# QEMU Monitor Protocol Library

***Release unknown version***

**John Snow**

**Aug 03, 2022**

# HOME

Welcome! `qemu.qmp` is a QEMU Monitor Protocol ("QMP") library written in Python, using asyncio. It is used to send QMP messages to running QEMU emulators. It requires Python 3.6+ and has no mandatory dependencies.

This library can be used to communicate with QEMU emulators, the QEMU Guest Agent (QGA), the QEMU Storage Daemon (QSD), or any other utility or application that speaks QMP.

This library makes as few assumptions as possible about the actual version or what type of endpoint it will be communicating with; i.e. this library does not contain command definitions and does not seek to be an SDK or a replacement for tools like libvirt or virsh. It is "simply" the protocol (QMP) and not the vocabulary (QAPI). It is up to the library user (you!) to know which commands and arguments you want to send.

CHAPTER

ONE

# WHO IS THIS LIBRARY FOR?

It is firstly for developers of QEMU themselves; as the test infrastructure of QEMU itself needs a convenient and scriptable interface for testing QEMU. This library was split out of the QEMU source tree in order to share a reference version of a QMP library that was usable both within and outside of the QEMU source tree.

Second, it's for those who are developing *for* QEMU by adding new architectures, devices, or functionality; as well as targeting those who are developing *with* QEMU, i.e. developers working on integrating QEMU features into other projects such as libvirt, KubeVirt, Kata Containers, etc. Occasionally, using existing virtual-machine (VM) management stacks that integrate QEMU+KVM can make developing, testing, and debugging features difficult. In these cases, having more 'raw' access to QEMU is beneficial. This library is for you.

Lastly, it's for power users who already use QEMU directly without the aid of libvirt because they require the raw control and power this affords them.

CHAPTER

TWO

# WHO ISN'T THIS LIBRARY FOR?

It is not designed for anyone looking for a turn-key solution for VM management. QEMU is a low-level component that resembles a particularly impressive Swiss Army knife. This library does not manage that complexity and is largely "VM-ignorant". It's not a replacement for projects like libvirt, virt-manager, GNOME Boxes, etc.

CHAPTER

THREE

# INSTALLING

This package can be installed from PyPI with pip:

```
> pip3 install qemu.qmp
```

CHAPTER

FOUR

# USAGE

Launch QEMU with a monitor, e.g.:

```
> qemu-system-x86_64 -qmp unix:qmp.sock,server=on,wait=off
```

Then, at its simplest, script-style usage looks like this:

```
import asyncio
from qemu.qmp import QMPClient

async def main():
    qmp = QMPClient('my-vm-nickname')
    await qmp.connect('qmp.sock')

    res = await qmp.execute('query-status')
    print(f"VM status: {res['status']}")

    await qmp.disconnect()

asyncio.run(main())
```

The above script will connect to the UNIX socket located at `qmp.sock`, query the VM's runstate, then print it out to the terminal:

```
> python3 example.py
VM status: running
```

For more complex usages, especially those that make full advantage of monitoring asynchronous events, refer to the online documentation or type `import qemu.qmp; help(qemu.qmp)` in your Python terminal of choice.

CHAPTER

FIVE

# CONTRIBUTING

Contributions are quite welcome! Please file bugs using the GitLab issue tracker. This project will accept GitLab merge requests, but due to the close association with the QEMU project, there are some additional guidelines:

1. Please use the "Signed-off-by" tag in your commit messages. See https://wiki.linuxfoundation.org/dco for more information on this requirement.
2. This repository won't squash merge requests into a single commit on pull; each commit should seek to be self-contained (within reason).
3. Owing to the above, each commit sent as part of a merge request should not introduce any temporary regressions, even if fixed later in the same merge request. This is done to preserve bisectability.
4. Please associate every merge request with at least one GitLab issue. This helps with generating Changelog text and staying organized. Thank you

## 5.1 Developing

Optional packages necessary for running code quality analysis for this package can be installed with the optional dependency group "devel": `pip install qemu.qmp[devel]`.

`make develop` can be used to install this package in editable mode (to the current environment) *and* bring in testing dependencies in one command.

`make check` can be used to run the available tests. Consult `make help` for other targets and tests that make sense for different occasions.

Before submitting a pull request, consider running `make check-tox && make check-pipenv` locally to spot any issues that will cause the CI to fail. These checks use their own virtual environments and won't pollute your working space.

CHAPTER

SIX

# STABILITY AND VERSIONING

This package uses a major.minor.micro SemVer versioning, with the following additional semantics during the alpha/beta period (Major version 0):

This package treats 0.0.z versions as "alpha" versions. Each micro version update may change the API incompatibly. Early users are advised to pin against explicit versions, but check for updates often.

A planned 0.1.z version will introduce the first "beta", whereafter each micro update will be backwards compatible, but each minor update will not be. The first beta version will be released after legacy.py is removed, and the API is tentatively "stable".

Thereafter, normal SemVer / PEP440 rules will apply; micro updates will always be bugfixes, and minor updates will be reserved for backwards compatible feature changes.

CHAPTER

SEVEN

# CHANGELOG

## 7.1 0.0.1 (2022-07-20)

- Initial public release. (API is still subject to change!)

CHAPTER

EIGHT

# QEMU.QMP: QEMU MONITOR PROTOCOL LIBRARY

Welcome! `qemu.qmp` is a QEMU Monitor Protocol ("QMP") library written in Python, using asyncio. It is used to send QMP messages to running QEMU emulators. It requires Python 3.6+ and has no mandatory dependencies.

This library can be used to communicate with QEMU emulators, the QEMU Guest Agent (QGA), the QEMU Storage Daemon (QSD), or any other utility or application that speaks QMP.

This library makes as few assumptions as possible about the actual version or what type of endpoint it will be communicating with; i.e. this library does not contain command definitions and does not seek to be an SDK or a replacement for tools like libvirt or virsh. It is "simply" the protocol (QMP) and not the vocabulary (QAPI). It is up to the library user (you!) to know which commands and arguments you want to send.

## 8.1 Who is this library for?

It is firstly for developers of QEMU themselves; as the test infrastructure of QEMU itself needs a convenient and scriptable interface for testing QEMU. This library was split out of the QEMU source tree in order to share a reference version of a QMP library that was usable both within and outside of the QEMU source tree.

Second, it's for those who are developing *for* QEMU by adding new architectures, devices, or functionality; as well as targeting those who are developing *with* QEMU, i.e. developers working on integrating QEMU features into other projects such as libvirt, KubeVirt, Kata Containers, etc. Occasionally, using existing virtual-machine (VM) management stacks that integrate QEMU+KVM can make developing, testing, and debugging features difficult. In these cases, having more 'raw' access to QEMU is beneficial. This library is for you.

Lastly, it's for power users who already use QEMU directly without the aid of libvirt because they require the raw control and power this affords them.

## 8.2 Who isn't this library for?

It is not designed for anyone looking for a turn-key solution for VM management. QEMU is a low-level component that resembles a particularly impressive Swiss Army knife. This library does not manage that complexity and is largely "VM-ignorant". It's not a replacement for projects like libvirt, virt-manager, GNOME Boxes, etc.

## 8.3 Installing

This package can be installed from PyPI with pip:

```
> pip3 install qemu.qmp
```

## 8.4 Usage

Launch QEMU with a monitor, e.g.:

```
> qemu-system-x86_64 -qmp unix:qmp.sock,server=on,wait=off
```

Then, at its simplest, script-style usage looks like this:

```
import asyncio
from qemu.qmp import QMPClient

async def main():
    qmp = QMPClient('my-vm-nickname')
    await qmp.connect('qmp.sock')

    res = await qmp.execute('query-status')
    print(f"VM status: {res['status']}")

    await qmp.disconnect()

asyncio.run(main())
```

The above script will connect to the UNIX socket located at `qmp.sock`, query the VM's runstate, then print it out to the terminal:

```
> python3 example.py
VM status: running
```

For more complex usages, especially those that make full advantage of monitoring asynchronous events, refer to the online documentation or type `import qemu.qmp; help(qemu.qmp)` in your Python terminal of choice.

## 8.5 Contributing

Contributions are quite welcome! Please file bugs using the GitLab issue tracker. This project will accept GitLab merge requests, but due to the close association with the QEMU project, there are some additional guidelines:

1. Please use the "Signed-off-by" tag in your commit messages. See https://wiki.linuxfoundation.org/dco for more information on this requirement.
2. This repository won't squash merge requests into a single commit on pull; each commit should seek to be self-contained (within reason).
3. Owing to the above, each commit sent as part of a merge request should not introduce any temporary regressions, even if fixed later in the same merge request. This is done to preserve bisectability.

4. Please associate every merge request with at least one GitLab issue. This helps with generating Changelog text and staying organized. Thank you

### 8.5.1 Developing

Optional packages necessary for running code quality analysis for this package can be installed with the optional dependency group "devel": `pip install qemu.qmp[devel]`.

`make develop` can be used to install this package in editable mode (to the current environment) *and* bring in testing dependencies in one command.

`make check` can be used to run the available tests. Consult `make help` for other targets and tests that make sense for different occasions.

Before submitting a pull request, consider running `make check-tox && make check-pipenv` locally to spot any issues that will cause the CI to fail. These checks use their own virtual environments and won't pollute your working space.

## 8.6 Stability and Versioning

This package uses a major.minor.micro SemVer versioning, with the following additional semantics during the alpha/beta period (Major version 0):

This package treats 0.0.z versions as "alpha" versions. Each micro version update may change the API incompatibly. Early users are advised to pin against explicit versions, but check for updates often.

A planned 0.1.z version will introduce the first "beta", whereafter each micro update will be backwards compatible, but each minor update will not be. The first beta version will be released after legacy.py is removed, and the API is tentatively "stable".

Thereafter, normal SemVer / PEP440 rules will apply; micro updates will always be bugfixes, and minor updates will be reserved for backwards compatible feature changes.

## 8.7 Changelog

### 8.7.1 0.0.1 (2022-07-20)

- Initial public release. (API is still subject to change!)

#### Overview

QEMU Monitor Protocol (QMP) development library & tooling.

This package provides a fairly low-level class for communicating asynchronously with QMP protocol servers, as implemented by QEMU, the QEMU Guest Agent, and the QEMU Storage Daemon.

*QMPClient* provides the main functionality of this package. All errors raised by this library derive from *QMPError*, see *qmp.error* for additional detail. See *qmp.events* for an in-depth tutorial on managing QMP events.

## Classes

### QMPClient

**class** qemu.qmp.**QMPClient**(*name: Optional[str] = None*)

Bases: *AsyncProtocol*[*Message*], *Events*

Implements a QMP client connection.

*QMPClient* can be used to either connect or listen to a QMP server, but always acts as the QMP client.

**Parameters**
**name** – Optional nickname for the connection, used to differentiate instances when logging.

Basic script-style usage looks like this:

```
import asyncio
from qemu.qmp import QMPClient

async def main():
    qmp = QMPClient('my_virtual_machine_name')
    await qmp.connect(('127.0.0.1', 1234))
    ...
    res = await qmp.execute('query-block')
    ...
    await qmp.disconnect()

asyncio.run(main())
```

A more advanced example that starts to take advantage of asyncio might look like this:

```
class Client:
    def __init__(self, name: str):
        self.qmp = QMPClient(name)

    async def watch_events(self):
        try:
            async for event in self.qmp.events:
                print(f"Event: {event['event']}")
        except asyncio.CancelledError:
            return

    async def run(self, address='/tmp/qemu.socket'):
        await self.qmp.connect(address)
        asyncio.create_task(self.watch_events())
        await self.qmp.runstate_changed.wait()
        await self.disconnect()
```

See *qmp.events* for more detail on event handling patterns.

**logger:** *logging.Logger* **= <Logger qemu.qmp.qmp_client (WARNING)>**

Logger object used for debugging messages.

**await_greeting:** *bool*

Whether or not to await a greeting after establishing a connection. Defaults to True; QGA servers expect this to be False.

**negotiate:** bool

Whether or not to perform capabilities negotiation upon connection. Implies *await_greeting*. Defaults to True; QGA servers expect this to be False.

**property greeting:** Optional[*Greeting*]

The *Greeting* from the QMP server, if any.

Defaults to None, and will be set after a greeting is received during the connection process. It is reset at the start of each connection attempt.

**async execute_msg**(*msg:* Message) → object

Execute a QMP command on the server and return its value.

**Parameters**
**msg** – The QMP *Message* to execute.

**Returns**
The command execution return value from the server. The type of object returned depends on the command that was issued, though most in QEMU return a dict.

**Raises**

- **ValueError** – If the QMP *Message* does not have either the 'execute' or 'exec-oob' fields set.
- ***ExecuteError*** – When the server returns an error response.
- ***ExecInterruptedError*** – If the connection was disrupted before receiving a reply from the server.

**classmethod make_execute_msg**(*cmd: str*, *arguments: Optional[Mapping[str, object]] = None*, *oob: bool = False*) → *Message*

Create an executable message to be sent by *execute_msg* later.

**Parameters**

- **cmd** – QMP command name.
- **arguments** – Arguments (if any). Must be JSON-serializable.
- **oob** – If True, execute "out of band".

**Returns**
A QMP *Message* that can be executed with *execute_msg()*.

**async execute**(*cmd: str*, *arguments: Optional[Mapping[str, object]] = None*, *oob: bool = False*) → object

Execute a QMP command on the server and return its value.

**Parameters**

- **cmd** – QMP command name.
- **arguments** – Arguments (if any). Must be JSON-serializable.
- **oob** – If True, execute "out of band".

**Returns**
The command execution return value from the server. The type of object returned depends on the command that was issued, though most in QEMU return a dict.

**Raises**

- ***ExecuteError*** – When the server returns an error response.

- *ExecInterruptedError* – If the connection was disrupted before receiving a reply from the server.

send_fd_scm(*fd: int*) → None

Send a file descriptor to the remote via SCM_RIGHTS.

This method does not close the file descriptor.

**Parameters**
**fd** – The file descriptor to send to QEMU.

This is an advanced feature of QEMU where file descriptors can be passed from client to server. This is usually used as a security measure to isolate the QEMU process from being able to open its own files. See the QMP commands `getfd` and `add-fd` for more information.

See `socket.socket.sendmsg` for more information on the Python implementation for sending file descriptors over a UNIX socket.

async accept() → None

Accept an incoming connection and begin processing message queues.

Used after a previous call to *start_server()* to accept an incoming connection. If this call fails, *runstate* is guaranteed to be set back to *IDLE*.

**Raises**

- *StateError* – When the *Runstate* is not *CONNECTING*.
- *QMPError* – When *start_server()* was not called first.
- *ConnectError* – When a connection or session cannot be established.

  This exception will wrap a more concrete one. In most cases, the wrapped exception will be `OSError` or `EOFError`. If a protocol-level failure occurs while establishing a new session, the wrapped error may also be an *QMPError*.

async connect(*address: Union[str, Tuple[str, int]]*, *ssl: Optional[SSLContext] = None*) → None

Connect to the server and begin processing message queues.

If this call fails, *runstate* is guaranteed to be set back to *IDLE*.

**Parameters**

- **address** – Address to connect to; UNIX socket path or TCP address/port.
- **ssl** – SSL context to use, if any.

**Raises**

- *StateError* – When the *Runstate* is not *IDLE*.
- *ConnectError* – When a connection or session cannot be established.

  This exception will wrap a more concrete one. In most cases, the wrapped exception will be `OSError` or `EOFError`. If a protocol-level failure occurs while establishing a new session, the wrapped error may also be an *QMPError*.

async disconnect() → None

Disconnect and wait for all tasks to fully stop.

If there was an exception that caused the reader/writers to terminate prematurely, it will be raised here.

**Raises**
`Exception` – When the reader or writer terminate unexpectedly. You can expect to see

EOFError if the server hangs up, or OSError for connection-related issues. If there was a QMP protocol-level problem, *ProtocolError* will be seen.

**listen**(**listeners:* EventListener) → Iterator[None]

Context manager: Temporarily listen with an *EventListener*.

Accepts one or more *EventListener* objects and registers them, activating them for the duration of the context block.

*EventListener* objects will have any pending events in their FIFO queue cleared upon exiting the context block, when they are deactivated.

**Parameters**
***listeners** – One or more EventListeners to activate.

**Raises**
***ListenerError*** – If the given listener(s) are already active.

**listener**(*names: Optional[Union[str, Iterable[str]]] = ()*, *event_filter: Optional[Callable[[*Message*], bool]] = None*) → Iterator[*EventListener*]

Context manager: Temporarily listen with a new *EventListener*.

Creates an *EventListener* object and registers it, activating it for the duration of the context block.

**Parameters**

- **names** – One or more names of events to listen for. When not provided, listen for ALL events.
- **event_filter** – An optional event filtering function. When names are also provided, this acts as a secondary filter.

**Returns**
The newly created and active *EventListener*.

**register_listener**(*listener:* EventListener) → None

Register and activate an *EventListener*.

**Parameters**
**listener** – The listener to activate.

**Raises**
***ListenerError*** – If the given listener is already registered.

**remove_listener**(*listener:* EventListener) → None

Unregister and deactivate an *EventListener*.

The removed listener will have its pending events cleared via *clear()*. The listener can be re-registered later when desired.

**Parameters**
**listener** – The listener to deactivate.

**Raises**
***ListenerError*** – If the given listener is not registered.

**property runstate:** ***Runstate***

The current *Runstate* of the connection.

**async runstate_changed**() → *Runstate*

Wait for the *runstate* to change, then return that *Runstate*.

**async start_server**(*address: Union[str, Tuple[str, int]]*, *ssl: Optional[SSLContext] = None*) → None

Start listening for an incoming connection, but do not wait for a peer.

This method starts listening for an incoming connection, but does not block waiting for a peer. This call will return immediately after binding and listening on a socket. A later call to `accept()` must be made in order to finalize the incoming connection.

**Parameters**

- **address** – Address to listen on; UNIX socket path or TCP address/port.
- **ssl** – SSL context to use, if any.

**Raises**

- ***StateError*** – When the *Runstate* is not *IDLE*.
- ***ConnectError*** – When the server could not start listening on this address.

  This exception will wrap a more concrete one. In most cases, the wrapped exception will be `OSError`.

**async start_server_and_accept**(*address: Union[str, Tuple[str, int]]*, *ssl: Optional[SSLContext] = None*) → None

Accept a connection and begin processing message queues.

If this call fails, *runstate* is guaranteed to be set back to *IDLE*. This method is precisely equivalent to calling *start_server()* followed by `accept()`.

**Parameters**

- **address** – Address to listen on; UNIX socket path or TCP address/port.
- **ssl** – SSL context to use, if any.

**Raises**

- ***StateError*** – When the *Runstate* is not *IDLE*.
- ***ConnectError*** – When a connection or session cannot be established.

  This exception will wrap a more concrete one. In most cases, the wrapped exception will be `OSError` or `EOFError`. If a protocol-level failure occurs while establishing a new session, the wrapped error may also be a *QMPError*.

**name: Optional[str]**

The nickname for this connection, if any. This name is used for differentiating instances in debug output.

**events: *EventListener***

Default, all-events *EventListener*. See *qmp.events* for more info.

## Message

**class** qemu.qmp.**Message**(*value: Union[bytes, Mapping[str, object]] = b'{}'*, ***, *eager: bool = True*)

Bases: `MutableMapping`[`str`, `object`]

Represents a single QMP protocol message.

QMP uses JSON objects as its basic communicative unit; so this Python object is a `MutableMapping`. It may be instantiated from either another mapping (like a `dict`), or from raw `bytes` that still need to be deserialized.

Once instantiated, it may be treated like any other `MutableMapping`:

```
>>> msg = Message(b'{"hello": "world"}')
>>> assert msg['hello'] == 'world'
>>> msg['id'] = 'foobar'
>>> print(msg)
{
  "hello": "world",
  "id": "foobar"
}
```

It can be converted to `bytes`:

```
>>> msg = Message({"hello": "world"})
>>> print(bytes(msg))
b'{"hello":"world","id":"foobar"}'
```

Or back into a garden-variety `dict`:

```
>>> dict(msg)
{'hello': 'world'}
```

Or pretty-printed:

```
>>> print(str(msg))
{
  "hello": "world"
}
```

**Parameters**

- **value** – Initial value, if any.
- **eager** – When `True`, attempt to serialize or deserialize the initial value immediately, so that conversion exceptions are raised during the call to `__init__()`.

### EventListener

**class** qemu.qmp.**EventListener**(*names: Optional[Union[str, Iterable[str]]] = None*, *event_filter: Optional[Callable[[Message], bool]] = None*)

Selectively listens for events with runtime configurable filtering.

This class is designed to be directly usable for the most common cases, but it can be extended to provide more rigorous control.

**Parameters**

- **names** – One or more names of events to listen for. When not provided, listen for ALL events.
- **event_filter** – An optional event filtering function. When names are also provided, this acts as a secondary filter.

When `names` and `event_filter` are both provided, the names will be filtered first, and then the filter function will be called second. The event filter function can assume that the format of the event is a known format.

**names:** `Set[str]`

Primary event filter, based on one or more event names.

**event_filter:** Optional[Callable[[*Message*], bool]]

Optional, secondary event filter.

**property history:** Tuple[*Message*, ...]

A read-only history of all events seen so far.

This represents *every* event, including those not yet witnessed via *get()* or `async for`. It persists between *clear()* calls and is immutable.

**accept**(*event:* Message) → bool

Determine if this listener accepts this event.

This method determines which events will appear in the stream. The default implementation simply checks the event against the list of names and the event_filter to decide if this *EventListener* accepts a given event. It can be overridden/extended to provide custom listener behavior.

User code is not expected to need to invoke this method.

**Parameters**
**event** – The event under consideration.

**Returns**
True, if this listener accepts this event.

**async put**(*event:* Message) → None

Conditionally put a new event into the FIFO queue.

This method is not designed to be invoked from user code, and it should not need to be overridden. It is a public interface so that *QMPClient* has an interface by which it can inform registered listeners of new events.

The event will be put into the queue if `accept()` returns True.

**Parameters**
**event** – The new event to put into the FIFO queue.

**async get**() → *Message*

Wait for the very next event in this stream.

If one is already available, return that one.

**empty**() → bool

Return True if there are no pending events.

**clear**() → List[*Message*]

Clear this listener of all pending events.

Called when an *EventListener* is being unregistered, this clears the pending FIFO queue synchronously. It can be also be used to manually clear any pending events, if desired.

**Returns**
The cleared events, if any.

> **Warning:** Take care when discarding events. Cleared events will be silently tossed on the floor. All events that were ever accepted by this listener are visible in *history()*.

### Runstate

**class** qemu.qmp.**Runstate**(*value*)

Bases: Enum

Protocol session runstate.

**IDLE = 0**

Fully quiesced and disconnected.

**CONNECTING = 1**

In the process of connecting or establishing a session.

**RUNNING = 2**

Fully connected and active session.

**DISCONNECTING = 3**

In the process of disconnecting. Runstate may be returned to *IDLE* by calling *disconnect()*.

### Exceptions

**exception** qemu.qmp.**QMPError**

Bases: Exception

Abstract error class for all errors originating from this package.

**exception** qemu.qmp.**StateError**(*error_message: str*, *state:* Runstate, *required:* Runstate)

Bases: *QMPError*

An API command (connect, execute, etc) was issued at an inappropriate time.

This error is raised when a command like `connect()` is called when the client is already connected.

**Parameters**

- **error_message** – Human-readable string describing the state violation.
- **state** – The actual *Runstate* seen at the time of the violation.
- **required** – The *Runstate* required to process this command.

**exception** qemu.qmp.**ConnectError**(*error_message: str*, *exc: Exception*)

Bases: *QMPError*

Raised when the initial connection process has failed.

This Exception always wraps a "root cause" exception that can be interrogated for additional information.

For example, when connecting to a non-existent socket:

```
await qmp.connect('not_found.sock')
# ConnectError: Failed to establish connection:
#               [Errno 2] No such file or directory
```

**Parameters**

- **error_message** – Human-readable string describing the error.
- **exc** – The root-cause exception.

**error_message:** *str*

Human-readable error string

**exc:** *Exception*

Wrapped root cause exception

**exception** qemu.qmp.**ExecuteError**(*error_response:* ErrorResponse, *sent:* Message, *received:* Message)

Bases: *QMPError*

Exception raised by *QMPClient.execute()* on RPC failure.

This exception is raised when the server received, interpreted, and replied to a command successfully; but the command itself returned a failure status.

For example:

```
await qmp.execute('block-dirty-bitmap-add',
                  {'node': 'foo', 'name': 'my_bitmap'})
# qemu.qmp.qmp_client.ExecuteError:
#     Cannot find device='foo' nor node-name='foo'
```

**Parameters**

- **error_response** – The RPC error response object.
- **sent** – The sent RPC message that caused the failure.
- **received** – The raw RPC error reply received.

**sent:** *Message*

The sent *Message* that caused the failure

**received:** *Message*

The received *Message* that indicated failure

**error:** *ErrorResponse*

The parsed error response

**error_class:** *str*

The QMP error class

**exception** qemu.qmp.**ExecInterruptedError**

Bases: *QMPError*

Exception raised by *execute()* (et al) when an RPC is interrupted.

This error is raised when an *execute()* statement could not be completed. This can occur because the connection itself was terminated before a reply was received. The true cause of the interruption will be available via *disconnect()*.

The QMP protocol does not make it possible to know if a command succeeded or failed after such an event; the client will need to query the server to determine the state of the server on a case-by-case basis.

For example, ECONNRESET might look like this:

```
try:
    await qmp.execute('query-block')
    # ExecInterruptedError: Disconnected
except ExecInterruptedError:
```

(continues on next page)

(continued from previous page)

```
    await qmp.disconnect()
    # ConnectionResetError: [Errno 104] Connection reset by peer
```

### Error classes

QMP Error Classes

This package seeks to provide semantic error classes that are intended to be used directly by clients when they would like to handle particular semantic failures (e.g. "failed to connect") without needing to know the enumeration of possible reasons for that failure.

QMPError serves as the ancestor for all exceptions raised by this package, and is suitable for use in handling semantic errors from this library. In most cases, individual public methods will attempt to catch and re-encapsulate various exceptions to provide a semantic error-handling interface.

---

**QMP Exception Hierarchy Reference**

```
Exception
    +- QMPError
        +- ConnectError
        +- StateError
        +- ExecInterruptedError
        +- ExecuteError
        +- ListenerError
        +- ProtocolError
            +- DeserializationError
            +- UnexpectedTypeError
            +- ServerParseError
            +- BadReplyError
            +- GreetingError
            +- NegotiationError
```

---

**exception** qemu.qmp.error.**QMPError**

Bases: Exception

Abstract error class for all errors originating from this package.

**exception** qemu.qmp.error.**ProtocolError**(*error_message: str*)

Bases: *QMPError*

Abstract error class for protocol failures.

Semantically, these errors are generally the fault of either the protocol server or as a result of a bug in this library.

**Parameters**
**error_message** – Human-readable string describing the error.

**error_message:** str

Human-readable error message, without any prefix.

### Events

QMP Events and EventListeners

Asynchronous QMP uses *EventListener* objects to listen for events. An *EventListener* is a FIFO event queue that can be pre-filtered to listen for only specific events. Each *EventListener* instance receives its own copy of events that it hears, so events may be consumed without fear or worry for depriving other listeners of events they need to hear.

### EventListener Tutorial

In all of the following examples, we assume that we have a *QMPClient* instantiated named `qmp` that is already connected. For example:

```
from qemu.qmp import QMPClient

qmp = QMPClient('example-vm')
await qmp.connect('127.0.0.1', 1234)
```

#### `listener()` context blocks with one name

The most basic usage is by using the *listener()* context manager to construct them:

```
with qmp.listener('STOP') as listener:
    await qmp.execute('stop')
    await listener.get()
```

The listener is active only for the duration of the 'with' block. This instance listens only for 'STOP' events.

#### `listener()` context blocks with two or more names

Multiple events can be selected for by providing any `Iterable[str]`:

```
with qmp.listener(('STOP', 'RESUME')) as listener:
    await qmp.execute('stop')
    event = await listener.get()
    assert event['event'] == 'STOP'

    await qmp.execute('cont')
    event = await listener.get()
    assert event['event'] == 'RESUME'
```

#### `listener()` context blocks with no names

By omitting names entirely, you can listen to ALL events.

```
with qmp.listener() as listener:
    await qmp.execute('stop')
    event = await listener.get()
    assert event['event'] == 'STOP'
```

This isn't a very good use case for this feature: In a non-trivial running system, we may not know what event will arrive next. Grabbing the top of a FIFO queue returning multiple kinds of events may be prone to error.

### Using async iterators to retrieve events

If you'd like to simply watch what events happen to arrive, you can use the listener as an async iterator:

```
with qmp.listener() as listener:
    async for event in listener:
        print(f"Event arrived: {event['event']}")
```

This is analogous to the following code:

```
with qmp.listener() as listener:
    while True:
        event = listener.get()
        print(f"Event arrived: {event['event']}")
```

This event stream will never end, so these blocks will never terminate. Even if the QMP connection errors out prematurely, this listener will go silent without raising an error.

### Using asyncio.Task to concurrently retrieve events

Since a listener's event stream will never terminate, it is not likely useful to use that form in a script. For longer-running clients, we can create event handlers by using `asyncio.Task` to create concurrent coroutines:

```
async def print_events(listener):
    try:
        async for event in listener:
            print(f"Event arrived: {event['event']}")
    except asyncio.CancelledError:
        return

with qmp.listener() as listener:
    task = asyncio.Task(print_events(listener))
    await qmp.execute('stop')
    await qmp.execute('cont')
    task.cancel()
    await task
```

However, there is no guarantee that these events will be received by the time we leave this context block. Once the context block is exited, the listener will cease to hear any new events, and becomes inert.

Be mindful of the timing: the above example will *probably*– but does not *guarantee*– that both STOP/RESUMED events will be printed. The example below outlines how to use listeners outside of a context block.

### Using `register_listener()` and `remove_listener()`

To create a listener with a longer lifetime, beyond the scope of a single block, create a listener and then call *register_listener()*:

```
class MyClient:
    def __init__(self, qmp):
        self.qmp = qmp
        self.listener = EventListener()

    async def print_events(self):
        try:
            async for event in self.listener:
                print(f"Event arrived: {event['event']}")
        except asyncio.CancelledError:
            return

    async def run(self):
        self.task = asyncio.Task(self.print_events)
        self.qmp.register_listener(self.listener)
        await qmp.execute('stop')
        await qmp.execute('cont')

    async def stop(self):
        self.task.cancel()
        await self.task
        self.qmp.remove_listener(self.listener)
```

The listener can be deactivated by using *remove_listener()*. When it is removed, any possible pending events are cleared and it can be re-registered at a later time.

### Using the built-in all events listener

The *QMPClient* object creates its own default listener named *events* that can be used for the same purpose without having to create your own:

```
async def print_events(listener):
    try:
        async for event in listener:
            print(f"Event arrived: {event['event']}")
    except asyncio.CancelledError:
        return

task = asyncio.Task(print_events(qmp.events))

await qmp.execute('stop')
await qmp.execute('cont')
```

(continues on next page)

(continued from previous page)

```
task.cancel()
await task
```

### Using both .get() and async iterators

The async iterator and *get()* methods pull events from the same FIFO queue. If you mix the usage of both, be aware: Events are emitted precisely once per listener.

If multiple contexts try to pull events from the same listener instance, events are still emitted only precisely once.

This restriction can be lifted by creating additional listeners.

### Creating multiple listeners

Additional *EventListener* objects can be created at-will. Each one receives its own copy of events, with separate FIFO event queues.

```
my_listener = EventListener()
qmp.register_listener(my_listener)

await qmp.execute('stop')
copy1 = await my_listener.get()
copy2 = await qmp.events.get()

assert copy1 == copy2
```

In this example, we await an event from both a user-created *EventListener* and the built-in events listener. Both receive the same event.

### Clearing listeners

*EventListener* objects can be cleared, clearing all events seen thus far:

```
await qmp.execute('stop')
discarded = qmp.events.clear()
await qmp.execute('cont')
event = await qmp.events.get()
assert event['event'] == 'RESUME'
assert discarded[0]['event'] == 'STOP'
```

*EventListener* objects are FIFO queues. If events are not consumed, they will remain in the queue until they are witnessed or discarded via *clear()*. FIFO queues will be drained automatically upon leaving a context block, or when calling *remove_listener()*.

Any events removed from the queue in this fashion will be returned by the clear call.

### Accessing listener history

*EventListener* objects record their history. Even after being cleared, you can obtain a record of all events seen so far:

```
await qmp.execute('stop')
await qmp.execute('cont')
qmp.events.clear()

assert len(qmp.events.history) == 2
assert qmp.events.history[0]['event'] == 'STOP'
assert qmp.events.history[1]['event'] == 'RESUME'
```

The history is updated immediately and does not require the event to be witnessed first.

### Using event filters

*EventListener* objects can be given complex filtering criteria if names are not sufficient:

```
def job1_filter(event) -> bool:
    event_data = event.get('data', {})
    event_job_id = event_data.get('id')
    return event_job_id == "job1"

with qmp.listener('JOB_STATUS_CHANGE', job1_filter) as listener:
    await qmp.execute('blockdev-backup', arguments={'job-id': 'job1', ...})
    async for event in listener:
        if event['data']['status'] == 'concluded':
            break
```

These filters might be most useful when parameterized. *EventListener* objects expect a function that takes only a single argument (the raw event, as a *Message*) and returns a bool; True if the event should be accepted into the stream. You can create a function that adapts this signature to accept configuration parameters:

```
def job_filter(job_id: str) -> EventFilter:
    def filter(event: Message) -> bool:
        return event['data']['id'] == job_id
    return filter

with qmp.listener('JOB_STATUS_CHANGE', job_filter('job2')) as listener:
    await qmp.execute('blockdev-backup', arguments={'job-id': 'job2', ...})
    async for event in listener:
        if event['data']['status'] == 'concluded':
            break
```

### Activating an existing listener with `listen()`

Listeners with complex, long configurations can also be created manually and activated temporarily by using *listen()* instead of *listener()*:

```
listener = EventListener(('BLOCK_JOB_COMPLETED', 'BLOCK_JOB_CANCELLED',
                          'BLOCK_JOB_ERROR', 'BLOCK_JOB_READY',
                          'BLOCK_JOB_PENDING', 'JOB_STATUS_CHANGE'))

with qmp.listen(listener):
    await qmp.execute('blockdev-backup', arguments={'job-id': 'job3', ...})
    async for event in listener:
        print(event)
        if event['event'] == 'BLOCK_JOB_COMPLETED':
            break
```

Any events that are not witnessed by the time the block is left will be cleared from the queue; entering the block is an implicit *register_listener()* and leaving the block is an implicit *remove_listener()*.

### Activating multiple existing listeners with `listen()`

While *listener()* is only capable of creating a single listener, *listen()* is capable of activating multiple listeners simultaneously:

```
def job_filter(job_id: str) -> EventFilter:
    def filter(event: Message) -> bool:
        return event['data']['id'] == job_id
    return filter

jobA = EventListener('JOB_STATUS_CHANGE', job_filter('jobA'))
jobB = EventListener('JOB_STATUS_CHANGE', job_filter('jobB'))

with qmp.listen(jobA, jobB):
    qmp.execute('blockdev-create', arguments={'job-id': 'jobA', ...})
    qmp.execute('blockdev-create', arguments={'job-id': 'jobB', ...})

    async for event in jobA.get():
        if event['data']['status'] == 'concluded':
            break
    async for event in jobB.get():
        if event['data']['status'] == 'concluded':
            break
```

Note that in the above example, we explicitly wait on jobA to conclude first, and then wait for jobB to do the same. All we have guaranteed is that the code that waits for jobA will not accidentally consume the event intended for the jobB waiter.

### Extending the `EventListener` class

In the case that a more specialized *`EventListener`* is desired to provide either more functionality or more compact syntax for specialized cases, it can be extended.

One of the key methods to extend or override is *`accept()`*. The default implementation checks an incoming message for:

1. A qualifying name, if any *`names`* were specified at initialization time
2. That *`event_filter()`* returns True.

This can be modified however you see fit to change the criteria for inclusion in the stream.

For convenience, a `JobListener` class could be created that simply bakes in configuration so it does not need to be repeated:

```
class JobListener(EventListener):
    def __init__(self, job_id: str):
        super().__init__(('BLOCK_JOB_COMPLETED', 'BLOCK_JOB_CANCELLED',
                          'BLOCK_JOB_ERROR', 'BLOCK_JOB_READY',
                          'BLOCK_JOB_PENDING', 'JOB_STATUS_CHANGE'))
        self.job_id = job_id

    def accept(self, event) -> bool:
        if not super().accept(event):
            return False
        if event['event'] in ('BLOCK_JOB_PENDING', 'JOB_STATUS_CHANGE'):
            return event['data']['id'] == job_id
        return event['data']['device'] == job_id
```

From here on out, you can conjure up a custom-purpose listener that listens only for job-related events for a specific job-id easily:

```
listener = JobListener('job4')
with qmp.listener(listener):
    await qmp.execute('blockdev-backup', arguments={'job-id': 'job4', ...})
    async for event in listener:
        print(event)
        if event['event'] == 'BLOCK_JOB_COMPLETED':
            break
```

### Experimental Interfaces & Design Issues

These interfaces are not ones I am sure I will keep or otherwise modify heavily.

### qmp.listen()'s type signature

*listen()* does not return anything, because it was assumed the caller already had a handle to the listener. However, for `qmp.listen(EventListener())` forms, the caller will not have saved a handle to the listener.

Because this function can accept *many* listeners, I found it hard to accurately type in a way where it could be used in both "one" or "many" forms conveniently and in a statically type-safe manner.

Ultimately, I removed the return altogether, but perhaps with more time I can work out a way to re-add it.

### API Reference

**class** qemu.qmp.events.**EventListener**(*names: Optional[Union[str, Iterable[str]]] = None*, *event_filter: Optional[Callable[[Message], bool]] = None*)

Bases: object

Selectively listens for events with runtime configurable filtering.

This class is designed to be directly usable for the most common cases, but it can be extended to provide more rigorous control.

**Parameters**

- **names** – One or more names of events to listen for. When not provided, listen for ALL events.
- **event_filter** – An optional event filtering function. When names are also provided, this acts as a secondary filter.

When `names` and `event_filter` are both provided, the names will be filtered first, and then the filter function will be called second. The event filter function can assume that the format of the event is a known format.

**accept**(*event:* Message) → bool

Determine if this listener accepts this event.

This method determines which events will appear in the stream. The default implementation simply checks the event against the list of names and the event_filter to decide if this *EventListener* accepts a given event. It can be overridden/extended to provide custom listener behavior.

User code is not expected to need to invoke this method.

**Parameters**
**event** – The event under consideration.

**Returns**
True, if this listener accepts this event.

**clear**() → List[*Message*]

Clear this listener of all pending events.

Called when an *EventListener* is being unregistered, this clears the pending FIFO queue synchronously. It can be also be used to manually clear any pending events, if desired.

**Returns**
The cleared events, if any.

> **Warning:** Take care when discarding events. Cleared events will be silently tossed on the floor. All events that were ever accepted by this listener are visible in *history()*.

**empty**() → bool

Return True if there are no pending events.

**event_filter:** Optional[Callable[[*Message*], bool]]

Optional, secondary event filter.

**async get**() → *Message*

Wait for the very next event in this stream.

If one is already available, return that one.

**property history:** Tuple[*Message*, ...]

A read-only history of all events seen so far.

This represents *every* event, including those not yet witnessed via *get()* or `async for`. It persists between *clear()* calls and is immutable.

**names:** Set[str]

Primary event filter, based on one or more event names.

**async put**(*event:* Message) → None

Conditionally put a new event into the FIFO queue.

This method is not designed to be invoked from user code, and it should not need to be overridden. It is a public interface so that *QMPClient* has an interface by which it can inform registered listeners of new events.

The event will be put into the queue if *accept()* returns True.

**Parameters**
**event** – The new event to put into the FIFO queue.

**class** qemu.qmp.events.**Events**

Bases: object

Events is a mix-in class that adds event functionality to the QMP class.

It's designed specifically as a mix-in for *QMPClient*, and it relies upon the class it is being mixed into having a 'logger' property.

**events:** *EventListener*

Default, all-events *EventListener*. See *qmp.events* for more info.

**listen**(**listeners:* EventListener) → Iterator[None]

Context manager: Temporarily listen with an *EventListener*.

Accepts one or more *EventListener* objects and registers them, activating them for the duration of the context block.

*EventListener* objects will have any pending events in their FIFO queue cleared upon exiting the context block, when they are deactivated.

**Parameters**
***listeners** – One or more EventListeners to activate.

**Raises**
*ListenerError* – If the given listener(s) are already active.

**listener**(*names: Optional[Union[str, Iterable[str]]] = ()*, *event_filter: Optional[Callable[[Message], bool]] = None*) → Iterator[*EventListener*]

Context manager: Temporarily listen with a new *EventListener*.

Creates an *EventListener* object and registers it, activating it for the duration of the context block.

**Parameters**

- **names** – One or more names of events to listen for. When not provided, listen for ALL events.
- **event_filter** – An optional event filtering function. When names are also provided, this acts as a secondary filter.

**Returns**
The newly created and active *EventListener*.

**register_listener**(*listener:* EventListener) → None

Register and activate an *EventListener*.

**Parameters**
**listener** – The listener to activate.

**Raises**
***ListenerError*** – If the given listener is already registered.

**remove_listener**(*listener:* EventListener) → None

Unregister and deactivate an *EventListener*.

The removed listener will have its pending events cleared via *clear()*. The listener can be re-registered later when desired.

**Parameters**
**listener** – The listener to deactivate.

**Raises**
***ListenerError*** – If the given listener is not registered.

**exception** qemu.qmp.events.**ListenerError**

Bases: *QMPError*

Generic error class for *EventListener*-related problems.

### Legacy API

(Legacy) Sync QMP Wrapper

This module provides the *QEMUMonitorProtocol* class, which is a synchronous wrapper around *QMPClient*.

Its design closely resembles that of the original QEMUMonitorProtocol class, originally written by Luiz Capitulino. It is provided here for compatibility with scripts inside the QEMU source tree that expect the old interface.

**class** qemu.qmp.legacy.**QEMUMonitorProtocol**(*address: Union[str, Tuple[str, int]]*, *server: bool = False*, *nickname: Optional[str] = None*)

Bases: object

Provide an API to connect to QEMU via QEMU Monitor Protocol (QMP) and then allow to handle commands and events.

**Parameters**

- **address** – QEMU address, can be either a unix socket path (string) or a tuple in the form ( address, port ) for a TCP connection
- **server** – Act as the socket server. (See 'accept')
- **nickname** – Optional nickname used for logging.

**accept**(*timeout: Optional[float] = 15.0*) → Dict[str, Any]

Await connection from QMP Monitor and perform capabilities negotiation.

**Parameters**
**timeout** – timeout in seconds (nonnegative float number, or None). If None, there is no timeout, and this may block forever.

**Returns**
QMP greeting dict

**Raises**
*ConnectError* – on connection errors

**clear_events**() → None

Clear current list of pending events.

**close**() → None

Close the connection.

**cmd**(*name: str*, *args: Optional[Dict[str, object]] = None*, *cmd_id: Optional[object] = None*) → Dict[str, Any]

Build a QMP command and send it to the QMP Monitor.

**Parameters**
- **name** – command name (string)
- **args** – command arguments (dict)
- **cmd_id** – command id (dict, list, string or int)

**cmd_obj**(*qmp_cmd: Dict[str, Any]*) → Dict[str, Any]

Send a QMP command to the QMP Monitor.

**Parameters**
**qmp_cmd** – QMP command to be sent as a Python dict

**Returns**
QMP response as a Python dict

**command**(*cmd: str*, ***kwds: object*) → object

Build and send a QMP command to the monitor, report errors if any

**connect**(*negotiate: bool = True*) → Optional[Dict[str, Any]]

Connect to the QMP Monitor and perform capabilities negotiation.

**Returns**
QMP greeting dict, or None if negotiate is false

**Raises**
*ConnectError* – on connection errors

**get_events**(*wait: Union[bool, float] = False*) → List[Dict[str, Any]]

Get a list of QMP events and clear all pending events.

**Parameters**
**wait** – If False or 0, do not wait. Return None if no events ready. If True, wait until we have at least one event. Otherwise, wait for up to the specified number of seconds for at least one event.

**Raises**
**asyncio.TimeoutError** – When a timeout is requested and the timeout period elapses.

**Returns**
A list of QMP events.

**classmethod parse_address**(*address: str*) → Union[str, Tuple[str, int]]

Parse a string into a QMP address.

Figure out if the argument is in the port:host form. If it's not, it's probably a file path.

**pull_event**(*wait: Union[bool, float] = False*) → Optional[Dict[str, Any]]

Pulls a single event.

**Parameters**
**wait** – If False or 0, do not wait. Return None if no events ready. If True, wait forever until the next event. Otherwise, wait for the specified number of seconds.

**Raises**
**asyncio.TimeoutError** – When a timeout is requested and the timeout period elapses.

**Returns**
The first available QMP event, or None.

**send_fd_scm**(*fd: int*) → None

Send a file descriptor to the remote via SCM_RIGHTS.

**settimeout**(*timeout: Optional[float]*) → None

Set the timeout for QMP RPC execution.

This timeout affects the *cmd*, *cmd_obj*, and *command* methods. The *accept*, *pull_event* and *get_events* methods have their own configurable timeouts.

**Parameters**
**timeout** – timeout in seconds, or None. None will wait indefinitely.

**exception** qemu.qmp.legacy.**QMPBadPortError**

Bases: *QMPError*

Unable to parse socket address: Port was non-numerical.

qemu.qmp.legacy.**QMPMessage**

QMPMessage is an entire QMP message of any kind.

alias of Dict[str, Any]

qemu.qmp.legacy.**QMPObject**

QMPObject is any object in a QMP message.

alias of Dict[str, object]

qemu.qmp.legacy.**QMPReturnValue**

QMPReturnValue is the 'return' value of a command.

### QMP Messages

QMP Message Format

This module provides the *Message* class, which represents a single QMP message sent to or from the server.

**class** qemu.qmp.message.**Message**(*value: Union[bytes, Mapping[str, object]] = b'{}'*, ***, *eager: bool = True*)

Bases: MutableMapping[str, object]

Represents a single QMP protocol message.

QMP uses JSON objects as its basic communicative unit; so this Python object is a MutableMapping. It may be instantiated from either another mapping (like a dict), or from raw bytes that still need to be deserialized.

Once instantiated, it may be treated like any other MutableMapping:

```
>>> msg = Message(b'{"hello": "world"}')
>>> assert msg['hello'] == 'world'
>>> msg['id'] = 'foobar'
>>> print(msg)
{
  "hello": "world",
  "id": "foobar"
}
```

It can be converted to bytes:

```
>>> msg = Message({"hello": "world"})
>>> print(bytes(msg))
b'{"hello":"world","id":"foobar"}'
```

Or back into a garden-variety dict:

```
>>> dict(msg)
{'hello': 'world'}
```

Or pretty-printed:

```
>>> print(str(msg))
{
  "hello": "world"
}
```

**Parameters**

- **value** – Initial value, if any.
- **eager** – When True, attempt to serialize or deserialize the initial value immediately, so that conversion exceptions are raised during the call to __init__().

**exception** qemu.qmp.message.**DeserializationError**(*error_message: str*, *raw: bytes*)

Bases: *ProtocolError*

A QMP message was not understood as JSON.

When this Exception is raised, __cause__ will be set to the json.JSONDecodeError Exception, which can be interrogated for further details.

**Parameters**

- **error_message** – Human-readable string describing the error.
- **raw** – The raw `bytes` that prompted the failure.

**raw:** `bytes`

The raw `bytes` that were not understood as JSON.

**error_message:** `str`

Human-readable error message, without any prefix.

**exception** qemu.qmp.message.**UnexpectedTypeError**(*error_message: str*, *value: object*)

Bases: *ProtocolError*

A QMP message was JSON, but not a JSON object.

**Parameters**

- **error_message** – Human-readable string describing the error.
- **value** – The deserialized JSON value that wasn't an object.

**error_message:** `str`

Human-readable error message, without any prefix.

**value:** `object`

The JSON value that was expected to be an object.

### QMP Data Models

QMP Data Models

This module provides simplistic data classes that represent the few structures that the QMP spec mandates; they are used to verify incoming data to make sure it conforms to spec.

**class** qemu.qmp.models.**Model**(*raw: Mapping[str, Any]*)

Bases: `object`

Abstract data model, representing some QMP object of some kind.

**Parameters**
**raw** – The raw object to be validated.

**Raises**

- **KeyError** – If any required fields are absent.
- **TypeError** – If any required fields have the wrong type.

**class** qemu.qmp.models.**Greeting**(*raw: Mapping[str, Any]*)

Bases: *Model*

Defined in qmp-spec.txt, section 2.2, "Server Greeting".

See 2.2 Server Greeting for details.

**Parameters**
**raw** – The raw Greeting object.

**Raises**

- **KeyError** – If any required fields are absent.
- **TypeError** – If any required fields have the wrong type.

**QMP:** *QMPGreeting*

'QMP' member

class qemu.qmp.models.**QMPGreeting**(*raw: Mapping[str, Any]*)

Bases: *Model*

Defined in qmp-spec.txt, section 2.2, "Server Greeting".

**Parameters**
**raw** – The raw QMPGreeting object.

**Raises**
- **KeyError** – If any required fields are absent.
- **TypeError** – If any required fields have the wrong type.

**version:** Mapping[str, object]

'version' member

**capabilities:** Sequence[object]

'capabilities' member

class qemu.qmp.models.**ErrorResponse**(*raw: Mapping[str, Any]*)

Bases: *Model*

Defined in qmp-spec.txt, section 2.4.2, "error".

**Parameters**
**raw** – The raw ErrorResponse object.

**Raises**
- **KeyError** – If any required fields are absent.
- **TypeError** – If any required fields have the wrong type.

**error:** *ErrorInfo*

'error' member

**id:** Optional[object]

'id' member

class qemu.qmp.models.**ErrorInfo**(*raw: Mapping[str, Any]*)

Bases: *Model*

Defined in qmp-spec.txt, section 2.4.2, "error".

**Parameters**
**raw** – The raw ErrorInfo object.

**Raises**
- **KeyError** – If any required fields are absent.
- **TypeError** – If any required fields have the wrong type.

**class_:** str

'class' member, with an underscore to avoid conflicts in Python.

**desc:** str

'desc' member

### Asyncio Protocol

Generic Asynchronous Message-based Protocol Support

This module provides a generic framework for sending and receiving messages over an asyncio stream. *AsyncProtocol* is an abstract class that implements the core mechanisms of a simple send/receive protocol, and is designed to be extended.

In this package, it is used as the implementation for the *QMPClient* class.

**class** qemu.qmp.protocol.**Runstate**(*value*)

Bases: Enum

Protocol session runstate.

**IDLE = 0**

Fully quiesced and disconnected.

**CONNECTING = 1**

In the process of connecting or establishing a session.

**RUNNING = 2**

Fully connected and active session.

**DISCONNECTING = 3**

In the process of disconnecting. Runstate may be returned to *IDLE* by calling *disconnect()*.

**exception** qemu.qmp.protocol.**ConnectError**(*error_message: str*, *exc: Exception*)

Bases: *QMPError*

Raised when the initial connection process has failed.

This Exception always wraps a "root cause" exception that can be interrogated for additional information.

For example, when connecting to a non-existent socket:

```
await qmp.connect('not_found.sock')
# ConnectError: Failed to establish connection:
#               [Errno 2] No such file or directory
```

**Parameters**

- **error_message** – Human-readable string describing the error.
- **exc** – The root-cause exception.

**error_message: str**

Human-readable error string

**exc: Exception**

Wrapped root cause exception

**exception** qemu.qmp.protocol.**StateError**(*error_message: str*, *state:* Runstate, *required:* Runstate)

Bases: *QMPError*

An API command (connect, execute, etc) was issued at an inappropriate time.

This error is raised when a command like *connect()* is called when the client is already connected.

**Parameters**

- **error_message** – Human-readable string describing the state violation.

- **state** – The actual *Runstate* seen at the time of the violation.
- **required** – The *Runstate* required to process this command.

qemu.qmp.protocol.**require**(*required_state:* Runstate) → Callable[[F], F]

Decorator: protect a method so it can only be run in a certain *Runstate*.

**Parameters**
**required_state** – The *Runstate* required to invoke this method.

**Raises**
***StateError*** – When the required *Runstate* is not met.

**class** qemu.qmp.protocol.**AsyncProtocol**(*name: Optional[str] = None*)

Bases: Generic[T]

AsyncProtocol implements a generic async message-based protocol.

This protocol assumes the basic unit of information transfer between client and server is a "message", the details of which are left up to the implementation. It assumes the sending and receiving of these messages is full-duplex and not necessarily correlated; i.e. it supports asynchronous inbound messages.

It is designed to be extended by a specific protocol which provides the implementations for how to read and send messages. These must be defined in *_do_recv()* and *_do_send()*, respectively.

Other callbacks have a default implementation, but are intended to be either extended or overridden:

- ***_establish_session***:
  The base implementation starts the reader/writer tasks. A protocol implementation can override this call, inserting actions to be taken prior to starting the reader/writer tasks before the super() call; actions needing to occur afterwards can be written after the super() call.
- ***_on_message***:
  Actions to be performed when a message is received.
- ***_cb_outbound***:
  Logging/Filtering hook for all outbound messages.
- ***_cb_inbound***:
  Logging/Filtering hook for all inbound messages. This hook runs *before* *_on_message()*.

**Parameters**
**name** – Name used for logging messages, if any. By default, messages will log to 'qemu.qmp.protocol', but each individual connection can be given its own logger by giving it a name; messages will then log to 'qemu.qmp.protocol.${name}'.

**_limit = 65536**

**name: Optional[str]**
The nickname for this connection, if any. This name is used for differentiating instances in debug output.

**logger = <Logger qemu.qmp.protocol (WARNING)>**
Logger object for debugging messages from this connection.

**_dc_task: Optional[asyncio.Future[None]]**
Disconnect task. The disconnect implementation runs in a task so that asynchronous disconnects (initiated by the reader/writer) are allowed to wait for the reader/writers to exit.

**property runstate: *Runstate***
The current *Runstate* of the connection.

**async runstate_changed**() → *Runstate*

Wait for the *runstate* to change, then return that *Runstate*.

**async start_server_and_accept**(*address: Union[str, Tuple[str, int]]*, *ssl: Optional[SSLContext] = None*) → None

Accept a connection and begin processing message queues.

If this call fails, *runstate* is guaranteed to be set back to *IDLE*. This method is precisely equivalent to calling *start_server()* followed by *accept()*.

**Parameters**

- **address** – Address to listen on; UNIX socket path or TCP address/port.
- **ssl** – SSL context to use, if any.

**Raises**

- ***StateError*** – When the *Runstate* is not *IDLE*.
- ***ConnectError*** – When a connection or session cannot be established.

  This exception will wrap a more concrete one. In most cases, the wrapped exception will be *OSError* or *EOFError*. If a protocol-level failure occurs while establishing a new session, the wrapped error may also be a *QMPError*.

**async start_server**(*address: Union[str, Tuple[str, int]]*, *ssl: Optional[SSLContext] = None*) → None

Start listening for an incoming connection, but do not wait for a peer.

This method starts listening for an incoming connection, but does not block waiting for a peer. This call will return immediately after binding and listening on a socket. A later call to *accept()* must be made in order to finalize the incoming connection.

**Parameters**

- **address** – Address to listen on; UNIX socket path or TCP address/port.
- **ssl** – SSL context to use, if any.

**Raises**

- ***StateError*** – When the *Runstate* is not *IDLE*.
- ***ConnectError*** – When the server could not start listening on this address.

  This exception will wrap a more concrete one. In most cases, the wrapped exception will be *OSError*.

**async accept**() → None

Accept an incoming connection and begin processing message queues.

Used after a previous call to *start_server()* to accept an incoming connection. If this call fails, *runstate* is guaranteed to be set back to *IDLE*.

**Raises**

- ***StateError*** – When the *Runstate* is not *CONNECTING*.
- ***QMPError*** – When *start_server()* was not called first.
- ***ConnectError*** – When a connection or session cannot be established.

  This exception will wrap a more concrete one. In most cases, the wrapped exception will be *OSError* or *EOFError*. If a protocol-level failure occurs while establishing a new session, the wrapped error may also be an *QMPError*.

**async connect**(*address: Union[str, Tuple[str, int]]*, *ssl: Optional[SSLContext] = None*) → None

Connect to the server and begin processing message queues.

If this call fails, *runstate* is guaranteed to be set back to *IDLE*.

**Parameters**

- **address** – Address to connect to; UNIX socket path or TCP address/port.
- **ssl** – SSL context to use, if any.

**Raises**

- ***StateError*** – When the *Runstate* is not *IDLE*.
- ***ConnectError*** – When a connection or session cannot be established.

  This exception will wrap a more concrete one. In most cases, the wrapped exception will be OSError or EOFError. If a protocol-level failure occurs while establishing a new session, the wrapped error may also be an *QMPError*.

**async disconnect**() → None

Disconnect and wait for all tasks to fully stop.

If there was an exception that caused the reader/writers to terminate prematurely, it will be raised here.

**Raises**

**Exception** – When the reader or writer terminate unexpectedly. You can expect to see EOFError if the server hangs up, or OSError for connection-related issues. If there was a QMP protocol-level problem, *ProtocolError* will be seen.

**async _session_guard**(*coro: Awaitable[None]*, *emsg: str*) → None

Async guard function used to roll back to *IDLE* on any error.

On any Exception, the state machine will be reset back to *IDLE*. Most Exceptions will be wrapped with *ConnectError*, but BaseException events will be left alone (This includes asyncio.CancelledError, even prior to Python 3.8).

**Parameters**

**error_message** – Human-readable string describing what connection phase failed.

**Raises**

- **BaseException** – When BaseException occurs in the guarded block.
- ***ConnectError*** – When any other error is encountered in the guarded block.

**property _runstate_event: Event**

**_set_state**(*state:* Runstate) → None

Change the *Runstate* of the protocol connection.

Signals the *runstate_changed* event.

**async _stop_server**() → None

Stop listening for / accepting new incoming connections.

**async _incoming**(*reader: StreamReader*, *writer: StreamWriter*) → None

Accept an incoming connection and signal the upper_half.

This method does the minimum necessary to accept a single incoming connection. It signals back to the upper_half ASAP so that any errors during session initialization can occur naturally in the caller's stack.

**Parameters**

- **reader** – Incoming asyncio.StreamReader
- **writer** – Incoming asyncio.StreamWriter

async _do_start_server(*address: Union[str, Tuple[str, int]]*, *ssl: Optional[SSLContext] = None*) → None

Start listening for an incoming connection, but do not wait for a peer.

This method starts listening for an incoming connection, but does not block waiting for a peer. This call will return immediately after binding and listening to a socket. A later call to accept() must be made in order to finalize the incoming connection.

**Parameters**

- **address** – Address to listen on; UNIX socket path or TCP address/port.
- **ssl** – SSL context to use, if any.

**Raises**

OSError – For stream-related errors.

async _do_accept() → None

Wait for and accept an incoming connection.

Requires that we have not yet accepted an incoming connection from the upper_half, but it's OK if the server is no longer running because the bottom_half has already accepted the connection.

async _do_connect(*address: Union[str, Tuple[str, int]]*, *ssl: Optional[SSLContext] = None*) → None

Acting as the transport client, initiate a connection to a server.

**Parameters**

- **address** – Address to connect to; UNIX socket path or TCP address/port.
- **ssl** – SSL context to use, if any.

**Raises**

OSError – For stream-related errors.

async _establish_session() → None

Establish a new session.

Starts the readers/writer tasks; subclasses may perform their own negotiations here. The Runstate will be RUNNING upon successful conclusion.

_schedule_disconnect() → None

Initiate a disconnect; idempotent.

This method is used both in the upper-half as a direct consequence of *disconnect()*, and in the bottom-half in the case of unhandled exceptions in the reader/writer tasks.

It can be invoked no matter what the *runstate* is.

async _wait_disconnect() → None

Waits for a previously scheduled disconnect to finish.

This method will gather any bottom half exceptions and re-raise the one that occurred first; presuming it to be the root cause of any subsequent Exceptions. It is intended to be used in the upper half of the call chain.

**Raises**

Exception – Arbitrary exception re-raised on behalf of the reader/writer.

**_cleanup**() → None

Fully reset this object to a clean state and return to *IDLE*.

**async _bh_disconnect**() → None

Disconnect and cancel all outstanding tasks.

It is designed to be called from its task context, *_dc_task*. By running in its own task, it is free to wait on any pending actions that may still need to occur in either the reader or writer tasks.

**async _bh_flush_writer**() → None

**async _bh_close_stream**(*error_pathway: bool = False*) → None

**async _bh_loop_forever**(*async_fn: Callable[[], Awaitable[None]]*, *name: str*) → None

Run one of the bottom-half methods in a loop forever.

If the bottom half ever raises any exception, schedule a disconnect that will terminate the entire loop.

**Parameters**

- **async_fn** – The bottom-half method to run in a loop.
- **name** – The name of this task, used for logging.

**async _bh_send_message**() → None

Wait for an outgoing message, then send it.

Designed to be run in *_bh_loop_forever()*.

**async _bh_recv_message**() → None

Wait for an incoming message and call *_on_message* to route it.

Designed to be run in *_bh_loop_forever()*.

**_cb_outbound**(*msg: T*) → T

Callback: outbound message hook.

This is intended for subclasses to be able to add arbitrary hooks to filter or manipulate outgoing messages. The base implementation does nothing but log the message without any manipulation of the message.

**Parameters**
**msg** – raw outbound message

**Returns**
final outbound message

**_cb_inbound**(*msg: T*) → T

Callback: inbound message hook.

This is intended for subclasses to be able to add arbitrary hooks to filter or manipulate incoming messages. The base implementation does nothing but log the message without any manipulation of the message.

This method does not "handle" incoming messages; it is a filter. The actual "endpoint" for incoming messages is *_on_message()*.

**Parameters**
**msg** – raw inbound message

**Returns**
processed inbound message

**async _readline**() → bytes

Wait for a newline from the incoming reader.

This method is provided as a convenience for upper-layer protocols, as many are line-based.

This method *may* return a sequence of bytes without a trailing newline if EOF occurs, but *some* bytes were received. In this case, the next call will raise EOFError. It is assumed that the layer 5 protocol will decide if there is anything meaningful to be done with a partial message.

**Raises**

- **OSError** – For stream-related errors.
- **EOFError** – If the reader stream is at EOF and there are no bytes to return.

**Returns**

bytes, including the newline.

**async _do_recv**() → T

Abstract: Read from the stream and return a message.

Very low-level; intended to only be called by *_recv()*.

**async _recv**() → T

Read an arbitrary protocol message.

**Warning:** This method is intended primarily for *_bh_recv_message()* to use in an asynchronous task loop. Using it outside of this loop will "steal" messages from the normal routing mechanism. It is safe to use prior to *_establish_session()*, but should not be used otherwise.

This method uses *_do_recv()* to retrieve the raw message, and then transforms it using *_cb_inbound()*.

**Returns**

A single (filtered, processed) protocol message.

**_do_send**(*msg: T*) → None

Abstract: Write a message to the stream.

Very low-level; intended to only be called by *_send()*.

**async _send**(*msg: T*) → None

Send an arbitrary protocol message.

This method will transform any outgoing messages according to *_cb_outbound()*.

**Warning:** Like *_recv()*, this method is intended to be called by the writer task loop that processes outgoing messages. Calling it directly may circumvent logic implemented by the caller meant to correlate outgoing and incoming messages.

**Raises**

**OSError** – For problems with the underlying stream.

**async _on_message**(*msg: T*) → None

Called to handle the receipt of a new message.

> **Caution:** This is executed from within the reader loop, so be advised that waiting on either the reader or writer task will lead to deadlock. Additionally, any unhandled exceptions will directly cause the loop to halt, so logic may be best-kept to a minimum if at all possible.

**Parameters**
**msg** – The incoming message, already logged/filtered.

### QMP Protocol

QMP Protocol Implementation

This module provides the *QMPClient* class, which can be used to connect and send commands to a QMP server such as QEMU. The QMP class can be used to either connect to a listening server, or used to listen and accept an incoming connection from that server.

**exception** qemu.qmp.qmp_client.**GreetingError**(*error_message: str*, *exc: Exception*)

Bases: _WrappedProtocolError

An exception occurred during the Greeting phase.

**Parameters**

- **error_message** – Human-readable string describing the error.
- **exc** – The root-cause exception.

**error_message: str**

Human-readable error message, without any prefix.

**exception** qemu.qmp.qmp_client.**NegotiationError**(*error_message: str*, *exc: Exception*)

Bases: _WrappedProtocolError

An exception occurred during the Negotiation phase.

**Parameters**

- **error_message** – Human-readable string describing the error.
- **exc** – The root-cause exception.

**error_message: str**

Human-readable error message, without any prefix.

**exception** qemu.qmp.qmp_client.**ExecuteError**(*error_response:* ErrorResponse, *sent:* Message, *received:* Message)

Bases: *QMPError*

Exception raised by *QMPClient.execute()* on RPC failure.

This exception is raised when the server received, interpreted, and replied to a command successfully; but the command itself returned a failure status.

For example:

```
await qmp.execute('block-dirty-bitmap-add',
                  {'node': 'foo', 'name': 'my_bitmap'})
# qemu.qmp.qmp_client.ExecuteError:
#     Cannot find device='foo' nor node-name='foo'
```

**Parameters**

- **error_response** – The RPC error response object.
- **sent** – The sent RPC message that caused the failure.
- **received** – The raw RPC error reply received.

**sent:** *Message*

The sent *Message* that caused the failure

**received:** *Message*

The received *Message* that indicated failure

**error:** *ErrorResponse*

The parsed error response

**error_class:** str

The QMP error class

**exception** qemu.qmp.qmp_client.**ExecInterruptedError**

Bases: *QMPError*

Exception raised by *execute()* (et al) when an RPC is interrupted.

This error is raised when an *execute()* statement could not be completed. This can occur because the connection itself was terminated before a reply was received. The true cause of the interruption will be available via *disconnect()*.

The QMP protocol does not make it possible to know if a command succeeded or failed after such an event; the client will need to query the server to determine the state of the server on a case-by-case basis.

For example, ECONNRESET might look like this:

```
try:
    await qmp.execute('query-block')
    # ExecInterruptedError: Disconnected
except ExecInterruptedError:
    await qmp.disconnect()
    # ConnectionResetError: [Errno 104] Connection reset by peer
```

**exception** qemu.qmp.qmp_client.**ServerParseError**(*error_message: str*, *msg:* Message)

Bases: _MsgProtocolError

The Server sent a *Message* indicating parsing failure.

i.e. A reply has arrived from the server, but it is missing the "ID" field, indicating a parsing error.

**Parameters**

- **error_message** – Human-readable string describing the error.
- **msg** – The QMP *Message* that caused the error.

**error_message:** str

Human-readable error message, without any prefix.

**exception** qemu.qmp.qmp_client.**BadReplyError**(*error_message: str*, *msg:* Message, *sent:* Message)

Bases: _MsgProtocolError

An execution reply was successfully routed, but not understood.

If a QMP message is received with an 'id' field to allow it to be routed, but is otherwise malformed, this exception will be raised.

A reply message is malformed if it is missing either the 'return' or 'error' keys, or if the 'error' value has missing keys or members of the wrong type.

**Parameters**

- **error_message** – Human-readable string describing the error.
- **msg** – The malformed reply that was received.
- **sent** – The message that was sent that prompted the error.

**sent**

The sent *Message* that caused the failure

**error_message: str**

Human-readable error message, without any prefix.

**class** qemu.qmp.qmp_client.**QMPClient**(*name: Optional[str] = None*)

Bases: *AsyncProtocol*[*Message*], *Events*

Implements a QMP client connection.

*QMPClient* can be used to either connect or listen to a QMP server, but always acts as the QMP client.

**Parameters**

**name** – Optional nickname for the connection, used to differentiate instances when logging.

Basic script-style usage looks like this:

```
import asyncio
from qemu.qmp import QMPClient

async def main():
    qmp = QMPClient('my_virtual_machine_name')
    await qmp.connect(('127.0.0.1', 1234))
    ...
    res = await qmp.execute('query-block')
    ...
    await qmp.disconnect()

asyncio.run(main())
```

A more advanced example that starts to take advantage of asyncio might look like this:

```
class Client:
    def __init__(self, name: str):
        self.qmp = QMPClient(name)

    async def watch_events(self):
        try:
            async for event in self.qmp.events:
                print(f"Event: {event['event']}")
        except asyncio.CancelledError:
            return

    async def run(self, address='/tmp/qemu.socket'):
```

(continues on next page)

(continued from previous page)

```
        await self.qmp.connect(address)
        asyncio.create_task(self.watch_events())
        await self.qmp.runstate_changed.wait()
        await self.disconnect()
```

See *qmp.events* for more detail on event handling patterns.

**logger:** logging.Logger = <Logger qemu.qmp.qmp_client (WARNING)>

Logger object used for debugging messages.

**await_greeting:** bool

Whether or not to await a greeting after establishing a connection. Defaults to True; QGA servers expect this to be False.

**negotiate:** bool

Whether or not to perform capabilities negotiation upon connection. Implies *await_greeting*. Defaults to True; QGA servers expect this to be False.

**property greeting:** Optional[*Greeting*]

The *Greeting* from the QMP server, if any.

Defaults to None, and will be set after a greeting is received during the connection process. It is reset at the start of each connection attempt.

**async execute_msg**(*msg:* Message) → object

Execute a QMP command on the server and return its value.

**Parameters**
**msg** – The QMP *Message* to execute.

**Returns**
The command execution return value from the server. The type of object returned depends on the command that was issued, though most in QEMU return a dict.

**Raises**

- **ValueError** – If the QMP *Message* does not have either the 'execute' or 'exec-oob' fields set.
- ***ExecuteError*** – When the server returns an error response.
- ***ExecInterruptedError*** – If the connection was disrupted before receiving a reply from the server.

**classmethod make_execute_msg**(*cmd: str*, *arguments: Optional[Mapping[str, object]] = None*, *oob: bool = False*) → *Message*

Create an executable message to be sent by *execute_msg* later.

**Parameters**

- **cmd** – QMP command name.
- **arguments** – Arguments (if any). Must be JSON-serializable.
- **oob** – If True, execute "out of band".

**Returns**
A QMP *Message* that can be executed with *execute_msg()*.

**async execute**(*cmd: str*, *arguments: Optional[Mapping[str, object]] = None*, *oob: bool = False*) → object

Execute a QMP command on the server and return its value.

**Parameters**

- **cmd** – QMP command name.
- **arguments** – Arguments (if any). Must be JSON-serializable.
- **oob** – If `True`, execute "out of band".

**Returns**

The command execution return value from the server. The type of object returned depends on the command that was issued, though most in QEMU return a `dict`.

**Raises**

- ***ExecuteError*** – When the server returns an error response.
- ***ExecInterruptedError*** – If the connection was disrupted before receiving a reply from the server.

**send_fd_scm**(*fd: int*) → None

Send a file descriptor to the remote via SCM_RIGHTS.

This method does not close the file descriptor.

**Parameters**

**fd** – The file descriptor to send to QEMU.

This is an advanced feature of QEMU where file descriptors can be passed from client to server. This is usually used as a security measure to isolate the QEMU process from being able to open its own files. See the QMP commands `getfd` and `add-fd` for more information.

See `socket.socket.sendmsg` for more information on the Python implementation for sending file descriptors over a UNIX socket.

**name: Optional[str]**

The nickname for this connection, if any. This name is used for differentiating instances in debug output.

**events: *EventListener***

Default, all-events *EventListener*. See *qmp.events* for more info.

### Utilities

Miscellaneous Utilities

This module provides asyncio utilities and compatibility wrappers for Python 3.6 to provide some features that otherwise become available in Python 3.7+.

Various logging and debugging utilities are also provided, such as *exception_summary()* and *pretty_traceback()*, used primarily for adding information into the logging stream.

**async** qemu.qmp.util.**flush**(*writer: StreamWriter*) → None

Utility function to ensure an `asyncio.StreamWriter` is *fully* drained.

`asyncio.StreamWriter.drain` only promises we will return to below the "high-water mark". This function ensures we flush the entire buffer – by setting the high water mark to 0 and then calling drain. The flow control limits are restored after the call is completed.

qemu.qmp.util.**upper_half**(*func: T*) → T

Do-nothing decorator that annotates a method as an "upper-half" method.

These methods must not call bottom-half functions directly, but can schedule them to run.

qemu.qmp.util.**bottom_half**(*func: T*) → T

Do-nothing decorator that annotates a method as a "bottom-half" method.

These methods must take great care to handle their own exceptions whenever possible. If they go unhandled, they will cause termination of the loop.

These methods do not, in general, have the ability to directly report information to a caller's context and will usually be collected as an `asyncio.Task` result instead.

They must not call upper-half functions directly.

qemu.qmp.util.**create_task**(*coro: Coroutine[Any, Any, T]*, *loop: Optional[AbstractEventLoop] = None*) → asyncio.Future[T]

Python 3.6-compatible `asyncio.create_task` wrapper.

**Parameters**

- **coro** – The coroutine to execute in a task.
- **loop** – Optionally, the loop to create the task in.

**Returns**
An `asyncio.Future` object.

qemu.qmp.util.**is_closing**(*writer: StreamWriter*) → bool

Python 3.6-compatible `asyncio.StreamWriter.is_closing` wrapper.

**Parameters**
**writer** – The `asyncio.StreamWriter` object.

**Returns**
`True` if the writer is closing, or closed.

**async** qemu.qmp.util.**wait_closed**(*writer: StreamWriter*) → None

Python 3.6-compatible `asyncio.StreamWriter.wait_closed` wrapper.

**Parameters**
**writer** – The `asyncio.StreamWriter` to wait on.

qemu.qmp.util.**asyncio_run**(*coro: Coroutine[Any, Any, T]*, ***, *debug: bool = False*) → T

Python 3.6-compatible `asyncio.run` wrapper.

**Parameters**
**coro** – A coroutine to execute now.

**Returns**
The return value from the coroutine.

qemu.qmp.util.**exception_summary**(*exc: BaseException*) → str

Return a summary string of an arbitrary exception.

It will be of the form "ExceptionType: Error Message" if the error string is non-empty, and just "ExceptionType" otherwise.

This code is based on CPython's implementation of `traceback.TracebackException.format_exception_only`.

qemu.qmp.util.**pretty_traceback**(*prefix: str = '| '*) → str

Formats the current traceback, indented to provide visual distinction.

This is useful for printing a traceback within a traceback for debugging purposes when encapsulating errors to deliver them up the stack; when those errors are printed, this helps provide a nice visual grouping to quickly identify the parts of the error that belong to the inner exception.

**Parameters**

**prefix** – The prefix to append to each line of the traceback.

**Returns**

A string, formatted something like the following:

```
| Traceback (most recent call last):
|   File "foobar.py", line 42, in arbitrary_example
|     foo.baz()
| ArbitraryError: [Errno 42] Something bad happened!
```

CHAPTER

NINE

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## q

# INDEX

## Symbols



## A



## B



## C

## D



## E



## F



## G



## H



## I



## L



## M



## N