Controller

`archon` provides a `wrapper` for the STA Archon controller and exposes the controller features as an `AMQPActor`. Note that `archon` uses `asyncio` throughout the codebase to enable asynchronicity.

`ArchonController` provides a mid-level interface to the Archon controller and implements the protocol for communicating with the controller.

```
>>> from archon.controller import ArchonController
>>> archon = ArchonController('my_controller', '10.7.45.25')
>>> archon
<archon.controller.controller.ArchonController at 0x7f5de43d3ee0>
```

After instantiating `ArchonController`, it must be started to establish the connection to the controller TCP server. When the controller is started, the parameters are reset, the internal status set to idle, and autoflushing is enabled.

```
>>> await archon.start()
```

Warning

`ArchonController` will only work with an Archon configuration script that matches the behaviour and parameters it expects. Since there are many ways to implement those features, it will **not** work with any script.

Sending commands

`send_command` allows to send a raw command to the controller, while managing command IDs, message formatting, and reply parsing. The replies from the controller are stored in `ArchonCommand.replies` as a list of `ArchonCommandReply`.

```
>>> cmd = archon.send_command('SYSTEM')
>>> cmd
<ArchonCommand (>01SYSTEM, status=ArchonCommandStatus.RUNNING)>
>>> await cmd
>>> cmd
<ArchonCommand (>00SYSTEM, status=ArchonCommandStatus.DONE)>
>>> cmd.replies
[<ArchonCommandReply (b'<01BACKPLANE_TYPE=1 BACKPLANE_REV=5 BACKPLANE_VERSION=1.0.11
```

`send_command` returns an `ArchonCommand` instance, which is a subclass of `Future`. The command is sent to the controller as soon as `send_command()` is called; awaiting the resulting `ArchonCommand` will asynchronously block until the command is done (successfully or not).

ArchonController keeps an internal list of all the running commands and associates replies with the corresponding command. Normally an Archon command expects a single reply, at which point the command is marked done. The only exception is the `FETCH` command which returns a number of binary replies. `ArchonController` handles this case in an efficient way when `fetch` is called.

If a command receives a failed reply, the status of the command will be set to `ArchonCommandStatus.FAILED`

```
>>> cmd = await archon.send_command('SYSTEM')
>>> cmd.status
ArchonCommandStatus.FAILED
```

All command statuses are instances of `ArchonCommandStatus`.

Normally a command will wait indefinitely for a reply. It's possible to set a timeout after which the command is failed with status `ArchonCommandStatus.TIMEDOUT` by passing `timeout=` to `send_command`.

Controller status and configuration

`ArchonController` also provides some mid-level routines to automate complex processes such as `loading` or `saving a configuration file`, retrieving the `controller status`, or reading the system status <.ArchonController.get_system>. All these methods are relatively straightforward to use and we refer the user to the API documentation.

`ArchonController` maintains an internal `status` for the controller (e.g., whether it's idle, exposing, or reading). This status is not retrieved from the device itself since the Archon firmware doesn't provide a way to query the timing script or parameters.

The status is a bitmask of `ControllerStatus` bits

```
>>> archon.status
<ControllerStatus.READOUT_PENDING|EXPOSING: 12>
```

Note that certain bits are incompatible and should never appear together (e.g., `EXPOSING` and `IDLE`). The internal status can be updated using `ArchonController.update_status`, which handles turning off incompatible bits, although it's not recommended to do so.

It's possible to "subscribe" to the controller status via the `yield_status` asynchronous generator. This generator will asynchronous yield the status of the device only when the bitmask changes.

```
async for status in controller.yield_status():
    print(status)
```

Exposing and reading the CCDs

To expose the CCDs attached to the controller use the `expose` method which accepts an exposure time

```
await archon.expose(15., readout=True)
```

The coroutine will turn off autoflushing, start the device integration routine, wait the desired exposure time, and turn off integration. When the integration starts the status bits `EXPOSING` and `READOUT_PENDING` are turned on. When the integration is complete the `EXPOSING` bit changes to `IDLE`. Note that `ArchonController.expose` returns immediately after the integration starts and returns a task that can be awaited until the readout finishes.

```
>>> expose_task = await archon.expose(15., readout=True)
# The exposure has started but expose() returns immediately.

>>> await expose_task  # This will block until the exposure and readout ends.
```

If `readout=True`, the readout routine will start immediately, at which point the status changes to `READING`. If `readout=False` one must manually call `readout` to start the chip readout.

Once readout is complete the buffer can be retrieved by calling `fetch`. `fetch` will identify the last completed buffer, retrieve its contents, and return a Numpy array with the appropriate number of lines and pixels.

```
>>> image = await archon.fetch()
>>> image
array([[12932, 11918, 12688, ..., 12998, 13486, 14235],
       [11619, 10529, 10613, ..., 10323, 12366, 11106],
       [11807, 10555, 10588, ..., 12059, 11573, 12342],
       ...,
       [ 9736, 10368, 10581, ..., 12534,  3538,  3768],
       [10467, 10653, 10537, ..., 10779,  3717,  3569],
       [11940, 10226, 10294, ..., 10596,  5329,  4796]], dtype=uint16)
```

Configuration

`ArchonController` is relatively agnostic to configuration parameters but some timeout values are defined in `etc/archon.yml` and accessible as

```
>>> from archon import config
{'archon': {'default_parameters': {}},
 'timeouts': {'controller_connect': 5,
              'expose_timeout': 2,
              'fetching_expected': 5,
              'fetching_max': 10,
              'flushing': 1.2,
              'readout_expected': 40,
              'readout_max': 60,
              'write_config_delay': 0.0001,
              'write_config_timeout': 2}}
```

Generally these values are reasonable and don’t need to be modified.