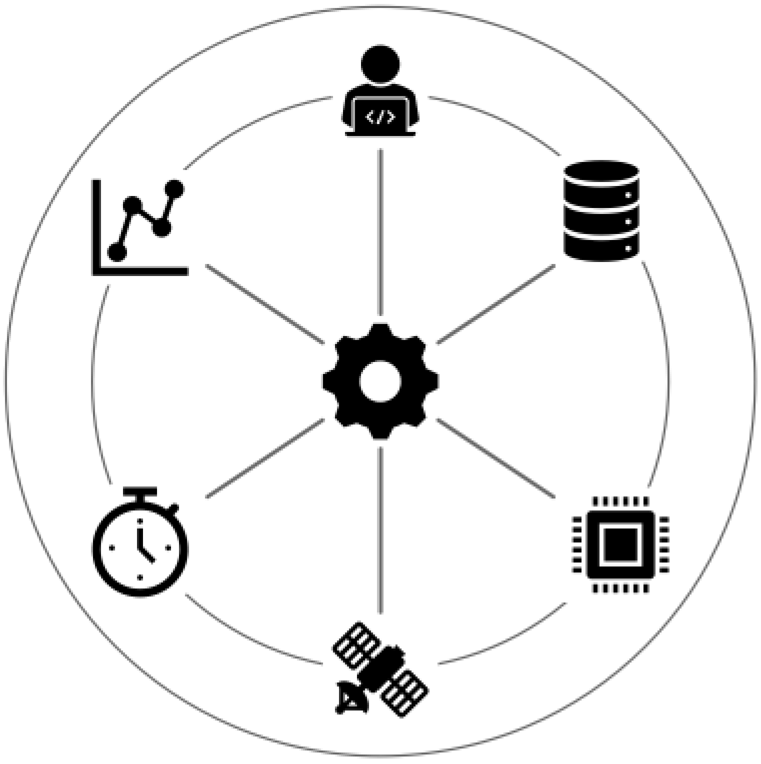


Common–EGSE

Developer Manual

Rik Huygen, Sara Regibo

Version 0.1, 21/05/2022

Table of Contents

Colophon

Copyright © 2022 by the KU Leuven PLATO CGSE Team

1st Edition — May 2022

This manual is written in PyCharm using the AsciiDoc plugin. The PDF Book version is processed with asciidoctor-pdf. The source code is available in a GitHub repository at XXXXX.

When you find an error or inconsistency or you have some improvements to the text, feel free to raise an issue or create a pull request. Any contribution is greatly appreciated and will be mentioned in the acknowledgement section.

Contributors

Name	Affiliation	Role
Rik Huygen	KU Leuven	
Sara Regibo	KU Leuven	
Pierre Royer	KU Leuven	
Sena Gomashie	SRON	
Jens Johansen	SRON	
Nicolas Beraud	IAS	
Pierre Guiot	IAS	
Angel Valverde	INTA	
Jesus Saiz	INTA	

Preface

During the development/implementation process it became clear that the project was bigger then we anticipated and we would need to document how the Common-EGSE developed into this full-blown product.

During the analysis, design and implementation phase we learned about so many libraries like ZeroMQ, Matplotlib, Numpy, Pandas, Rich, Textual, … but also the standard Python library is so extremely rich of well-designed modules and language concepts that we integrated into our design and implementation.

Many of the applied concepts might seem obvious for the core developers of a/this product, but it might not be for the contributors of e.g. device drivers. A full in-depth description of the system and how it was developed is therefore indispensable.

The text itself gruw over the years, it started in reStructuredText and was processed with Sphinx, then we moved to Markdown and Mkdocs because of its simplicity and thinking this would encourage us to write and update the text more often. Finally, we ended up using Asciidoc which is a bit of a compromise between the previous two.

List of TODO topics

- ☐ List the terminal commands in a simple overview table or something
- ☐ Explain System Preferences in the settings.yaml file (and the local settings)
- ☐ The Logger uses ZeroMQ to pass messages. That means these this ZeroMQ needs to be closed when your application ends.
- ☐ Where do we descibe the Synoptics Manager, where does this process fit into the design fiures etc.
- ☐ Somewhere we need to describe all the modules in the CGSE. For example, the `device.py` module defines a lot of interesting classes, including Exceptions, the device connection interfaces, and the device transport.
- ☐ Somewhere it shall be clearly explained where all these processes need to be started, on the egse-server or egse-client, manually or by Systemd, … or clicking an icon?
- ☐ Should we maybe divide the developer manual in part 1 with coding advice and part 2 with the description of the CGSE code?
- ☐ Explain how the Failure — Success — Response classes work
- ☐ Explain how you can run the GUIs and even some control servers on your local machine using port forwarding. As an example, use the TCS or the Hexapod. This is only for hardware device connections, it is not for core services, run them as normal on your local machine.
- ☐ Find better names for the Parts in the manual
- ☐ WHere do we discuss devices with Ethernet interfaces and USB interfaces, the differences, the pros and cons, are there other still in use, like GPIO?

Documents and Acronyms

Applicable documents

[AD-01]	PLATO Common-EGSE Requirements Specification, PLATO-KUL-PL-RS-0001, issue 1.4, 18/03/2020
[AD-02]	PLATO Common-EGSE Design Description, PLATO-KUL-PL-DD-0001, issue 1.2, 13/03/2020
[AD-03]	PLATO Common-EGSE Interface Control Document, PLATO-KUL-PL-ICD-0002, issue 0.1, 19/03/2020

Reference Documents

[RD-01]	PLATO Common-EGSE Installation Guide, PLATO-KUL-PL-MAN-0002
[RD-02]	PLATO Common-EGSE User Manual, PLATO-KUL-PL-MAN-0001
[RD-03]	Common-EGSE on-line Documentation https://ivs-kuleuven.github.io/plato-cgse-docs/

Acronyms

AEU	Ancillary Electronics Unit
API	Application Programming Interface
CAM	Camera
COT	Commercial off-the-shelf
DPU	Data Processing Unit
DSI	Diagnostic SpaceWire Interface
EGSE	Electrical Ground Support Equipment
OS	Operating System
PLM	Payload Module
REP	Read-Evaluate-Print Loop, e.g. the Python interpreter prompt
RMAP	Remote Memory Access Protocol
SpW	SpaceWire

SVM	Service Module
TBC	To Be Confirmed
TBD	To Be Decided or To Be Defined
TBW	To Be Written
TV	Thermal Vacuum
USB	Universal Serial Bus

Caveats

Place general warnings here on topics where the user might make specific assumption on the code or the usage of the code.

Setup

1. It's not a good idea to create keys with spaces and special characters, although it is allowed in a dictionary and it works without problems, the key will not be available as an attribute because it will violate the Python syntax.

   ```
   >>> from egse.setup import Setup
   >>> s = Setup()
   >>> s["a key with spaces"] = 42
   >>> print(s)
   NavigableDict
   └──── a key with spaces: 42
   >>> s['a key with spaces']
   42
   >>> s.a key with spaces
     Input In [18]
       s.a key with spaces
           ^
   SyntaxError: invalid syntax
   ```

2. When submitting a Setup to the configuration manager, the Setup is automatically pushed to the GitHub repository with the message provided with the `submit_setup()` function. No need anymore to create a pull request. There are however two thing to keep in mind here:

   a. do not add a Setup manually to your PLATO_CONF_FILE_LOCATION or to the repository. That will invalidate the repo and the cache that is maintained by the configuration manager

   b. you should do a git pull regularly on your local machine and also on the egse-client if the folder is not NFS mounted

Part I — Development Notions

Chapter 1. Style Guide

This part of the developer guide contains instructions for coding styles that are adopted for this project.

The style guide that we use for this project is PEP8. This is the standard for Python code and all IDEs, parsers and code formatters understand and work with this standard. PEP8 leaves room for project specific styles. A good style guide that we can follow is the Google Style Guide.

The following sections will give the most used conventions with a few examples of good and bad.

1.1. TL;DR

Type	**Style**	**Example**
Classes	CapWords	ProcessManager, ImageViewer, CommandList, Observation, MetaData
Methods & Functions	lowercase with underscores	get_value, set_mask, create_image
Variables	lowercase with underscores	key, last_value, model, index, user_info
Constants	UPPERCASE with underscores	MAX_LINES, BLACK, COMMANDING_PORT
Modules & packages	lowercase **no** underscores	dataset, commanding, multiprocessing

1.2. General

- name the class or variable or function with what it is, what it does or what it contains. A variable named `user_list` might look good at first, but what if at some point you want to change the list to a set so it can not contain duplicates. Are you going to rename everything into `user_set` or would `user_info` be a better name?
- never use dashes in any name, they will raise a `SyntaxError: invalid syntax`.
- we introduce a number of relaxations to not brake backward compatibility for the sake of a naming convention. As described in A Foolish Consistency is the Hobgoblin of Little Minds: *Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is the most important. […] do not break backwards compatibility just to comply with this PEP!*

1.3. Classes

Always use CamelCase (Python uses CapWords) for class names. When using acronyms, keep them all UPPER case.

- class names should be nouns, like Observation
- make sure to name classes distinctively
- stick to one word for a concept when naming classes, i.e. words like `Manager` or `Controller` or `Organizer` all mean similar things. Choose one word for the concept and stick to it.
- if a word is already part of a package or module, don't use the same word in the class name again.

Good names are: `Observation`, `CalibrationFile`, `MetaData`, `Message`, `ReferenceFrame`, `URLParser`.

1.4. Methods and Functions

A function or a method does something (and should only do one thing, SRP=Single Responsibility Principle), it is an action, so the name should reflect that action.

Always use lowercase words separated with underscores.

Good names are: `time_in_ms()`, `get_commanding_port()`, `is_connected()`, `parse_time()`, `setup_mask()`.

When working with legacy code or code from another project, names may be in camelCase (with the first letter a lower case letter). So we can in this case use also `getCommandPort()` or `isConnected()` as method and function names.

1.5. Variables

Use the same naming convention as functions and methods, i.e. lowercase with underscores.

Good names are: `key`, `value`, `user_info`, `model`, `last_value`

Bad names: `NSegments`, `outNoise`

Take care not to use builtins: `list`, `type`, `filter`, `lambda`, `map`, `dict`, …

Private variables (for classes) start with an underscore: `_name` or `_total_n_args`.

In the same spirit as method and function names, the variables can also be in camelCase for specific cases.

1.6. CONSTANTS

Use ALL_UPPER_CASE with underscores for constants. Use constants always within a name space, not globally.

Good names: `MAX_LINES`, `BLACK`, `YELLOW`, `ESL_LINK_MODE_DISABLED`

1.7. Modules and Packages

Use simple words for modules, preferably just one word like `datasets` or `commanding` or `storage` or `extensions`. If two words are unavoidable, just concatenate them, like `multiprocessing` or `sampledata` or `testdata`. If needed for readability, use an underscore to separate the words, e.g. `image_analysis`.

1.8. Import Statements

- group and sort import statements
- never use the form `from <module> import *`
- always use absolute imports in scripts

Be careful that you do not name any modules the same as a module in the Python standard library. This can result in strange effects and may result in an `AttributeError`. Suppose you have named a module `math` in the `egse` directory and it is imported and used further in the code as follows:

```
from egse import math

...

  # in some expression further down the code you might use

  math.exp(a)
```

This will result in the following runtime error:

```
  File "some_module.py", line 8, in <module>
    print(math.exp(a))
AttributeError: module 'egse.math' has no attribute 'exp'
```

Of course this is an obvious example, but it might be more obscure like e.g. in this GitHub issue: 'module' object has no attribute 'Cmd'.

Chapter 2. Best Practices for Error and Exception Handling

All errors and exceptions should be handled to prevent the Application from crashing. This is definitely true for server applications and services, e.g. device control servers, the storage and configuration manager. But also GUI applications should not crash due to an unhandled exception. It is important that, at least on the server side, all exceptions are logged.

2.1. When do I catch exceptions?

Exceptions shouldn't really be caught unless you have a really good plan for how to recover. If you have no such plan, let the exception propagate to a higher level in your software until you know how to handle it. In your own code, try to avoid raising an exception in the first place, design your code and classes such that you minimise throwing exceptions.

If some code path simply must broadly catch all exceptions (i.e. catch the base `Exception` class) — for example, the top-level loop for some long-running persistent process — then each such caught exception must write the full stack trace to the log, along with a timestamp. Not just the exception type and message, but the full stack trace. This can easily be done in Python as follows:

```
try:
    main_loop()
except Exception:
    logging.exception("Caught exception at the top level main_loop().")
```

This will log the exception and stacktrace with logging level ERROR.

For all other except clauses — which really should be the vast majority — the caught exception type must be as specific as possible. Something like `KeyError`, `ConnectionTimeout`, etc. That should be the exception that you have a plan for, that you can handle and recover from at this level in you code.

2.2. How do I catch Exceptions?

Use the `try`/`except` blocks around code that can potentially generate an exception *and* your code can recover from that exception.

Any resources such as open files or connections can be closed or cleaned up in the *finally* clause. Remember that the code within `finally` will always be executed, regardless of an Exception is thrown or not. In the example below, the function checks if there is internet connection by opening a socket to a host which is expected to be available at all times. The socket is closed in the finally clause even when the exception is raised.

```python
import socket
import logging

def has_internet(host="8.8.8.8", port=53, timeout=3):
    """Returns True if we have internet connection, False otherwise."""
    try:
        socket_ = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        socket_.settimeout(timeout)
        socket_.connect((host, port))
        return True
    except socket.error as ex:
        logging.info(f"No Internet: Unable to open socket to {host}:{port} [{ex}]")
        return False
    finally:
        if socket_ is not None:
            socket_.close()
```

2.3. What about the 'with' Statement?

When you use a `with` statement in your code, resources are automatically closed when an Exception is thrown, but the Exception is still thrown, so you should put the `with` block inside a `try`/`except` block.

As of Python 3 the `socket` class can be used as a context manager. The example above can thus be rewritten as follows:

```python
import socket
import logging

def has_internet(host="8.8.8.8", port=53, timeout=3):
    """Returns True if we have internet connection, False otherwise."""
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as socket_:
            socket_.settimeout(timeout)
            socket_.connect((host, port))
            return True
    except socket.error as ex:
        logging.warning(f"No Internet: Unable to open socket to {host}:{port} [{ex
}]")
        return False
```

Another example from our own code base shows how to handle a Hexapod PUNA Proxy error. Suppose you want to send some commands to the PUNA within a context manager as follows:

```
from egse.hexapod.symetrie.puna import PunaProxy

with PunaProxy() as proxy:
    proxy.move_absolute(0, 0, 2, 0, 0, 0)
    # Send other commands to the Puna Hexapod.
```

When you execute the above code and the PUNA control server is not active, the Proxy will not be able to connect and the context manager will raise a `ConnectionError`.

```
PunaProxy could not connect to its control server at tcp://localhost:6700. No
commands have been loaded.
Control Server seems to be off-line, abandoning
Traceback (most recent call last):
  File "t.py", line 3, in <module>
    with PunaProxy() as proxy:
  File "/Users/rik/Git/plato-common-egse/src/egse/proxy.py", line 129, in __enter__
    raise ConnectionError("Proxy is not connected when entering the context.")
ConnectionError: Proxy is not connected when entering the context.
```

So, the `with .. as` statement should be put into a `try .. except` clause. Since we probably cannot handle this exception at this level, we just re-raise the connection error.

```
from egse.hexapod.symetrie.puna import PunaProxy

try:
    with PunaProxy() as proxy:
        proxy.move_absolute(0,0,2,0,0,0)
        # Send other commands to the Puna Hexapod.
except ConnectionError as exc:
    logger.exception("Could not send commands to the Hexapod PUNA because the
control server is not reachable.")
    raise  ①
```

① use a single `raise` statement, don't repeat the `ConnectionError` here

2.4. Why not just return an error code?

In languages like the C programming language is it custom to return error codes or `-1` as a return code from a function to indicate a problem has occurred. The main drawback here is that your code, when calling such a function, must always check it's return codes, which is often forgotten or ignored.

In Python, throw exceptions instead of returning an error code. Exceptions ensure that failures do not go unnoticed because calling code didn't check a return code.

2.5. When to Test instead of Try?

The question basically is if we should check for a common condition without possibly throwing an exception. Python does this different than other languages and prefers the use of exceptions. There are two different opinions about this, EAFP and LBYL. From the Python documentation:

EAFP: **it’s Easier to Ask for Forgiveness than Permission.**

This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many try and except statements. The technique contrasts with the LBYL style common to many other languages such as C.

LBYL: **Look Before You Leap.**

This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the EAFP approach and is characterized by the presence of many if statements. In a multi-threaded environment, the LBYL approach can risk introducing a race condition between “the looking” and “the leaping”.

Consider the following two cases:

```
XXXXX: put example with if statement here
```

```
XXXXX: put the same example with try..except
```

If you expect that 90% of the time your code will just run as expected, use the `try/except` approach. It will be faster if exceptions really are exceptional. If you expect an abnormal condition more than 50% of the time, then using `if` is probably better.

In other words, the method to choose depends on how often you expect the event to occur.

- Use exception handling if the event doesn’t occur very often, that is, if the event is truly exceptional and indicates an error (such as an unexpected end-of-file). When you use exception handling, less code is executed in normal conditions.
- Check for error conditions in code if the event happens routinely and could be considered part of normal execution. When you check for common error conditions, less code is executed because you avoid exceptions.

2.6. When to re-throw an Exception?

Sometimes you just want to do something and rethrow the same Exception. This is easy in Python as shown in the following example.

```
try:
    # do some work here
except SomeException:
    logging.warning("...", exc_info=True)
    raise
```

In some cases, it is best to have the stacktrace printed out with the logging message. I've include the `exc_info=True` in the example.

2.7. What about Performance?

It is nearly free to set up a `try/except` block (an exception manager), while an `if` statement always costs you.

Bear in mind that Python internally uses exceptions frequently. So, when you use an `if` statement to check e.g. for the existence of an attribute (the `hasattr()` method), this builtin function will call `getattr(obj, name)` and catch `AttributeError`. So, instead of doing the following:

```
if hasattr(command, 'name'):
    command_name = getattr(command, 'name')
else:
    command_name = None
```

you can better use the `try/except`.

```
try:
    command_name = getattr(command, 'name')
except AttributeError:
    command_name = None
```

2.8. Can I raise my own Exception?

As a general rule, try to use builtin exceptions from Python, especially `ValueError`, `IndexError`, `NameError`, and `KeyError`. Don't invent your own 'parameter' or 'arguments' exceptions if the cause of the exception is clear from the builtin. The hierarchy of Exceptions can be found in the Python documentation at Builtin-Exceptions > Exception Hierarchy.

When the connection with a builtin exception is not clear however, create your own exception from the `Exception` class.

```
class DeviceNotFoundError(Exception):
    """Raised when a device could not be located or loaded."""
    pass
```

NOTE Even if we are talking about Exceptions all the time, your own Exceptions should end with `Error` instead of `Exception`. The standard Python documentation also has a section on User Defined Exceptions that you might want to read.

In some situations you might want to group many possible sources of internal errors into a single exception with a clear message. For example, you might want to write a library module that throws its own exception to hide the implementation details, i.e. the user of your library shouldn't have to care which extra libraries you use to get the job done.

Since this will hide the original exception, if you throw your own exception, make sure that it contains every bit of information from the originally caught exception. You'll be grateful for that when you read the log files that are send to you for debugging.

The example below is taken from the actual source code. This code catches all kinds of exceptions that can be raised when connecting to a hardware device over a TCP socket. The caller is mainly interested of course if the connection could be established or not, but we always include the information from the original exception with the `raise..from` clause.

```
def connect(self):

    # Sanity checks

    if self.is_connection_open:
        raise PMACException("Socket is already open")
    if self.hostname in (None, ""):
        raise PMACException("ERROR: hostname not initialized")
    if self.port in (None, 0):
        raise PMACException("ERROR: port number not initialized")

    # Create a new socket instance

    try:
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.setblocking(1)
        self.sock.settimeout(3)
    except socket.error as e_socket:
        raise PMACException("ERROR: Failed to create socket.") from e_socket

    # Attempt to establish a connection to the remote host

    try:
        logger.debug(f'Connecting a socket to host "{self.hostname}" using port
{self.port}')
        self.sock.connect((self.hostname, self.port))
    except ConnectionRefusedError as e_cr:
        raise PMACException(f"ERROR: Connection refused to {self.hostname}") from
e_cr
    except socket.gaierror as e_gai:
        raise PMACException(f"ERROR: socket address info error for {self.hostname}")
from e_gai
    except socket.herror as e_h:
        raise PMACException(f"ERROR: socket host address error for {self.hostname}")
from e_h
    except socket.timeout as e_timeout:
        raise PMACException(f"ERROR: socket timeout error for {self.hostname}") from
e_timeout
    except OSError as e_ose:
        raise PMACException(f"ERROR: OSError caught ({e_ose}).") from e_ose

    self.is_connection_open = True
```

2.9. When should I use Assertions?

Use assertions only to check for invariants. Assertions are meant for development and should not replace checking conditions or catching exceptions which are meant for production. A good guideline to use `assert` statements is when they are triggering a bug in your code. When your code assumes something and acts upon the assumption, it's recommended to protect this assumption with an assert. This assert failing means your assumption isn't correct, which means your code isn't correct.

```
def _load_register_map(self):
    # This method shall only be called when self._name is 'N-FEE' or 'F-FEE'.
    assert self._name == 'N-FEE' or self._name == 'F-FEE'
```

Another example is:

```
# If this assertion fails, there is a flaw in the algorithm above
assert tot_n_args == n_args + n_kwargs, (
    f"Total number of arguments ({tot_n_args}) doesn't match "
    f"# args ({n_args}) + # kwargs ({n_kwargs})"
)
```

Remember also that running Python with the `-O` option will remove or disable assertions. Therefore, *never* put expressions from your normal code flow in an assertion. They will not be executed when the optimizer is used and your code will break gracefully.

2.10. What are Errors?

This is a naming convention thing… names of user defined sub-classes of Exception should end with `Error`.

2.11. Cascading Exceptions

TBW

2.12. Logging Exceptions

Generally, you should not log exceptions at lower levels, but instead throw exceptions and rely on some top level code to do the logging. Otherwise, you'll end up with the same exception logged multiple times at different layers in your application.

- https://docs.sentry.io/error-reporting/quickstart/?platform=python

2.13. Resources

Some of the explanations were taken shamelessly from the following resources:

- https://docs.microsoft.com/en-us/dotnet/standard/exceptions/best-practices-for-exceptions
- https://stackoverflow.com/questions/1835756/using-try-vs-if-in-python

- https://stackoverflow.com/questions/24752395/python-raise-from-usage
- https://realpython.com/the-most-diabolical-python-antipattern/

Chapter 3. Writing docstrings

This might need to go into the style guide or we leave it here as a stand-alone section.

- Explain which format of docstring we have chosen → Google
- What needs to go into the docstring and what not
- where will the docstring show up → in PyCharm help, in the REPL, pdoc3

Chapter 4. Setting up ssh access and GitHub deploy keys

4.1. plato-common-egse

- ssh-keygen -t ed25519 -C "<your email address>"
  - name of the file: `id_cgse_egse_server_<TH>`
- copy the content of `id_cgse_egse_server_<TH>.pub` into the deploy keys @ GitHub **name is plato-data@egse-server (id_cgse_egse_server_<TH>)
- add the following lines to ~/.ssh/config

```
Host repo-common-egse
    Hostname github.com
    IdentityFile ~/.ssh/id_cgse_egse_server_<TH>
```

- make sure the files in `~/.ssh` have the right permissions

```
chmod 644 ~/.ssh/config
```

- add a remote for doing the updates

```
git remote add updates git@repo-common-egse:IvS-KULeuven/plato-common-egse.git
```

- then perform an update:
  - if this is the first-time-install:

    ```
    git fetch updates
    git checkout tags/2022.2-<TH>-CGSE -b 2022.2-<TH>-CGSE-branch
    python -m pip uninstall Common-EGSE
    python setup.py install --force --home=/cgse/
    ```

  - for existing installations, a new command has been implemented that automates the above update procedure:

    ```
    update_cgse ops --tag=2022.2-<TH>-CGSE
    ```

WARNING

You might experience the following problem starting the core egse services on the egse-server with Systemd:

```
systemctl restart log_cs sm_cs cm_cs pm_cs
```

- the problem was reported as (code=exited, status=200/CHDIR)
- the service files contain a WorkingDirectory pointing to `~/workdir` and that folder did not exist on the machine.

4.2. plato-test-scripts

- ssh-keygen -t ed25519 -C ["rik.huygen@kuleuven.be](mailto:%22rik.huygen@kuleuven.be)"
  - name of the file: `id_ts_egseclient2_<TH>`
- copy the content of `id_ts_egseclient2_<TH>.pub` into the deploy keys @ GitHub
  - name is plato-data@egse-client (id_ts_egse_client_<TH>)
- add the following lines to ~/.ssh/config

```
Host repo-test-scripts
    Hostname github.com
    IdentityFile ~/.ssh/id_ts_egseclient2_csl
```

- make sure the files in ~/.ssh have the right permissions

```
chmod 644 ~/.ssh/config
```

- add a remote for doing the updates

```
git remote add updates [git@repo-test-scripts:IvS-KULeuven/plato-test-
scripts.git](mailto:git@repo-common-egse:IvS-KULeuven/plato-common-egse.git)
```

- if you had not yet installed the repo, clone it

```
git clone **[git**@repo-test-scripts:IvS-KULeuven/plato-test-
scripts.**git**](mailto:git@repo-test-scripts:IvS-KULeuven/plato-test-
scripts.git)
```

- on a first-time-installation, perform an update as follows:

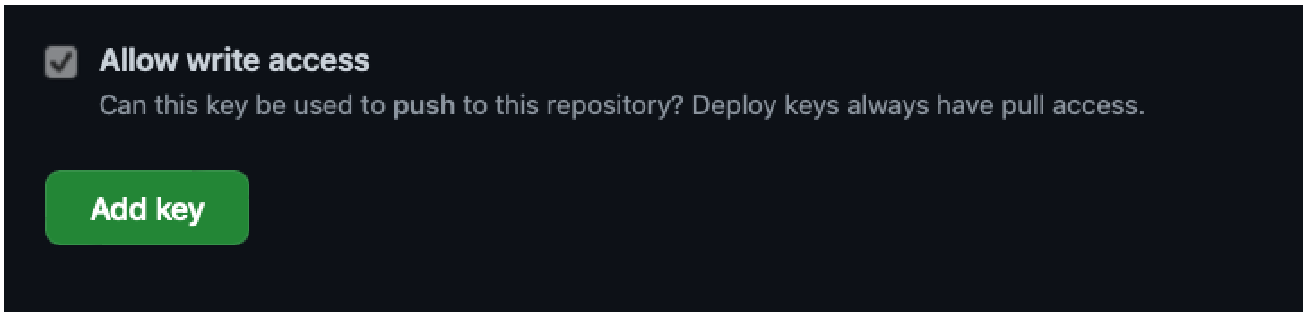
```
git fetch updates
git rebase updates/develop
python -m pip install -e
```

- otherwise:

```
update_ts
```

4.3. plato-cgse-conf

Basically the same procedure as for the previous two repos, except for plato-cgse-conf using the configuration manager: when you add the deploy key to GitHub, you must check the *Allow write access* checkbox. That will allow the configuration manager to upload new Setups to the repo.


Allow write access
Can this key be used to push to this repository? Deploy keys always have pull access.
Add key

For the configuration manager, we will also name the remote ‘upload’.

```
git remote add upload git@repo-cgse-conf:IvS-KULeuven/plato-cgse-conf.git
```

The following environment variable is needed in `/cgse/env.txt`:

```
export PLATO_CONF_REPO_LOCATION=/home/plato-data/git/plato-cgse-conf
```

Make sure that the branch has the upload/main as its tracking branch:

```
$ git branch -u upload/main
$ git branch -vv
* main 17fb23c [upload/main] change filter wheels parameters
```

Part II — Core Concepts

Chapter 5. Core Services

What are considered core services?

The core services are provided by four control servers that run on the egse-server machine: the storage manager, the configuration manager, the process manager, and the log manager. The services are described in full detail In the following sections.

5.1. The Configuration Manager

The configuration manager is a core service…

- The GlobalState
- The Setup
- The Observation concept

5.2. The Storage Manager

The storage manager is responsible for storing the following data:

- all housekeeping from the devices that are connected by a control server
- all housekeeping and CCD data that is transferred from the N-FEE and the F-FEE during camera tests.
- Origin = defined in the ICD, include a list of predefined origin strings

5.2.1. File Naming

5.2.2. File Formats

5.3. The Process Manager

The job of the process manager is to keep track of running processes during the Camera Tests.

There are known processes that should be running all the time, i.e. the configuration manager, and the Storage Manager. Then, there are device control servers that are dependent on the Site and the Setup at that site. The process manager needs to know which device control servers are available and how to contact them. That information is available in the configuration manager. We need to decide how the interface between the PM and the CM looks like and what information is exchanged in which format.

5.4. The Synoptics Manager

Synoptics = in a form of a summary or synopsis; taking or involving a comprehensive mental view. According to the Oxford Dictionary.

- what is this?
- Which parameters are Synoptic?
- From the device to the Grafana screen, what is the data flow, where are the name changes, where are the calibrations….

5.5. The Telemetry (TM) Dictionary

The `tm-dicionary.csv` file (further referred to as the "telemetry ™ dictionary") provides an overview of all housekeeping (HK) and metrics parameters in the EGSE system. It is used:

- By the `get_housekeeping` function (in `egse.hk`) to know in which file the values of the requested HK parameter should be looked for;
- To create a translation table to convert — in the `get_housekeeping` function of the device protocols — the original names from the device itself to the EGSE-conform name (see further);
- For the HK that should be included in the synoptics: to create a translation table to convert the original device-specific (but EGSE-conform) names to the corresponding synoptical name in the Synoptics Manager (in `egse.synoptics`).

5.5.1. The File's Content

For each device we need to add all HK parameters to the TM dictionary. For each of these parameters you need to add one line with the following information (in the designated columns):

Column name	Expected content
TM source	Arbitrary (but clear) name for the device. Ideally this name is short but clear enough for outsiders to understand what the device/process is for.
Storage mnemonic	Storage mnemonic of the device. This will show up in the filename of the device HK file and can be found in the settings file (`settings.yaml`) in the block for that specific device/process.
CAM EGSE mnemonic	EGSE-conform parameter name (see next Sect.) for the parameter. Note that the same name should be used for the HK parameter and the corresponding metrics.
Original name in EGSE	In the `get_housekeeping` method of the device protocols, it is - in some cases (e.g. for the N-FEE HK) - possible that you have a dictionary with all/most of the required HK parameters, but with a non-EGSE-conform name. The latter should go in this column.
Name of corresponding timestamp	In the device HK files, one of the columns holds the timestamp for the considered HK parameter. The name of that timestamp column should go in this column of the TM dictionary.

Column name	Expected content
Origin of synoptics at CSL	Should only be filled for the entries in the TM dictionary for the Synoptics Manager. This is the original EGSE-conform name of the synoptical parameter in the CSL-specific HK file comprising this HK parameter. Leave empty for all other devices!
Origin of synoptics at SRON	Should only be filled for the entries in the TM dictionary for the Synoptics Manager. This is the original EGSE-conform name of the synoptical parameter in the SRON-specific HK file comprising this HK parameter. Leave empty for all other devices!
Origin of synoptics at IAS	Should only be filled for the entries in the TM dictionary for the Synoptics Manager. This is the original EGSE-conform name of the synoptical parameter in the IAS-specific HK file comprising this HK parameter. Leave empty for all other devices!
Origin of synoptics at INTA	Should only be filled for the entries in the TM dictionary for the Synoptics Manager. This is the original EGSE-conform name of the synoptical parameter in the INTA-specific HK file comprising this HK parameter. Leave empty for all other devices!
Description	Short description of what the parameter represents.
MON screen	Name of the Grafana dashboard in which the parameter can be inspected.
unit cal1	Unit in which the parameter is expressed. Try to be consistent in the use of the names (e.g. Volts, Ampère, Seconds, Degrees, DegCelsius, etc.).
offset b cal1	For raw parameters that can be calibrated with a linear relationship, this column holds the offset `b` in the relation `calibrated = a * raw + b`.
slope a cal1	For raw parameters that can be calibrated with a linear relationship, this column holds the slope `a` in the relation `calibrated = a * raw + b`.
calibration function	Not used at the moment. Can be left emtpy.
MAX nonops	Maximum non-operational value. Should be expressed in the same unit as the parameter itself.
MIN nonops	Minimum non-operational value. Should be expressed in the same unit as the parameter itself.
MAX ops	Maximum operational value. Should be expressed in the same unit as the parameter itself.
MIN ops	Minimum operational value. Should be expressed in the same unit as the parameter itself.
Comment	Any additional comment about the parameter that is interesting enough to be mentioned but not interesting enough for it to be included in the description of the parameter.

Since the TM dictionary grows longer and longer, the included devices/processes are ordered as follows (so it is easier to find back the telemetry parameters that apply to your TH):

- Devices/processes that all test houses have in common: AEU, N-FEE, TCS, Synoptics Manager, etc.
- Devices that are CSL-specific;
- Devices that are SRON-specific;
- Devices that are IAS-specific;
- Devices that are INTA-specific.

5.5.2. EGSE-Conform Parameter Names

The correct (i.e. EGSE-conform) naming of the telemetry should be taken care of in the `get_housekeeping` method of the device protocols.

Common Parameters

A limited set of devices/processes is shared by (almost) all test houses. Their telemetry should have the following prefix:

Device/process	**Prefix**
Configuration Manager	CM_
AEU (Ancillary Electrical Unit)	GAEU_
N-FEE (Normal Front-End Electronics)	NFEE_
TCS (Thermal Control System)	GTCS_
FOV (source position)	FOV_
Synoptics Manager	GSYN_

TH-Specific Parameters

Some devices are used in only one or two test houses. Their telemetry should have TH-specific prefix:

TH	**Prefix**
CSL	GCSL_
SRON	GSRON_
IAS	GIAS_
INTA	GINTA_

5.5.3. Synoptics

The Synoptics Manager groups a pre-defined set of HK values in a single file. It’s not the original

EGSE-conform names that are use in the synoptics, but names with the prefix `GSYN_`. The following information is comprised in the synoptics:

- Acquired by common devices/processes:
- Calibrated temperatures from the N-FEE;
- Calibrated temperatures from the TCS;
- Source position (commanded + actual).
- Acquired by TH-specific devices:
- Calibrated temperatures from the TH DAQs;
- Information about the OGSE (intensity, lamp and laser status, shutter status, measured power).

For the first type of telemetry parameters, their original EGSE-conform name should be put into the column `CAM EGSE mnemonic`, as they are not TH-specific.

The second type of telemetry parameters is measured with TH-specific devices. The original TH-specific EGSE-conform name should go in the column `Origin of synoptics at ...`.

5.5.4. Translation Tables

The translation tables that were mentioned in the introduction, can be created by the `read_conversion_dict` function in `egse.hk`. It takes the following input parameters:

- `storage_mnemonic`: Storage mnemonic of the device/process generating the HK;
- `use_site`: Boolean indicating whether you want the translation table for the TH-specific telemetry rather than the common telemetry (`False` by default).

To apply the actual translation, you can use the `convert_hk_names` function from `egse.hk`, which takes the following input parameters:

- `original_hk`: HK dictionary with the original names;
- `conversion_dict`: Conversion table you got as output from the `read_conversion_dict` function.

5.5.5. Sending HK to Synoptics

When you want to include HK of your devices, you need to take the following actions:

- Make sure that the TM dictionary is complete (as described above);
- In the device protocol:
  - At initialisation: establish a connection with the Synoptics Manager: `self.synoptics = SynopticsManagerProxy()`
  - In `get_housekeeping` (both take the dictionary with HK as input):
    - For TH-specific HK: `self.synoptics.store_th_synoptics(hk_for_synoptics)`;

  - For common HK: `self.synoptics.store_common_synoptics(hk_for_synoptics)`.

Please, do not introduce new synoptics without further discussion!

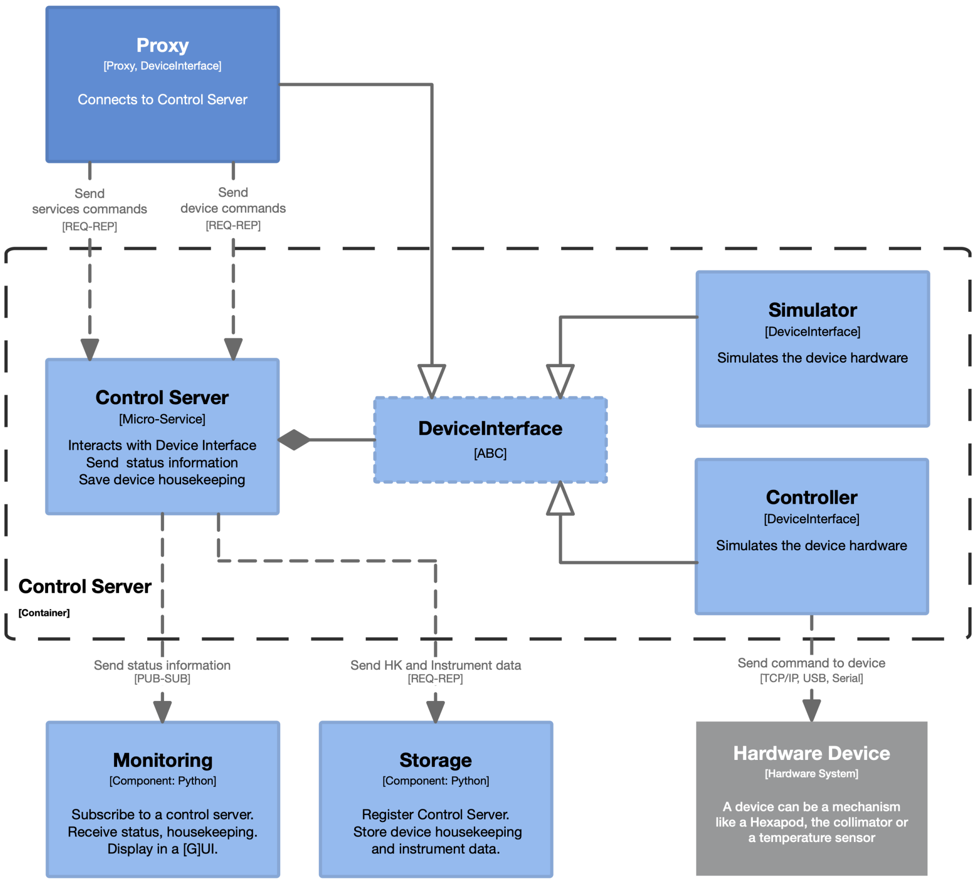
5.6. The Log Manager

TBW

- local logging using the standard logging module
- log messages sent to the log manager log_cs via ZeroMQ
- why am I not seeing some log messages in my terminal or Python console? → only level INFO and higher
- should I configure the logger, how?
- Cutelog and Textualog → user manual?

Chapter 6. The Commanding Concept

- Proxy – Protocol – Controller
- Control Servers
- YAML command files
- Dynamic commanding


Proxy
[Proxy, DeviceInterface]
Connects to Control Server
Send services commands [REQ-REP]
Send device commands [REQ-REP]
Simulator
[DeviceInterface]
Simulates the device hardware
Control Server
[Micro-Service]
Interacts with Device Interface
Send status information
Save device housekeeping
DeviceInterface
[ABC]
Controller
[DeviceInterface]
Simulates the device hardware
Control Server
[Container]
Send status information [PUB-SUB]
Send HK and Instrument data [REQ-REP]
Send command to device [TCP/IP, USB, Serial]
Monitoring
[Component: Python]
Subscribe to a control server.
Receive status, housekeeping.
Display in a [G]UI.
Storage
[Component: Python]
Register Control Server.
Store device housekeeping and instrument data.
Hardware Device
[Hardware System]
A device can be a mechanism like a Hexapod, the collimator or a temperature sensor

6.1. The Control Server

TODO:

- Explain what the CS does during startup and what happens when this fails.
- description of the `control.py` module

6.2. The Proxy Class

The Proxy class automatically tries to connect to its control server. When the control server is up and running, the Proxy loads the device commands from the control server. When the control server is not running, a warning is issued, and the device commands are not loaded. If you then try to call one of the device methods, you will get a `NotImplementedError`. The ZeroMQ connection initiated by the Proxy will just sit there until the control server comes up, so you can fix the problem with the control server and call the `load_commands()` method on the Proxy again to load the commands.

6.3. The Protocol Class

The DPU Protocol starts a thread called *DPU Processor* Thread. This thread is a loop that continuously reads packets from the FEE and writes RMAP commands when there is a command put on the queue. When the control server receives a quit command, this thread also needs to be terminated gracefully since it has a connection and registration with the Storage manager. Therefore, the Protocol class has a `quit()` method which is called by the control server when it is terminated. The DPU Protocol class implements this `quit()` method to notify the DPU Processor Thread. Other Protocol implementations usually don't implement the `quit()` method.

Another example of a control server that starts a sub-process through its Protocol class in the TCS control server. The control server start the TCS Telemetry process to handle reading housekeeping from the TCS EGSE using a separate port [6667 by default].

6.4. Response, Failure, Success

The `control.py` module defines a number of classes that can be used to send a response on a device command to the client. Responses are handle by the `Protocol` class in the `handle_device_method()` method.

The main reason why these classes are provided is because Exceptions that happen in the server process should somehow propagate to the client process. A `Response` object can contain a return value if the command was executed successfully, or it can contain an Exception object when the device command has failed. If the command has failed or not can be checked by the client with the `successful()` method.

The `Response` class has three sub-classes, `Success`, `Failure`, and `Message`. A `Success` object is returned by the server process with the actual response in the property `return_code`. In the case of an Exception, the server process logs the error message and packs the Exception object in a `Failure` object.

XXXXX: insert a class diagram of Response and it's three sub-classes here

Chapter 7. The Setup

TODO:

- What is the difference between a Setup and the Settings?
  - Settings are for constants, IP addresses, system definitions, …
  - Setup contains configuration items, conversion coefficients and tables, identification of devices, everything that defines the configuration of the system and that can change from test to test and has influence on the data, HK, …

The content of this section needs to be split between the developer manual and the user manual. Usage shall go into the User Manual, the Developer Manual shall explain how the Setup is coded and how it is used in code. What about the `GlobalState.setup`?

The *Setup* is the name we give to the set of configuration items that make up your test environment. A configuration item is e.g. a device like a temperature controller, a mechanism like a Hexapod, a camera (which is the SUT: System Under Test). The *Setup* groups all these items in one *configuration*.[1]

This section will explain what the Setup contains, when and where it is used, how the different Setups are managed and how you can create a new Setup.

7.1. What is the Setup?

Explain the Setup is a `NavigableDict` and what that means to the usage etc. See also further in this section, it's partly explained. This is explained further in …

The *Setup* is defined as a hierarchy of configuration items and their specific settings and definitions. All the hardware and software that make up your complete test setup will be described in the YAML file. As a quick example, the lines below define a typical setup for the Hexapod PUNA that is used at CSL:

```
Setup:
    gse:
        hexapod:
            device_name: Symetrie Puna Hexapod
            device: class//egse.hexapod.symetrie.puna.PunaProxy
            ID: 172543
            firmware: 3.14
            time_request_granularity: 0.1
```

The *Setup* is implemented as YAML configuration file and loaded into a special Python dictionary. The dictionary is special because it allows you to navigate with a 'dot' notation in addition to keys. The following two lines are equivalent (assuming the Setup is loaded in the variable `setup`):

```
>>> setup.gse.hexapod.ID
172543

>>> setup["gse"]["hexapod"]["ID"]
172543
```

Another advantage of this special dictionary is that some fields are interpreted and loaded for you. For example, the device field of the Hexapod starts with `class//` and provides the class name for the Hexapod device. When you access this field, the class will automatically be instantiate for you and you can start commanding or querying the device. The following example initiates a homing command on the Hexapod controller:

```
>>> setup.gse.hexapod.device.homing()
```

When you want to know which configuration items are defined in the Setup at e.g. the `gse` level, use the `keys()` function:

```
>>> setup.gse.keys()
dict_keys(['hexapod', 'stages', 'thorlabs', 'shutter', 'ogse', 'filter_wheel'])
```

When you want a full printout of the `setup.gse`, use the `print` function. This will print out the dictionary structure as loaded from the YAML file.

```
>>> print(setup.gse)
```

7.2. What goes into the Setup?

The *Setup* will contain configuration information of your test equipment. This includes calibration and conversion information, identification, alignment information, etc. All items that can change from one setup to the next should be fully described in the *Setup*.

The following list gives an idea of what is contained and managed by a Setup. This list is not comprehensive.

- site: identification of the Site, e.g. CSL, IAS, INTA, SRON.
- position: if you have different test setups, each of them should be identifiable, e.g. at CSL there are four positions with different setups.
- gse: the ground support equipment like mechanisms, temperature controllers and sensors/heaters, optical equipment, shutter, hexapod, power meters, SpaceWire interface, TEB, etc.

- camera: this is the System Under Test (SUT) and it has different sub-items like FEE, TOU, DPU software, model and type, all with their specific characteristics.

For all of these items the Setup shall hold enough information to uniquely identify the configuration item, but also to reproduce the state of that item, i.e. version numbers of firmware software, transformation matrices for alignment, conversion coefficients, calibration tables, etc.

7.3. How to use the Setup

As described above, the Setup is a special dictionary that contains all the configuration information of your test equipment. This information resides in several files maintained by the configuration control server. You can request a list of Setups that are available from the configuration manager with the `list_setups()` function. This is a convenience function that will print the list in your Python session.

```
>>> list_setups()
```

The above command will contact the configuration control server and request the list of Setups. To load the current active Setup into your session, use the `load_setup()` method. This function takes one optional argument to load a specific Setup. Now you can work with this setup as explained above, accessing its content and navigate through the hierarchy with the 'dot' notation.

If you need to make a change to one of the configuration items in the Setup, you can just assign a new value to that field. Suppose we have upgraded the firmware of the PUNA Hexapod that is used in this setup, we can use the following commands to make that change:

```
>>> setup.gse.hexapod.Firmware = 3.1415
```

The change then needs to be submitted to the configuration control server who will put it under configuration control, i.e. assign a new unique Setup identifier and save the entier Setup into a new YAML file.

```
>>> setup = submit_setup(setup, description="Updated firmware of PUNA Hexapod")
```

7.4. How to create and manage Setups

Whenever you make a change to an item in the Setup, or you add a configuration item, a new Setup will be created with a new unique `id` as soon as you submit the Setup to the configuration manager.

The configuration control server has no insight or knowledge about the content of a Setup, so you can freely add new items when needed. The simplest way to start with a Setup and adapt it to your current test equipment environment, is to load a Setup that closely resembles your current environment, and only make the changes necessary for the new Setup, then submit the new Setup to the configuration control server.

If you need to start from scratch, create a new empty Setup or feed it with a dictionary that contains already some of the information for the Setup:

```
>>> from egse.setup import Setup
>>> setup = Setup({"gse": {"hexapod": {"ID": 42, "name": "PUNA"}}})
>>> print(setup)
gse:
    hexapod:
        ID: 42
        name: PUNA
```

If you need to set the firmware version for the Hexapod controller.

```
>>> setup.gse.hexapod.firmware = "2020.07"
>>> print(setup)
gse:
    hexapod:
        ID: 42
        name: PUNA
        firmware: 2020.07
```

This way it is easy to update and maintain your Setup. When ready, submit to the configuration control server as shown above.

If you want to save your Setup temporarily on your local system, use the `to_yaml_file()` method of Setup. This will save the Setup in your working directory.

```
>>> setup.to_yaml_file(filename="SETUP-42-FIXED.yaml")
```

⚠ Explain here how the user should submit a Setup from the client machin. That will send the Setup to the configuration manager and automatically push the new Setup to the GitHub repository provided the proper permissions are in place, i.e. a deploy key with write access. Where shall this be described?

[1] Both *Setup* and *configuration* are overloaded words, if not clear from the context, I'll try to explain them when used.

Chapter 8. The GlobalState

- Why would we want a global state?
- They always tell me that we should not use global variables!
- What is in the `GlobalState`?
- What can we do with the `GlobalState`?

8.1. Singleton versus Shared State

From within several places in the code, we needed access to a certain state and act accordingly. One of these states is the `dry_run` which allows us to execute command sequences (test scripts) without actually sending instructions to the hardware, but just logging that the command would have been executed with its arguments. We could have gone with a singleton pattern, i.e. a class for which only one instance exists in your session, but a singleton is a difficult pattern to test and control. Another possible solution is to use a class for which all instances have a shared state. That means even if the class is instantiated multiple times, its state is the same for all those instances. This pattern is also known as the Borg pattern or the shared-state pattern.

The name `GlobalState` is maybe not such a good name as this class actually shares state between its instances, but this shared state is not global in the sense of a global variable. The objects can be instantiated from anywhere at anytime, which is what makes them globally available.

8.2. What is in the GlobalState?

8.2.1. The Setup

The complete configuration of the test setup. From the Setup you can access all devices that are known by the configuration manager, and you have access to calibration settings. The Setup is fully described in the API documentation of the class at `egse.setup`. The Setup that comes with the GlobalState is loaded from the Configuration Manager. Use the `GlobalState.load_setup()` to load the current Setup that is active on the Configuration Manager.

If you need to work with different Setups, `GlobalState` is not the right place to be. Load any Setup directly from the `Setup` class with the static methods `from_dict(my_dict)` or `from_yaml_file(filename)`.

8.2.2. Performing a Dry Run

At some point we need to check the test scripts that we write in order to see if the proper commands will be executed with their proper arguments. But we don’t want the commands to be send to the mechanisms or controllers. We want to do a dry run where the script is executed as normal, but no instructions are sent to any device.

8.2.3. Retrieve the Command Sequence

Whenever a building block is executed, a command sequence is generated and stored in the `GlobalState`. There are two functions that access this command sequence: (1) the `execute()` function will copy the command sequence into the test dataset, and (2) the `generate_command_sequence()` will return the command sequence as a list (TODO: this will probably get its own class eventually).

Part III — Device Commanding

Chapter 9. Device Control Servers

- Mechanisms
- Sensors
- Heaters
- Facility

9.1. The Device Interface Classes

- Naming convention
- SCPI
- Where are device drivers configured?
- Direct communication with devices
- Device simulators
- Device Interfaces
- Description of the `device.py` module

9.1.1. The Connection Interface

A **connection interface** defines the commands that are used to establish a connection, terminate a connection, and check if a connection has been established. We have two main connection interfaces: (1) a connection to a hardware device, e.g. a temperature controller or a filter wheel, and (2) a connection to a control server. A control server is the single point access to a hardware device.

Connection Interface for Devices

A *Controller* is a class that connects directly to the hardware. This can be through e.g. an Ethernet TCP socket or a USB interface or any other connection. The details of a connection are buried in low level device interface classes which use all kinds of different protocols. The Controller will use these low level device classes to control the connection, and then provide a uniform interface to connect and disconnect from the device to the user.

So, we defined a generic interface for working with device connections. The interface defines the following commands: `connect`, `disconnect`, and `is_connected`.

Use the `connect()` method to establish a connection to the hardware controller of the device.

Use the `disconnect()` method to terminate the connection to the hardware controller.

The previous two commands raise a device specific error (Exception) when a connection can not be established or terminated.

Use the `is_connected()` method to check if a connection with the controller is established. This command returns `True` if there is a working connection with the device, `False` otherwise.

This interface shall be implemented by device controller classes, device simulators, and `Proxy` sub-classes. Examples of device controllers are `PunaController` and `ThorlabsPM100Controller`, while their simulators are called `PunaSimulator` and `ThorlabsPM100Simulator` and the `Proxy` sub-classes `PunaProxy` and `ThorlabsPM100Proxy`.

In the example below we make a connection to the PUNA Hexapod and issue a homing and an absolute movement command, then close the connection.

```
from egse.hexapod import HexapodError
from egse.hexapod.symetrie.puna import PunaController

puna = PunaController()
try:
    puna.connect()
    puna.homing()
    puna.move_absolute(0, 0, 18, 0, 0, 0)
except HexapodError as exc:
    logger.error(f"There was an error during the interaction with the PUNA Hexapod:
{exc}")
    # do recovery action here if needed
finally:
    if puna.is_connected():
        puna.disconnect()
```

Most of the Controllers can also be used as a context manager. The following code does the same as the example above. The connection is automatically established and terminated by the context manager.

```
from egse.hexapod.symetrie.puna import PunaController

with PunaController() as puna:
    puna.homing()
    puna.move_absolute(0, 0, 18, 0, 0, 0)
```

Connection Interface for `Proxy` Classes

`Proxy` classes make a connection to a control server, e.g. the `PunaProxy` class will make a connection to the `PunaControlServer`. The control server will in turn be connected to either a Controller or a Simulator. The `Proxy` classes are called this way, because they act as a gateway for device commands, i.e. the proxy forwards device commands to the Controllers through the control server.

As stated above, a `Proxy` sub-class implements the device connection interface with the `connect()`, disconnect(), and is_connected() methods. That is because a Proxy completely mimics a Controller device interface. The Proxy however also establishes and manages a connection with its control server, but we cannot use the same connection interface for this type of connection.

The Proxy connection to its control server is defined by the ControlServerConnectionInterface and is implemented in the Proxy base class. This interface defines the following commands:

- connect_cs,
- disconnect_cs,
- reconnect_cs,
- reset_cs_connection,
- and is_cs_connected.

Use the connect_cs() and disconnect_cs() methods to establish and terminate a connection to the control server. The reconnect_cs() method currently basically does the same as the connect_cs() method, it is provided as a convenience to make the flow of connecting and reconnecting clearer in your code when needed.

Use the reset_cs_connection() method when the connection is in an undefined state. This method will try to connect and reconnect to the control server for a number of retries before failing.

Use the is_cs_connected() method to check if a connection with the control server is established. This command returns True if there is a working connection with the server, False otherwise.

This interface is implemented in the Proxy base class and there is no need to worry about this in your Proxy sub-class implementation.

Also a Proxy can be used as a context manager. The example for the PunaController above can therefore be rewritten for the PunaProxy:

```
from egse.hexapod.symetrie.puna import PunaProxy

with PunaProxy() as puna:
    puna.homing()
    puna.move_absolute(0, 0, 18, 0, 0, 0)
```

Note however that the Context Manager in this case will connect and disconnect to the control server, leaving the connection to the hardware device untouched.

Chapter 10. The System Under Test (SUT) – DPU Processor and FEE Simulator

- Commanding is the same as all other devices, DPU Control Server and Controller, Proxy, etc
- DPU Processor as a separate thread to communicate to the FEE (Simulator)
- DPU Protocol starts the DPU Processor process
- DPU Controller and DPU Processor communicate via three Queues, the command queue, the response queue, and a priority queue.
- Timing: timecode packet, HK packet, [Data packets], Commanding through RMAP requests

Commands that are given on the Python prompt are by definition synchronous meaning when you call a function or execute a building block, the function or building block will wait for its return value before finishing. This is usually not a problem, because we most of the time have a pretty good idea how long a calculation or action will take. Most functions return within a few hundred milliseconds or less, which is practically immediate. When commanding the FEE however, things are more complicated. The FEE has strict timing when it comes to commanding. During the sync period (usually 6.25s unless commanded different) there is a slot of at least 2s at the end of the period, which is reserved for commanding. The sync period starts with a timecode packet followed by the FEE housekeeping packet. That takes just a few milliseconds. Depending on the FEE mode, we can then expect data packets filling up until 4s after the time code. There can be less or no data packets, leaving more time for commanding. Commanding the FEE is done by sending SpaceWire RMAP requests to the FEE in that 2s+ timeslot. When we send a command from the Python prompt to the FEE, it arrives in the Queue at the DPU Processor and will be send to the FEE in the next 2s+ command timeslot. When the Command enters the Queue at the beginning of the readout period, i.e. right after the timecode, maximum 4s will pass before the command is actually send to and executed on the FEE. All this time, the Python prompt will be blocked while waiting for the response. The next command can only be sent on return of the previous. So, we have two issues to solve, (1) the duration and blocking of all the steps in the commanding chain, and (2) sending more than just one command to the FEE in the same timeslot.

Part IV — Unit Testing and beyond

Chapter 11. The Test Suite

- pytest

Chapter 12. Code Reviews

This section is not finished and needs further updates. Please send me any comments and suggestions for improvement. Thanks, Rik.

This is a proposal for a software source code review for the Common-EGSE. Code review is in principle a continuing process. With every pull request in GitHub there should be one of your colleagues doing a quick review of your changes before they are accepted and merged. We will add a section in Contributing to the Code > Pull Requests explaining how to review the code changes from a pull request.

This section handles a slightly more formal code review that is required at certain milestones in the development process.

We will have a code review at TBD week.

The code review is done primarily for the following reasons:

- Share knowledge: you write code for yourself, your colleagues, test operators, scientists and engineers. Each of them should have a certain understanding of the system, but not all at the same detail.
- Maintainability: code should be understandable and at least two developers should know what the code does and why. These two developers are you and one of your colleagues.
- Finding bugs and design flaws.
- Consistent error and exception handling.
- Consistent logging.
- Finding functionality creep.
- Proper testing: test coverage, functionality testing.
- Development Principles: SOLID, DRY.
- Meet coding standards.

Please remember that the purpose of the code review is **not to reject** the code, but **to improve** the code quality. Focus is on how easy it is to understand the code.

12.1. Who is part of this Code Review?

Developers: Rik Huygen, Sara Regibo, …

Instrument Experts: Pierre Royer, Bart Vandenbussche

Do we need more reviewers? The EGSE engineers from INTA and SRON? Somebody from the PCOT?

12.2. Planning

For each reviewer I will prepare an issue where you can check off the parts which have been reviewed. WHERE TO PUT THE REVIEW REPORTS/COMMENTS?

It is very important that the review is done in a timely fashion. We don’t want to be bothered with this for weeks.

Proposal for reviewer/review items:

Sara Regibo:

- [ ] Commanding Concept

Nicolas Beraud:

- [ ] Hexapod package
- [ ] Stages package

Rik Huygen:

- [ ] Image Viewer
- [ ] Powermeter
- [ ] Shutter

Pierre Royer:

- [ ] GlobalState
- [ ] Setup

Bart Vandenbussche:

- [ ] Image Viewer functionality

12.3. What needs to be reviewed?

TODO: Make a checklist!

- **documentation**:
  - API documentation at GitHub.io
- **docstrings**: Do you understand from the docstring of the functions and public methods what the functionality is, what is needed as input and what is returned, if we need to catch exceptions?
- **coding style**: Do I understand what the code does, is the control flow not too complicated, …
- **whatch-outs**:

    - mutable default parameters

Be constructive!
Be specific!

The goal is to ship good and maintainable code, it's not the goal to prove how good or clever we are.

12.4. Prerequisites

Before the code review, all the code will be run through a number of automated steps:

- check for trailing white space;
- check for end of file blank lines;
- check format of the YAML files;
- run the code through black to make sure we have a consistent formatting;
- run the code through flake8 to make sure the style guide is being followed;
- run all the test harnesses, preferably with hardware attached, but also with the simulators. Have the test coverage report ready.

There will be a specific release for the code review with tag `code-review-2020-Q2`.

Part V — TODO

Everything in this part of the manual needs to be worked on and relocated in one of the above parts/sections.

Chapter 13. Terminal Commands

A lot of testing and monitoring can be done from the command terminal and doesn't need a specific GUI or PyCharm. This section will explain how to check status of most of the control servers, how to inspect log files and HDF5 files, how to generate reports and how you can update the Common-EGSE and the test scripts.

13.1. What goes into this section

- See cheatsheet CGSE → included below
- describe all _cs commands and their options, when to use them and when NOT to use them
- describe all _ui commands and their options
- what other terminal commands are used
  - ps -ef|grep _cs
  - kill PID or %1
  - python -m egse (and all the other info modules)
  - python ../src/scripts/check_hdf5_files.py
  - python ../src/scripts/create_hdf5_report.py
  - export_grafana_dashboards
  - update_cgse → should become `cgse update`
  - update_ts → should become `ts update`
- often used git commands
  - git status
  - git describe —tags
  - git stash [pop]
  - git fetch update
  - git rebase update/develop
- Textualog, see Section 5.6

CGSE *Common-EGSE*

CHECKING

Python version — 3.8+

Python virtual environment	`venv`
Environment variables	

PATH	`/cgse/bin:$PATH`
PYTHONPATH	`/cgse/lib/python`
PLATO_LOCAL_SETTINGS	`/cgse/local_settings.yaml`
PLATO_DATA_STORAGE_LOCATION	`/data/{SITE}`
PLATO_CONF_DATA_LOCATION	`/data/{SITE}/conf`
PLATO_LOG_FILE_LOCATION	`/data/{SITE}/log`

CONFIGURATION

Check the settings	`python -m egse.settings [--local]`
SITE ID	`edit $PLATO_LOCAL_SETTINGS`
List the available Setups	`cm_cs list-setups`
OS info	`python -m egse.system`

SETUP

Check the current Setup	`python -m egse.setup [--use-cm]`
List the available Setups	`cm_cs list-setups`
Load a Setup	`cm_cs load-setup <setup_id>`

UPDATE

Use the following commands from the Project folder

DEVELOPMENT MODE

Pull changes from upstream	`git pull upstream develop`
Install	`python setup.py clean --all`
	`python setup.py develop`

OPERATIONAL MODE

(e.g. release-tag: 2024.3)

Pull changes from upstream

```
git fetch upstream
git checkout tags/{release_tag} -b {release_tag}-branch
```

Install

```
python setup.py clean --all
python setup.py install --home=/cgse
```

CORE SERVICES

Running the core services

```
invoke status-core-egse
invoke start-core-egse    📍
invoke stop-core-egse     📍
```

Configuration Manager	`cm_cs status`
Storage Manager	`sm_cs status`
Process Manager	`pm_cs status`
Log Manager	`log_cs status`

DEVICE CONTROL

Start control servers in simulator mode	`--simulator`
AEU Test EGSE	`aeu_cs start|stop`
TCS EGSE	`tcs_cs start|stop`
PUNA Hexapod	`puna_cs start|stop`
ZONDA Hexapod	`zonda_cs start|stop`
HUBER Stages	`smc9300_cs start|stop`

GUIs

CORE SERVICES

Configuration Manager GUI	`cm_ui`
Process Manager GUI	`pm_ui`
Setup GUI	`setup_ui`

DEVICE GUIs

* Option to select device type	`--type proxy|simulator|direct`
AEU Test EGSE	`aeu_ui`
TCS EGSE	`tcs_ui`
PUNA Hexapod*	`puna_ui`
ZONDA Hexapod*	`zonda_ui`
HUBER Stages*	`smc9300_ui`
Filter Wheel 8SMC4*	`fw8smc4_ui`
Shutter KSC101*	`ksc101_ui`
Power Meter PM100A*	`pm100a_ui`
LakeShore Model 336*	`lsci336_ui`

OTHER GUIs

HDF5 GUI (N-FEE)	`hdf5_ui [{filename}]`
Source Position GUI	`fpa_ui`
Visited Positions GUI	`vis_pos_ui`

📍 use only when working isolated on your local machine

Most commands have a —`help` option.
`{var}` means to change this block with the value of `var`.

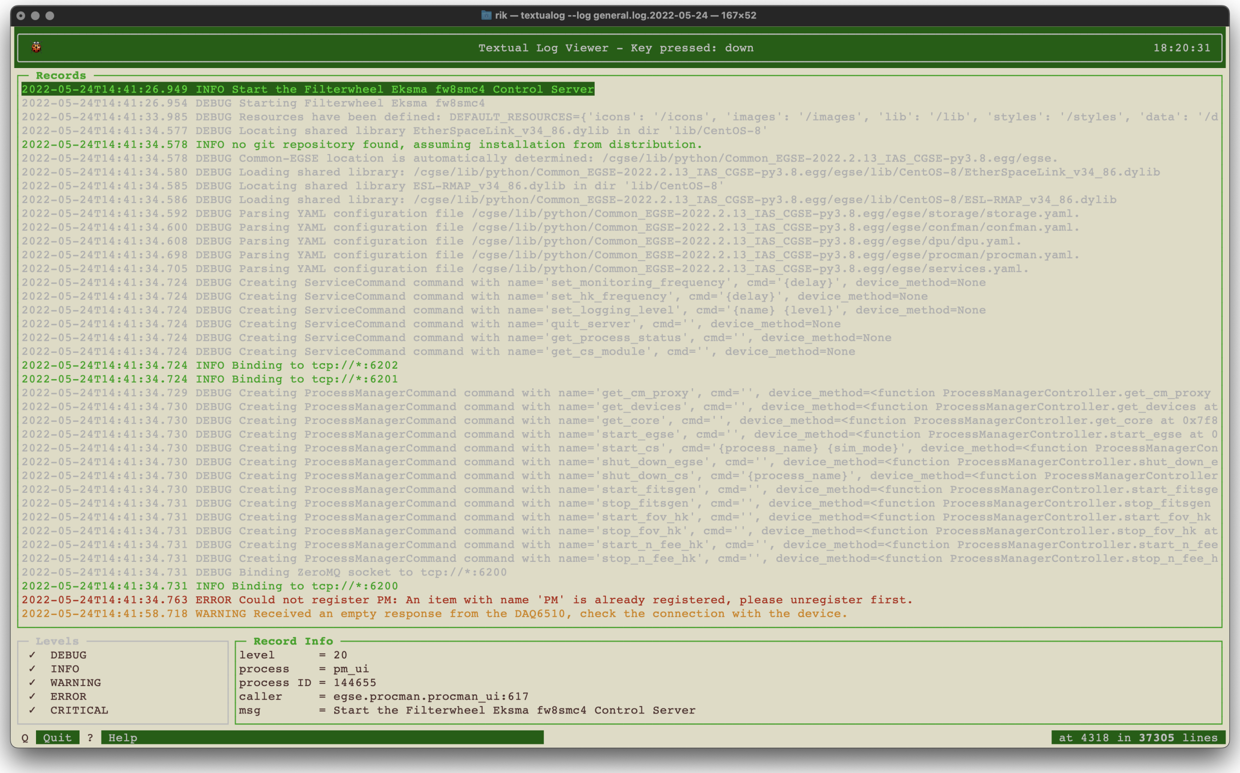
Chapter 14. Stories

This section contains short stories of debugging sessions and working with the system that are otherwise difficult to explain in the previous sections.

14.1. Seemingly unrelated Process Manager Crash

Related to issue: xxxx

The filter wheel control server (`fw8smc4_cs`) didn't start when using the process manager GUI (`pm_ui`). When the button was clicked nothing happened, but we got some error messages in the logger:


rik — textualog --log general.log.2022-05-24 — 167×52
Textual Log Viewer - Key pressed: down
18:20:31
Records
Levels
Record Info
Q Quit
? Help
at 4318 in 37305 lines

```
2022-05-24T14:41:26.949 INFO Start the Filterwheel Eksma fw8smc4 Control Server
2022-05-24T14:41:26.954 DEBUG Starting Filterwheel Eksma fw8smc4
2022-05-24T14:41:33.985 DEBUG Resources have been defined: DEFAULT_RESOURCES={'icons': '/icons', 'images': '/images', 'lib': '/lib', 'styles': '/styles', 'data': '/d
2022-05-24T14:41:34.577 DEBUG Locating shared library EtherSpaceLink_v34_86.dylib in dir 'lib/CentOS-8'
2022-05-24T14:41:34.578 INFO no git repository found, assuming installation from distribution.
2022-05-24T14:41:34.578 DEBUG Common-EGSE location is automatically determined: /cgse/lib/python/Common_EGSE-2022.2.13_IAS_CGSE-py3.8.egg/egse.
2022-05-24T14:41:34.580 DEBUG Loading shared library: /cgse/lib/python/Common_EGSE-2022.2.13_IAS_CGSE-py3.8.egg/egse/lib/CentOS-8/EtherSpaceLink_v34_86.dylib
2022-05-24T14:41:34.585 DEBUG Locating shared library ESL-RMAP_v34_86.dylib in dir 'lib/CentOS-8'
2022-05-24T14:41:34.586 DEBUG Loading shared library: /cgse/lib/python/Common_EGSE-2022.2.13_IAS_CGSE-py3.8.egg/egse/lib/CentOS-8/ESL-RMAP_v34_86.dylib
2022-05-24T14:41:34.592 DEBUG Parsing YAML configuration file /cgse/lib/python/Common_EGSE-2022.2.13_IAS_CGSE-py3.8.egg/egse/storage/storage.yaml.
2022-05-24T14:41:34.600 DEBUG Parsing YAML configuration file /cgse/lib/python/Common_EGSE-2022.2.13_IAS_CGSE-py3.8.egg/egse/confman/confman.yaml.
2022-05-24T14:41:34.608 DEBUG Parsing YAML configuration file /cgse/lib/python/Common_EGSE-2022.2.13_IAS_CGSE-py3.8.egg/egse/dpu/dpu.yaml.
2022-05-24T14:41:34.698 DEBUG Parsing YAML configuration file /cgse/lib/python/Common_EGSE-2022.2.13_IAS_CGSE-py3.8.egg/egse/procman/procman.yaml.
2022-05-24T14:41:34.705 DEBUG Parsing YAML configuration file /cgse/lib/python/Common_EGSE-2022.2.13_IAS_CGSE-py3.8.egg/egse/services.yaml.
2022-05-24T14:41:34.724 DEBUG Creating ServiceCommand command with name='set_monitoring_frequency', cmd='{delay}', device_method=None
2022-05-24T14:41:34.724 DEBUG Creating ServiceCommand command with name='set_hk_frequency', cmd='{delay}', device_method=None
2022-05-24T14:41:34.724 DEBUG Creating ServiceCommand command with name='set_logging_level', cmd='{name} {level}', device_method=None
2022-05-24T14:41:34.724 DEBUG Creating ServiceCommand command with name='quit_server', cmd='', device_method=None
2022-05-24T14:41:34.724 DEBUG Creating ServiceCommand command with name='get_process_status', cmd='', device_method=None
2022-05-24T14:41:34.724 DEBUG Creating ServiceCommand command with name='get_cs_module', cmd='', device_method=None
2022-05-24T14:41:34.724 INFO Binding to tcp://*:6202
2022-05-24T14:41:34.724 INFO Binding to tcp://*:6201
2022-05-24T14:41:34.729 DEBUG Creating ProcessManagerCommand command with name='get_cm_proxy', cmd='', device_method=<function ProcessManagerController.get_cm_proxy
2022-05-24T14:41:34.730 DEBUG Creating ProcessManagerCommand command with name='get_devices', cmd='', device_method=<function ProcessManagerController.get_devices at
2022-05-24T14:41:34.730 DEBUG Creating ProcessManagerCommand command with name='get_core', cmd='', device_method=<function ProcessManagerController.get_core at 0x7f8
2022-05-24T14:41:34.730 DEBUG Creating ProcessManagerCommand command with name='start_egse', cmd='', device_method=<function ProcessManagerController.start_egse at 0
2022-05-24T14:41:34.730 DEBUG Creating ProcessManagerCommand command with name='start_cs', cmd='{process_name} {sim_mode}', device_method=<function ProcessManagerCon
2022-05-24T14:41:34.730 DEBUG Creating ProcessManagerCommand command with name='shut_down_egse', cmd='', device_method=<function ProcessManagerController.shut_down_e
2022-05-24T14:41:34.730 DEBUG Creating ProcessManagerCommand command with name='shut_down_cs', cmd='{process_name}', device_method=<function ProcessManagerController
2022-05-24T14:41:34.730 DEBUG Creating ProcessManagerCommand command with name='start_fitsgen', cmd='', device_method=<function ProcessManagerController.start_fitsge
2022-05-24T14:41:34.731 DEBUG Creating ProcessManagerCommand command with name='stop_fitsgen', cmd='', device_method=<function ProcessManagerController.stop_fitsgen
2022-05-24T14:41:34.731 DEBUG Creating ProcessManagerCommand command with name='start_fov_hk', cmd='', device_method=<function ProcessManagerController.start_fov_hk
2022-05-24T14:41:34.731 DEBUG Creating ProcessManagerCommand command with name='stop_fov_hk', cmd='', device_method=<function ProcessManagerController.stop_fov_hk at
2022-05-24T14:41:34.731 DEBUG Creating ProcessManagerCommand command with name='start_n_fee_hk', cmd='', device_method=<function ProcessManagerController.start_n_fee
2022-05-24T14:41:34.731 DEBUG Creating ProcessManagerCommand command with name='stop_n_fee_hk', cmd='', device_method=<function ProcessManagerController.stop_n_fee_h
2022-05-24T14:41:34.731 DEBUG Binding ZeroMQ socket to tcp://*:6200
2022-05-24T14:41:34.731 INFO Binding to tcp://*:6200
2022-05-24T14:41:34.763 ERROR Could not register PM: An item with name 'PM' is already registered, please unregister first.
2022-05-24T14:41:58.718 WARNING Received an empty response from the DAQ6510, check the connection with the device.

 ✓  DEBUG           level      = 20
 ✓  INFO            process    = pm_ui
 ✓  WARNING         process ID = 144655
 ✓  ERROR           caller     = egse.procman.procman_ui:617
 ✓  CRITICAL        msg        = Start the Filterwheel Eksma fw8smc4 Control Server
```

There was no clear error message except that the PM was already registered at the storage manager. We thought that could be due to the fact the pm_ui crashed already several times before, or maybe a STORAGE_MNEMONIC in the Settings that was not set correctly. So we checked the Settings YAML file, and also the local settings file. The command to do that is:

```
python -m egse.settings
```

We looked at the Setups and inspected the source of the filter wheel control server, but could not relate the problem to anything in the code. Especially, since the control server could be started without problem from the terminal.

```
fw8smc4_cs start
```

So we started to look into all kinds of settings that were important when starting a control server from the process manager, e.g. the DEVICE_SETTINGS variable that is expected in the module and should contain the definition of the ControlServer. XXXXX: xref to document/section explaining this. We fixed a number of these things, see for example PR #1963, but still the problem remained.

We were still confused about the message that the PM was already registered at the Storage Manager. Why does this appear here? Why would the process manager try to register again. Looking at the debug messages before the error, we saw that the process manager actually restarted. That's of course the reason why it tries to register to the Storage, and the fact that it is already registered is probably due to the fact that the process manager crashed. So, instructing the `pm_cs` to start the `fw8smc4_cs` (by clicking the button in the `pm_ui`) crashes the process manager. And it is of course immediately restarted by the Systemd services.

So, we checked the logging messages of the Systemd for the `pm_cs.services`:

```
May 24 14:46:35 plato-arrakis pm_cs[2526576]: libximc.so: cannot open shared object
file: No such file or directory
May 24 14:46:35 plato-arrakis pm_cs[2526576]: Can't load libximc library. Please add
all shared libraries to the appropriate places. It is decribed in detail in
developers' documentation. On Linux make sure you installed libximc-dev package.
May 24 14:46:35 plato-arrakis pm_cs[2526576]: make sure that the architecture of the
system and the interpreter is the same
May 24 14:46:35 plato-arrakis pm_cs[2526576]: System Exit with code None.
```

The problem seems to be that the `pm_cs` can not load the `libximc` shared library which is needed for the filter wheel control server. This works in the terminal, because there the `LD_LIBRARY_PATH` is set, but it is not known to the process manager control server. To fix this, the environment variable must be set in the `/cgse/env.txt` file that is used in the services file to start the `pm_cs.services`. The content of the files:

```
[plato-data@plato-arrakis ~]$ cat /cgse/env.txt
PYTHONPATH=/cgse/lib/python/
PLATO_LOCAL_SETTINGS=/cgse/local_settings.yaml
PLATO_CONF_DATA_LOCATION=/data/IAS/conf
PLATO_CONF_REPO_LOCATION=/home/plato-data/git/plato-cgse-conf
PLATO_DATA_STORAGE_LOCATION=/data/IAS
PLATO_LOG_FILE_LOCATION=/data/IAS/log
LD_LIBRARY_PATH=/home/plato-data/git/plato-common-
egse/src/egse/lib/ximc/libximc.framework
```

and

```
[plato-data@plato-arrakis ~]$ cat /etc/systemd/system/pm_cs.service
[Unit]
Description=Process Manager Control Server
After=network-online.target cm_cs.service

[Service]
Type=simple
Restart=always
RestartSec=3
User=plato-data
Group=plato-data
EnvironmentFile=/cgse/env.txt
WorkingDirectory=/home/plato-data/workdir
ExecStartPre=/bin/sleep 3
ExecStart=/cgse/bin/pm_cs start

[Install]
Alias=pm_cs.service
WantedBy=multi-user.target
```

After this fix, the `fw8smc4_cs` could be started from the process manager GUI.

Chapter 15. Miscellaneous

Below this point we have miscellaneous topics that still need their place in the developer manual.

This file contains sections that are not assigned a proper location in the developer document. When we are ready for one of the sections, just move it out into its dedicated file.

Sections below are copied from the Word document PLATO-KUL-PL-MAN-0003.

Chapter 16. System Summary

16.1. System Configuration

- How to determine the version that is installed?
  - `from egse.version import VERSION`
  - `from egse.version import RELEASE`
  - The VERSION and RELEASE are defined and maintained in the settings.yaml file in the egse package..
- Which files to edit for configuring the system?
  - `egse/settings.yaml` do not edit this, instead edit the local_settings file pointed to by PLATO_LOCAL_SETTINGS environment variable.
- Where do I describe which components are part of the system?
- What if CSL has two Keithley DAQ6510 devices? How will they be distinguished?

16.2. System Components

This section describes the main components of the Common-EGSE system.

- Storage Manager
- Configuration Manager
- Process Manager
- Device Control Servers
- System Under Test (SUT)

16.3. Data Flows

The Common-EGSE provides functionality to configure, control and/or readout the following devices:

- Camera N-FEE and F-FEE, i.e. configure the camera and readout the CCDs
- CAM TCS EGSE, i.e. thermal control of the TOU and FEE
- AEU EGSE, i.e. provide secondary power and synchronisation to the FEE
- Optical, e.g. control and attenuate laser light source
- Mechanisms, e.g. hexapod, translation and rotation stages, gimbal
- Thermal control or monitoring of test setup

All data that is output by any of these devices will be archived in global and/or test specific files. The file types and formats are described in section Error! Reference source not found. Error!

Reference source not found..

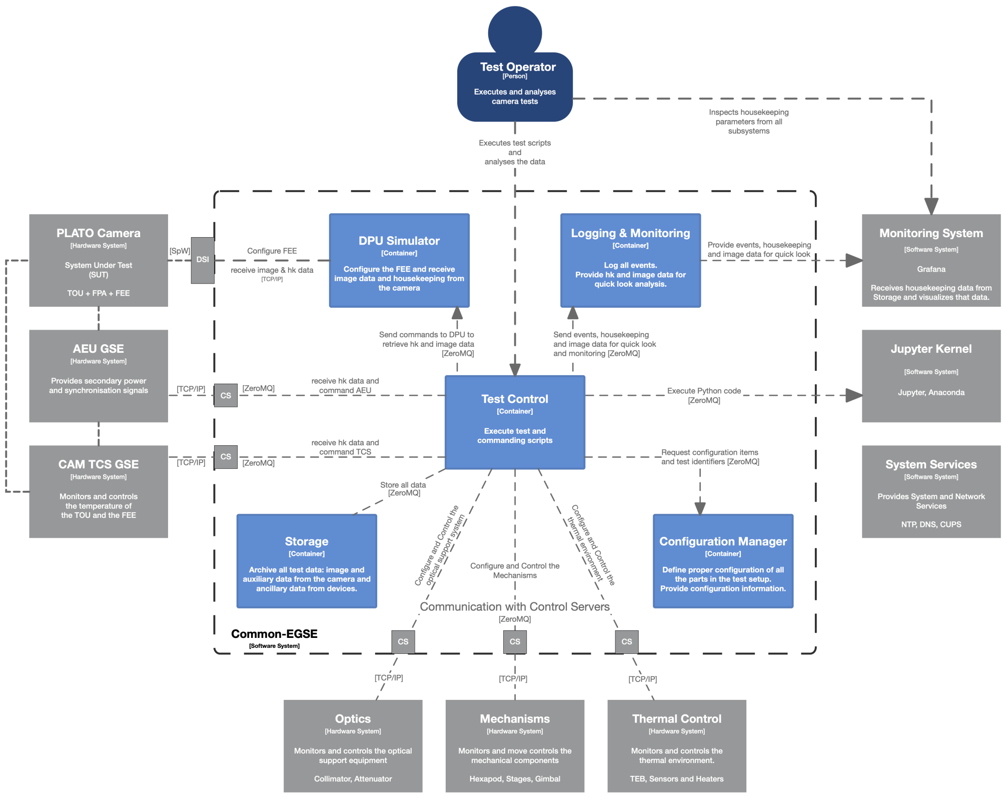

Test Operator
[Person]
Executes and analyses camera tests
Inspects housekeeping parameters from all subsystems
Executes test scripts and analyses the data
PLATO Camera
[Hardware System]
System Under Test (SUT)
TOU + FPA + FEE
[SpW]
DSI
Configure FEE
receive image & hk data
[TCP/IP]
DPU Simulator
[Container]
Configure the FEE and receive image data and housekeeping from the camera
Logging & Monitoring
[Container]
Log all events. Provide hk and image data for quick look analysis.
Provide events, housekeeping and image data for quick look
Monitoring System
[Software System]
Grafana
Receives housekeeping data from Storage and visualizes that data.
AEU GSE
[Hardware System]
Provides secondary power and synchronisation signals
Send commands to DPU to retrieve hk and image data [ZeroMQ]
Send events, housekeeping and image data for quick look and monitoring [ZeroMQ]
Jupyter Kernel
[Software System]
Jupyter, Anaconda
[TCP/IP]
CS
[ZeroMQ]
receive hk data and command AEU
Test Control
[Container]
Execute test and commanding scripts
Execute Python code [ZeroMQ]
CAM TCS GSE
[Hardware System]
Monitors and controls the temperature of the TOU and the FEE
receive hk data and command TCS
Request configuration items and test identifiers [ZeroMQ]
System Services
[Software System]
Provides System and Network Services
NTP, DNS, CUPS
Store all data [ZeroMQ]
Storage
[Container]
Archive all test data: image and auxiliary data from the camera and ancillary data from devices.
Configure and Control the optical support system
Configure and Control the Mechanisms
Configure and Control the thermal environment
Configuration Manager
[Container]
Define proper configuration of all the parts in the test setup. Provide configuration information.
Communication with Control Servers [ZeroMQ]
Common-EGSE
[Software System]
Optics
[Hardware System]
Monitors and controls the optical support equipment
Collimator, Attenuator
Mechanisms
[Hardware System]
Monitors and move controls the mechanical components
Hexapod, Stages, Gimbal
Thermal Control
[Hardware System]
Monitors and controls the thermal environment.
TEB, Sensors and Heaters

Figure 1. The main components of the Common-EGSE (bleu) and how they interface to each other (internal interfaces) and to the test setup (grey) and the PLATO camera (SUT) (external interfaces).

The Common-EGSE consists of several components, each with specific responsibilities, that communicate over a ZeroMQ network protocol. The following components have been identified and are part of the core functionality of the Common-EGSE:

- Test Control: execution of test and commanding scripts
- Configuration Manager: control the systems configuration
- DPU Simulator: configuration of the FEE and readout of the CCD and housekeeping data
- Logging: central logging component for status information and events
- Monitoring: monitor crucial housekeeping and telemetry and perform limit checks
- Storage: archive all data like images, housekeeping, telemetry, SpaceWire packets, commanding sequences etc.

Figure 1 above summarises the main components in the core Common-EGSE and test setup and defines their connections.

Chapter 17. Installation and Update

The Common-EGSE software system is installed and updated via GitHub. The installation is more complex than a simple download-and-install procedure and is fully described in its installation guide [RD-01].

Chapter 18. Getting Started

If you work with the system for the first time, you should go through the user manual [RD-02] to get familiar with the Common-EGSE setup, services and interactions. This section will explain how to log onto the Common-EGSE and how to prepare your development environment.

18.1. Log on to the System

- Who does login to the system?
- How, as which user, privileges is the user logged in?
- What services are running anyway, started at system boot?
- Developer Desktop versus operational desktop

18.2. Setting up the development environment

- Git
- Fork and clone the GitHub repository
- Install the Common-EGSE system – see the installation guide [RD-01]
- PYTHONPATH
- Being in the right directory
- PyCharm or another IDE

Index