account_warp_timer(). Every such event has it's own id from the following list:
  - REPLAY_ASYNC_EVENT_BH. Bottom-half callback. This event synchronizes callbacks that affect virtual machine state, but normally called asynchronously. Followed by:
    - 8-byte operation id.
  - REPLAY_ASYNC_EVENT_INPUT. Input device event. Contains parameters of keyboard and mouse input operations (key press/release, mouse pointer movement). Followed by:

        * 9-16 bytes depending of input event.
    - REPLAY_ASYNC_EVENT_INPUT_SYNC. Internal input synchronization event.
    - REPLAY_ASYNC_EVENT_CHAR_READ. Character (e.g., serial port) device input initiated by the sender. Followed by:
        * 1-byte character device id.
        * Array with bytes were read.
    - REPLAY_ASYNC_EVENT_BLOCK. Block device operation. Used to synchronize operations with disk and flash drives with CPU. Followed by:
        * 8-byte operation id.
    - REPLAY_ASYNC_EVENT_NET. Incoming network packet. Followed by:
        * 1-byte network adapter id.
        * 4-byte packet flags.
        * Array with packet bytes.
- EVENT_SHUTDOWN. Occurs when user sends shutdown event to qemu, e.g., by closing the window.
- EVENT_CHAR_WRITE. Used to synchronize character output operations. Followed by:
    - 4-byte output function return value.
    - 4-byte offset in the output array.
- EVENT_CHAR_READ_ALL. Used to synchronize character input operations, initiated by qemu. Followed by:
    - Array with bytes that were read.
- EVENT_CHAR_READ_ALL_ERROR. Unsuccessful character input operation, initiated by qemu. Followed by:
    - 4-byte error code.
- EVENT_CLOCK + clock_id. Group of events for host clock read operations. Followed by:
    - 8-byte clock value.
- EVENT_CHECKPOINT + checkpoint_id. Checkpoint for synchronization of CPU, internal threads, and asynchronous input events.
- EVENT_END. Last event in the log.

BIBLIOGRAPHY

[LoPAR] Linux on Power Architecture Reference document (LoPAR) revision 2.9.

[SEVAPI] Secure Encrypted Virtualization API

[APMVOL2] AMD64 Architecture Programmer's Manual Volume 2: System Programming

D-BUS INTERFACES INDEX

d

v

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W