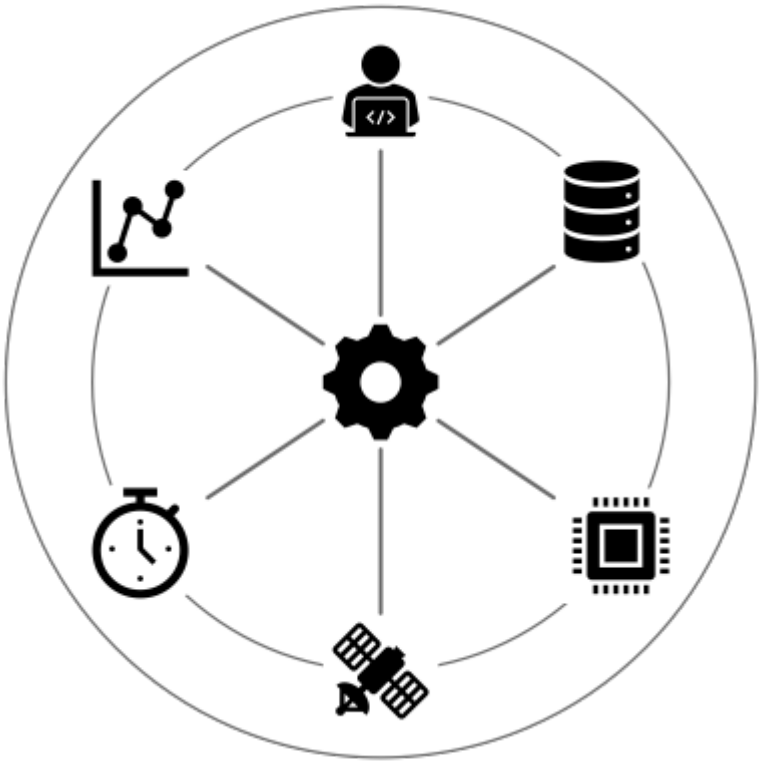


Common–EGSE

Developer Manual

Rik Huygen, Sara Regibo

Version 1.6, 04/05/2023

Table of Contents

Changelog

04/05/2023 — v1.6

- Added explanation about data distribution and monitoring message queues used by the DPU Processor to publish information and data, see Chapter 13.
- Added some more meat into the description of the arguments for the `@dynamic_command` decorator, see Section 6.6.2.
- Added an explanation of the communication with and commanding of the N-FEE in the inner loop of the DPUProcessor, see Section 13.1.1.

03/04/2023 — v1.5

- Added a section 'Services' in the Commanding Concepts section, see Section 6.4.

12/03/2023 — v1.4

- Added description of the arguments that can be used with the `@dynamic_command` decorator, see Section 6.6.2.
- Added a section on creating multiple control servers for identical devices, see Section 6.1.1.

09/02/2023 — v1.3

- Updated section on version and release numbers, see Chapter 2.
- Fixed some formatting issues in the CCD numbering section.

Colophon

1st Edition — February 2023

This manual is written in PyCharm using the AsciiDoc plugin. The PDF Book version is processed with asciidoctor-pdf.

The manual is available as HTML from ivs-kuleuven/github.io. The HTML pages are generated with Hugo which is an OSS static web-pages generator. From this site, you can also download the PDF books.

The source code is available in a GitHub repository at ivs-kuleuven/plato-cgse-doc.

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.

Conventions used in this Book

We try to be consistent with the following typographical conventions:

Italic

Indicates a new term or …

Constant width

Used for code listings, as well as within paragraphs to refer to program elements like variable and function names, data type, environment variables (`ALL_CAPS`), statements and keywords.

Constant width between angle brackets `<text>`

Indicates `text` that should be replaced with user-supplied values or by values determined by context. The brackets should thereby be omitted.

When you see a `$ ⋯` in code listings, this is a command you need to execute in a terminal (omitting the dollar sign itself). When you see `>>> ⋯` in code listings, that is a Python expression that you need to execute in a Python REPL (here omitting the three brackets).

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.

Introduction

This document is the developer guide for the PLATO Common-EGSE, used at CSL (Liège, Belgium) and the test houses at IAS (Paris, France), INTA (Madrid, Spain) and SRON (Groningen, The Netherlands) and describes the Common-EGSE system from the developer's point of view. This document explains how to contribute to the code, which versions of libraries that are used, compiler options for device libraries written in C, access to the GitHub repo, coding style,and much more.

The current release contains 303 Python files, together they contain 95389 lines of which there are 49360 lines of Python code. About 50% of the code files contain actual code, the rest is documentation, comments and blank lines. It is impossible for one person to fully understand all details of the system and only for that reason is it essential to have good and comprehensive documentation.

Please, read the installation instructions to know how to install the software and configure your Python installation.

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?
- [x] 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?
- [ ] Describe how to detect which process is holding a connection on Linux and macOS. This

happens sometimes when a process crashes or when the developer did not properly close all connection, or when a process is hanging. You will get a `OSError: [Errno 48] Address already in use`.

- ▢ Describe the RegisterMap of the N-FEE and how and when it is synced with the DPU Processor. What is the equivalent of the FPGA (loading the Registers on every long pulse) in the DPU Processor (DPU Internals vs NFEEState).
- ▢ Describe what the num_cycles parameter is and how this is used in the DPU Processor. That this parameter can be negative and what this means and why this decision (to not have too many if statements in the run() of the DPU Processor.)
- ▢ Describe that the DPU Processor is time critical
- ▢ Describe why certain control servers start a sub-process → TCS, DPU, …

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
CGSE	Common-EGSE
COT	Commercial off-the-shelf
DPU	Data Processing Unit
DSI	Diagnostic SpaceWire Interface
EGSE	Electrical Ground Support Equipment
OS	Operating System
PID	Process Identifier
PPID	Parent Process Identifier
PLM	Payload Module
REPL	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
TS	Test Scripts
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: `get_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. Version numbers

The version of Python that you use to run the code shall be >= 3.8. You can determine the version of Python with the following terminal command:

```
$ python3 -V
Python 3.8.3
```

The version of the Common-EGSE (CGSE) and the test scripts (TS) that are installed on your system can be determined as follows:

```
$ python3 -m egse.version
CGSE version in Settings: 2023.6.0+CGSE
CGSE installed version = 2023.6.0+cgse

$ python3 -m camtest.version
CAMTEST version in Settings: 2023.6.0+TS
CAMTEST git version = 2023.6.0+TS-0-g975d38e
```

The version numbers that are shown above have different parts that all have their meaning. We use semantic versioning to define our version numbers. Let's use the CGSE version [2023.6.0+CGSE] to explain each of the parts:

- 2023.6.0 is the core version number with major.minor.patch numbers. The major number is the year in which the software was released. The minor number is the week number in which the configuration control board (CCB) gathered to approve a new release. The patch is used for urgent bug fixes within the week between two CCBs.
- For release candidates a postfix can be added like 2023.6.0-rc.1 meaning that we are dealing with the first release candidate for the 2023.6.0 release. We should avoid using release candidates in the operational environment because these versions might not have been fully tested. In an official release of the software, this part in the versioning will be omitted.
- CGSE is a release for the Common-EGSE repository. The Test scripts will have TS here. There is also CONF for the plato-cgse-conf repository which contains all Setups and related files.

Then we have some additional information in the version number of the test scripts, in the above example [2023.6.0+TS-0-g975d38e].

- 0 is the number of commits that have been pushed and merged to the GitHub repository since the creation of the release. This number shall be 0 at least for the CGSE on an operational system. If this number != 0, it means you have not installed the release properly as explained in the installation instructions and you are probably using a development version with less tested or even un-tested code. This number can differ from 0 for the test scripts as we do not yet have a proper installation procedure for the releases.
- g975d38e is the abbreviated git hash number[1] for this release. The first letter g indicates the

use of `git`.

The VERSION and RELEASE are also defined in the `settings.yaml` and should match the first three parts of the version number explained above. If not, it was probably forgotten to update these numbers when preparing the release. You can use these version numbers in your code by importing from `egse.version` and `camtest.version`.

```
from egse.version import VERSION as CGSE_VERSION
from camtest.version import VERSION as CAMTEST_VERSION
```

[1] The full git hash is a much longer number and is an SHA-1 hash — a checksum of the content of the commit plus a header.

Chapter 3. 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.

3.1. 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 your code.

3.2. 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.

```
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()
```

3.3. 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:

```
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 only log the exception and 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

3.4. 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 the calling code didn't check a return value.

3.5. 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:

```
if response[-2:] != '\r\n':
    raise ConnectionError(f"Missing termination characters in response: {response}")
```

```
def convert_to_float(value: str) -> float:
    try:
        return float(value)
    except ValueError:
        return math.nan
```

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.

3.6. 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  ①
```

① use only a `raise` statement, without the `SomeException` added. This will rethrow the exact same exception that was catched.

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.

3.7. 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
```

3.8. 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
```

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
```

3.9. 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 in ('N-FEE', '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.

3.10. What are Errors?

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

3.11. Cascading Exceptions

TBW

3.12. 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

3.13. 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 4. 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

Part II — Core Concepts

Chapter 5. Core Services

What are considered core services?

The core services are provided by five control servers that run on the egse-server machine: the log manager, the configuration manager, the storage manager, the process manager, and the synoptics manager. Each of these services is installed as a *systemd* service on the operational machine. That means they are monitored by the system and when one of these processes crashes, it is automatically restarted. As a developer, you can start the core services on your local machine or laptop with the `invoke` command from a terminal.

```
$ cd ~/git/plato-common-egse
$ invoke start-core-egse
```

The services are described in full detail in the following sections.

5.1. 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?

5.2. The Configuration Manager

The configuration manager is a core service…

- The GlobalState
- The Setup
- The Observation concept

5.3. 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
- Persistence: CSV / HDF5 / FITS / TXT / SQLite

5.3.1. File Naming

- see also the ICD

5.3.2. File Formats

- see also the ICD

5.4. The Process Manager

The job of the process manager is to keep track and monitor the execution 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.

5.5. 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….

In all involved test facilities, the EGSE is used to perform the same set of basic operations: monitoring temperatures, changing the intensity of the source and point it somewhere, acquiring images, etc. However, the devices that are used to perform these tasks are not everywhere the same (e.g. the OGSE with its lamp and filterwheels, the DAQs for temperature acquisition, etc.).

Each (device) control server has a dedicated CSV file in which the housekeeping information is stored and often the name of the parameters indicates the test facility at which they were acquired.

It is not inconvenient if user need to memorise the HK names for all test facilities, it also makes the test and analysis scripts more complex.

The Synoptics Manager stores this information in one centralised location (the synoptics file)

with generic parameter names.

5.5.1. Synoptical Parameters

The housekeeping parameters that are stored in the synoptics are:

- Calibrated temperatures, acquired by the FEE, TCS, and facility DAQs;
- OGSE information (source intensity, measured power, whether the lamp and/or the laser are on);
- Source position, both actual and commanded, as field angles (θ,φ).

5.6. 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`).

5.6.1. 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.

Column name	Expected content
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.
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.

Column name	Expected content
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.

5.6.2. 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_
CSL1	GCSL1_

TH	Prefix
CSL2	GCSL2_
SRON	GSRON_
IAS	GIAS_
INTA	GINTA_

5.6.3. 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 ...`.

5.6.4. 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.

5.6.5. Sending HK to Synoptics

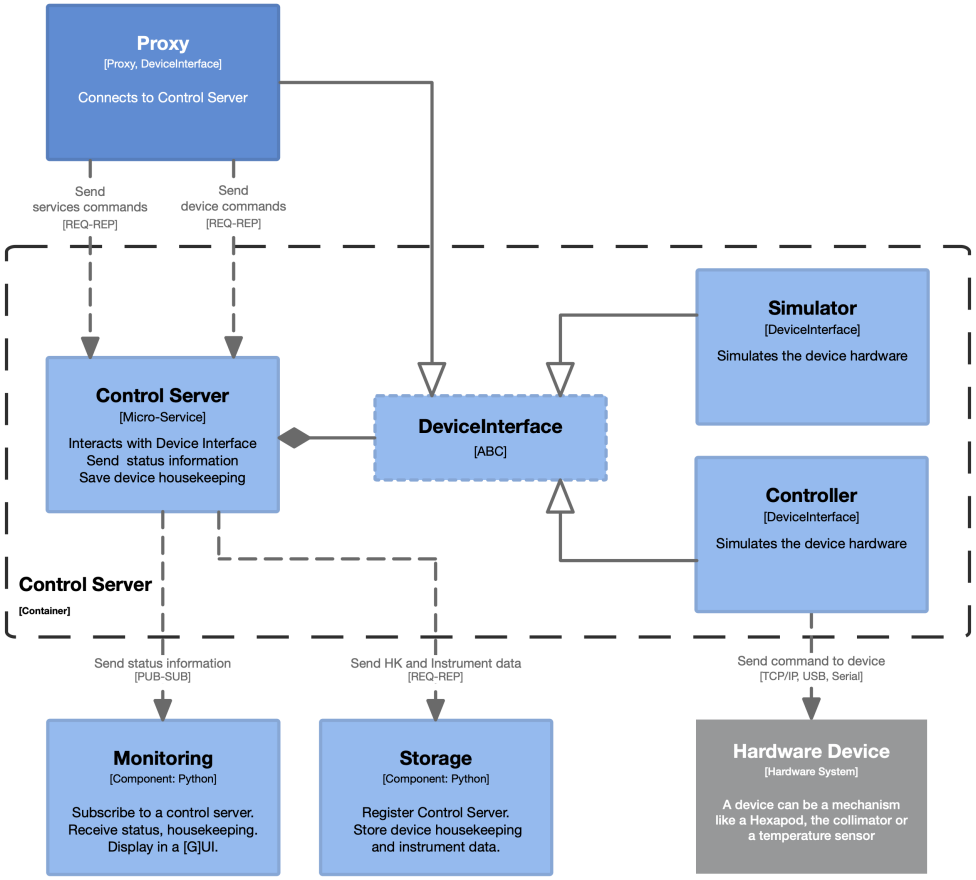
When you want to include HK of your devices, you need to take the following actions:

- 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!

Chapter 6. The Commanding Concept

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



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

6.1.1. Create multiple control servers for identical devices

This section describes what you can do when you have identical devices, such as two LakeShore Temperature Controllers or multiple power supplies, and need to use them in the same project at the same site. While the control servers are configured in the Settings YAML file with, among other things, IP address and port numbers, they still need to be addressed in the Setup for configuring your equipment at a certain time and location. Specifically, you will need to create

the proper `Proxy` class to communicate with your control server.

The specificity of the equipment used in your lab is defined in the Setup, and from the Setup, you will usually create the `Controller` or `Proxy` objects to access and control your equipment. This can be done in two different ways, which are explained in the next two sections.

Directly define the device controller class

In the Setup, you will define your devices as in the excerpt below. This will create a `PunaProxy` object whenever you request the device from the Setup.

```
gse:
    hexapod:
        device: class//egse.hexapod.symetrie.PunaProxy
```

Some devices contain more than one controller and need several control servers to manage that device. An example of this type is the AEU Test EGSE. This device contains six power supply units and two analog wave generators. When requesting a `Proxy` object to access the device, say PSU 2, the PSU identifier (the number '2' in this case) needs to be passed into the `__init__()` method of the `Proxy` class. To accomplish this, the Setup for this device contains an additional device argument:

```
gse:
    aeu:
        psu2:
            device: class//egse.aeu.aeu.PSUProxy
            device_args: [2]
```

All the values that are given in the list for the `device_args` are passed as positional arguments to the `__init__()` method of the `PSUProxy` class. When requesting the device from the Setup, you will get a `PSUProxy` object. The super class was however instantiated with the port number that was defined for PSU 2 in the settings file.

```
>>> psu = setup.gse.aeu.psu2.device
>>> psu
<egse.aeu.aeu.PSUProxy object at 0x7f83dafc2460>
>>> psu.name
'PSU2'
>>> psu.get_endpoint()
'tcp://localhost:30010'
```

Use a factory to create a device

In the Setup you can, instead of a `class//` definition, use a `factory//` definition. A Factory is a class with a specific interface to create the class object that you intended based on a number of

arguments that are also specified in the Setup. Let's first look at an example for the PUNA Hexapod. The following excerpt from a Setup defines the hexapod device as a `ControllerFactory`. There are also two device arguments defined, a device name and an idenditier.

```
gse:
    hexapod:
        device: factory//egse.hexapod.symetrie.ControllerFactory
        device_args:
            device_name: PUNA Hexapod
            device_id: 1A
```

So, what happens when you access this device from the Setup. Let's look at the following code snippet from your Python REPL. The Setup has been loaded already in the `setup` variable. To access the device, use the dot (.) notation.

```
1 >>> puna = setup.gse.hexapod.device
2 >>> puna
3 <egse.hexapod.symetrie.puna.PunaController object at 0x7f83e039ba60>
4 >>> setup.gse.hexapod.device_args.device_id = "1B"
5 >>> puna = setup.gse.hexapod.device
6 >>> puna
7 <egse.hexapod.symetrie.punaplus.PunaPlusController object at 0x7f83dab8ed00>
```

When we –with the given Setup– request the hexapod device, we get a `PunaController` object as you can see on line 3, when we change the `device_id` to '1B' and request the hexapod device, we get the `PunaPlusController` object (line 7). Changing the `device_id` resulted in a different class created by the Factroy. Any Factory class that is used this way in a Setup shall implement the `create(..)` method which is defined by the `DeviceFactoryInterface` class. Our `ControllerFactory` for the hexapod inherits from this interface and implements the method.

```
class DeviceFactoryInterface:
    def create(self, **kwargs):
        ...


class ControllerFactory(DeviceFactoryInterface):
    """
    A factory class that will create the Controller that matches the given device
name
    and identifier.

    The device name is matched against the string 'puna' or 'zonda'. If the device
name
    doesn't contain one of these names, a ValueError will be raised. If the
device_name
    matches against 'puna', the additional `device_id` argument shall also be
```

```
specified.
    """
    def create(self, device_name: str, *, device_id: str = None, **_ignored):

        if "puna" in device_name.lower():
            from egse.hexapod.symetrie.puna import PunaController
            from egse.hexapod.symetrie.punaplus import PunaPlusController

            if not device_id.startswith("H_"):
                device_id = f"H_{device_id}"

            hostname = PUNA_SETTINGS[device_id]["HOSTNAME"]
            port = PUNA_SETTINGS[device_id]["PORT"]
            controller_type = PUNA_SETTINGS[device_id]["TYPE"]

            if controller_type == "ALPHA":
                return PunaController(hostname=hostname, port=port)
            elif controller_type == "ALPHA_PLUS":
                return PunaPlusController(hostname=hostname, port=port)
            else:
                raise ValueError(
                    f"Unknown controller_type ({controller_type}) for {device_name}
"
                    f" {device_id}"
                )

        elif "zonda" in device_name.lower():
            from egse.hexapod.symetrie.zonda import ZondaController
            return ZondaController()

        else:
            raise ValueError(f"Unknown device name: {device_name}")
```

The `create(..)` method returns a device controller object for either the PUNA Alhpa or Alpha+ Controller, or for the ZONDA Controller. Each of these represent a Symétrie hexapod. The decision for which type of controller to return is first based on the device name, then on settings that depend on the device identifier. Both of these parameters are specified in the Setup (see above) and passed into the `create()` method of the factory class at the time the user requests the device from the setup.

The `TYPE`, `HOSTNAME`, and `PORT` are loaded from the `PUNA_SETTINGS` dictionary. These settings are device specific and are defined in the local setting for your site. The `PUNA_SETTINGS` is a dictionary that represents part of the overall Settings. The factory class uses these device configuration settings to decide which controller object shall be instantiated and what the parameters for the creation shall be.

```
└──── PMAC Controller
    ├──── H_1A
```

```
    │    ├──── TYPE: ALPHA
    │    ├──── HOSTNAME: XXX.XXX.XXX.X1A
    │    └──── PORT: 1025
    └──── H_1B
         ├──── TYPE: ALPHA_PLUS
         ├──── HOSTNAME: XXX.XXX.XXX.X1B
         └──── PORT: 23
```

You might think that it would be simpler to specify the device configuration settings also in the Setup and pass all this information to the create method when requesting the device from the Setup. However, this would undermine the specific difference between Settings and Setup and also would invalidate the direct creation of the controller object without the use of a factory class.

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. Services

The control servers are the single point access to your equipment and devices. Commanding the device is handled by the Protocol and Controller classes on the server side and by the Proxy class on the client side. In addition to commanding your devices, you would also like to have some control on the behaviour of your device control servers. That is where the Service slass comes into play. The Services provide common commands that act on the control server instead of on

the device. These commands are send through a Service Proxy that you can request for each of the control servers. Below is a code snippet asking for the service proxy of the hexapod PUNA and requesting the status of the control server process.

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

>>> with PunaProxy() as puna, puna.get_service_proxy() as srv:
...     print(srv.get_process_status())
{
    'timestamp': '2023-03-29T08:00:31.708+0000',
    'delay': 1000,
    'PID': 39242,
    'Up': 486.7373969554901,
    'UUID': UUID('a74c0f52-ce06-11ed-a5d1-acde48001122'),
    'RSS': 115720192,
    'USS': 98033664,
    'CPU User': 10.516171776,
    'CPU System': 3.300619776,
    'CPU count': 6,
    'CPU%': 2.9
}
```

Status information for the control servers is also sent out on the monitoring port, so if you need to monitor the process status, its better to subscribe to the monitoring protocol.

There are two commands that you can use to change the frequency by which the control server is sending out information, one is for changing the housekeeping frequency, one is for changing the monitoring frequency.

```
set_hk_frequency(freq: float) -> float
```

This function sets the housekeeping frequency (in Hz) to the given freq value. This is only approximate since the frequency is converted into a delay time and the actual execution of the `get_housekeeping()` function is subject to the load on the server and the overhead of the timing. What will happen is that the delay time that will be applied is never shorter than the average execution time of the `get_housekeeping()` method increased with a 200ms overhead to allow the control server to respond to other commands.

```
set_monitoring_frequency(freq: float) -> float
```

This function works similar to the `set_hk_frequency()` but sets the delay time on the `get_status()` function of the device protocol.

When you want to optimize these frequencies, make sure you have an idea of the execution time of either function. If for instance the `get_status()` function takes 500ms to complete, it will be

impossible to request a monitoring update frequency of 2Hz.

6.5. 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.

6.6. Dynamic Commanding

Two of the requirements that were formulated when we started the analysis and design of the CGSE was that (1) it shouldn't be too difficult to implement device controllers and (2) the definition of the commands shall be done at one place only. So we started out with a design where the commands were defined in a YAML file (called the *command yaml file*) and the code would use this information to build the API for the device. That would make it —from the device developer point of view— rather easy to create the commanding interface for a new device. Unfortunately, the implementation was not that easy or straightforward. We are working in a client-server or even microservices environment which means the device commanding should be available on both the server side (what we call the Controller) and on the client side (what we call the Proxy).

Both the Controller and the Proxy will have to implement the same interface, but we didn't want to bother the developer with having to implement the interface twice. The Proxy (client) side should be easy, because it doesn't need any processing or device control, it only needs to forward the command and its arguments to the server to be executed and then wait for the response. On the server it is slightly more complicated since there we have to interact with a device that can be connected by Ethernet, USB, serial,... with many different commanding protocols.

We have currently two designs and implementations that handle dynamic commanding. The first implementation we came up with is the `@dynamic_interface` which is a decorator that is used on the methods in the device interface class. How this works is explained in the section below on Section 6.6.1. The latest implementation is the `@dynamic_command` which also decorates the interface methods, but does a lot more work with respect to command string generation and response processing. How this work is explained in Section 6.6.2. We have also written up a section on the transition from the old implementation into the new dynamic command

implementation. See Section 6.6.3.

6.6.1. The Dynamic Interface Decorator

A `@dynamic_interface` decorator adds a private static variable to a function or method. The variable name that is attributed is `__dynamic_interface` and it is used as a marker on the function.

The static variable is used by the Proxy class to check if a method is meant to be overridden dynamically. The idea behind this is to loosen the contract of an abstract base class (ABC) into an interface. For an ABC, the abstract methods must be implemented at construction/initialization. This is not possible for the Proxy subclasses as they load their commands (i.e. methods) from the control server, and the method will be added to the Proxy interface after loading. Nevertheless, we like the interface already defined for auto-completion during development or interactive use.

When a Proxy subclass that implements an interface with methods decorated by the `@dynamic_interface` does overwrite one or more of the decorated methods statically, these methods will not be dynamically overwritten when loading the interface from the control server. A warning will be logged instead.

The methods of the Proxy sub-class are added by the base class method `load_commands()`. This method asks the control server to send all device command definitions. The control server replies with a dictionary of user-defined methods and their names as keys. The commands are attributed to the Proxy class as methods with the key as their method names, unless the attribute name already exists in the Proxy and is not decorated as a dynamic interface. The following code snippet demonstrates how the command methods are dynamic added/attributed to the Proxy class.

```
def _add_commands(self):
    for key in self._commands:
        if hasattr(self, key):
            attribute = getattr(self, key)
            if (
                isinstance(attribute, types.MethodType) and
                not hasattr(attribute, "__dynamic_interface")
            ):
                continue
        command = self._commands[key]
        new_method = MethodType(command.client_call, self)
        new_method = set_docstring(new_method, command)
        setattr(self, key, new_method)
```

The command from the dictionary is not just added as a new attribute of course (that would be too easy). Instead, each command is defined as an instance of a client-server command class (`ClientServerCommand`). The ClientServerCommand has two methods `client_call()` and `server_call()` and —you guessed it— all the dynamically attached methods are the `client_call()` method for that command, but carrying the name of the command, i.e. the key in the dictionary.

The `command.client_call` is a function, i.e. an unbound class method. We are now trying to associate this function with the class instance (`self`) of the Proxy sub-class. We need to create a method that is *bound* to the class instance. Think of it as *binding* the class instance to the first argument of the function (the `self` argument).

The `MethodType()` returns a new callable object that holds both the function and the instance, and if you call this object, its first argument is automatically bound to the `self` argument.

The `setattr(self, key, new_method)` can not be replaced by a plain assignment because the name of the method is a string and we can not do something like `self.key = new_method` because the method name would be `key`.

So, during the creation of the Proxy object, the command list is requested from the control server, and when the command is not already an attribute of the Proxy sub-class, its `client_call()` method is connected as a new method to the Proxy sub-class.

The `ClientServerCommand` class is defined in the module `egse.command` and the `client_call()` method does something like in the code snippet below. The arguments that are passed into the function are first validated. This validation is done against the input given in the *command yaml file*. Then a command execution object is created which is sent to the control server. The response from the server is returned.

```
def client_call(self, other, *args, **kwargs):
    try:
        self.validate_arguments(*args, **kwargs)
    except CommandError as e_ce:
        logger.error(str(e_ce))
        return None

    ce = CommandExecution(self, *args, **kwargs)
    return other.send(ce)
```

At the control server, the command execution is received by the Protocol class and the command and the given arguments are extracted. The Protocol then calls the `server_call()` method of the command with the arguments. The (simplified) code snippet below illustrates what happens in the `execute()` method of the Protocol class at the server. The `self` variable is the instance of the device protocol class.

```
def execute(self):
    ...
    ce = self.receive()
    cmd = ce.get_cmd()
    args = ce.get_args()
    kwargs = ce.get_kwargs()
    ...
    cmd.server_call(self, *args, **kwargs)
```

The `server_call()` now calls the method from the Controller class that was associated with the `ClientServerCommand` while parsing and processing the *command yaml file*. That method is called with the arguments passed into the command execution.

- [] make a graph illustrating the round-trip of the commanding chain.

6.6.2. The Dynamic Command Decorator

One of the problems of the dynamic interface decorator is that during construction of the Proxy class, the control server must be contacted to request the list of commands, when this fails and the control server is not available, the Proxy is not functional. You will first get a warning log message of the following type:

```
<Device>Proxy could not connect to its control server at tcp://localhost:6700. No
commands have been loaded.
```

When you then try to execute a command, e.g. `puna_proxy.get_speed()`, it will result in a `NotImplementedError`. The reason is that the commands have not been loaded from the control server and therefore the default interface method is executed resulting in this error.

Mainly for this reason we have enhanced the dynamic commanding protocol with the `@dynamic_command` decorator. This decorator behaves differently on the client side (the Proxy) and the server side (the Controller). There is no need anymore to load the available commands from the control server, so, proxies for devices can be started when the control server is not yet running and will gracefully (re-)connect to the device when the control server becomes available.

NOTE

add more description of what the decorator is, what happens behind the scenes, how it is perceived on the client and on the server side etc...

So, since the `@dynamic_command` is the successor of the `@dynamic_interface`, how do we refactor our code for this new commanding scheme. That will be explained in the next section.

6.6.3. Update your code to use the @dynamic_command decorator

- Each method in the interface needs to be changed from a `@dynamic_interface` to a `@dynamic_command`. Remember the `@dynamic_interface` only marked the method with the `__dynamic_interface` attribute. The `@dynamic_command` defines the type of command, how the command string is contructed and how the response needs to be decoded. As an example, consider the `get_current_position()` method in the `HuberSMC9300Interface` class.

```
    @dynamic_interface
    def get_current_position(self, axis) -> float:
        """
        Returns the current position for this axis as a float.
        """
```

```
        raise NotImplementedError
```

The method takes an argument `axis` and returns the current position as a float. The implementation is found in the `HuberSMC9300Controller` class.

```
    def get_current_position(self, axis) -> float:
        cmd_string = cmd_q_pos.get_cmd_string(axis)
        retStr = self.huber.get_response(cmd_string)

        # The response will look like '<axis>:<retPos>;'
        # where <axis> is the axis number and <retPos> is the current position
        # as a float
        retPos = float(retStr[2:-1])

        return retPos
```

In the new scheme, the method in the interface class will look like below and the method in the controller class will be removed.

```
    @dynamic_command(cmd_type="query", cmd_string="?p${axis}",
                     process_cmd_string=process_cmd_string,
                     process_response=decode_axis_float)
    def get_current_position(self, axis: int) -> float:
        """
        Returns the current position for this axis as a float.
        """
        raise NotImplementedError
```

The command is a *query* command and therefore expects a response from the device. The command string is given in the form of a string template where 'axis' is replaced by the value of the `axis` argument of the method. The cmd_string will become `"?p1"` when `axis=1`. The `process_cmd_string` is a function that will prepare the command string for sending to the device. In the case of the HUBER stages, it appends `\r\n` before sending.

The `decode_axis_float` is a function used to decode the response from the device. It's a function that can be used to process all responses in the form of `<axis>:<float>;`.

```
def decode_axis_float(response: bytes) -> float:
    """
    Decodes the response of type '<axis>:<float>;' and strips off the newline.
    """

    response = response.decode().rstrip()

    return float(response[2:-1])
```

So, we removed the method from the *Controller* class and converted the *Interface* method from a `@dynamic_interface` to a `@dynamic_command`. The *Controller* class still inherits from the *Interface* class, but additionally it now also needs to inherit from the `DynamicCommandMixin` class which is defined in the `egse.mixin` module.

```
class HuberSMC9300Controller(HuberSMC9300Interface, DynamicCommandMixin):
    ...
```

- The Proxy class needs to inherit from `DynamicProxy` instead of `Proxy:

```
class HuberSMC9300Proxy(DynamicProxy, HuberSMC9300Interface):
    ...
```

- Finally, remove all device command definitions from the device YAML file, in our case `smc9300.yaml`.

Thus far we have touch on the following arguments that can be used with `@dynamic_command`.

- `cmd_type` can be one of the following 'read', 'write', 'transaction', and 'query'. This is the only required argument for the decorator.
- `cmd_string` defines the formatting of the eventual command string that will be passed to the transport functions. The `cmd_string` is a template string that contains `$`-based substitutions for the function arguments. When you specify the `use_format=True` keyword, the `cmd_string` will be formatted using the format() function instead of the template substitution. The format option is less secure, but provides the functionality to format the arguments.

  A template string looks like:

  ```
  cmd_string="CREATE:SENS:TEMP ${name} ${type} default=${default}"
  ```

  The same `cmd_string` as a format option:

  ```
  cmd_string="CREATE:SENS:TEMP {name} {type} default={default:0.4f}"
  use_format=True
  ```

- `process_response` is a pure function[1] to process the response from the device before it is returned. This function shall take at least one keyword argument `response=` which is the response from the device as a byte array. The value that is returned by the function is eventually also the response of the command that was executed. This return value is device implementation specific and part of the interface definition.
- `process_cmd_string` is a pure function to process the given command string to make it ready for sending to the device. This function is called after the arguments to the command have been filled into the template string given by `cmd_string`. The function shall take a string as the

only positional argument and return a string that will be sent to the device.

Then, there are these additional arguments that can be used with `@dynamic_command` to tune the command string and pre- and post-processing.

- `use_format` defines if the `cmd_str` shall be formatted as a template string or with the `format()` method.
- `process_kwargs` is a function that processes the keyword arguments and returns a string representation of those arguments. By default, keyword arguments are expanded in a string containing `key=value` pairs separated by spaces. This function is used when the arguments are given as `**kwargs` to the function definition.
- `pre_cmd` specifies a function that will be executed before the processed command string is sent to the device.
- `post_cmd` specifies a function that will be executed after the command is executed, i.e. sent to the device and potentially retrieved a response.

The `pre_cmd` and `post_cmd` keywords specify a callable/function to be executed before and/or after the actual command was executed. These functions are called with specific keyword arguments that allow additional device interaction and response processing. The `pre_cmd` function is called with the keyword argument `transport=` which passes the device transport. This allows the function to interact with the device again through the methods defined by the DeviceTransport interface. Additionally, the name of the called function (`function_name`), the processed command string (`cmd_string`) and the original positional (`args`) and keyword arguments (`kwargs`) are passed into the `pre_cmd` function. These additional arguments should not be changed, but can be used by the function for processing, logging, etc. The `pre_cmd` function must not return anything.

The `post_cmd` function is called with the keyword arguments `transport=` and `response=`. The response argument contains the response from the command that was previously sent to the device. The `post_cmd` function can use this response to parse its content and act against this content, although possible, it is usually not a good idea to alter the content of the response argument. The `post_cmd` function shall return (i.e. pass through) the response or return it's own information e.g. a status that was retrieved from the device with an additional command sent over transport.

We now have walked through all the steps to upgrade your device commanding. The next thing to do is testing!

[1] A pure function is a function that has no side effects and always returns the same output for the same input.

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 machine. 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 Settings Class

- The `settings.py` module...
- The `settings.yaml` file is located in the `egse` module.
- The local settings ....
- reloading settings is currently not possible

The *Settings* contain all static information needed to configure your system and environment. That is first of all the version and release numbers of the CGSE, and the site identifier and other information about the site. The Settings also contain all the IP addresses and port number for all the known devices, together with other static information like the device name, default settings for the device like speed, timeout, delay time, firmware version, etc. We will go into more details about the content later, let's now first look at the format and usage of the Settings.

The Settings are maintained in a YAML file which is located in the `egse` module. The default Settings file is named `settings.yaml` and we call them *Global Settings*. You can for your project or test house also define *Local Settings* with higher precedence that will overwrite the global settings when loaded. The location of the local settings YAML file is defined by the environment variable `$PLATO_LOCAL_SETTINGS` and is usually defined as follows:

```
$ export PLATO_LOCAL_SETTINGS=/cgse/local_settings.yaml
```

In your code, the Settings are accessed by `Settings` class from the `egse.settings` Python module. This module defines the `Settings` class which is used to access all information in the Settings. You can load the Settings in your session with the following command:

```
from egse.settings import Settings

settings = Settings.load()
```

This will load the global settings first and then the local settings if the environment variable is defined. Remember that local settings will take precedence. You only need to define the settings that actually change for your local installation, respect the full hierarchy when specifying those settings. You are allowed to define new entries at any level in the Settings hierarchy.

In a terminal you can check your settings as follows:

```
$ python3 -m egse.settings
AttributeDict
 ├──── Common-EGSE
 │     ├──── VERSION: 2022.3.0-rc.13+CGSE
 │     └──── RELEASE: November 2022, 2022.3.0-rc.13+CGSE
 ├──── SITE
 │     ├──── ID: CSL1
```

```
 |    ├──── SSH_SERVER: localhost
 |    └──── SSH_PORT: 22
...
Memoized locations:
['/cgse/lib/python/Common_EGSE-2022.3.0rc13+cgse-py3.8.egg/egse/settings.yaml',
'/cgse/local_settings.yaml']
```

The memoized locations are the settings files that have been loaded and memorized. Once the application has started and the settings have been loaded, they can only be reloaded by explicitly forcing a reload as follows:

```
settings = Settings.load(force=True)
```

This does however not guarantee that the settings will propagate properly throughout the application or to client apps. Settings can be saved in local variables or class instances that have no knowledge of a settings reload.

Other things to document XXXXX:

- IP addresses or HOSTNAMES in the global settings shall by default all be set to `localhost`.

Chapter 9. The GlobalState

- What can we do with the `GlobalState`?
- How to use the GlobalState in test scripts?

This section needs to be updated!

9.1. Singleton versus Shared State

From within several places deep in the code of the test scripts, we need access to a certain state of the system and act accordingly. The main state to access is the Setup which provides all configuration and calibration parameters of the test equipment and of the SUT.

Another use is the `dry_run` state which allows you 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.

An alternative to provide such functionality is to pass arguments with the required state into all levels until the level/function where the information is needed, even if the argument is never used in most of the intermediate levels.

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.

9.2. What is in the GlobalState?

The following sections describe the different states and function provided by the GlobalState. Remember the GlobalState is intended to be used within functions and methods of the test scripts in order to access global state information. That means the GlobalState needs to be initialised for some functions to work properly. Don't use the GlobalState in Common-EGSE modules. If you need access to the Setup from the CGSE, explicitly request the Setup from the configuration manager using the `get_setup()` function.

9.2.1. The Setup

You can access the Setup from the GlobalState as follows:

```
from camtest import GlobalState
```

```
setup = GlobalState.setup
```

The GlobalState requests the Setup from the configuration manager which keeps track of the currently active Setup. On system startup however, when the configuration manager is not running, the above code will return `None`. Fix your system startup and load a proper Setup into the configuration manager if this is the case.

You can load the Setup also with the function `load_setup()` and this will then automatically populate the GlobalState:

```
from egse.setup import load_setup

setup = load_setup()
```

From the Setup you can access all devices that are known by the configuration manager, and you have access to configuration and 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. If you need to work with different Setups simultaneously, `GlobalState` is not the right place to be. In this case you should work directly with the `Setup` class. You can get any Setup with the `get_setup()` function without loading that Setup in the configuration manager.

```
from egse.setup import get_setup

setup_67 = get_setup(67)
```

There is a subtle difference between the `get_setup()` and the `load_setup()` function. The `get_setup()` function retrieves the requested Setup from the configuration manager without replacing the currently active Setup in the configuration manager control server. The `load_setup()` returns the requested Setup **and** replaces the currently active Setup in the configuration manager.

Alternatively, you can load any Setup directly from a dictionary or a YAML file using the static methods `from_dict(my_dict)` or `from_yaml_file(filename)` of the `Setup` class.

There is a full section on the Setup, how to populate it, how it is managed and controlled and what it shall contain and what not. Check out the section on The Setup to get more details.

9.2.2. 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 intended arguments. But we don't want the commands to be sent 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.

9.2.3. 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).

Chapter 10. Data Strategy

- ▢ where is the data stored
- ▢ what is the folder structure
- ▢ /data and /archive
- ▢ rsync
- ▢ data propagation

  [ ]

Chapter 11. Date, Time and Timestamps

This section explains how and where time is used in the CGSE code and in the data.

What goes in?

- [x] the format_datetime() function
- [ ] timestamps in the CSV files
- [ ] timestamps in the HDF5 files
- [ ] timestamps in the FITS files
- [ ] timestamps in the log files
- [ ] timestamps in Grafana
- [ ] UTC versus local time
- [ ] time synchronisation → NTP or otherwise

11.1. Formatting the date and time

In the `egse.system` module we have provided a few functions to work with datetime in a consistent way. The most important is the `format_datetime()` function which is used for instance to create the timestamps for all CSV housekeeping data files.

```
>>> format_datetime()
'2022-06-01T07:47:25.672+0000'
```

The format that is returned by default is `"YYYY-mm-ddTHH:MM:SS.µs+0000"` which means the time is expressed in UTC. The function has a few optional parameters that allows to tune the returned string. This should be used only for specific purposes, use the default always to generate a timestamp for your data.

If you need to parse the timestamp string returned by the `format_datetime()` function, use the following format string: `"%Y-%m-%dT%H:%M:%S.%f%z"`. We might provide a `parse_datetime()` function in the future to standardize and simplify the parsing of datetime objects.

```
>>> datetime.strptime(format_datetime(), "%Y-%m-%dT%H:%M:%S.%f%z")
datetime.datetime(2022, 6, 1, 8, 22, 3, 686000, tzinfo=datetime.timezone.utc)
```

If you need a specific date without time, the `format_datetime()` function takes a string argument that you can use to get the date of today, yesterday, tomorrow and the day before yesterday. This might be useful in scheduling tasks.

```
>>> format_datetime('yesterday)
```

```
'20220531'
>>> format_datetime('yesterday', fmt="%d/%m/%Y")
'31/05/2022'
```

Part III — Device Commanding

Chapter 12. Device Control Servers

- Mechanisms
- Sensors
- Heaters
- Facility

12.1. 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

12.1.1. 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 13. The System Under Test (SUT)

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

13.1. DPU Control Server and DPU Processor

The DPU Control Server (used to be called the DPU Simulator in older documents) acts like any other control server. It's Protocol class (`DPUProtocol`) starts a `DPUController` which implements the commands that are defined in the `DPUInterface` class. Specific DPU commands are sent to the control server using the `DPUProxy` class. The `DPUProxy` class also implements the `DPUInterface`.

The difference with normal device control servers lies in the additional sub-process which handles the communication with the N-FEE. This separate process, the `DPUProcessor`, is also started by the `DPUProtocol` and both the `DPUProcessor` and the `DPUController` communicate via three multiprocessing Queues, the command queue, the response queue, and a priority command queue. The main task of the `DPUProcessor` is to communicate with the N-FEE, i.e. command it via SpaceWire RMAP Requests and retrieve housekeeping and data packets that are sent to the Storage manager. The commands that the `DPUProcessor` must execute are passed through the command queue by the `DPUController`. Every readout cycle the `DPUProcessor` checks the command queue and passes any available command to the N-FEE as an RMAP Request.

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 N-FEE however, things are more complicated. The N-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 N-FEE housekeeping packet. That takes just a few milliseconds. Depending on the N-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 N-FEE is done by sending SpaceWire RMAP requests to the N-FEE in that 2s+ timeslot. When we send a command from the Python prompt to the N-FEE, it arrives in the command queue at the `DPUProcessor` and will be sent to the N-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 N-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 N-FEE in the same timeslot.

XXXXX: add here how we solved this!

The priority queue is also a command queue, but the commands are not sent to the N-FEE. Instead, the commands are executed in the `DPUProcessor` and are used to either get information about the state of the N-FEE or set/get the internal state of the `DPUProcessor`. The state of the N-FEE is mirrored and kept up-to-date by the `DPUProcessor` during the readout cycle. Priority commands are typically to request e.g. the N-FEE mode, or to set an internal DPU parameter. While the command queue is checked only during the allowed RMAP communication period, the priority queue is checked and executed several times during the readout cycle.

The DPU Processor has two ZeroMQ message queues on which it publishes specific information. One message queue is the data distribution queue on which the following information is published. All messages published on this queue are multipart messages with the following IDs to which you can subscribe.

SYNC_TIMECODE

Whenever the N-FEE sends a timecode to the DPU, the timecode is published on the message queue as a tuple with (timecode, timestamp). The timecode is an integer number cycled on the sequence from 0 to 63 and incremented on every sync pulse, both 200ms and 400ms pulses. Since the N-FEE does not have an internal clock that keeps the time, we associate a timestamp with the time of reception of the timecode by the DPU Processor. The timecode itself is created and send by the N-FEE to the DPU over the SpaceWire interface within 1µs after receiving the sync pulse from the AEU. Reception at the DPU Processor, creating the timestamp etc. results in a delay of approximate 15ms.

SYNC_HK_PACKET

Right after the timecode, the N-FEE sends out a housekeeping packet. When this packet is received at the DPU Processor, it is published on the data distribution message queue as a tuple with (HousekeepingPacket, timestamp). Again, the timestamp is the time of reception at the DPU Processor.

SYNC_DATA_PACKET

When the N-FEE is in FULL_IMAGE mode with `digitise_en=1` it will send out data packets containing the image data from the readout. Depending on the number of rows that are digitised the N-FEE will send out a variable number of data packets. Each of these packets are published on the data distribution queue as a tuple with (DataPacket, timestamp). The DataPackets can be of type DataDataPackets or OverscanDataPackets.

N_FEE_REGISTER_MAP

At the start of every readout cycle, the DPU Processor puts a Numpy array of type UINT8 on the message queue. This Numpy array is the memory map that contains and defines the Register Map. Note, that the data from this memory map is a copy, a mirror, of the memory map in the N-FEE.

NUM_CYCLES

The internal DPU Processor counter `num_cycles` is used to allow the user to command a number of readout cycles in full image mode. This counter is needed because the N-FEE has no concept of '*number-of-images*', i.e. you can not ash the N-FEE to take 10 images.Nevertheless, we wanted this functionality in our test scripts, and that is what `num_cycles` is. If you command `num_cycles=10` the N-FEE will be put into full image mode, generate image data for 10 readout cycles, and after that instructed to go to DUMP mode (see [dump-mode]).

The data distribution message queue is used by processes that need to handle image data like the DPU GUI. There is a second message queue used by the DPU Processor to publish monitoring information. In addition to SYNC_TIMECODE, SYNC_HK_PACKET, NUM_CYCLES, the following information is published on this message queue:

HDF5_FILENAMES

A list of HDF5 path names that are ready and available for processing.

SYNC_ERROR_FLAGS

On each readout (every 200ms and 400ms pulse), after all housekeeping and data packets have been received from the N-FEE, the DPU Processor reads out the housekeeping memory area from the N-FEE. This memory area is at that time just updated by the N-FEE and therefore can contain new information with respect to the beginning of the readout. Especially the error flags –which are in the housekeeping information– are of interest because they can indicate errors that happened during the readout while the original HK packet might not yet contain any errors. The error flags are sent as a tuple with (error flags, frame counter, timestamp). The error flags is an integer that represents a bitfield of (currently) 16 flags.

Any process can subscribe to the monitoring queue easily by using the DPUMonitoring class. For example, the following code snippet connects to the DPU Processor monitoring queue, waits until the next timecode is received, then executes some code and waits for the next timecode.

```
from egse.dpu import DPUMonitoring

with DPUMonitoring() as moni:
    while moni.wait_for_timecode():
        ...
```

If you need more control over the monitoring and the actions, you can subscribe to the DPU Processor monitoring queue directly. The following code snippet waits for an error flag and raises an alert message on Slack whenever the error flag is not 0:

```
import zmq
import pickle
from egse.zmq import MessageIdentifier
from egse.slack import send_alert
```

```
hostname = "localhost"
port = 30102
sub_id = MessageIdentifier.SYNC_ERROR_FLAGS.to_bytes(1, byteorder='big')
context = zmq.Context()

receiver = context.socket(zmq.SUB)
receiver.connect(f"tcp://{hostname}:{port}")
receiver.subscribe(sub_id)

while True:
    try:
        sync_id, message = receiver.recv_multipart()
        sync_id = int.from_bytes(sync_id, byteorder='big')
        error_flags, frame_counter, timestamp = pickle.loads(message)
        msg = f"{MessageIdentifier(sync_id).name}, {error_flags = } for
{frame_counter = }"
        print(msg)
        if error_flags:
           send_alert(msg)
    except KeyboardInterrupt:
        print("KeyboardInterrupt caught!")
        break

receiver.close(linger=0)
```

13.1.1. The Inner Loop

The inner loop is the workhorse of the DPUProcessor, it has three main functions (1) reading information and data that is provided by the N-FEE, (2) sending that data to the storage manager, and (3) send commands to the N-FEE. In pseudo language, the inner loop performs the following functions:

```
while True:
    try:
        timecode = read_timecode()  ①
        save_timecode(timecode)

        hk_packet = read_housekeeping_packet()  ②
        save_housekeeping(hk_packet)

        update_internals()  ③
        process_high_priority_commands()  ④

        if FULL_IMAGE_MODE:  ⑤
            until last_data_packet:
                data_packet = read_data_packet()
                save_data_packet(data_packet)
```

```
        hk_data = read_hk_data()  ⑥
        save_hk_data(hk_data)

        for each cycle:  ⑦
            save_attributes(obsid, num_cycles, register_map)
            publish_data_and_monitoring_info()
    except:
        report error but do not abort ⑧

    process_high_priority_commands()  ⑨

    if commands_on_queue:
        send_commands_to_nfee()  ⑩
```

1. The first information that the N-FEE sends out is the timecode. The timecode is an integer that cycles through the values from 0 to 63 (a six bit number). The timecode is sent within 2µs from the sync pulse that the N-FEE receives from the N-AEU, but it is not associated with a timestamp. Since the N-FEE has no concept of on-board time, the DPUProcessor will have to associate a timestamp with the timecode. That timestamp is the time of reception of the timecode and it will be typically a few ms delayed. This timestamp is saved in the HDF5 file as an attribute of the timecode.
2. The next thing the N-FEE sends out over the SpaceWire interface is a housekeeping packet. It is sent immediately after the timecode. Also here, the DPUProcessor generates a timestamp at reception and saves this timestamp together with the complete housekeeping packet.
3. The DPUProcessor keeps internal information and bookkeeping parameters in order to mimic the state of the N-FEE FPGA. This information is updated from the register map and is dependent on where we are in the readout cycle, e.g. start of the cycle, long-or-short pulse, etc.
4. Since the DPUProcessor keeps itself in sync with the state of the N-FEE, there is no need to consult the N-FEE about its state over an expensive SpaceWire RMAP request. So, if a user wants to know the operating mode of the N-FEE or other state information like e.g. the current frame number or the synchronisation mode, this information comes directly from the DPUProcessor internal state instead of passing the request through to the N-FEE. That is what high priority commands do, they do not wait for the safe time range where we can send RMAP commands, but return immediately with the answer loaded from the internal state of the DPUProcessor.
5. When the N-FEE is in full image mode, the CCDs are read out and image data is sent out over the SpaceWire. Depending on the amount of data this can take up to about 4 seconds (a little bit more is possible). The amount of image data depends on a few register parameters, namely, `v_start` and `v_end` define the number of rows that are read out, `h_end` defines the number of columns to read out per CCD side, `sensor_sel` specifies which CCD side has to be readout (or both). So, we can read out only part of the CCDs and this will result in less data being transferred and less time to spent in this loop. Reading out only part of the CCD is called *partial readout mode*.

6. When the N-FEE has finished reading out the CCDs and sending out the image data, the housekeeping area in the N-FEE memory map will be updated with the current state of the HK parameters. This updated housekeeping will be important especially for status information and error flags. The error flags can contain information about transmission buffers that overflow, EDAC[1], link errors etc. So, at this point we read out the updated housekeeping data and stores that next to the original HK packet.
7. For each readout cycle we have some additional things to do before we will send RMAP commands to the N-FEE. This pseudo statement might be a little misleading, it's not a `foreach` loop, but a condition that we need to execute the two enclosing statements for every readout cycle (aka frame). So, for each frame we will save important attributes that are used by the data processing, and we will also publish the data we have and any required additional information on the message queue. Other processes can subscribe to the message queue to be informed about the state of the N-FEE or to receive data for visualisation or processing (e.g. the DPU GUI or the FITS generation).
8. In principle, the pseudo commands that we have described upto now define one complete cycle of N-FEE information, i.e. timecode, housekeeping, and data. If we would never have to change the state of the N-FEE and would not send commands over the SpaceWire, the loop would repeat here. Of course, we want to do other things and communicate with the N-FEE and that will be described in the following paragraphs 8-10, but also important is to understand that things can go wrong and exceptions can be raised because of SpaceWire hick-ups, timeouts, disk full errors from the storage, and many more. That is why we put all the previous commands in a `try` clause (the statements between the `try` and `except` keywords). Whenever something goes wrong and an exception is raised, the `except` clause is executed and the error is logged with possibly extra debugging information. At this point we **do not** (as is usually done elsewhere) abort and exit the inner loop, but we continue with the execution of the remaining statements.

   We do not want to abort because the DPUProcessor should never stop running. The idea is that, whenever an error occurs, the remaining statements from the `try` clause are skipped, the error is reported, we have the opportunity for high priority commands or to send commands to the N-FEE, and then cycle through until the next timecode. In case of a link corruption or when the DPU to N-FEE communication is out of sync, it might take a few loops to recover on the next timecode. But, eventually, a next timecode will be detected and the DPUProcessor can recover its communication and its state.

   If we do not receive information from the N-FEE on the SpaceWire, this is handled as an exception and a `NoBytesReceivedError` will be raised, also, waiting for the next timecode can raise a `TimecodeTimeoutError`. In these two cases, we will fall into the `except` clause, but no error will be logged and the execution will continue as normal after the `except` clause. This scenario will happen very frequently while waiting for SpaceWire packets or for the next timecode.

9. After receiving the image data and updated housekeeping its time again to process any high priority commands from the queue.
10. The last thing to do before repeating the inner loop is to send any RMAP commands that are waiting on the command queue. One of the first commands that will be sent by default is to

clear any error flags, after that a command will be read from the queue, processed and send to the N-FEE as an RMAP request. All commands that are sent to the N-FEE are also saved in the HDF5 file.

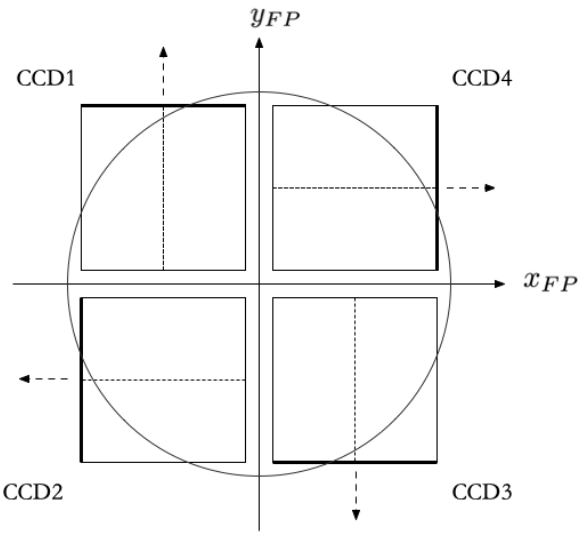
That concludes our inner loop for the DPUProcessor. Please keep in mind that this description is a simplification of the real inner loop and you will need to carefully study the code before making any changes.

13.2. The N-FEE Simulator

TBW

13.3. CCD Numbering

Throughout the PLATO camera test campaigns, the CCD numbering, used throughout the CGSE, will remain the same (see figure below), but the underlying numbering, used by MSSL for the N-FEE will be different for EM, PFM, and FM.



13.3.1. MSSL Numbering

The MSSL numbering of the CCDs is used in:

- the SpW packets with the raw image data;
- the N-FEE register map (used for configuration and queries).

13.3.2. Conversions MSSL CGSE

Originally, the conversion back and forth between the MSSL and CGSE numbering was configured in `egse.fee` in the following variables:

- `CCD_BIN_TO_ID`: for the conversion from MSSL numbering to CGSE numbering (used in the FITS

persistence layer, DPU UI and HDF5 viewer);

- `CCD_BIN_TO_IDX`: for the conversion from MSSL numbering to display index in the DPU UI;
- `CCD_ID_TO_BIN`: for the conversion from CGSE numbering to MSSL numbering;
- ~~`CCD_IDX_TO_BIN`~~.

The `DEFAULT_CCD_READOUT_ORDER` variable holds the default order of the CCD readout, as defined in the N-FEE register.

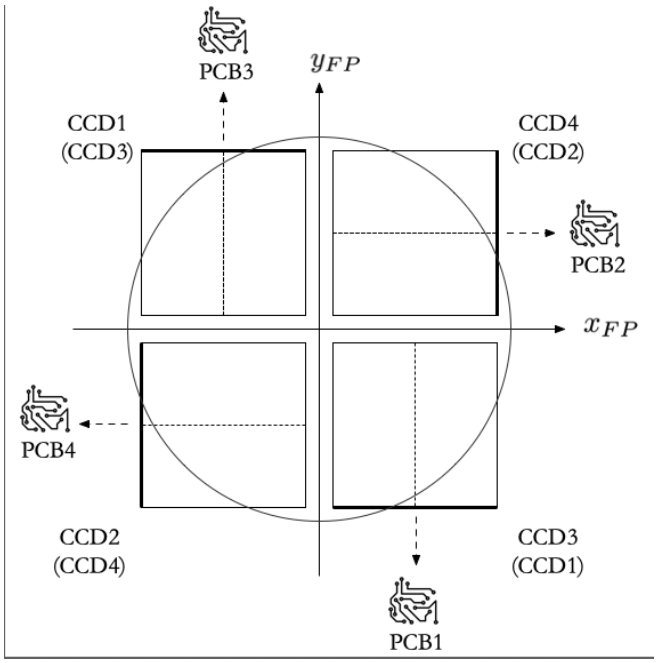
13.3.3. Register map

13.3.4. Moving to the Setup

The idea is that these conversions will be moved to the setup file, under `setup.camera.fee.ccd_numbering`. The name of the relevant parameters is the same as above, but in small case.

13.3.5. EM

The CCD numbering in the CGSE and the connections to the PCBs for EM are shown in the figure below. Between brackets are the CCD numbers by MSSL. Not only do the CCD numbers not match between MSSL and the CGSE, but the connections between the CCDs and the corresponding PCBs is incorrect. The latter will be corrected for in PFM.



Reference Documents

The entries in the N-FEE register map for EM are stipulated in the following reference document:

N-FEE ICD	PLATO-MSSL-PL-ICD-0002 (issue 12.0, 05/05/2022)
Register map (Attachment)	PLATO-MSSL-PL-PL-FI-0003_1.0_N-FEE_EM1_EM2_Register_MAP.xlsx

Common EGSE

CDD conversions

- CCD_BIN_TO_ID: [3, 4, 1, 2]
- CCD_BIN_TO_IDX: [2, 3, 0, 1]
- CCD_ID_TO_BIN: [0, 0b10, 0b11, 0b00, 0b01]
- ~~CCD_IDX_TO_BIN: [0b10, 0b11, 0b00, 0b01]~~

CCD IDs

- CCD1: 0b10
- CCD2: 0b11
- CCD3: 0b00
- CCD4: 0b01

Default readout order

Deviations from the N-FEE ICD

Incorrect connection between the PCBs and the CCDs:

- According to the spreadsheet for EM (see above), the default CCD readout order is 228 (decimal value). This is supposed to correspond to the requested CCD readout scheme 1 - 2 - 3 - 4 (e.g. when going to dump mode), but - due to the incorrect connections between the PCBs and the CCDs - this actually corresponds to 3 - 4 - 1 - 2.
- To compensate for this, the default CCD order as configured in the setup does not correspond to what is in the spreadsheet: instead of using 0b11100100 (228), we use 0b01001110 (0x4E or 78).

CGSE Configuration

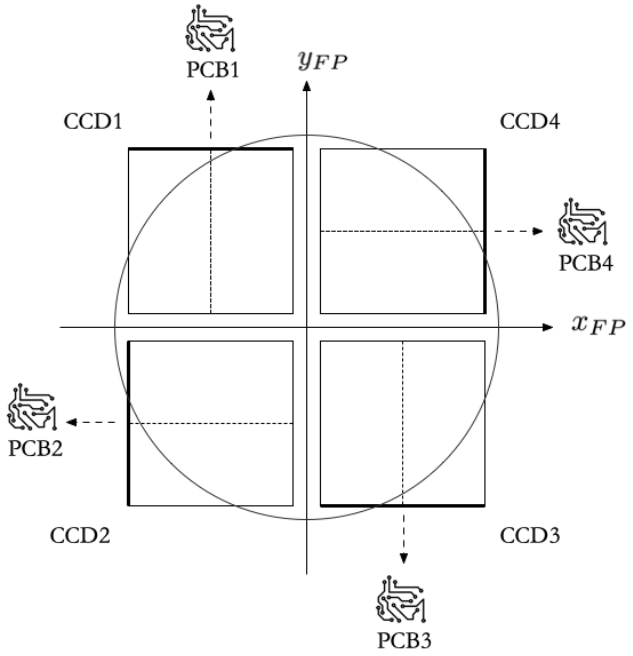
- Register map: plato-cgse-conf/data/common/n-fee: n_fee_register_em_v2.yaml
- HK map: plato-cgse-conf/data/common/n-fee: n_fee_hk_em_v2.yaml
- Sensor calibration: plato-cgse-conf/data/common/n-fee: nfee_sensor_calibration_em_v3.yaml

13.3.6. PFM

For PFM, the following modifications will be applied w.r.t. EM:

- The CCDs will be connected to the PCB with the same number;
- Part of the entries in the N-FEE register map will move places.

The CCD numbering and the connections to the PCBs for PFM are shown in the figure below. Note that the underlying MSSL numbering is the same as for EM.



Reference Documents

The entries in the N-FEE register map for PFM are stipulated in the following reference document:

N-FEE ICD	PLATO-MSSL-PL-ICD-0002 (issue 12.0, 05/05/2022)
Register map (Attachment)	PLATO-MSSL-PL-PL-FI-0003_1.0_N-FEE_PFM_Register_MAP.xlsx

Common EGSE

CDD conversions

- CCD_BIN_TO_ID:
- CCD_BIN_TO_IDX:
- CCD_ID_TO_BIN:
- ~~CCD_IDX_TO_BIN:~~

CCD IDs

- CCD1:
- CCD2:
- CCD3:
- CCD4:

Default readout order

CGSE Configuration

- Register map: plato-cgse-conf/data/common/n-fee: n_fee_register_pfm_v1.yaml
- HK map: plato-cgse-conf/data/common/n-fee: n_fee_hk_pfm_v1.yaml
- Sensor calibration: plato-cgse-conf/data/common/n-fee: nfee_sensor_calibration_em_v3.yaml

---

13.3.7. FM

The CCD numbering and the connections to the PCBs for FM are the same as for PFM.

Reference Documents

The entries in the N-FEE register map for FM are stipulated in the following reference document:

N-FEE ICD	PLATO-MSSL-PL-ICD-0002 (issue 12.0, 05/05/2022)
Register map (Attachment)	PLATO-MSSL-PL-PL-FI-0003_1.0_N-FEE_PFM_Register_MAP.xlsx

Common EGSE

CDD conversions

- CCD_BIN_TO_ID:
- CCD_BIN_TO_IDX:
- CCD_ID_TO_BIN:
- ~~CCD_IDX_TO_BIN:~~

CCD IDs

- CCD1:
- CCD2:
- CCD3:

- CCD4:

CGSE Configuration

- Register map: plato-cgse-conf/data/common/n-fee: n_fee_register_fm_v1.yaml
- HK map: plato-cgse-conf/data/common/n-fee: n_fee_hk_fm_v1.yaml
- Sensor calibration: plato-cgse-conf/data/common/n-fee: nfee_sensor_calibration_em_v3.yaml

Default readout order

[1] EDAC stands for Error Detection And Correction

Chapter 14. Device Simulators

- Different ways to write a simulator
  - implement a Simulator class that can replace a Controller class
  - implement a simulator process

14.1. The OGSE Simulator

The OGSE simulator by default listens on TCP port 4181 for incoming connections from the OGSE Controller. The controller can be used stand-alone in a Python REPL for testing or can be part of the OGSE Control Server as the last step in the commanding chain.

Start the OGSE simulator as follows:

```
$ ogse_sim start
```

The OGSE simulator will listen for incoming connections. There can be only one process connected to the OGSE simulator. This behaviour is similar as the hardware device, which only accepts one connection. When a connected process disconnects, the OGSE simulator will accept a new connection.

The OGSE simulator can be killed by pressing CTRL-C in the Terminal where the simulator is running. Alternatively, you can send a TERM or HUP signal to the process:

```
$ ps -ef|grep ogse
459800007 21334  1908   0  2:20PM ttys002    0:01.02
/Library/Frameworks/Python.framework/Versions/3.8/Resources/Python.app/Contents/MacO
S/Python /Users/rik/git/plato-common-egse/venv38/bin/ogse_sim start
$ kill -TERM 21334
```

The OGSE simulator has of course no interlock jack that you can pull out to immediately power off the system. Nevertheless, we can simulate this behaviour by sending a user signal to the `ogse_sim` process. Sending a USR1 signal will open the interlock when it is closed, and close it when it is open. The USR1 signal works as a toggle for the interlock.

```
$ kill -USR1 <PID>  ①
```

① replace `<PID>` with the correct process identifier

Check the interlock state with the command method `get_interlock()`:

```
>>> ogse = OGSEController()
>>> ogse.connect()
```

```
>>> ogse.get_interlock()
'interlock: OPEN'
```

The following commands have been implemented in the simulator:

status()	return the status of the power, lamp, interlock, laser, power meters, and attenuator
get_interlock()	state is OPEN or CLOSE
get_power()	state is On or OFF
get_lamp()	state is ON or OFF
get_laser()	state is ON or OFF
get_lamp_fault()	state is ERROR or NO-ERROR
get_controller_fault()	state is ERROR or NO-ERROR
get_psu()	state is ON or OFF
get_operate()	state is On or OFF
get_flags()	the state of all parameters encoded in a single number formatted in hexadecimal and binary
get_power_and_temperature()	returns the power and temperature reading of both power-meters
ldls_status()	returns the state of the connection to the LDLS device, state is OK or ERROR
pm_status()	returns the state of the connection to the power-meter devices, state is OK or ERROR
att_get_level()	returns a dictionary with the following keys: att_moving [bool], att_factor [float], and att_index [int]
att_status()	returns the state of the connection to attenuator device, state is OK or ERROR
att_set_level_index(<index>)	Sets attenuator to the level closest to <index>
att_set_level_factor(<factor>)	Sets attenuator to the level closest to <factor>
att_set_level_position(<wheel1>, <wheel2>)	sets the two filter wheels to the given position, each wheel has 8 positions, allowed values are 1 to 8.
att_level_up()	selects the attenuation one step higher than the current value, it has no effect if the current level is already the highest
att_level_down()	selects the attenuation one step lower than the current value, it has no effect if the current level is already the lowest
version()	returns the version of the hardware controller software
power_on()	turns the power supply on

power_off()	turns the power supply off
operate_on()	turns the laser on
operate_off()	turns the laser off
exit()	
quit()	
reset()	

Part IV — Monitoring

Chapter 15. Metrics, Prometheus and Grafana

The CGSE provides mechanisms for monitoring metrics from different processes and devices with Grafana. This chapter will explain how the metrics are defined for your favorite process and device, how their values are gathered and propagated to the Prometheus server and how Grafana queries Prometheus to create timeseries that are displayed in your browser.

15.1. Setup Prometheus

- ☐ what should go in the `prometheus-egse-server.yml`
- ☐ what about the timestamps that are associated with the metrics?

15.2. Define your metrics

Depending on your needs, you can choose different methods for the timestamp of your metrics. We will discuss three ways to get and use metrics timestamps:

1. the timestamp for the metric comes with the device housekeeping.
2. the timestamp is created by the device controller when the metric is retrieved.
3. the timestamp is created by Prometheus and is the time of querying the http server that is started by each of the control servers.

Option 1. is the best way for timestamps for your metrics when the accuracy of the time when the metric was retrieved is important. The timestamp then itself is a metric (Gauge) and might need to be converted to a float that represents the time in the expected way, depending on the target application.

```
import prometheus_client as prom

x = prom.Gauge("x", "the metric")
x_ts = prom.Gauge("x_ts", "the metric timestamp")

# Option 1: timestamp is a metric

x.set( <the value of the parameter> )
x_ts.set( <the timestamp for the value as a float> )
```

Option 2. can be used when the device housekeeping doesn't contain the timestamp of the metrics. This method can be a few to a few tens of milliseconds off depending on the device response time. For some devices and depending on the duration of the query and the number of parameters this can be even longer. One example is the DAQ6510 when tens of temperatures are scanned several times before returning an average value. A full scan can take up to 20s or more.

```
import prometheus_client as prom
import datetime

x = prom.Gauge("x", "the metric")
x_ts = prom.Gauge("x_ts", "the metric timestamp")

# Option 2. timestamp is set by the device controller

x.set( <the value of the parameter> )
x_ts.set(datetime.datetime.now(tz=datetime.timezone.utc).timestamp())
```

Option 3. is used when the exact time is not really important and may be a few (tens of) seconds off. This is the case for most temperature measurements evolving slowly.

The time used by Prometheus is the unixtime which means localtime (not UTC) and Epoch 01/01/1970, excluding leap seconds. This is the same as what is returned by the Python `time.time()` function.

Part V — Test Scripts

This section of the development manual is about developing test scripts to run the PLATO camera performance tests. If you were looking for unit tests, see Part VI — Unit Testing and beyond.

Chapter 16. Observations

- ▢ What are observations?
- ▢ Why do we need them?

Chapter 17. Building Blocks

- [x] what are building blocks?
- [x] why do we need them?
- [x] where do they live?
- [x] how are they implemented?
  - only keyword argument
- [x] how are they executed?
  - all keyword arguments must be explicitly provided → explain why
- [ ] generate_command_sequence()????
- [ ] need to be executed within an observation (between start_ and end_observation() or by execute()) to enforce the creation of a unique obsid.

A building block is a Python function implementing the commands corresponding to a logical entity within a test or denoting an entire test (called the “test-script”). The function is decorated with a `@building_block` and has several benifits and restrictions that will be explained in this section. A typical small building block is shown below:

```
@building_block
def set_trp1(temperature: float) -> None:
    """
    Set the temperature setpoint for TRP1.

    Args:
        temperature: Temperature setpoint for TRP1 [°C].

    Raises:
        If a task is already running in the TCS EGSE, an Abort is raised.
    """

    if is_task_running():
        raise Abort("Cannot change the TRP1 setpoint, a task is running on the TCS")

    tcs: TCSInterface = GlobalState.setup.gse.tcs.device
    tcs.set_parameter(name="ch1_tset", value=temperature)
    tcs.commit()
```

A building block can only run within the context of an observation and all the arguments need to be explicitly provided as keyword arguments. The reason for this is the concept of *explicit is better than implicit* which definitely holds for commanding test equipment and certainly a million dollar PLATO Camera. The requirement to explicitly provide all arguments with their

name reduces the chance for making errors and prevents changes in argument defaults. The above building block will therefore be executed as follows:

```
from camtest.commanding import tcs

start_observation("Configure the TCS EGSE")
...
tcs.set_trp1(temperature=-70.0)
...
end_observation()
```

All functions that result in a change in the systems behaviour or a change in the test equipment shall be decorated as a building block. Status functions and so-called getters usually are not building blocks.

Building blocks implement some safeguards, imposing a number of limitations on the code:

- Building blocks cannot be called recursively. Beware of building blocks calling other building blocks. Avoid too many layers. Avoid functions calling building blocks.
- At run time, the names and the values of every argument of a building block must be explicitly given, even if the argument is `None`. Building blocks with many parameters are hence strongly discouraged.
- Positional arguments are rejected in building blocks, all arguments must be specified as keyword arguments.
- All arguments must be given the default value None in the building block definition.

The building block concept is part of the PLATO Test Scripts and not of the Common-EGSE. All building blocks that are defined will therefore live in the `plato-test-scripts` repository, mostly in the `camtest.commanding` module. The `@building_block` decorator itself is defined in the module `camtest.core.exec`.

The building block decorator is defined as `def building_block(func: Callable) → Callable:` which means the decorator must be used without brackets `()` or arguments. The function which is decorated must be defined with keyword only arguments that all have a default value of `None`.

The building block decorator performs the following actions and checks for the function at import time:

- the decorated function `func` is attributed as a building block, i.e. the `func` will get the attribute `BUILDING_BLOCK_FUNC` set to `True`. This is internally used in the `execute` function to ensure the executed function is a building block.
- the decorator will check if a YAML file exists with the definition of default keyword arguments. The location of the YAML file is fixed.

Possible future change

> Consider putting building block defaults in one *defaults.yaml* file located at the same location as the building block definition. That would create one file for all defaults at that location, instead of one file for each building block. What to do when two building blocks have identical names, but defined in a different module?

The building block decorator performs the following actions and checks for the function at runtime:

- check that no positional arguments are passed into the function,
- fill the default value for keyword arguments that were not passed into the function, provided the function has defaults,
- check if there are still missing arguments, required keyword arguments that have no default and are not provided in the call.

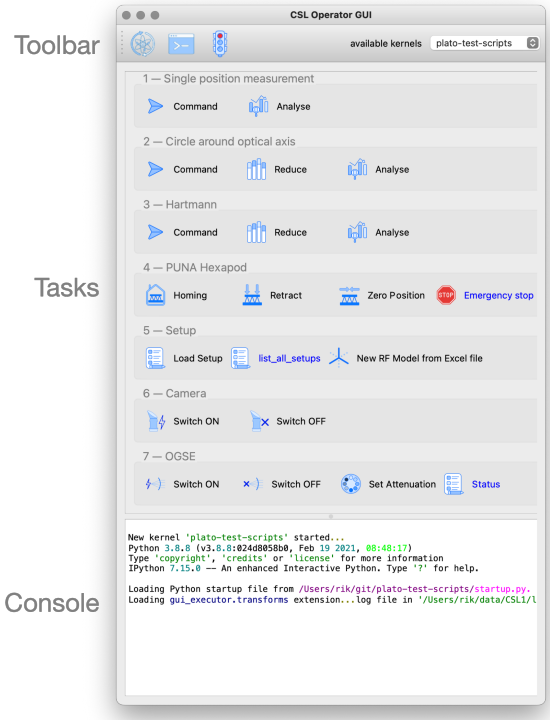
Then, before the function is actually called, the `start_building_block()` function is called. This will register the building block in the observation context and add the command to the command sequence in the GlobalState if this is a dry run. A corresponding `end_building_block()` will de-register the function from the observation context and the command sequence.

The overhead for a function decorated as a building block is about 350µs.

- [ ] higher level test house independent commanding scripts

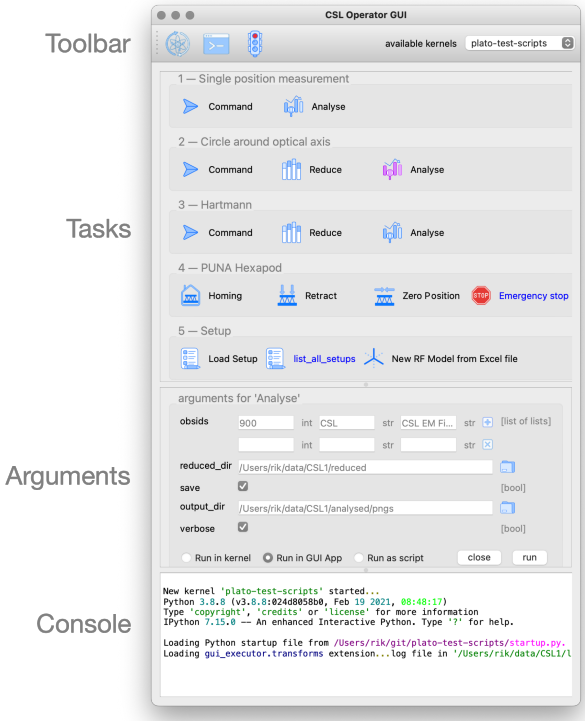
Chapter 18. The Tasks GUI

The Tasks GUI is a generic GUI that allows the test operator to execute standard tasks by clicking the task button and provide the necessary arguments. An example of such a Task GUI is given in the screenshot below.



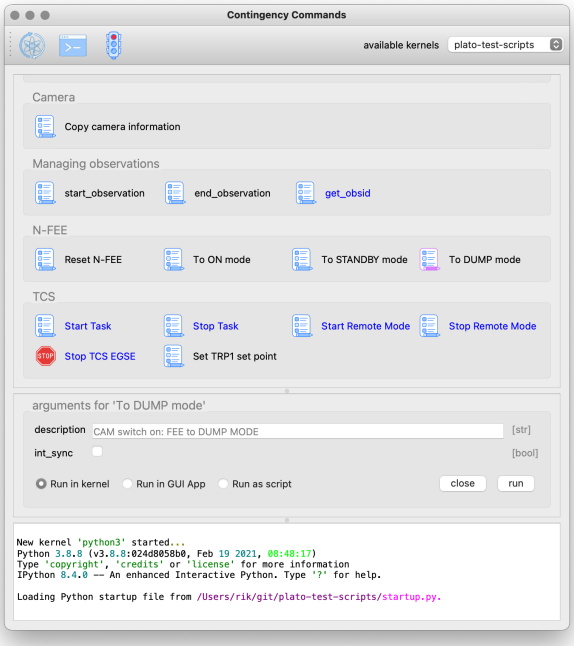
The Task GUI is organised in three panels that are arranged vertically in the window. The top panel is the toolbar to control the Jupyter kernel, the panel at the center contains all the task buttons organised by category, the bottom panel is the console where the output and error messages of the tasks will appear.

When you click on a task button, a fourth panel will appear above the console panel. This is the arguments panel where you can specify all arguments required for the task. Default arguments are shown in gray and can be left to use or overwritten to change. The arguments panel of the selected 'Analyse' task for the 'Circle around optical axis' group is shown below.



This 'Analyse' task accepts 5 arguments: a list of observation identifiers,the location of the reduced data, an option to save the generated plots in a given output folder, and a flag to make the tasks output more verbose. Since this task will produce plots, we want to execute it as a GUI App. Finally, the 'Close' button hides the arguments panel again and deselcts the task button, and the 'Run' button executes the task.

Other task GUIs will be developed for specific purposes. The *CSL Operator GUI* is specifically developed for CSL to execute alignment and focusing measurements. Other test houses will develop their own tasks and we have also generic task GUIs like the *Contingency GUI* that is used at all test houses and quickly switches devices on or off, and allows to perform simple device configurations. An early version of the GUI is shown below, you can clearly see this GUI has the familiar layout and functionality of any Task GUI.



In the rest of this chapter we will explain how these GUIs are created and how you can develop your own version for your specific tasks.

18.1. The package Layout

To build up the Task GUI, we distinguish the task button as a function, several of these functions can be grouped in a Python module (a `.py` file) and all the modules plus additional information needed for the Task GUI is kept in a Python package. The *CSL Operator GUI* shown above, is located in the package `camtest.csl` and has the following layout:

```
camtest.csl
    ├──────────────── __init__.py
    ├──────────────── camera.py
    ├──────────────── circle.py
    ├──────────────── hartmann.py
    ├──────────────── hexapod.py
    ├──────────────── huber.py
    ├──────────────── icons
    ├──────────────── ogse.py
    ├──────────────── refmodel.py
    └──────────────── single.py
```

Each of these `.py` files form a group of buttons in the above Task GUI. The `__init__.py` file is special, it defines `camtest.csl` as a package, and it defines the command to start the Task GUI [see XXXXX]. The `icons` folder contains the graphics for the task buttons and the application icon.

18.2. Defining a Task

Let's build our own simple Task GUI and start with the most stated and useless function, *Hello, World!*. We will eventually build a Task GUI with tasks of increasing complexity and guide you through the different steps.

Create a folder `yakka`[1] that will be our Task GUI package. In the folder create an empty file `__init__.py` and a file named `hello.py`.

```
yakka
    ├──────────────── __init__.py
    └──────────────── hello.py
```

The `hello.py` file shall contain the following code:

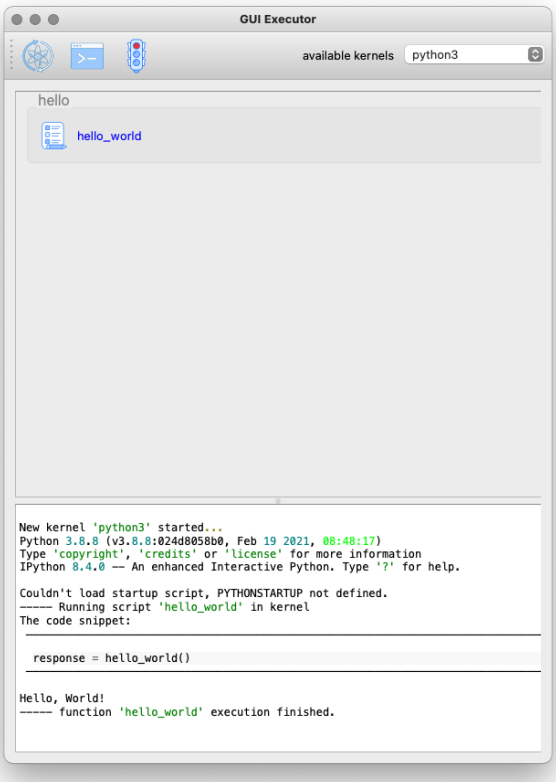
```
from gui_executor.exec import exec_ui

@exec_ui(immediate_run=True) ①
def hello_world():
    print("Hello, World!")
```

① Each task button in the Task GUI is actually a function that is decorated with the `@exec_ui` decorator.

Make sure you are at the same directory level as the `yakka` folder and the execute the following command from your terminal. That will start the Task GUI as shown in the screenshot below.

```
PYTHONPATH=. gui-executor --module-path yakka
```



We see the task appearing in the screenshot above. The task text is blue which means it will run immediately when clicked. The tasks name is the name of the function and the task group name is the name of the `.py` file. The icon is the standard icon used for the task buttons. When you click the task button, the Console shows the following output:

```
----- Running script 'hello_world' in kernel ①
The code snippet:
 ──────────────────────────────────────────────────────────────────────────────
──
  response = hello_world()  ②
 ──────────────────────────────────────────────────────────────────────────────
──

Hello, World! ③
----- function 'hello_world' execution finished. ④
```

What do we see in this output:

① The script is run in the kernel, that is the Jupyter kernel which is started when the Task GUI starts up. By default, the 'python3' kernel is used as you can see in the toolbar.

② The code snippet that is run is shown between two horizontal lines. We see that the function is called without arguments and the return is catched in the variable `response` (which is `None` since the function doesn't return anything).

③ The string 'Hello, World!' is printed when the function was executed.

④ A mark that the function execution has finished.

Let's add another task that takes an argument 'name' as a string with the default value of "John".

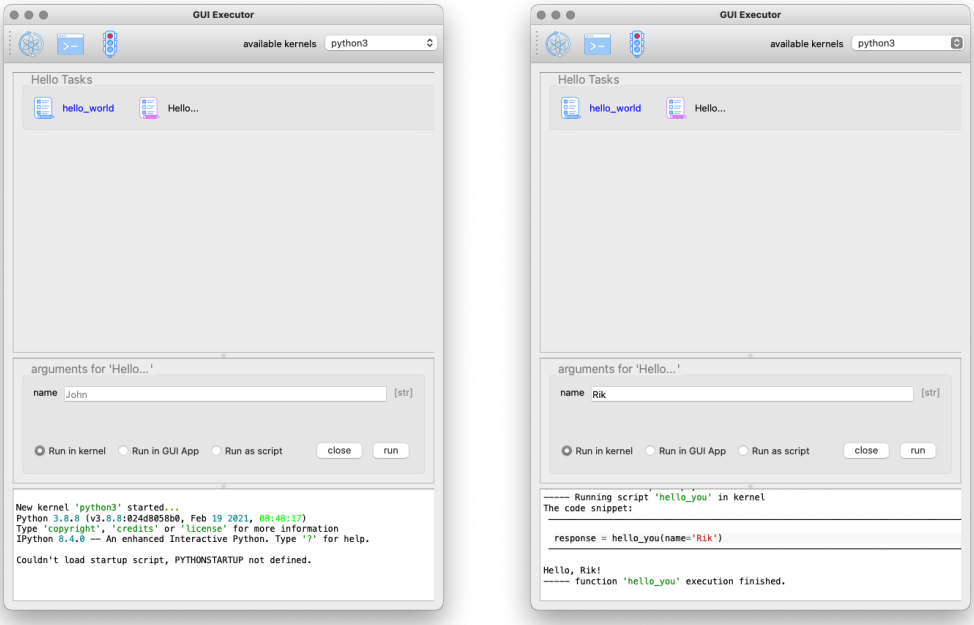
```
from gui_executor.exec import exec_ui

<strong>UI_MODULE_DISPLAY_NAME = "Hello Tasks"</strong>  ①

@exec_ui(immediate_run=True)
def hello_world():
    print("Hello, World!")

<strong>@exec_ui(display_name="Hello...")  ②
def hello_you(name: str = "John"):  ③
    print(f"Hello, {name}!")
</strong>
```

1. if the global variable `UI_MODULE_DISPLAY_NAME` is defined, its value will be used as the name of the group of buttons in this module.
2. You can give the task button a proper name instead of the function name.
3. The type hint for the argument and the default value are used in the arguments panel in the GUI.



In the screenshot above, you can see the effect of the small changes we made in the `hello.py`. The tasks button group is now called 'Hello Tasks' and the new task we added got the 'Hello...' name instead of the function name. The new task icon has a different color because it's is selected. You can also see in the arguments panel that the type hint is picked up and shown in grey and the default name is also filled in grey in the text field. When I put my name there and press the 'Run'

button, you can see that the function is called with the proper argument.

18.3. The @exec_ui decorator

TBW

- immediate_run
- description
- display_name
- use_kernel, use_gui_app, use_script_app
- input_request
- icons

18.4. Type Hints and Defaults

TBW

- supported type hints
- defining a new type hint
- ListList
- Callback

18.5. Choosing your icons

TBW

- PNG or SVG
- normal and selected
- what when `immediate_run=True`

18.6. The Jupyter Kernels

TBW

- Switching kernels and why
- creating a new kernel

18.7. Running your tasks

TBW

- Running in the kernel

- Running as a script
- Running as a GUI App

18.8. The __init__.py

TBW

- creating a function to start the GUI in the background
- add the command to the `setup.py`

[1] 'yakka' means 'work' in informal Australian English

Part VI — Unit Testing and beyond

Chapter 19. The Test Suite

- pytest

Chapter 20. Testing device Interfaces

There are several ways to test device interfaces since we have layered access to the devices. The direct access to a device is usually implemented in a `<device_name>_devif.py` module. If the device has an Ethernet connection, the class is usually called `<device_name>EthernetInterface`, e.g. `OGSEEthernetInterface`, for USB connected devices this would be `<device_name>USBInterface`, e.g. PM100AUSBInterface`.

When you have direct access to the device and it is not used in an operational environment, you can test the interface from the `Controller` class

Chapter 21. 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.

21.1. 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?

21.2. 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

21.3. 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.

21.4. 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 VII — Miscellaneous

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

Chapter 22. 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.

22.1. 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
  - 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`
- Textualog, see Section 5.1

22.2. Often used terminal commands

Command	Description
`ps -ef|egrep "_cs|das|fov|fee|fits"`	Check which control servers are running. The names of all control servers end with _cs, except `das`, `fov_hk`, `n_fee_hk`, and `fitsgen`.
`kill [-9] PID`	Terminate the command with the given process identifier (PID). The `-9` option is used to force a kill.
`kill %1`	After you have sent a command to the background with `^Z`, this command must still be killed. The `%1` represents the jobs number.
`df -h /data /archive`	Check the available disk space on the data and archive disks. The `-h` option given numbers in a human readable format.

Command	Description
`curl localhost:<port>`	Most of the control servers generate metrics data that will be ingested in a Prometheus time-series database. The metrics can be inspected with this `curl` command. Run the command on the egse-server or replace `localhost` with the egse-server IP address. The `port` is dependent on the metrics that you want to inspect, each control server has a specific metrics port.

22.3. Often used git commands

The `git` commands need to be executed in the project folder or a sub-folder thereof, e.g. `~/git/plato-common-egse`.

Command	Description
`git status`	Check the status of the working directory. This is mainly used to list the files that have been changed and need to be added and committed before pushing to the repository. Sometimes these files need to be stashed before an update.
`git stash [pop]`	Temporarily saves (stashes) the files that were changed in your working directory. Use this command to clear your working directory before updating the project. The `pop` option is used to put back the saved files after update.
`git describe --tags [--long]`	Print the most recent tag which represents your installed version or release. If the `--long` option is used, the additional information is the number of commits since the tag and the abbreviated commit hash.
`git remote -v`	Check the remote repositories that are known on your local copy.
`git branch [-v] [-vv]`	Print the branches known on your local repo. The current branch has an asterisk '*' in front of its name. The `-v` option adds information on the last commit, when doubling this option, `-vv`, also the remote tracking branch will be printed.
`git fetch updates`	Fetch all changes from the remote `updates`
`git rebase updates/develop`	Apply the changes from `updates/develop` on the current branch. This is equivalent to merging.

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**.

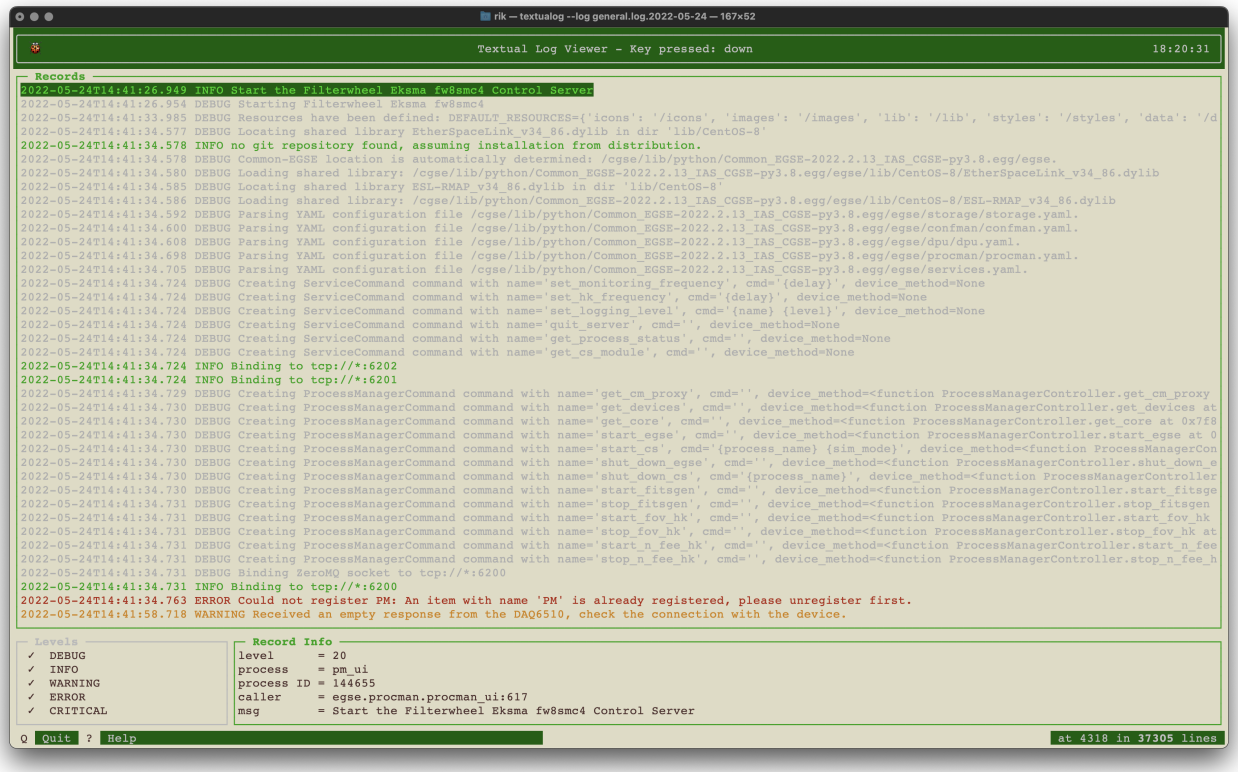
Chapter 23. Stories

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

23.1. 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:



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:

PYTHONPATH PLATO_CONF_DATA_LOCATION PLATO_CONF_REPO_LOCATION PLATO_DATA_STORAGE_LOCATION PLATO_LOG_FILE_LOCATION LD_LIBRARY_PATH

```
[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 24. 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 25. System Configuration

- 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?

25.1. 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)

25.2. 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..

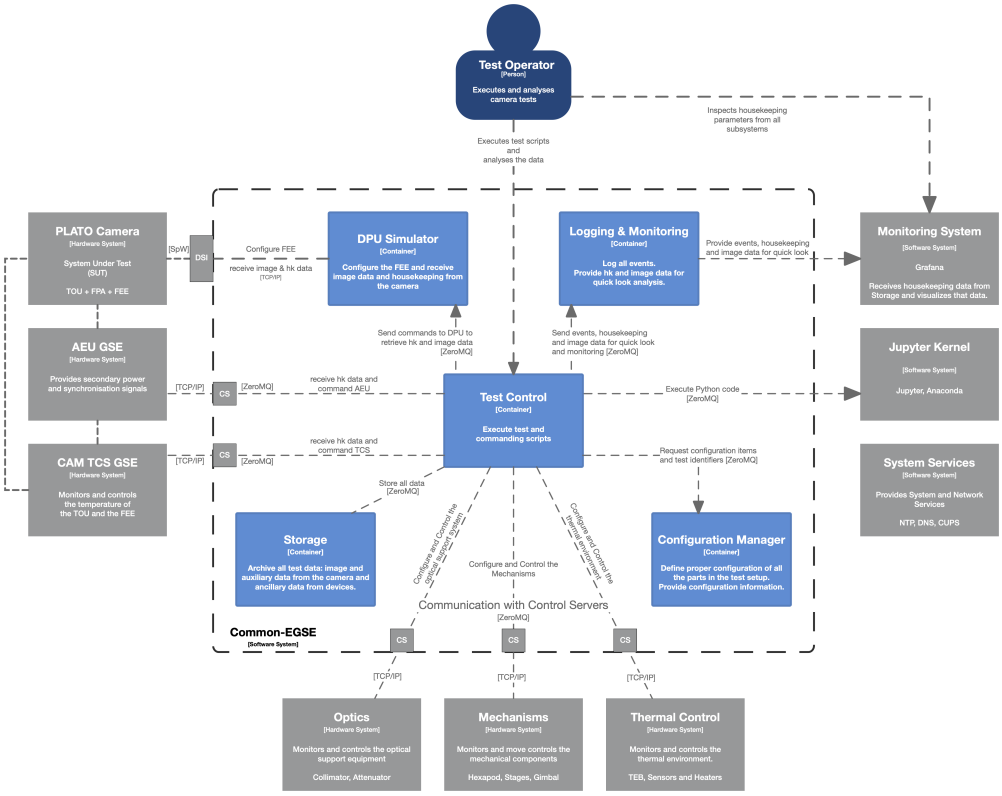


Figure 1. The main components of the Common-EGSE (bleu) and how they interface to each other (internal interfaces) and to the test setup (grey) and the PLATO camera (SUT) (external interfaces).

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 26. 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 27. 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.

27.1. 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

27.2. 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