Manual for the MOONS FPU grid driver Python library module

(EtherCAN interface version v2.1.0)

Johannes Nix

September 23, 2019

Contents

1. Introduction

1.1. Related documents

[1] VLT-TRE-MON-14620-1018,	FPU Verification Requirements
[2] VLT-TRE-MON-14620-3017,	MOONS Fibre Positioner Verification Software Design
[3] VLT-TRE-MON-14620-3016,	Fibre Positioner Communication Protocol
[4] FPUMovementSafetySpecification	Specification of safety levels for FPU movements
[5] VLT-ICD-MON-14620-4303	RFE to Instrument Software Interface Control Document

1.2. Purpose of this document

This document describes the Python programming interface for the MOONS Fibre Positioner Unit EtherCAN interface module, protocol version 2.

The EtherCAN interface allows to control and move Fibre Positioner Units (FPUs) in the MOONS FPU grid by a software interface. This programming interface is implemented in C++. It is designed to be used in several different contexts: Primarily, for simultaneously controlling more than 1000 FPUs in the final MOONS astronomic instrument. Important secondary uses are verifying and calibrating the components of the positioning system, developing and testing of algorithms for complex higher layers of the instrument control system, and engineering and development of related systems and verification software such as image analysis. For this secondary uses, a Python module is available, which derives the lion's share of its functionality by the C++ EtherCAN interface, while adding a small amount of functionality which is only required for testing, or still experimental.

Using the final verification software will not require the users to know how to program Python. However, several parts of the verification software system and the MOONS instrument control software (ICS) are initially developed and tested in Python. The Python interface module documented here has the function to make the functionality of the C++ EtherCAN interface accessible from Python. As such, the goal of this document is to describe a toolbox which can be used to manipulate and test the FPUs as easily as possible, while the verification software is being developed. The underlying hardware protocol will only be exposed as much as necessary to understand the possible states of the hardware.

1.3. Required knowledge

What will be helpful to know before It is assumed that the reader has basic knowledge of Python, or, alternatively, a similar object-oriented language, like Java or C#. In the case that using Python scripts is new to the reader, it is probably helpful to consult the tutorial for Python 2.7 at `https://docs.python.org/2.7/tutorial/`. Both the current version of the EtherCAN interface as well as this document use Python 2.7. Python 2, which is reaching its end of life in 2020, has some differences to Python 3. However this description will proceed in a way that differences to Python 3 are minimised.

Also, in order to retrieve the current version of the library, the user needs to have a minimum knowledge of either subversion or the git version control system. This is explained, for example, in "getting a git repository[1]".

1.4. Order of presentation

The presentation is divided in four parts, as follows:

1. The first part represents a tutorial aiming at explaining quickly how to use the software. This also introduces basic concepts which are essential for understanding the subsequent manual part. In section 2, a minimal example is presented. As additional required knowledge, in section 3 the rules for using the software protection layer are explained. It is recommended to read all of these sections before starting to operate the FPUs, and make sure that the datum switches work before using any datum operation.

2. The second part forms a manual which provides more complete information on how to operate the FPUs, and how to solve problems. In section 4, it is explained how the EtherCAN interface module is configured. The following sections are based heavily on the information given in the tutorial: Section 5 describes an important procedure which verifies that the datum switches are working. Section 6 explains how to perform a number of common tasks. Section 7 explains how errors are handled, and can be processed in scripts. Section 9 details how comprehensive logs of the commands send to each FPU can be accessed and interpreted. Section 8 explains how the FPU health and performance counters are accessed, and what they mean. Section 10 explains how collisions and limit switch breaches are handled. The final section of that part, section 11, explains how multiple FPUs can be handled, especially in complex recovery scenarios.

3. The third part is the reference part, section III. It lists all available commands with a comprehensive list of all their options, and also documents the hierarchy of exceptions. Additionally, section 24 contains a detailed reverse chronological list of changes to commands.

4. Finally, there is an Appendix part. It lists additional examples (section A), gives a full list of software dependencies and how to install the software from scratch (section B), and explains how to run the test gateway (section C).

[1]https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository

1.5. Compatibility

The driver software described in this manual uses the CAN protocol 2, and requires a matching firmware version for protocol 2 in order to work. Firmware versions for protocol 1 and 2 cannot be mixed, and an FPU with firmware version 1 will not respond correctly to a version 2 driver.

A conceptually large internal change is that the state of the FPU is now tracked by the FPU firmware, not more by the driver software. This has the visible effect that even when the driver program was terminated, it is not necessary to initialise FPUs again if they were not powered down.

For the Python interface however, there are very little visible changes:

- The `getPositions()` command has been completely replaced by the `pingFPUs()` command.
- The `getCounterDeviations()` command is obsolete, because after a `findDatum()` operation, the residual error is reported automatically and stored in the FPU state.
- Recovery of alpha limit breaches is substantially easier (and should also, due to changes to the movement ranges, be much safer).
- All movement operations transmit the last direction of movement, and store it in the FPU state struct.
- FPUs with aborted movements can be recovered by using the new `enableMove` command.
- There is a new `checkIntegrity()` command which returns a checksum of the FPU firmware.
- FPUs can now be locked and unlocked, which is needed in instrument operation.
- The serial numbers and firmware versions do not need to be retrieved repeatedly, they are also stored in the FPU state data.
- A further small visible change is that error codes of CAN commands have different symbol names. Because this part of the API is now more or less stable, these symbols may be accessed via the `last_status` flag, and used in test scripts. The driver error codes and the exception hierarchy which were already exposed in version 1 are unchanged.

Part I.

Tutorial

Note: This tutorial part assumes that the driver software is already installed and configured to run. When this is not the case, section 4 on page 18 explains how to do this.

2. A minimal example

Contents

The following listing shows a minimal example of the EtherCAN interface usage, and explains the underlying concepts step by step.

Importantly, the example assumes that the FPUs have been powered on before, and the datum switches have been verified to work reliably. Section 5 on page 27 explains how the proper operation of the datum switches can be verified. This check should be always done *before* any `findDatum()` command is sent, because non-working or damaged switches can lead to immediate damage of the FPU.

```python
from __future__ import print_function
import FpuGridDriver
from FpuGridDriver import TEST_GATEWAY_ADRESS_LIST as gw_address
from fpu_commands import list_positions, \
       list_deviations, list_states, gen_wf
# print the driver version, just to check
print("The FPU driver version is:", FpuGridDriver.__version__,
      ", CAN PROTOCOL version:", FpuGridDriver.CAN_PROTOCOL_VERSION)

NUM_FPUS = 1 # set number of FPUs to 1
gd = FpuGridDriver.GridDriver(NUM_FPUS)
# connect to gateway with default IP 192.168.0.10
print("connecting grid:", gd.connect(address_list=gw_address))

print("getting grid state:")
grid_state = gd.getGridState()
```

```
17
18
19 print("issuing findDatum:")
20 gd.findDatum(grid_state)
21 print("findDatum finished")
22
23 # We use grid_state to display the starting position
24 print("the starting position (in degrees) is:",
25        gd.countedAngles(grid_state))
26
27 # Generate a waveform which moves the alpha arm by +45
28 # degree, and the beta arm by +15 degree.
29
30 alpha_move = 45
31 beta_move = 15
32
33 # Generate the required waveform for one FPU
34 waveform = gen_wf(alpha_move, beta_move)
35
36 # upload the waveform to the FPU
37 gd.configMotion(waveform, grid_state)
38
39 # start the movement
40 print("starting movement by (45,15) degree")
41 gd.executeMotion(grid_state)
42
43 print("the reached position (in degrees) is:",
44        gd.countedAngles(grid_state))
45
46 print("ready")
```

2.1. Import statements

The import statements in lines 1 – 4 load the following modules and configurations:

- In line 1, the print function and the division operator is configured to work as in Python 3.
- "`import FpuGridDriver`" in line 2 loads the driver library module. (If this import command fails, check that both `LD_LIBRARY_PATH` and `PYTHONPATH` have the right values.)
- The line "`from FpuGridDriver import TEST_GATEWAY_ADRESS_LIST`" loads a default address which refers to the EtherCAN gateway which is used. This address would be different when one wants to select a different gateway. The default address used by the

EtherCAN gateway card is 192.168.0.10, and the card must be connected to a private (internal) network.

- The line "`from fpu_commands import ...`" imports a few utility commands which help to display positions and to generate movement waveforms.

2.2. Grid driver object

In line 10 of the example script, the used number of FPUs is set to one, and in line 11, the FPU grid EtherCAN interface is initialised with that value. The returned object, which is here referenced by the Python variable "gd", is always required to access the FPUs.

2.3. Driver methods

The driver object allows to invoke method calls (methods are also often called member functions) which control the FPU hardware. The method names always start with a dot after the name of the driver object. So, `gd.abc()` would call method "abc" of the driver object. Like normal function calls, they usually have parameters.

In this case, the method `connect()` connects the EtherCAN interface to one or more EtherCAN gateways and starts listening to messages from the FPUs. The parameter `gw_address` defines to *which* gateways the EtherCAN interface listens to. Section 13.2 on 67 explains the default parameters which are used.

As can be seen in the example, the methods have return values. In case of serious errors, the methods raise a Python exception.

The Python program does not need to explicitly disconnect and de-initialise the EtherCAN interface, this happens automatically when the program terminates[1].

2.4. Grid state variable

Almost all commands to the EtherCAN interface receive and return a snapshot of the current state of the used FPU (or, if there are several, of all FPUs). That snapshot is passed in a variable called `grid_state`. This variable contains all the information a user might need about the state of the FPUs - their current positions, which commands they will accept right now, whether there are any collisions, whether the step counters have been zeroed, whether a limit switch was hit, and so on. With each invocation of a method, the old grid state is passed as a reference parameter, and its new value is returned in the same variable.

To retrieve the current grid state information, the method "getGridState()" is used, as is done in line 16. In our listing, the grid state data is assigned to the variable named `grid_state`. The `getGridState()` function always returns immediately, as it just takes a snapshot of the

[1]However if required, you can use the statements `gd.disconnect()` and `del gd`, which disconnects and deletes the driver object.

state data. Because `grid_state` is used so frequently, a short-hand alternative name which many of the example scripts use is the name `gs`. Using the name “gs” is a mere convention: Technically, these names are simply Python variable names, which can be arbitrarily chosen by the author of a script.

2.5. Moving the FPU around

The example script then proceeds to use three methods to move the FPUs:

`findDatum()` in line 20 moves the FPU to the datum position. After this, the internal step counters are set to zero, both for the alpha and for the beta arm.

If a datum search is not possible because the arm positions do not allow to do that safely, a `DE_PROTECTION_ERROR` exception is returned.

`configMotion()` is a method which sends a table with movement instructions, also called “waveform table”, to the FPU. In the listing, this is done in line 37. The waveform table is stored in the variable `waveform`, which is defined in line 34.

`executeMotion()` in line 41 is the command which starts the movement of the FPU. This command returns when the movement is complete and the FPUs have stopped to move. In case of an error, the command generates a Python exception.

2.6. Utility functions

The example also shows two functions for generating movement data, and displaying information about the FPU state:

`countedAngles()` is a method which takes a `grid_state` variable, refreshes its content, and displays the current positions of the FPUs as alpha and beta angles, in degree units. In these coordinates, the datum positions serve as a point of reference. Their value are normally -180 degree for the alpha datum position, and 0 degree for the beta datum positions.

`gen_wf()` is a function which takes an (alpha,beta) pair of angle values, scaled in degrees, and generates a valid waveform which can be send to the FPU. The waveform which is passed is a regular Python data structure and can be displayed and manipulated like any other Python object. The user only needs to take care that it contains valid movement data when it is passed to the `configMotion()` method!

The generated waveforms do not match the input angles exactly, as they are necessarily subject to rounding, which is explained below in further detail.

2.7. Interactive inspection of the FPU state

The grid state variable holds a large amount of detail information about the current state of FPUs. This is by design, because upper layers of the software need to be able to deal with complex scenarios such as collisions, or even a pile-up of many collided FPUs, and correct them. However, only a part of this information is required by the verification system.

When trying to diagnose and understand errors, it is often helpful to access this state information in an interactive way. To do this, we will make use of Python's capability to run any commands which appear in a script equally well when entered interactively. In addition, the Python interpreter can be started with the "-i" (interactive) option which has the effect that after after all commands in a script have been executed, or after any exception has been thrown, the interpreter opens a command line or "read-eval-print-loop" in which commands can be entered and executed in the interactive mode. Every time when a typed-in expression or function call returns a value, the value is printed out before the command prompt appears again.

For example, let's assume that the script on page 4 is shortened to the first 36 lines and stored to a file named `short_script.py`. We then might run the following interactive session (where the lines starting with "$" and ">>>" are interactive input, and everything else is output):

```
$ python -i short_script.py
initializing driver:  DE_OK
connecting grid: DE_OK
getting grid state:
issuing findDatum:
findDatum finished
the starting position (in degrees) is: [(-180.0, 0.0)]
>>> grid_state.FPU[0]
{ 'last_updated' : 662758.71746,
  'pending_command_set' : 0,
  'state' : 'FPST_READY_FORWARD',
  'last_command' : 1,
  'last_status' : 0,
  'alpha_steps' : 0,
  'beta_steps' : 0,
  'alpha_deviation' : 14,
  'beta_deviation' : -3,
  'timeout_count' : 0,
  'step_timing_errcount' : 0,
  'can_overflow_errcount' : 0,
  'direction_alpha' : 6,
  'direction_beta' : 6,
  'num_waveform_segments' : 19,
  'num_active_timeouts' : 0,
  'sequence_number' : 26,
  'ping_ok' : 1,
  'movement_complete' : 0,
  'alpha_was_referenced' : 1,
  'beta_was_referenced' : 1,
  'is_locked' : 0,
  'alpha_datum_switch_active' : 0,
  'beta_datum_switch_active' : 0,
  'at_alpha_limit' : 0,
  'beta_collision' : 0,
  'waveform_valid' : 1,
  'waveform_ready' : 1,
  'waveform_reversed' : 0,
  'register_address' : 0,
  'register_value' : 0,
  'firmware_version' : 2.0.0,
  'firmware_date' : '2018-01-01',
  'serial_number' : "MP000",
  'crc32' : 00000000,
  'checksum_ok' : 0,
  'sequence_number' : 26,  }
>>>
>>> gd.countedAngles(grid_state)
[(-180.0, 0.0)]
>>> gd.executeMotion(grid_state)
ethercanif.E_EtherCANErrCode.DE_OK
>>> gd.countedAngles(grid_state)
[(-135.00385502825844, 15.003335899819655)]
>>> grid_state.FPU[0]
{ 'last_updated' : 662997.43726,
  'pending_command_set' : 0,
  'state' : 'FPST_RESTING',
  'last_command' : 7,
  'last_status' : 0,
  'alpha_steps' : 5125,
  'beta_steps' : 1265,
  'alpha_deviation' : -15,
  'beta_deviation' : -13,
  'timeout_count' : 0,
  'step_timing_errcount' : 0,
  'can_overflow_errcount' : 0,
  'direction_alpha' : 6,
  'direction_beta' : 6,
  'num_waveform_segments' : 19,
  'num_active_timeouts' : 0,
  'sequence_number' : 29,
  'ping_ok' : 1,
  'movement_complete' : 1,
  'alpha_was_referenced' : 1,
  'beta_was_referenced' : 1,
  'is_locked' : 0,
  'alpha_datum_switch_active' : 0,
  'beta_datum_switch_active' : 0,
  'at_alpha_limit' : 0,
  'beta_collision' : 0,
  'waveform_valid' : 1,
  'waveform_ready' : 0,
  'waveform_reversed' : 0,
  'register_address' : 0,
  'register_value' : 0,
  'firmware_version' : 2.0.0,
  'firmware_date' : '2018-01-01',
  'serial_number' : "MP000",
  'crc32' : 00000000,
  'checksum_ok' : 0,
  'sequence_number' : 29,  }
>>>
```

In this listing, entering the expression grid_state.FPU[0] displays the state parameters of FPU 0 (which is the only one we have because NUM_FPUS was set to 1).

Just to pin-point two of the more important bits of information, the fields "alpha_steps" and "beta_steps" contain the current step counts of an FPU, and the fields "alpha_datum_switch_active" and "beta_datum_switch_active" contain the current position of the datum switches. The field "state" contains the current state of the FPU, "firmware_version" contains the version of the FPU firmware, and "serial_number" the serial number of the FPU, if it was stored in the firmware. One can see that after issuing the executeMotion command, the state of the FPU changes from READY_FORWARD to RESTING. Actually, the output of the list_angles() utility function mentioned before is normally (that is, if at least one datum search was completed) just a scaled and shifted version of the "alpha_steps" and "beta_steps" fields. The flags alpha_was_referenced and beta_was_referenced are set to True when the datum position was reached at least once since the EtherCAN interface was started, and no collisions, limit switch breaches, or abortMotion commands have happened since.

Capturing interactive output

To investigate and communicate any difficulties or errors, it is sometimes helpful to capture the interactive output. A helpful tool for this is the Linux "script" command, which does exactly this, and is installed on the MOONS workstation. For further information, please consult the "script" man page.

2.8. FPU states

The state of an FPU is summarised in a enumeration value, which is stored in the attribute "state". For example, the state of FPU 0 can be retrieved by

```
>>> grid_state.FPU[0].state
```

Generally, each FPU is always in exactly one of 12 different states. Changes can be described by the concept of a "finite state machine": Each state allows a number of different commands, and commands can change the FPU state from the current state into another allowed state. Some of the states are visited during any normal operation of the FPU, and some are reached as a result of errors. Table 2.1 on page 12 lists the different states.

Figure 2.1 on page 13 shows, as a reference, the states of the FPU, and the possible state transitions and commands which are allowed in protocol version 2.

State name	Description
UNKNOWN	The state of the FPU is not known, the EtherCAN interface needs to be connected to the EtherCAN gateway first and retrieve the state.
UNINITIALIZED	The FPU has been pinged successfully, but it was so far not zeroed to the datum position, so no movements are possible.
LOCKED	The FPU was locked, which inhibits movement of this FPU.[a]
DATUM_SEARCH	The FPU is currently searching for the datum position.
AT_DATUM	The FPU has reached the datum position.
LOADING	The FPU is currently configuring a new movement waveform.
READY_FORWARD	The FPU has successfully been configured with a waveform and is ready to move in the forward direction of the waveform table.
READY_REVERSE	The FPU is ready to move in reverse direction along the waveform table.
MOVING	The FPU is currently moving.
RESTING	The FPU has finished its movement.
ABORTED	The movement was aborted, either by an internal error in the FPU controller, or by an explicit abortMotion command.
OBSTACLE_ERROR	The FPU has either detected a collision of the beta arm, or the alpha arm has hit the limit switch. In both cases, the movement was aborted.

[a]This is a facility provided by protocol version 2

Table 2.1.: List of FPU states and their definitions

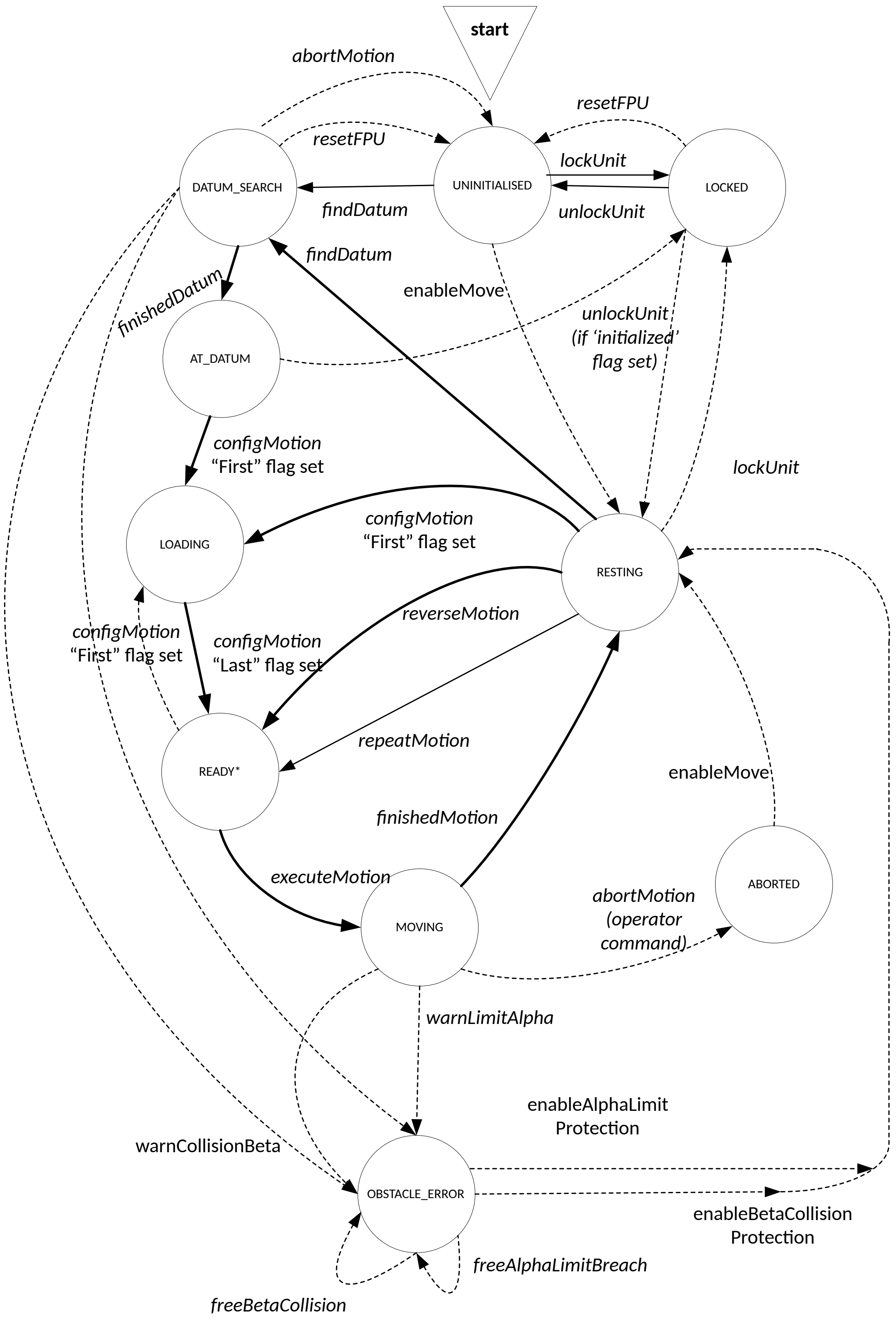

Figure 2.1.: Allowed state transitions of a single FPU for CAN protocol version 2, and the associated EtherCAN interface commands and CAN messages.

3. Rules to maintain integrity of the software protection

Contents

The software protection makes it easier to ensure that the FPUs are not damaged by out-of-bounds movements or operations that are not supported by the firmware.

3.1. Rules to observe

The protection relies on a few conditions that must be strictly observed at all times to keep the tracked positions valid. These conditions are as follows:

- Never use the same serial number for more than one FPU.
- Never move FPUs circumventing the EtherCAN interface. If FPUs are moved outside of the EtherCAN interface, their positions need to be stored in the database again, using the `fpu-admin` tool. If any driver instance is controlling such an FPU, before correcting the stored positions: exit the driver, and only after correcting re-start the driver.
- Do not use any obsolete versions of the EtherCAN interface.
- Never power-cycle (turn off and on again) an FPU while the EtherCAN interface is connected. If an FPU needs to be re-started, exit the program which uses the driver, restart the FPU, and start up the program again. Alternatively, one can use the `resetFPUs()` command of the EtherCAN interface. This ensures that the EtherCAN interface correctly interprets the changed step count which is caused by the firmware reset.
- When first using an FPU which was newly connected to a gateway, always use the `trackedAngles()` method to check that it shows the correct position, before moving the FPU.
- The protection layer cannot defend FPUs from failures of the datum switches. Always have checked that these switches work before using a `findDatum()` command. The following section 5 explains in detail how a provisional check can be done using EtherCAN

interface commands. Checks which address the reliability of the hardware are out of the scope of this document - please refer to the FPU hardware documentation.

WARNING: Do not power-cycle the FPU while it is connected to the EtherCAN interface software.

3.2. Allowed operations and covered cases

If the above conditions are observed, the following operations are allowed, and designed to include the following error conditions:

- Moving an FPU using `findDatum()` or `executeMotion()`.
- Reversing movements.
- Movements which change direction multiple times.
- Resetting FPUs, using the resetFPUs command.
- Cancelling movements using `<Ctrl>-<c>` or `abortMotion()`.
- Exiting and re-starting the driver program.
- Power-cycling FPUs while the driver program is not running.
- Controlling multiple FPU grids by different EtherCAN interface instances.
- Trying to configure invalid waveforms.
- Running into limit switch breaches and collisions of the arms.
- Resuming protected operations after temporarily disabling the protection flag.
- Recovering from beta collisions, using the `freeBetaCollision()` command.
- Temporary or permanent loss of connectivity on the CAN bus.
- Loss of the Ethernet connection to the EtherCAN gateways.
- Recovery of alpha arm limit switch breaches.
- Disconnecting an FPU from one EtherCAN gateway, and connecting it to another gateway connected to the same host workstation.

- Hard power-off or loss of power of the workstation which hosts the driver program or the database (note that this can disrupt other software which is not designed for that)[1].

3.3. Handling ambiguous positions

When movements are interrupted, due to errors like collisions, or due to `abortMotion` messages, it is possible that positions become ambiguous. Normally, ambiguous positions are resolved by issuing a `findDatum()` command. It can however happen that such an ambiguous position prevents the software protection from allowing a new movement. For example, if the beta arm is only known to be in a range between -10.0° and +10.0° degrees, the software protection cannot decide whether it should be moved clock-wise or anti-clockwise. In this case, an error is thrown.

There are three ways to handle such a situation:

1. The easiest approach is to move the FPU so that its range of possible location does not prevent the `findDatum()` command any more to move in a unambiguous direction. For example, if in the case given above, the FPU is moved by 15° degrees anti-clockwise, its new position will still be ambiguous, but it will be in the range from +5.0° to +25.0° degrees. (When the FPU is not in initialized state, it will be necessary to pass the keyword parameter `allow_uninitialized=True` to the corresponding `configMotion()` command, and also to issue an `enableMove() command`. However, it is not at all necessary to disable the software protection.)

   Now, the findDatum command will simply perform an clock-wise datum search, without needing any further information.

2. Alternatively, in a few cases, it can be necessary to exit the driver, measure the position, and assign a new position to the FPU, using the `fpu-admin init` command.

3. In theory, it is also possible to disable the software protection. This is almost never a good idea. Disabling the software protection should only be done when it is necessary to move the FPU outside of its safe range of positions. This is only needed when explicitly testing hardware protection functions.

The next section explains how to test the proper functioning of the datum switch hardware in a safe way.

[1]The hosting workstation needs to use a journaling file system, such as ext4 or ZFS, and should not use disk encryption.

Part II.

Manual

4. Setting up the library module

Contents

This section deals with retrieving, installing and configuring the EtherCAN interface module. On the MOONS workstation at UKATC, it can be skipped because the FPU EtherCAN interface is already installed. In this case, you can proceed directly with section 2 on page 4.

4.1. Installing required dependencies for a new installation

For a new installation, a number of required dependencies should be installed on a 64-bit Linux system. The detailed list of dependencies, and how to install them is explained in Appendix B on page 120.

4.2. Getting the source code

To set up the library module, it is first necessary to retrieve a current copy of its source code.

4.2.1. Using a packed archive

Without direct access, this can be done by unpacking a tar archive with the latest sources.

4.2.2. Using network access

On the moons-pc01 workstation, this can be done using the command

```
git clone /home/jnix/fpu_driver
```

This creates a new directory with the name `fpu_driver` and populates it with the source code.

Alternatively, a fresh copy can be retrieved from the UKATC dalriada server. Doing this this requires a personal user account on that server. Assuming the account name is `carl`, this is done by the command

```
git clone carl@dalriada/sw/sw4/jnix/MOONS/fpu_EtherCAN interface
```

After entering the password for user `carl`, the source code is copied into the new folder `fpu_driver`.

Refreshing the source code from git To refresh the source code, change the working directory to `fpu_driver`, and issue the command

```
git pull
```

4.2.3. Using the ESO subversion repository

This repository contains the current development version of the module for hardware protocol 2. It can be retrieved using the URL `http://svnhq1.hq.eso.org/p1/branches/MOONS/FibrePositioner/MOONS2/ICS/mpifps/`.

The "`make`" commands described in the following need to be run in the top-level folder. This uses a temporary `Makefile` which ignores the other ICS components. Running the code from this development branch requires to run either the included EtherCAN hardware simulation (as described in Appendix C), or compatible FPU firmware for hardware protocol v2.

4.3. Building specific revisions

In the default case, the library is build from the master branch of the git repository. The master branch contains the most current working version. In some cases, it might be required to use a specific revision of the library. Assuming that we need, as an example, to build version 1.5.2 of the library, checking out this revision is done with the command

```
git checkout v2.1.1
```

After this, one can proceed with building the library as normal. To switch back to the master branch, issue the command[1]

```
git checkout master
```

4.4. Building the library module

To build the Python library module, change the working directory to `fpu_driver`, and issue the command

```
make wrapper
```

In order for this to work, the required dependencies need to be installed. For the MOONS workstation at UK ATC, this is always the case. To install these dependencies on other computers, follow the detailed instructions in the appendix section B on page 120. A more concise summary is given in the text file "`python/doc/INSTALL`".

4.5. Enabling debug output of CAN messages

Beginning with EtherCAN interface version 0.5.3, a very comprehensive log of sent and received CAN messages is enabled by default. For long running tests, the logging might need to be set to a lower level of detail in order to keep the used disk space in reasonable limits. See section 9 on page 51 for the location of these logs, and any further details.

4.6. Environment configuration

For the Python module to work, the directory `/user/local/lib` (or, more general, the path to which the boost::python library is installed to) needs to be added to the environment variable `LD_LIBRARY_PATH`.

Further, the library uses a lock file to prevent that different processes send simultaneously commands to the same gateway. To access the lock file, all user accounts which use the driver software need to be member of the Unix group defined in the environment variable `MOONS_GATEWAY_ACCESS_GROUP`. The default name for that group is "`moons`".

Also, it is probably useful to add the `python` directory within the driver package, and also the current directory to the environment variable `PYTHONPATH`, so that the EtherCAN interface module can be found independently from the current working directory. Assuming a standard Ubuntu or Debian setup, this can be done by adding the following lines to the file `.bashrc` in the user-specific home directory:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
export PYTHONPATH=$PYTHONPATH:$HOME/fpu_driver/python:.
```

[1] previously, the master branch tracked the CAN protocol version 1. The master branch has been changed to track the can protocol 2 branch on September 19, 2019.

To run the fpu-admin Python script, which manages the FPU position database, it is also recommended to include the above directory in the PATH environment variable, like this:

```
export PATH=$PATH:$HOME/fpu_driver/python
```

Further, it is recommended to set the default log directory to point to a file system which is writable and has a lot of free space, for example:

```
export FPU_INTERFACE_DEFAULT_LOGDIR=/moonsdata/logfiles
```

One more setting is required: This version of the EtherCAN interface uses a position database, which tracks positions and state of FPUs at any point in time. Importantly, this database needs to be *global* (that is, for all users) for the workstation on which the EtherCAN interface software is running. To configure it, the environment variable `FPU_DATABASE` needs to be set in the configuration file `/etc/profile` as follows:

```
FPU_DATABASE=/var/lib/fpudb
export FPU_DATABASE
```

If this variable is not set, the FPU position database will not be used. In this case, the hardware protection capability, which is implemented in the Python wrapper, is not available.

The directory specified has to be created to be writable by any user which runs an FPU EtherCAN interface. It is therefore necessary to set up any user which runs the FPU EtherCAN interface to belong to the same user group, and make the directory writable by any member of this group, using the Unix “chmod” and “chgrp” commands.

Note on memory usage As a side note, the FPU position database uses LMDB to enable fail-safe storage and logging. This is a fast, thread-safe key-value storage which uses large memory-mapped files. LMDB can often use a large amount of virtual memory. This is not a bug and especially not a memory leak, as only a small fraction of this data is actually loaded into RAM.

4.7. Supported firmware versions

This EtherCAN interface requires FPU firmware version 2.0.x to support all regular features. Older versions of the FPU firmware are not fully supported, and do not handle CAN message priorities correctly. See section 23 for details which functionality is missing for older versions.

4.8. Hardware configuration and addressing

Messages to an individual FPU are sent using a logical number, the FPU ID, which starts at 0 and goes up to a maximum value. Gaps in the sequence of used logical numbers are not supported.

The messages from the EtherCAN interface to FPUs are transported by two layers of hardware.

First, the messages are transmitted by an Ethernet socket connection to up to three EtherCAN gateways. To connect to each gateway, its IP address and port number are used. The default IP number, to which the EtherCAN gateway card is configured, is `"192.168.0.10"`, and the connection uses port 4700 by default. In this case, the EtherCAN gateway (or if several are used, all of them) needs to be connected to a private network interface to which IP packets to network `192.168.0.0/24` are routed.

From the selected gateway, the messages are then forwarded to up to five CAN buses. Each bus connected to a gateway is identified with a number as well, and is connected to up to 72 (currently, 67) FPUs. To identify each FPU in a setup with many FPUs, they have to use a CAN ID which is unique on each bus, *and is different from zero.* This ID is physically configured by manually toggling a DIP switch in the FPU's electronics board. The current EtherCAN interface uses a fixed continuous mapping (basically, array indexing) from the translation of FPU IDs to gateway number, bus number, and CAN ID[2].

This results in that each FPU has a physical address with three elements: gateway number, bus number, and CAN ID.

For addressing a single FPU, this simply means that its CAN ID needs to be set to one, which means its DIP switch needs to be set to the delivered configuration. That FPU then needs to be connected to the first bus on the first gateway.

Note: The current hard-wired address mapping between FPU logical numbers and the (gateway number, bus number, CAN ID) tuple will be made more flexible in future versions of the EtherCAN interface. The current mapping will remain the default value.

4.9. Configuring serial numbers and setting initial positions

Initialising an FPU (Note: This step can be skipped if the LMDB protection feature is not used, for example when only testing electronic components rather than mechanical hardware. In this case, only a limited "unprotected" driver layer is available.)

Before any FPU can be used with the EtherCAN interface, two commands need to be performed to configure the hardware protection feature:

1. The serial number of the FPU needs to be flashed to the NVRAM of the FPU motion controller. This allows the software to recognise the FPU.

2. The initial position of the alpha and beta arm needs to be stored in the FPU database.

To do this, the FPU needs to be connected to the EtherCAN gateway with a known logical ID, determined by the CAN bus ID explained above. Also, the gateway needs to be connected to the workstation.

[2]Future versions of the software will relax this restriction

Let's assume that the serial number is represented by the shell variable `serialnumber`, the logical FPU id by the variable `fpu_id`, the initial alpha arm position by `alpha_position`, and the beta arm position by `beta_position`. The angular positions are given in degrees, and with the coordinate offsets used in the instrument. By default, this means the alpha datum position is at -180° degrees, and the beta datum position at zero degrees.

Then, the following two commands need to be issued:

```
fpu-admin flash ${serialnumber} ${fpu_id}
fpu-admin init ${serialnumber}${alpha_position} ${beta_position}
```

The first command writes the serial number to the NVRAM of the FPU controller. This needs to be done only once (except when the electronics PCB of that FPU is replaced by another PCB). The serial number must never be used for another FPU. (As a protective measure, the command checks that the serial number wasn't used before. This check can be disabled by passing the --reuse_sn flag.)

The second command stores the position of the FPU in the position database. The position has to be qualitatively correct whether the alpha arm is in the safe zone and whether the beta arm needs to move into a clock-wise or anti-clockwise direction to datum. It is also possible to specify position ranges as measurement uncertainty intervals in the form of Python list literals, enclosed in double quotes:

```
fpu-admin init ${serialnumber} "[0, 1.5]" "[3, 6.5]"
```

Importantly, storing the position has to be repeated when it should occur that the stored position is not matching the actual position. Apart from driver bugs, this can only happen if the FPU is moved by something else than the FPU EtherCAN interface.

Viewing FPU position dates The database record can be viewed and verified by using the command

```
fpu-admin list1 ${serialnumber}
```

Assuming the initial positions were alpha = -179° degrees and beta = 1° degrees, this should show data similar to the following output:

```
jnix@nippes:~/MOONS/fpu_driver/python$ ./fpu-admin list1 PT01
['./fpu-admin', 'list1', 'PT01']
('PT01', 'apos') : [[-179.0, -179.0], -180.0]
('PT01', 'bpos') : [[1.0, 1.0], 0.0]
('PT01', 'wf_reversed') : False
('PT01', 'alimits') : [[-180.0, 179.6], -180.0]
('PT01', 'blimits') : [[-179.3, 150.3], 0.0]
('PT01', 'bretries') : 3
('PT01', 'beta_retry_count_cw') : 0
('PT01', 'beta_retry_count_acw') : 0
  ....
```

What can be seen from the above example is that both the alpha position ('apos') and the beta position ('bpos') are stored as a braced pair of identical numbers, followed by an additional number. The number pair indicates the position, and the additional number indicates the offset of the datum position, which is by default -180° degrees for the alpha arm, and 0° degrees for the beta arm. One could ask why the arm positions are represented as a number pair. The reason is simple: The database is designed so that it can, if required, represent uncertainty around the arm positions as an position interval. In that case, the number pairs would represent start and end of an interval of possible positions.

The intervals "alimits" and "blimits" describe the ranges which are considered safe by the software. They can be adjusted on an per-FPU basis if required, using `fpu-admin alimits` and `fpu-admin blimits`.

The `fpu-admin` tool can also be used to print a list of datum aberrations (the difference between the ideal datum position, and the step count at which datum was actually found), and a time-series of counters. This is described in the reference part in section 8. More commands and options are shown if the tool is run with the command-line option "-h" or "–help".

4.10. Database maintenance

The position database is required to operate the FPUs in the normal (protected) operation mode, and will also hold some data gathered during calibration of the FPUs. It is therefore necessary to back it up regularly. It can be backed up by copying the content of the directory defined in the `$FPU_DATABASE` environment variable. For this, the dedicated backup tool for LMDB, `mdb_copy` should to be used. `mdb_copy` is safe for backing up a database which is in use at the same time.

The database can be used by several EtherCAN interface instances at once, each of them controlling different FPUs. When the involved EtherCAN interfaces are not running, FPUs can be switched from one gateway to another one - the information in the database will remain valid.

4.11. Software protection flag

Protecting instrument hardware The FPU control chain uses a layered approach to make sure that fibre positioner units do not get damaged.

Hardware protection At the bottom layer, there is an electronic collision protection, which will switch off the motors and send a warning message if two FPUs touch each other. This protection level is called "hardware protection".

However, even if a direct damage should be avoided by this protection, activating it involves that the motors are suddenly stopped, which decreases the precision and can significantly increase wear and tear, thus reducing the life time of the stopped FPU. Therefore, the hardware protection should be used only at an emergency level. Furthermore, some situations are not covered by it, specifically if the FPU arms are moved into the dead zones.

Software protection For these two reasons, the EtherCAN interface software performs comprehensive additional checks which should ideally ensure that no out-of-range movements occur. These checks are performed every time an FPU is moved, or configured to move, and are named "software protection". For all normal movements of FPUs, this protection should be left active.

Deactivating the software protection In some situations, specifically if the hardware protection or the alpha datum switch needs to be tested, it can be necessary to switch off and deactivate this software protection layer. This can be done using the "`soft_protection`" keyword argument when calling the corresponding EtherCAN interface methods. Passing the value "`soft_protection=False`" will switch the software protection check off. When this is done, the entire responsibility with operating the FPUs safely is with the user. It is highly recommended to refer to additional documentation (especially FPUMovementSafetySpecification.docx, the actual RFE to software ICD document, and additional hardware and firmware documentation), to assess which operations are safe.

It is also possible to de-activate any protection by using a special driver class with the name `UnprotectedGridDriver`. This class is useful for testing purposes, but should not be used with valuable hardware.

4.12. Communication protocol versions

The FPU EtherCAN interface sends commands to the fibre positioner units, which use a specific communication protocol. This protocol needs to match the FPUs firmware and its capabilities. The current version of the EtherCAN interface (version 2.0.x) uses "communication protocol version 2.0", which has some differences to earlier versions.

A significant difference in the hardware protection capabilities is that cases in which the alpha arm of the FPU hits the limit switch can be recovered in exactly the same way as a beta arm collision. This is explained in section 10 on page 55.

The functionality described in this document is supported by EtherCAN interface versions equal or newer than 2.1.0 and an FPU firmware that supports the CAN protocol version 2.0.x or later. Earlier firmware versions do not work with this driver software version.

The next sections are intended as a tutorial on how to control and operate FPUs. As usual, the operations are presented so that the most basic concepts are introduced first, and more difficult commands and concepts which build on that later. For this reason, it is first shown in section 2 how to perform basic movements, before explaining in section 5 how the switch operation is tested. Nevertheless, it is highly recommended to have test the switch operation tested before doing any further operations with the FPU.

WARNING: Before first issuing a `findDatum()` command, make positively sure that both the alpha arm datum switch and the beta datum switch work. Especially for the alpha arms, defective datum switches will likely cause the datum command to damage the hardware. The EtherCAN interface software cannot protect against this.

5. Checking the datum and limit switches

During initial testing of each FPU, it is advisable to test the function of the switches – especially the alpha limit switch – without performing a datum operation. The reason for this is that if the alpha limit switch should not work, the datum operation would probably damage the FPU irreparably.

The following section describes how to check that the datum and limit switches of an FPU are working.

5.1. How to retrieve the switch state

To test the switches, the FPU needs to be be driven manually to a position where the corresponding switch is required to engage. At this point, one can use the command

```
gd.getSwitchStates(grid_state, fpuset=[])
```

to retrieve the state of the switches.

5.2. Required information

To begin this test, it is necessary to know the exact safe operation ranges of the alpha and beta arm. These values depend on hardware and FPU firmware revisions and are currently still subject to change. They are described in document [4], "FPUMovementSafetySpecification.docx". The following description assumes that the position after performing a datum operation is (-180°, 0°) degrees, that both switches are off at positions (-175°, 5.0°), and both switches on at positions (-180.2°, -0.2°).

5.3. Test procedure, explained step-by-step

To test the datum switches, perform the following steps:

1. If the EtherCAN interface is connected, disconnect it first.
2. Make sure that the FPU database contains the correct alpha and beta arm positions for the FPU which is examined. Do this using the command

```
fpu-admin ${serialnumber} list1
```

The result should look like that:

```
['./fpu-admin', 'list1', 'PT01']
('PT01', 'apos') : [[-180.0, -180.0], -180.0]
('PT01', 'bpos') : [[0.0, 0.0], 0]
('PT01', 'wf_reversed') : False
('PT01', 'alimits') : [[-180.0, 179.6], -180.0]
('PT01', 'blimits') : [[-179.3, 150.3], 0.0]
....
```

This indicates that the alpha arm is at position -180° degrees, using a datum offset of -180° degrees, and the beta arm is at 0° degrees.

If the positions are not correct, proceed as described in section 4.9 on page 22. The further explanation assumes that the FPU position is at the numbers shown above.

3. Connect the EtherCAN interface and run a `pingFPUs()` command.

4. Move the FPU so that it is within the normal range, and with certainty not touching the datum switches.

   As a normal movement configuration requires that a datum search has been performed before, it is necessary send an `anableMove()` command `first`, and also to pass the `allow_uninitialized=T`
   flag to the configMotion command:

   ```
   w = gen_wf(5,5)
   gd.enableMove(0, gs)
   gd.configMotion(w, gs, allow_uninitialized=True)
   gd.executeMotion(gs)
   ```

5. Verify that both switches are off:

   ```
   gd.getSwitchStates(gs)
   ```

6. Now, move the FPU so that it just activates the datum switches. The movement interval needed depends on firmware version and offsets used. In addition to the options used in the first movement, it is now necessary to disable the protection layer by passing the "`soft_protection=False`" keyword argument. With firmware version 1.3 and an initial position of (-180, 0), this will be reached by using the command

   ```
   w = gen_wf(-5.2,-5.2)
   gd.enableMove(0, gs)
   gd.configMotion(w, gs, allow_uninitialized=True,
                   soft_protection=False)
   gd.executeMotion(gs)
   ```

   The FPU should now be at an absolute position of (-180.2, -0.2), and touching both switches.

7. Now, verify that both switches are on:

   ```
   gd.getSwitchStates(gs)
   ```

   Finally, the FPU arms can be moved back into the safe zone, using the command

   ```
   gd.enableMove(0, gs)
   w = gen_wf(5.2,5.2)
   gd.configMotion(w, gs, allow_uninitialized=True,
                   soft_protection=False)
   gd.executeMotion(gs)
   ```

If this test does not have the expected result, the cause is likely to be a hardware failure, which makes the FPU unusable until it is resolved.

6. Common tasks

Contents

The following sections describe several common tasks and describes which commands can be used to perform them.

6.1. Checking the connection

At some points, it is of interest what the state of the connection to the CAN gateway and of the FPUs is. This can be achieved by the following commands:

```
grid_state = gd.getGridState()
grid_state.interface_state
```

The return value is the connection state of the EtherCAN interface, which is `DS_CONNECTED` if the connection is live. In the case that the connection fails, it is still possible to retrieve state information on the FPUs, but it cannot be refreshed any more until the connection is re-established using the `connect()` method of the driver object.

Of course, it is possible that the connection between EtherCAN interface and EtherCAN gateway works, but that an FPU does not respond, for example, if the CAN bus connection wiring is broken, or the FPU has no power. This can be checked by the commands:

```
  gd.pingFPUs(grid_state)
  grid_state.FPU[0].ping_ok
```

In the case that the FPU responds, the returned value is `True`. If the FPU does not respond, the value of the `ping_ok` becomes `False`.

Errors both during making the socket connection, and when sending commands to the FPUs cause a `ConnectionFailure` exception to be thrown, like in this log file:

```
  $ python -i test_mockup/test_asyncMotion.py
connect: Connection refused
Traceback (most recent call last):
  File "test_mockup/test_asyncMotion.py", line 12, in <module>
    print("connecting grid:", gd.connect())
  File "/sw/sw4/jnix/MOONS/fpu_driver/python/FpuGridDriver.py", line 65, in connect
    return self._gd.connect(address_list)
    fpu_driver.ConnectionFailure: DE_NO_CONNECTION: The FPU Driver
    is not connected to a gateway.
>>>
```

The associated error message and enumeration symbol should be sufficient to determine the cause of the problem. A comprehensive list of exceptions and error codes is given in section 21 on page 96.

Recovery form connection errors If the connection to a gateway is lost, the `connect()` method of the EtherCAN interface needs to be called again to re-establish it.

If a specific FPU does not respond, each command to it will result in a time-out, until it responds again. Other FPUs will be unaffected. Failures to respond can be caused by cabling problems, a missing termination resistance on the CAN bus, or an invalid CAN id configuration (which is explained in section 4.8 on page 21). Another very likely cause of command time-out errors are hardware failures which prevent FPU operations from completing. In this case, the involved FPUs are probably broken.

6.2. Checking the FPU firmware, its integrity, and the FPU serial numbers

After having a working connection, it is useful to check for the correct firmware version, the firmware integrity, and the serial number of the addressed FPUs.

6.2.1. Checking the protocol version

The value `FpuGridDriver.CAN_PROTOCOL_VERSION` returns the major version number of the CAN protocol which the driver implements. Because a script might not work as expected, it can be useful to check it automatically in the top level python script like that:

```
assert(FpuGridDriver.CAN_PROTOCOL_VERSION == 2)
```

6.2.2. Checking the firmware version

Next, the version of the firmware running on the FPUs can be checked with the command

```
gd.printFirmwareVersion(grid_state)
```

It returns a list of FPUs with the corresponding version of the version and data of the firmware in the FPU's motion controller.

It is possible that some tests rely on a certain minimum firmware version being present on the controllers. This can be checked with the method

```
gd.minFirmwareVersion() >= (2, 0, 3)
```

6.2.3. Checking the firmware integrity

As best practice, any important tests should also check beforehand the integrity of the flashed firmware. This can be done with the following command:

```
gd.checkIntegrity(grid_state)
```

FPUs with equal firmware versions should have equal check sums, of course (different serial numbers do not affect the checksum).

6.2.4. Listing the serial numbers

Finally, in any verification procedures, it is important to positively verify that the FPU specimens under test have their identity correctly attached. This can be done with the command:

```
gd.printSerialNumbers(grid_state)
```

All the above commands accept a keyword argument "`fpuset`" which allows to limit the result to a certain subset of FPUs.

6.3. Getting the current position

We already saw how to use the `pingFPUs()` method of the driver object. This method also automatically retrieves the current step counters of the FPU alpha and beta arm motors. There are two utility functions which display them: The command

```
list_positions(grid_state)
```

shows a list of tuples with the values of the alpha and beta step counters, and whether the counter has been initialised by a datum operation. The first element of the tuple corresponds to the alpha step counter, the second to the beta step counter, the third is a Boolean value which indicates whether the alpha counter was initialised, and the fourth indicates the beta counter initialisation. The keyword option “`show_zeroed=False`” disables display of the datum flags. As any other keyword arguments in python, it is passed like this:

```
list_positions(grid_state, show_zeroed=False)
```

Often, it is more practical to know the value scaled to an approximate angle. The scaled values are displayed by the driver method

```
gd.countedAngles(grid_state)
```

This method applies an offset to the step counts so that the alpha datum position results in a value of -180 degree, and the beta datum position of 0 degree. This offset is a convention which is used in all of the MOONS project to describe arm positions.

If one or both arms have not been initialised, by default a non-a-number (NaN) value is shown. Any floating point computation which involves a NaN value will result in another NaN value, so that using invalid positions is avoided. In the case that the uninitialised values are needed, the option "`show_uninitialized=True`" can be used.

One can wonder how this angle is calibrated in respect to motor and gear non-linearities, and what is the relative position to the datum position. The answer is, these values are not calibrated - `countedAngles()` shows only an approximately scaled value, with a scaling factor derived from the gear ratio. Also, the displayed values are only relative to the datum position, which is defined to have the step counter values (0,0). If the datum position was not searched at least once before, both values are probably meaningless, and this is why `countedAngles()` returns a pair of NaN values.

6.4. Moving the FPU to the datum position

6.4.1. Performing a datum search

Automatic search Because the FPU's step counters need to be zeroed before any accurate positioning is possible, a sensible operation early after powering on the FPU is to move the FPUs to the datum position. This is done by the command

```
gd.findDatum(grid_state)
```

It is also recommend to move the FPUs to the datum position before powering off.

A detailed description of all possible parameters for the datum search can be found in section 16.1 on page 76.

6.4.2. Independent operation of alpha and Beta arm

The FPU verification procedure requires that the alpha and beta arms of the FPU can be moved independently to the datum position. To perform this, the `findDatum()` command has an additional keyword parameter `selected_arm`. It indicates which arm should be moved to the zero position. The possible values of this keyword argument are `DASEL_BOTH`, `DASEL_ALPHA`, and `DASEL_BETA`. `DASEL_BOTH` is the default behaviour: Both alpha and beta arm are moved to the datum position. The value `DASEL_ALPHA` moves only the alpha arm, and the value `DASEL_BETA` will move only the beta arm.

6.4.3. Restricting a datum search to a subset of FPUs

In some cases, it can be necessary to move only a subset of FPUs to the datum position, leaving all the other FPUs untouched. This can be done by passing a `fpuset` keyword argument to the findDatum() method call, which contains a list of the FPUs which are to be moved, like this:

```
gd.findDatum(grid_state, fpuset=[3,5])
```

Here, only FPUs 3 and 5 will be moved to the datum position.

The fpuset can actually be passed to all movement commands including executeMotion(), and to all commands which act on a group of FPUs, like the resetFPUs(). A more detailed explanation can be found in section 12.2 on page 62.

6.4.4. Moving an FPU which has not been datumed

In some cases, it might be necessary to move an FPU which was not datumed. Normally, this is prevented by the fact that an FPU does not accept any movement configuration when it is in UNINITIALIZED state. In the case that an uninitialized FPU needs to be moved, this can be enabled by sending it a enableMove() command. This is explained further in section 6.6.2 on page 43.

6.5. Moving the FPU

6.5.1. Overview

To move the FPU in a precise and safe way, four steps are necessary:

1. Moving the FPU to the datum position, as explained above.
2. Generating a valid movement waveform table. For a single FPU, this is basically a list which defines for a number of short time segments how many steps the FPU stepper motors should move, and in which direction.
3. Sending the waveform to the FPU
4. Starting the movement

There are two options for defining the waveform. The simpler one is to automatically generate a protocol-conforming waveform where we merely pass two values which define how much the alpha and beta arms should move.

6.5.2. Definition of movement directions

The FPU EtherCAN interface uses uniformly the following convention to define the direction of movements and the sign of angles:

- When looking from above (from the direction of the incoming light in the telescope), a *counter-clockwise* movement will have a *positive* sign, and use positive step counts.

- Correspondingly, a *clockwise* movement will have a *negative* sign, and use negative step counts.
- Consequently, any position which is counter-clockwise from the datum position will be denoted by a positive angle, and any position which is reached after rotating clockwise from the zero position, will have negative angles.
- The zero position itself is defined as the datum position for the beta arm. For the alpha arm, the position is configurable. By convention, the alpha datum position is at -180.0°.

6.5.3. Moving by an angle

```
1  alpha_move = 45
2  beta_move = 15
3
4  # Generate the required waveform for one FPU
5  waveform = gen_wf(alpha_move, beta_move)
6
7  # upload the waveform to the FPU
8  gd.configMotion(waveform, grid_state)
9
10 # start the movement
11 print("starting movement by (45,15) degree")
12 gd.executeMotion(grid_state)
```

The above listing shows how to generate a waveform which moves the FPU by a certain (alpha,beta) angle difference.

Lines 1 and 2 define variables with the angles we want to move. In line 5, a new waveform table called waveform is generated using the gen_wf() utility function. By default, this function generates the quickest valid movement by the angles requested.

In line 8, the configMotion() method is used to send the waveform to the FPU. As usual, the grid_state variable is passed, updated by the command, and the resulting state of the FPU is returned in this variable. (In certain situations, it is necessary to pass additional flags, if an FPU is not in a state which normally allows movements. The details for this are explained in section 17.1 on page 79 ff.).

Finally, in line 12, the method executeMotion() is called, which starts the movement of the FPU (or, if we have more than one FPU, the movement of all of them). In the normal case, this command blocks and returns when the FPUs have finished moving. In the case that there is an error, for example a collision, a Python exception will be raised.

6.5.4. Precision of generated waveforms

Please note that, because the angle values which are arguments to the gen_wf() function always need to be rounded when they are converted to actual motor steps, *the resulting move-*

ment will be only approximately by that angle. The resulting error of around 1/100th degree will add up if two or more such movements are combined. To move by an exact distance, either a waveform with an absolute input needs to be generated and configured (which minimise the rounding error down to half a motor step), or – and even better – one needs to pass integer stepcount numbers into the waveform generation. The latter can be achieved by passing the keyword parameter `units='steps'` when calling `gen_wf()`.

The `countedAngles()` functions explained above will always return the true position of the motors as a floating-point degree value, with a accuracy which is only limited by the representation of floating-point numbers. This allows to correct for the rounding so that errors do not sum up.

To summarise, the approximate rounded waveforms generated by `gen_wf()` with degree values as input are not ideal for exact procedures like engineering measurements. Such procedures must carefully take the error stemming from rounding to step counts into account, and to correct for it so that it does not accumulate.

6.5.5. Generating waveforms for multiple FPUs

The function `gen_wf()` can also generate waveform tables for multiple FPUs. This is done by passing a list, or a numpy array, of angles to both the alpha and the beta parameter:

```
  # Generate a waveform for three FPUs
waveform = gen_wf([15.0, 10.0, 5.0], [0, -5.0, 13.5])
```

Here, the first element of the first list defines the alpha movement angle for the first FPU, the second element the alpha angle of the second FPU, and so on.

See section 19.6 for further details.

6.5.6. Restricting a movement to a subset of FPUs

In the verification system, it is necessary to move FPUs selectively.

A movement can be restricted to a subset of FPUs by issuing a `configMotion()` command which has all-zero entries for the FPUs which should not move. This avoids that a wavetable which was previously stored on the FPU becomes mistakenly activated.

However a more convenient way to restrict the effect of the `executeMotion()` command to a subset of FPUs is to pass an `fpuset` parameter which contains a list of FPUs that should move, in exactly the same way as described for the `findDatum()` command in section 6.4.3 on page 34. As an example, the command

```
gd.executeMotion(grid_state, fpuset=[2,4,8,17])
```

will move only FPUs 2,4,8, and 17, and ignore the states of the other FPUs.

6.5.7. Manually defining a waveform table

Waveform syntax In the example above, the waveform was generated by the `gen_wf()` utility function. For normal movements, this is a good option, and the actual value of the step counters can always be retrieved using the `list_positions()` utility function. Sometimes, a much closer control of the FPU movements might be required. This can be done with a code fragment like this:

```
wtable = { 0: [ ( 125, -125),
           ( 140, -140),
           ( 130, -125),
           ( 125, -125),
           ( 10, -20),
           ],
}
gd.configMotion(wtable, grid_state)
```

Here, the Python variable wtable is assigned with a waveform. The required format for a waveform is *a dictionary with a list of 2-tuples with the alpha and beta step counts.* Because that sounds a bit complex, let's dissect this structure into its components:

- At the top layer, we have a Python dictionary. Generally, dictionaries contain a set of different *keys*, and for each key there is an associated *value*. For a waveform, the key of each dictionary entry, an integer, is the numerical ID of the FPU we are addressing. Because we only have one FPU, and the numbering starts with zero, the key is simply zero.
- The corresponding value for the key zero is a Python *list*, as indicated by the brackets and a comma-separated sequence. The list contains exactly one entry for each time-slice of the movement operation[1].
- Each list entry is a tuple with two elements, both of which are numbers. The first element is the number of alpha steps, and the second element is the number of beta steps.

  A positive number means that the FPU should move counter-clockwise (when viewed from above), and a negative number means it should move clock-wise (when viewed from above). If the value is zero, then the FPU will rest.

Now, we can decipher the above structure into "in the first time slice of the movement, the alpha arm should move 125 steps counter-clockwise, and the beta arm should move 125 steps clockwise," and so on.

[1]With the current default values, each time slice has a fixed duration of 250 milliseconds

6.5.8. Waveform validity rules

When you define waveforms, they have to conform to a fairly large number of rules and conditions to make sure that the FPU firmware can actually execute them reliably. These rules are versioned, and the used rule set can be selected by passing the `ruleset_version` keyword argument to the `configMotion()` method. The rules for each version are defined as follows:

Waveform validity rule set version 1 This rule set is a stricter rule-set which is designed to filter out any waveforms which could cause undesired behaviour in the motors. However, it assumes some capabilities of the FPU firmware which are not met by firmware versions smaller than 1.5.0.

1. All step numbers need to be integer values.
2. The maximum number of entries in a waveform table is 256 (for the CAN protocol version 2).
3. A waveform for one FPU can include several movement parts for each arm. A movement part is a sequence of step counts which are non-zero and have the same direction. A waveform can contain an arbitrary number of movement sequences with non-zero entries.
4. The first non-zero entry in any movement sequence needs to have the minimum step count. Currently, this minimum is set to a step count of 125, which can be changed as a EtherCAN interface parameter. (To be precise, a starting speed which is up to 5 % higher than this minimum speed is still acceptable, which is needed for feeding in output from the path planning library – this is also a EtherCAN interface parameter, see section 13 on page 64. Minimum and maximum value will be configurable in later versions of the firmware.)
5. The last entry of a movement part in one direction can have a step count between zero an the minimum step count. It must not be larger than the minimum step count.
6. A single entry with a step count equal to or smaller than the minimum step count is allowed. It can only be preceded or followed by zero-valued entries.
7. Except the the last non-zero entry in a movement part, all step numbers in a waveform need to have an absolute value which has at least the minimum value of 125, and at most the maximum value of 500. Also, arbitrary sequences of zeros are allowed.
8. Between consecutive step number values, the absolute value of the larger number can only be at most 40% larger than the absolute value of the smaller number. Also, a change from zero to the minimum value, or from a value not larger than the minimum value to zero is allowed. (This maximum rate of change is also a EtherCAN interface parameter, see section 13.)

9. When changing the movement direction, the absolute number of steps before the change needs to be between zero and the minimum step count, followed by a wave table segment with a step count of zero, followed by the minimum step count with the opposite sign.

Waveform validity rule set version 2 This rule set is a more liberal selection of rules, which is designed to test preliminary path generation algorithms. It is compliant with all known restrictions of FPU firmware version 1.4.4.

1. Parameters are a minimum step count (which is 125, by default), a maximum step count (which is 500, by default), and a maximum starting speed at 138, which is the minimum step count plus 10.0 %. These limits are EtherCAN interface parameters, see section 13 on page 64.
2. All step numbers need to be integer values.
3. The maximum number of entries in a waveform table is 256.[2]
4. A waveform for one FPU can include several movement parts for each arm. A movement part is a sequence of step counts which are non-zero and have the same direction. A waveform can contain an arbitrary number of movement sequences with non-zero entries.
5. Except the the last non-zero entry in a movement part, all step numbers in a waveform need to have an absolute value which has at least the minimum value of 125, and at most the maximum value of 500. Also, arbitrary sequences of zeros are allowed.
6. The first non-zero entry in any movement sequence needs to be between the minimum step count (125) and the starting speed (138).
7. Except for the very last entry in a waveform, the last non-zero entry in any movement sequence needs to be between the minimum step count (125) and the starting speed (138).
8. Only the very last entry of a waveform can have a step count between zero and the minimum step count. It also can be larger than the minimum step count (causing a stop with a high deceleration or jerk value).
9. A single entry with a step count equal to or larger than the minimum step count, but not larger than the starting speed is allowed. It can only be preceded or followed by zero-valued entries.
10. When changing the movement direction, the absolute number of steps before the change needs to be at least the minimum step count, followed by a wave table segment with a step count of zero, followed by a value between the minimum step count and the starting speed with the opposite sign.

[2]In the old version 1 of the communication protocol, the limit was 128 entries.

11. Between consecutive step number values, the absolute value of the larger number can only be at most 40% larger than the absolute value of the smaller number. This applies to both sections with increasing speed, and sections with decreasing speed. Also, a change from zero to the minimum start speed value, or from a value larger or equal to the minimum start value to zero is allowed. This maximum rate of change is also a EtherCAN interface parameter, see section 13.

Waveform validity rule set version 3 This rule set tries to accommodate for some characteristics of the DNF algorithm waveforms. It is not yet known whether it is fully supported by the FPU hardware. This rule set is identical to ruleset version 2, with two exceptions:

- Completely stopping a movement is allowed from any speed.
- When decelerating (reducing the velocity), the square of the limit for accelerating is allowed. With a default acceleration limit of 40 %, this amounts to 95 %.

Waveform validity rule set version 4 This rule set matches the capabilities of firmware version 1.5.0, and firmware for the hardware prtocol 2, and is the default. It is almost identical to ruleset version 1, with the sole difference that direction changes between adjacent segments are allowed if the absolute speeds before and after are not too large.

1. All step numbers need to be integer values.
2. The maximum number of entries in a waveform table is 256 (for CAN protocol version 2).
3. A waveform for one FPU can include several movement parts for each arm. A movement part is a sequence of step counts which are non-zero and have the same direction. A waveform can contain an arbitrary number of movement sequences with non-zero entries.
4. The first non-zero entry in any movement sequence needs to have the minimum step count. Currently, this minimum is set to a step count of 125, which can be changed as a EtherCAN interface parameter. (To be precise, a starting speed which is up to 5 % higher than this minimum speed is still acceptable, which is needed for feeding in output from the path planning library – this is also a EtherCAN interface parameter, see section 13 on page 64. Minimum and maximum value will be configurable in later versions of the firmware.)
5. The last entry of a movement part in one direction can have a step count between zero an the minimum step count. It must not be larger than the minimum step count.
6. A single entry with a step count equal to or smaller than the minimum step count is allowed. It can only be preceded or followed by zero-valued entries.

7. Except the the last non-zero entry in a movement part, all step numbers in a waveform need to have an absolute value which has at least the minimum value of 125, and at most the maximum value of 500. Also, arbitrary sequences of zeros are allowed.

8. Between consecutive step number values, the absolute value of the larger number can only be at most 40% larger than the absolute value of the smaller number. Also, a change from zero to the minimum value, or from a value not larger than the minimum value to zero is allowed. (This maximum rate of change is also a EtherCAN interface parameter, see section 13.)

9. When changing the movement direction, the absolute number of steps before the change needs to be between zero and 120 steps, followed by either a step number of zero, or a step number between zero and 120 steps in the opposite direction. Alternatively, the number of steps can be the minimum step count, followed by a pause, followed by the minimum step count in the other direction.

Waveform validity rule set version 5 This most recent rule set, which is now the default variant, matches firmware version 2.

It tries to match the physical limits of the FPUs better than earlier rule sets, while maximising the range of movement for the path analysis algorithms. The path analysis algorithms are severely inhibited by low acceleration limits, and profit from the capability to do high accelerations at low speeds.

The minimum step count is 62, and the maximum step count is 250. The maximum change in step numbers per segment is 100 steps. This allows for a total acceleration of 400 steps / second.

At the same time, the maximum supported speed change for the FPU arms is probably relatively constant, with the limit that the torque of the motor gets smaller and smaller with high rotational frequencies. This suggested to use a constant acceleration limit.

The rules are otherwise the same as for rule set version 4, with the difference that rather than a relative increase, an absolute difference in step counts per segments is the limit. Note that this allows for larger accelerations if the temporal length of waveform segments is decreased.

Handling of validation failures If the `configMotion` method is called with an invalid waveform, a `InvalidWaveformException` is thrown, with a more specific error code and message. The error message should be specific enough to determine the reason of the validation failure. A complete list of waveform error messages and codes is given in section 21.2 on page 96 ff.

It is also possible that an error is generated because an FPU is in an invalid state, even if the waveform which is sent is generally OK. Details of this are discussed in section 17.1 on page 79.

6.5.9. Errors during movements

Movement operations can result in errors caused by collisions, limit switch breaches, connection failures, or hardware failures. Such errors generate an exception of type `MovementError` during the movement. A detailed description of error handling is given in the sections 7. Section 10 describes how to resolve collisions and limit switch breaches.

6.6. Aborting a movement

6.6.1. Stopping an FPU

In the case that one wants to abort a `findDatum()` or a `executeMotion()` operation, one can press `<Control>-<c>`. This terminates the movement in about 0.1 seconds. [3]

If it is necessary to abort a movement from Python in a program – for example, from another Python thread – this can be done using the `abortMotion()` method of the grid driver object. This method is thread-safe.

For other driver methods, a `<Control>-<c>` normally aborts the current Python script when the method returns.[4]

6.6.2. Resolving an ABORTED state

After an `abortMotion()` command, FPUs which were moving are put into the state `ABORTED`. They will not be able to move again before this state is resolved. In protocol version 2, this can (and should) be done using the `enableMove` command, so that it is not necessary to reset or power-cycle the FPU. This command can be used as following:

```
# fpu_id = 7
gd.unlockFPU(7, grid_state)
```

This sends the `enableMove()` command to FPU id number 7, and puts it into `RESTING` state.

The command can also be used to bypass the requirement that all FPUs after powering on need to be put into `AT_DATUM` state first, before they can be moved normally. Requiring that a datum operation was performed first is safer and required to make the operation precise, but it may not be possible in some cases.

[3] Under the hood, this generates a `SIGINT` signal which is during the operation of both commands caught be the Python interpreter. The Python interpreter then sends an `abortMotion()` command to the driver object, which results in an `ABORT_MOTION` CAN message. Sending a SIGINT signal to python by other means would have the same effect.

[4] In some cases, it can happen that the Python program is stuck on a failed socket connection. Terminating such a connection can take considerable time because sockets are handled by the Linux kernel and by default, the kernel tries extremely hard to keep and re-animate an unreliable connection, even if the physical link is temporarily broken. To terminate such a hung program, it can be suspended using `<Control-z>` (which sends a SIGSTOP signal) and then terminated using the UNIX `kill` shell command, or using `<Control-\>`, which generates a `SIGQUIT` signal.

6.7. Reversing a movement

Often, an FPU has been moved away from the datum position, and it is desired to move it back to the datum position. This can be done using the following method:

```
gd.reverseMotion(grid_state)
```

Reversing the waveform requires that there is, firstly, a current waveform configured, and, secondly, that this waveform remains valid. After any normal, successful movement, a waveform remains valid for repetition or reversal. After any collision, limit switch breach, or abortion of a movement, the waveform becomes invalid. Generally spoken, a waveform becomes invalid when either, the movement it defines was interrupted and not completed, or when the step counters become invalid (for example due to a collision when the FPU is resting).

6.8. Repeating a movement

Sometimes, a waveform for moving in a certain direction has been configured, and this movement just needs to be repeated. This is achieved by the following method:

```
gd.repeatMotion(grid_state)
```

6.9. Configuring absolute movements or paths

6.9.1. Avoidance of rounding errors

Because the input to the `configMotion(()` method is in relative angles, and the movement angles are always rounded to the nearest corresponding step count, rounding errors from the successive relative movements can sum up, unless they are either given in integer step units, or carefully corrected by the reported actual position before each new movement. This is easily understood by considering that

$$\mathrm{round}(1.4 + 1.4 + 1.4) \neq \mathrm{round}(1.4) + \mathrm{round}(1.4) + \mathrm{round}(1.4)$$

- the left hand side evaluates to 4, the right hand side to 3, and the only difference is the order of summation versus rounding. Individual relative movements correspond to the situation where angles are first rounded, and summed only after that.

For executing planned movement paths and doing science observations, this is not precise enough. Furthermore, precise movements require a gearbox correction which depends on the absolute position of the arms. Therefore, an alternative movement configuration method, named `configPaths()` is provided, which avoids any accumulation of rounding errors, and allows for transparent corrections. The reached target positions will always be within 0.5 steps of the desired target positions.

6.9.2. Loading and executing paths generated by the path generator software

The following code snippets shows how to load and execute paths which were generated by the prototype version of the path generator software. It assumes that the initialisation as in the `initialiseGrid.py` sample file has been performed, and that the file `"targets_7fp_case_2_1_PATHS.txt"`, is an output file from the path generator which contains FPU paths in degrees, and start from the default position of $(\alpha, \beta) = (0.0°, 0.0°)$.

```
1  import wflib
2  from numpy import ones
3  gd.findDatum(gs, timeout=DATUM_TIMEOUT_DISABLE)
4
5  wf = gen_wf(ones(7) * 180.0, 0)
6  gd.configMotion(wf, gs)
7  gd.executeMotion(gs)
8
9  gd.trackedAngles(gs)
10 list_positions(gs)
11
12 p = wflib.load_paths("targets_7fp_case_2_1_PATHS.txt")
13
14 gd.configPaths(p, gs)
```

For executing a path, the current position of the FPUs has always to be exactly at the point where the path starts, with a maximum error of at most 0.5 steps. We assume here that the generated paths start from the default position at (0°, 0°). Also, the FPUs need to be calibrated by executing a datum command. To go to the default position, the FPU's alpha arms need to be moved by +180.0°.

Lines 1 to 7 in the code sample show how to accomplish that. First, two auxiliary libraries are imported. `wflib` provides the loading routine for the path file format. It depends on the Python mocpath module being installed and available on the PYTHONPATH. Also, the `ones()` function from numpy library is used to generate a movement to the default position. In this example, we assume a number of seven FPUs which are active.

The call to `gen_wf()` generates a waveform which moves all FPUs from the datum position to the default position.

After this, the `load_paths()` function in line 12 loads the path data into a Python data structure. In the next statement, the driver's `configPaths()` method checks this path for conformity with the waveform rules for the current firmware, and, if everything is right, it sends the data to the FPUs. As the firmware and knowledge about the possible FPU control parameters is still evolving, it is possible to switch the checks to different rule sets which are identified with a version number. As an example, ruleset version number 0 disables all checking for dynamic properties, so that only checks on the safe range of the movements are performed.

This rule set version number needs to be passed to the ruleset_version keyword parameter of the load_paths() method, like this:

```
gd.configPaths(p, gs, ruleset_version=0)
```

Apart from the rule set version, the configPaths() method accepts all parameters which the configMotion() does accept. The full range-checking for safety of waveforms is used.

Identical to a relative waveform, a path can be reversed by the reverseMotion() command. At this level, the EtherCAN interface does not check paths for conflicts or collisions, so that the path planning software needs to ensure that no conflicts will occur.

6.9.3. Manually configuring movement paths

It is possible to generate the absolute movement paths which are input to the configPaths() method manually. The input format is as follows:

```
    paths = { fpu_id1 : ([a0,a1,a2,a3, ...., an],
                         [b0,b1,b2,b3, ...., bn]),
              fpu_id2 : .... }
```

Here, fpu_id1 is the zero-based number of the FPU in the driver, a0,a2,a2 ... an are the alpha angles, and b0, b1, b2, b3, ... bn are the beta angles. The angles are in radian (that's the format which the path planning software uses), and a0 and b0 have to match the current position of the corresponding FPU exactly - otherwise the path is rejected.

After executing the path using executeMotion(), an and bn will be the final position when the waveform finishes executing, rounded to whole steps using round(), and with a maximum error of half a step.

6.10. Locking and unlocking FPUs

In the instrument operation, it can be necessary to block specific FPUs from further movements. This is for example the case if an FPU is damaged, or close to a damaged unit. The affected FPUs can be brought into a safe position, and then their activity disabled, so that they would not participate in movements. The way to do this is using the lockFPU() and unlockFPU() driver methods:

```
gd.lockFPU(fpu_id, grid_state)

gd.unlockFPUs(fpu_id, grid_state)
```

Locking an FPU is possible in any state except when they are currently moving (in this case, they would need to be stopped first, using an abortMotion() command). When locked, an FPU ignores any command except the unlockFPU() or the resetFPUs() command.

When FPUs are unlocked, they return, as far as possible, to their previous state.

6.11. Example scripts

Section A on page 118 describes a number of additional example scripts which can be used and modified to perform specific tests.

The script `CalibrationPositioning.py` is a small program which takes several options and can perform some basic burn-in or lifetime testing of FPUs. It can be used as a starting point for more elaborated engineering and verification software. Apart from the FPU EtherCAN interface commands documented in this document, all other programming constructs and functions are explained in the Python Library Reference[5].

[5]see `https://docs/python.org/2.7/`

7. Error handling and exceptions

When a driver methods encounters an error, such as an invalid command, a Python exception is raised. Exceptions are grouped into a hierarchy, and the exceptions generated by the driver are listed in the reference part in section 21.1 on page 96. Each exception is accompanied by a specific symbol and a string which in more detail describes the cause of the error.

Exceptions fall roughly into the following categories:

SetupError An invalid configuration

InvalidParameterError Invalid parameters to a command, for example addressing an FPU ID which is not existent or not configured.

InvalidStateException Issuing a command which is invalid at the current state of an FPU, or of the EtherCAN interface as a whole.

MovementError Errors which result from collisions, limit switch breaches, or abortions of movements.

SystemFailure Rarely, errors which are caused by the depletion of memory or other system resources.

ConnectionFailure Errors resulting from a failure of network connection, or FPUs failing to respond.

The detailed explanation for each error code is replicated in section 21.2 on page 96.

Exceptions can be handled automatically to support automatic testing, and correct discovery of fault conditions, such as collisions. A full explanation on how to handle exceptions in Python can be found at `https://docs.python.org/2.7/tutorial/errors.html`.

8. Health log of FPU counters

For each `executeMotion()` and `findDatum()` command, the FPU driver maintains a set of counters which are stored in the position database, and a time series of that counters in a second database called “health log”.

The content of this database can be listed using the command

```
fpu-admin healthlog  ${serialnumber}
```

Each database record is indexed by the FPU serial number and the count of the `findDatum()` command.

This constant of each record the following fields:

unixtime The UNIX time (number of seconds since January 1, 1970). This corresponds to the time stamps in the EtherCAN interface log.

datum_count Count of datum operations for this FPU.

executed_waveforms Number of executed waveforms.

alpha_starts Number of times the alpha arm started to move.

beta_starts Number of times the alpha arm started to move.

total_alpha_steps Total step count of the alpha arm.

total_beta_steps Total step count of the beta arm.

alpha_direction_reversals Number of times the alpha arm reversed direction.

beta_direction_reversals Number of times the beta arm reversed direction.

sign_alpha_last_direction Sign of last alpha movement.

sign_beta_last_direction Sign of last beta movement.

collisions Number of collisions detected for this FPU.

limit_breaches Number of alpha limit breaches detected.

datum_timeout Number of time-outs for the datum command (including both CAN time-outs, and hardware time-outs).

movement_timeout Number of time-outs during movements.

can_timeout Number of CAN timeouts during both datum and movements.

alpha_aberration_count Number of counted alpha aberrations, that is the number of steps which the alpha step counter had when the datum switch was found the last time. This count does not include uninitialised datum searches.

beta_aberration_count Number of counted beta aberrations.

datum_sum_alpha_aberration Sum of alpha aberrations, which can be used to compute the mean.

datum_sum_beta_aberration Sum of beta aberrations.

datum_sqsum_alpha_aberration Sum of squares of the alpha aberration counts, usable to compute the standard deviation.

datum_sqsum_beta_aberration Sum of squares of the beta aberration counts.

9. Logging

9.1. Log files

Contents

By default, the EtherCAN interface logs commands and CAN messages to text files which are created in the subdirectory `_log` of the current work directory. Each log file is tagged with a ISO 8601 time stamp[1] of the driver start-up time, and log data is organised in three different files:

`_{start_timestamp}-fpu_control.log` contains all send high-level commands, and the resulting state of the FPU grid.

`_{start_timestamp}-fpu_tx.log` contains all CAN messages which were sent to individual FPUs.

`_{start_timestamp}-fpu_rx.log` contains all CAN messages which were received, including errors messages sent because of collisions and limit switch breaches.

9.2. Logging options

Details of logging can be adjusted by passing keyword arguments to the driver constructor. The available options are:

logLevel – sets the logging level. Possible log levels are listed in table 9.1 on page 52. The default level is currently `LOG_TRACE_CAN_COMMANDS`. If the environment variable `FPU_DRIVER_DEFAULT_LOGLE` is set, its value is used as the default level.

[1] `http://en.wikipedia.org/wiki/ISO_8601`

log level	**Description**
LOG_ERROR	Log only critical errors and important warnings, such as collision messages and message time-outs.
LOG_INFO	Also log summary of each command send to the FPU grid, and overall statistics for FPU states (e.g. number of FPUs which have reached the datum position)
LOG_GRIDSTATE	Additionally, log the movement targets for each FPU and detailed state of the whole FPU grid on completion of each command. This level will generate a larger amount of data but will not affect responsiveness of the message processing within the EtherCAN interface. It is tailored to reconstruct any problem with collisions or hardware defects during normal instrument operation and, when necessary, help to improve collision recovery strategies.
LOG_VERBOSE	Log details of each command sent to each FPU (e.g. sent waveform tables). Also, logs the grid state while waiting for the completion of movement commands. This level will generate a large amount of data but should usually not affect responsiveness of the EtherCAN interface. This level of logging is appropriate e.g. when debugging the generation of waveform data from the path analysis layer.
LOG_DEBUG	Log details on CAN response time-outs and any information which might be helpful to diagnose problems. This level is intended for debugging the EtherCAN interface software.
LOG_TRACE_CAN_MESSAGES	Additionally, log hex dump of binary data of each CAN message as it is sent to the FPUs and each CAN response. This data will be logged to the …rx.log and …tx.log files. This level will generate a very large amount of data and is intended for debugging issues with the CAN message generation, the CAN protocol itself, or issues with the FPU firmware. Because messages are sent from within high-priority event loops, enabling this level will degrade the responsiveness of the EtherCAN interface and might delay processing of critical error responses. It is not designed to be used during normal instrument operation.

Table 9.1.: Log levels of the FPU EtherCAN interface

log_dir – sets the directory or folder into which the log files are written. A tilde followed by a slash ("~/") expands to the user home directory, also environment variables (e.g. `$LOG_DIR` are expanded. The default value is `./_logs`. If the leaf directory does not exist when the driver is started, it is automatically created.

If the environment variable `FPU_DRIVER_DEFAULT_LOGDIR` is set, its value is used to set the default log directory. For long-running tests with multiple FPUs, this directory should be adjusted to point to a file system path which has plenty of free space.

start_timestamp Allows to change the timestamp prefix which is inserted into the names of the individual log files. Adjusting this parameter might be used to set different names for driver instances which connect to different gateways, for example. The default value is `"ISO8601"` which is expanded to a string of the form `yyyy-mm-ddThh:MM:SS`, denominating the current time in ISO format.

9.3. Display of time in the logs

9.3.1. Used time format in logs

At the beginning of each line, the log files show the value of the UNIX time as time stamp, which is the fractional number of seconds since January 1, 1970 in the UTC time zone.

9.3.2. Converting to and from UNIX time stamps

To convert the timestamps in the log files, the following shell commands are useful:

```
  # display the current time in the local time zone
moons@moons-pc:~/MOONS/fpu_driver$ date
Fri 18 May 12:34:27 BST 2018
```

display the current system time, in the local time zone.

```
  # display the current time as UNIX time value
moons@moons-pc:~/MOONS/fpu_driver$ date ``+%s''
1526643274
```

display the time as UNIX time stamp.

```
  # convert the UNIX time stamp "152657976" to local time
moons@moons-pc:~/MOONS/fpu_driver$ date -d ``@1526643274''
Fri 18 May 12:34:34 BST 2018
```

allows to convert a time stamp found in a log file to the local time.

```
  # convert the UNIX time stamp "152657976" to UTC, and display it
moons@moons-pc:~/MOONS/fpu_driver$ TZ=UTC date -d ``@1526643274''
Fri 18 May 11:34:34 UTC 2018
```

displays a given time stamp in the UTC (universal coordinated time) time zone.

9.3.3. Relationship to internal time-keeping

Calendar dates and times are not always continuous, due to clock adjustments and introduction of leap seconds. In some cases, this can cause problems, for example spurious time-outs can appear.. Therefore, the EtherCAN interface uses internally a different, monotonic clock, which is basically the number of seconds since the system start. When certain fields are logged, such as the time when an FPU received its last command, these times are converted to the UNIX timestamp, too.

10. Recovery from collisions and limit breaches

Contents

Because the FPUs are controlled individually, and the EtherCAN interface has absolutely no concept of their geometric configuration, it cannot take care to move them so that collisions are avoided. Instead, the hardware needs to handle collisions in a way that no damage results and communicate such situations back to the higher layers of the software. The following section explains how this is achieved.

10.1. What happens when a collision or limit switch breach occurs

10.1.1. Detection of obstacles

Situations in which the movement of the hardware is obstructed are either collisions between FPUs or limit switch breaches of the alpha arm. Limit switch breaches are detected when the alpha arm detects a transition where the limit switch is on and goes off for a clock-wise movement, or an activation of the switch for a counterclockwise movement. All other cases of obstruction are considered collisions, which is detected by electrical circuits that protect the beta arm. This includes movements where the beta arm is moved out of its allowed range and hits the hard stop.

When a collision or limit switch breach occurs, the FPU electronic hardware stops any movement, and sends a message to the EtherCAN interface. On receiving such a message, the EtherCAN interface changes the state of the FPU in the internal `grid_state` structure. This is reflected in the status flags of the FPU, which were discussed above in section 2.7 on page 8. The state field of the FPU is set to the value OBSTACLE_ERROR, the flags beta_collision and at_alpha_limit are set accordingly, and the flag was_referenced is cleared.

10.1.2. Errors are turned into exceptions

If any movement operation or findDatum() command is going on, this command returns with the updated state information on the collision, and, depending on the used driver method, without further waiting for the movement to complete. When the driver call returns to the Python wrapper, the error is checked for and transformed into a Python exception of the class fpu_driver.MovementError. The sub-classes (CollisionError, LimitBreachError, AbortMotionError, StepTimingError) allow to discern the cause of the error in try . . . except clauses. If the generated exception is not caught, this terminates the Python script.

10.1.3. Collisions and limit switch breaches while the FPUs are not moving

It is also possible that a collision occurs while an FPU is *not* moving and the EtherCAN interface not executing a command. In this case, the internal grid_state data kept by the EtherCAN interface is changed, too. However, any grid_state variable within Python which was returned form an earlier command, will not be updated automatically. When the user tries to launch a new command, the changed state is detected, and if the new command is not valid in a collided state, the called method updates the grid_state variable and also raises a second exception, without trying to execute the command.[1] The current state can always be retrieved or refreshed by calling the getGridState() method.

The following recipe assumes that the task of determining in which direction FPUs should be moved after a collision is already solved. In the general case, this can be a complex question. The grid_state structure is designed to provide a host of information for making this decision.

10.2. Resolution of a beta arm collision

.

In case of a collision of a beta arm, the driver provides two special member functions of the grid driver object for resolving this. The following snippet shows how to use them:

```python
from FpuGridDriver import REQD_CLOCKWISE, REQD_ANTI_CLOCKWISE
fpu_id = 3
grid_state.FPU[fpu_id].direction_beta
gd.freeBetaCollision(fpu_id, REQD_CLOCKWISE, grid_state)
# required for firmware version 1
```

[1]Because movement methods return on the first collision, but in a multi-FPU grid additional collisions can occur after that, when controlling multiple FPUs it is a good idea to retrieve an updated grid state structure after handling any collision.

```
6   gd.pingFPUs(grid_state)
7   gd.enableBetaCollisionProtection(grid_state)
```

The effect of these lines is as follows:

In line 1, the symbols `REQD_CLOCKWISE` and `REQD_ANTI_CLOCKWISE` are imported. They are used to define the movement direction. Then, the parameter `fpu_id` is set to the numerical ID of the FPU, which is zero for a single-FPU grid. We retrieve the last direction of movement of the arm using the `direction_beta` field of the FPU status record. Then, calling the method `freeBetaCollision()` moves the collided FPU into the direction passed in the second parameter - clockwise, or anti-clockwise, as determined from assessing the FPUs arm positions.

Calling `freeBetaCollision()` might need to be repeated a few times, and verified using visual inspection, or camera pictures. When the collision is resolved, the FPU can be switched to the normal state using the `enableBetaCollisionProtection()` command. After this, the FPU can be moved normally. Because any previously configured waveforms become invalid, a new movement needs to be configured using `configMotion()` at this point. To make movements precise, a new datum search is required at that point.

10.3. Resolution of an alpha limit switch breach

For firmware and CAN protocol version 2, limit switch breaches can be handled in the same way as beta arm collisions, using the corresponding commands `freeAlphaLimitBreach()` and `enableAlphaLimitProtection()` for the alpha arm:

```python
from FpuGridDriver import REQD_CLOCKWISE, REQD_ANTI_CLOCKWISE
fpu_id = 3
grid_state.FPU[fpu_id].direction_alpha
gd.freeAlphaLimitBreach(grid_state)
# required for firmware version 1
gd.pingFPUs(grid_state)
gd.enableAlphaLimitProtection(grid_state)
```

WARNING: When resolving an alpha arm limit switch breach, make sure to move the alpha arm away from the hard stop. No collision protection is active in this state!

10.4. Changes in CAN protocol version 2

The following are the main changes between protocol version 2 and the older version 1:

- Alpha limit switch breaches can be recovered automatically. The risk of damage caused by starting a datum search during a limit switch breach is reduced.
- In protocol version 2, transitions and allowed commands are checked much stricter than in protocol 1. The stricter checking allows to make stronger assumptions about the current state, and that makes it possible to perform automatic recovery of multiple collided FPUs.
- Protocol 2 is able to detect lost messages in almost all situations. This is accompanied with robustness changes of message handling in the FPU firmware and the EtherCAN gateway.
- It is possible to lock and unlock individual FPUs, which can be used to make sure that they are not moved unintentionally.
- Waveforms can be longer and waveform upload time can be tweaked to be faster.

11. Outline on management of multiple FPUs

This section attempts to give a broad overview on how a grid with 1000 or more FPUs can be managed by higher levels of the instrument software.

11.1. Retrieving individual states

The EtherCAN interface software can effortlessly send and receive commands to up to 1005 FPUs. The only requirement is that the grid driver object is initialised with that number of FPUs, and the corresponding number of EtherCAN gateways is connected. When the `grid_state` variable is retrieved, it contains the states of *all* FPUs in the `grid_state.FPU[]` sequence. So, in order to know what the state of FPU number 900 is, we simply have to query `grid_state.FPU[900]`.

11.2. Accessing the counts of states in an FPU grid

To make managing such a high number of FPUs easier, there exist a few additional functions. One feature is that `grid_state` variable has a member named `Counts`, which is an array that summarises the state of all FPUs. As discussed in section 2.7 on page 8 and the following section, each FPU has a field with the name "`state`". Its type is an enumeration value describing its state. Table 2.1 on page 12 lists these enumeration values.

When the enumeration symbols are converted to integer indices using `int()`, the array elements of `Counts` contain for each index the number of FPUs which are in the state defined by that index. For convenience, `str(grid_state)` displays `Count` as a dictionary with the state names as keys, and the count numbers as values. So, the expression

```
grid_state.Counts[int(FpuGridDriver.FPST_RESTING)]
```

returns the number of FPUs in `RESTING` state. Because FPUs are always in exactly one state, the numbers in `Counts` always sum up to the number of FPUs.

11.3. Summarising the FPU grid state

In addition, there is a function with the name `getGridStateSummary(grid_state)` which maps the different states of the set of FPUs to a single value. Such a mapping reduces the dimensionality of the input data, and therefore there is more than one way to define it. The idea followed here is simply to define an ordering of all state enumeration values. The used ordering is

```
UNINITIALIZED < OBSTACLE_ERROR < ABORTED < DATUM_SEARCH
        < AT_DATUM < READY_FORWARD < MOVING < RESTING
```

The return value of the function delivers the *smallest* state value in which at least one FPU is. That means, if a single FPU is in state UNINITIALZED, and all other FPUs are in state READY_FORWARD, the function would describe the state of the grid as GS_UNINITIALIZED. This allows to get quickly an overview which operations can be done consistently. For example, if the grid is in state GS_READY_FORWARD, all FPUs are ready to move. In the same way, if 990 FPUs are in state "RESTING", 14 in state "ABORTED", and one in state "OBSTACLE_ERROR", the resulting overall state is OBSTACLE_ERROR.

11.4. Thread-safety and concurrency

All EtherCAN interface methods are thread-safe, so they can be used in a GUI environment, for example.

It is possible to inquire the state of the grid and of course the known positions of all FPUs while movements are happening, using the getGridState() member function of the grid EtherCAN interface object. Currently, commands are only processed one at a time - this restriction does not has purely technical reasons, but simply makes management substantially less complex. The exception from this is the abortMotion() method, which can be sent at any time, from any thread.

11.5. Waiting for movement operations to finish

Finally, there exists a low-level EtherCAN interface method, waitForState() which waits either for a specific time-out to happen, or for a specific state of the grid to be reached. It always returns the updated grid state.

The combination of these facilities makes it easy to perform stream-lined group operations, while providing full information on the detailed state of things in cases when, for example, FPUs have a collision and need to be disentangled guided by geometric information and a high-level resolution strategy.

Part III.

Reference

12. Aspects relevant for all commands

12.1. Retrieving the EtherCAN interface version

There are two attributes of the `FpuGridDriver` module which allow to retrieve version information:

The standard attribute `FpuGridDriver.__version__` returns the latest version label of the EtherCAN interface. The first letter is stripped, so that “v2.0.3” turns into “2.0.3”. This version number is authoritative when testing whether a EtherCAN interface API or documentation is valid for the installed module.

The attribute `FpuGridDriver.CAN_PROTOCOL_VERSION` displays the major version number of the CAN protocol against which the EtherCAN interface was compiled, it can be “1” or “2”. This version number needs to match the version of the firmware which is running on the FPUs, otherwise the EtherCAN interface will not work (and will even fail to produce meaningful error diagnostics). The version numbers of the FPU firmware can be retrieved and checked using the `printFirmwareVersion()` and `minFirmwareVersion()` commands which are described in sections 15.2 and 15.3 on page 71.

12.2. Addressing a subset of FPUs

In some situations, it is required to send commands only to a sub-set if fibre positioners. In the ICS, this is the case in certain recovery scenarios. In the FPU verification system, the need for this arises when several FPUs are installed on the kinetic mount and connected to the same gateway. Here, we want to move and test only one FPU at a time. The normal mode of commands like `findDatum()` and `executeMotion()` would move all FPUs which are connected.

To restrict the group-wise commands to a sub-set of FPUs, all corresponding commands have a parameter `fpuset`. This parameter is by default an empty list, which has the effect that the command is sent to all FPUs. If the parameter is set to an integer list of FPU Ids, the command is sent only to the listed FPUs. For example, the command

```
gd.findDatum(gs, fpuset=[1,3,5,7,11])
```

will send a `findDatum()` command only to the FPUs with the ids 1,3,5,7, and 11.

An invalid FPU Id will cause a `INVALID_PARAMETER_EXCEPTION` to be thrown.

For details how subsets are handled in the `configMotion()` command, see section 17.1.

This feature requires a EtherCAN interface version >= 0.6.0 and works with all firmware versions (as long as the firmware supports the specific command mode).

Note that using the `fpuset` parameter is not in all cases the best option if FPUs need to be handled independently. An alternative approach id to instantiate separate driver objects which connect to different gateways. In such an set-up, each gateway will need an individual IP address, and should also have an individual MAC address. This can be done either within the same python program, or in different programs which can operate completely independently.

13. Driver and connection initialisation

Contents

13.1. Instantiating the driver class

The driver is initialised by calling the class constructor `FpuGridDriver.GridDriver()`, and assigning it to a Python variable:

```
from FpuGridDriver import GridDriver

NUM_FPUS = 20 # set number of FPUs to 20
gd = GridDriver(NUM_FPUS)
```

The constructor has one integer parameter, which is the number of FPUs which will be used. The maximum value for this number is defined at compile time in the EtherCAN interface, and is currently 1005.

The numbering of FPUs and buses and the way hardware addresses are mapped to FPU IDs are currently fixed. Future versions will allow for arbitrary numbering.

If the running OS is lacking system resources, for example if the memory is exhausted, initializing the EtherCAN interface might return either a Python MemoryError exception, or an `fpu_driver.SystemFailure` exception. In the latter case, the detailed resource which is lacking will be indicated in the error message, and in the log output.

The class constructor has several optional arguments:

```
gd = GridDriver(NUM_FPUS,
                SocketTimeOutSeconds=20.0,
                confirm_each_step=True,
                waveform_upload_pause_us=0,
                configmotion_max_retry_count=5,
                configmotion_max_resend_count=10,
                min_bus_repeat_delay_ms = 2,
                min_fpu_repeat_delay_ms = 4,
                alpha_datum_offset=ALPHA_DATUM_OFFSET,
                logLevel=DEFAULT_LOGLEVEL,
                log_dir=DEFAULT_LOGDIR,
                motor_minimum_frequency=MOTOR_MIN_STEP_FREQUENCY,
                motor_maximum_frequency=MOTOR_MAX_STEP_FREQUENCY,
                motor_max_start_frequency=MOTOR_MAX_START_FREQUENCY,
                motor_max_rel_increase=MAX_ACCELERATION_FACTOR,
                firmware_version_address_offset=0x61,
                protection_logfile="_{start_timestamp}-fpu_protection.log",
                control_logfile="_{start_timestamp}-fpu_control.log",
                tx_logfile = "_{start_timestamp}-fpu_tx.log",
                rx_logfile = "_{start_timestamp}-fpu_rx.log",
                start_timestamp="ISO8601")
```

Arguments which control the logging are explained in section 9. Additionally, the following keyword arguments can be passed:

alpha_datum_offset This parameter defines the alpha angle convention at the datum position. By default, it is set to -180.0, this means that the alpha arm at the datum position will have an alpha angle of $-180.0°$. This angle is primarily used when waveform tables and movements are logged by the EtherCAN interface (see also the keyword argument with the same name in the countedAngles() method and the list_angles() utility function).

SocketTimeOutSeconds The time-out value which defines after how much time a socket connection is considered lost and the connection is closed. This should only happen when there is a networking problem in the connections to the EtherCAN gateways. (The time-out threshold which defines the maximum waiting time for a non-responding FPU is defined by the protocol).

motor_minimum_frequency=500 Is the minimum allowed frequency of the stepper motor which the EtherCAN interface will accept when checking waveforms for validity. Together with the length of a waveform segment this frequency directly defines the minimum number of steps which need to occur in a waveform section. For example, in combination with a waveform segment length of 250 milliseconds and a minimum frequency of 500 Hz, at least 125 steps are needed. This limit is equal for the alpha and beta arms. The translation of this number of steps per second is defined by the gear ratio, and is different for alpha and beta arm.

Important: **This is an unsafe low-level control value which is not subject to any further safety checks. Changing this parameter to a value which is too low will drive the FPU motors into physical resonances which can lead to a loss of control. Any changed value needs to be agreed and vetted by a hardware engineer.**

motor_maximum_frequency=2000 As above, this value defines the *maximum* number of motor steps per second. The above warning about safety applies to this parameter as well.

motor_max_start_frequency=525 Analogous to the above two items, this parameter defines the maximum number of steps at the beginning and ending of a waveform. Ideally, the value should be equal to the minimum value. However, due to the data flow in the control pipeline, it is sometimes possible that steps are slightly incremented or decremented in rounding and gearbox correction steps. Therefore, this value provides some tolerance for such deviations. The above warning for `motor_minimum_frequency` applies to this parameter, too.

motor_max_rel_increase=1.4 This defines the maximum relative factor with which the speed can change between two waveform sections. It applies symmetrically to both increasing and decreasing speeds: The larger value must be not larger than the smaller value times this factor.

waveform_upload_pause_us This keyword parameter allows to configure a small pause between sending waveform steps to the same FPU. The value is an integer number in microsecond units, and for EtherCAN interface version 1.2.0 and newer, it is 50 000 by default. If the value is zero, no waiting is done.

This parameter is deprecated because its effect is is somewhat unreliable, use `min_fpu_repeat_delay_ms` instead.

confirm_each_step This keyword parameter is a Boolean value which controls whether the EtherCAN interface waits for a confirming response for each waveform element that is sent to an FPU. Because each response requires a CAN message, this can roughly reduce the free capacity of the CAN buses by a factor of two. For the 1.2.0 EtherCAN interface and subsequent versions, the default value is `True`.

Important: As of EtherCAN interface version 1.3.0, this option is not yet fully implemented, because the correct implementation requires significant structural change to the EtherCAN interface. Correspondingly, the waveform upload can fail.

can_command_priority This parameter allows to set the minimum priority value for CAN commands from the driver to the FPUs. The lower the value, the higher the actual priority. The default value is 3, which will work for firmware versions >= 1.6.0. Earlier firmware versions will require a value of zero - otherwise, commands will not be recognised. To avoid problems with missing CAN responses or capacity problems in the firmware, it is recommended to use the default value.

configmotion_max_resend_count This parameter defines how often the `configMotion()` command should silently try to resend CAN messages if a CAN time-out is encountered. Setting this value to a number higher than zero increases robustness against connection problems during path configuration. (It does however not solve connection problems in general).

configmotion_max_retry_count Similar to the preceding parameter, this parameter defines how often the `configMotion()` command should try to repair a failed configuration command by issuing the command again for the FPUs which reported failures. As above, this can enhance robustness, because configMotion messages are the most frequent type of messages, but it cannot recover from lost movement commands.

min_bus_repeat_delay_ms This parameter allows to set a rate limit on CAN commands which go to the same CAN bus. The value needs to be between zero and 255. If set to a non-zero integer value of n, the driver will make sure that "dummy delay" messages are inserted before each CAN command which have the effect that no CAN command goes to this bus within less than $n - 1$ milliseconds. The larger this value is, the smaller is the number of FPU commands which can be send per second, and the longer the upload of waveforms will take. The best throughput is probably achieved if messages are sent in a round-robin manner to the different buses on the same gateway - this will minimise the waiting time for each bus.

min_fpu_repeat_delay_ms Similar as above, this parameter allows to set a rate limit for commands to individual FPUs. If several commands are sent to the same FPU, a value of n makes sure that delay commands are inserted which should warrant that no two commands arrive at the same FPU within less than $n - 1$ milliseconds. If the value of this parameter is equal or less than `min_bus_repeat_delay_ms`, it has no extra effect. This limit provides more reliable control than limits set using `waveform_upload_pause_us`, and is therefore preferable.

This rate limit allows to quickly check whether timing of messages arriving at the same FPU might cause errors or different behaviour.

Note that the above rate control commands also affect `abortMotion()` commands. Typical sensible rate limit values are about one or two milliseconds for all commands to the same bus, and up to about 20 milliseconds for commands to the same FPUs. Depending on the EtherCAN gateway implementation, it might be possible to set the limit to zero.

13.2. connect()

The method `FpuGridDriver.connect()` establishes a connection to a list of EtherCAN gateways. The configuration of gateways ip address and port numbers can be passed by the keyword parameter `address_list`. By default, this is using the IP `192.168.0.10` and the port number `4700`, which as of May 2018 matches the current gateway prototype setup. This default gateway setting equals the constant `FpuGridDriver.TEST_GATEWAY_ADRESS_LIST`. For using the EtherCAN mock-up, the value `FpuGridDriver.MOCK_GATEWAY_ADRESS_LIST` can be used.

If the connection fails hard, the command will return a `ConnectionFailure` exception. It is in some cases possible that the initial `connect()` command is successful, but the first command which accesses the FPU grid fails because setting up the connection has timed out. In this case, the failing command will also return a `ConnectionFailure` exception.

It is possible that different driver instances are initialised at the same time, which connect to different gateways, that is, each instance controls its own grid, or group of FPUs.

14. Configuring FPU parameters

The following commands allow to set certain parameters in the FPU controller to values which are different from default values. They might be used for engineering. Normally, they can be only set while an FPU is in UNINITIALIZED state, and they should be set to the same values for all FPUs in an instrument grid.

14.1. writeSerialNumber()

The command

```
gd.writeSerialNumber(fpu_id, snstring, grid_state)
```

sends the serial number in the string snstring to the FPU with ID fpu_id and writes it to the FPU controller NVRAM.

For CAN protocol version 2, the serial number can be any unique sequence of up to six printable ASCII characters, for example "PT1001".

The command will be rejected and will return the error value DE_DUPLICATE_SERIAL_NUMBER when the serial number already is stored in another connected FPU in the grid.

This command is supported by the UnprotectedGridDriver class, which does not provide protected movement commands. This allows to configure FPUs which initially do not have a unique serial number, because they have identical firmware.

The command is only allowed when an FPU is in state UNINITIALISED.

14.2. setTicksPerSegment()

The command

```
gd.setTicksPerSegment(nticks,  grid_state, fpuset=[])
```

sets the number of clock ticks of the FPU motion controller per movement segment in a waveform table to the passed value, with a base unit of 100 nanoseconds. The default value is 250 milliseconds.

The command is only allowed when an FPU is in state UNINITIALISED.

14.3. setStepsPerSegment()

The command

```
gd.setStepsPerSegment(min_steps, max_steps,  grid_state, fpuset=[])
```

This command sets the minimum number and maximum number of motor steps in a time segment of the planned path. Together with setTicksPerSegment, this defines the minimum step frequency which the motor controller uses. The maximum number is (depending on the firmware version) used for bonds checking before responding to configMotion messages. The minimum value is not actually a lower bound for waveforms: Frames which have a smaller number of steps than the minimum number of steps are allowed, as long as they are permitted by the used waveform rules and assuming that the firmware can approximate the movement with a combination of pulses and pauses.

The command is only allowed when an FPU is in state UNINITIALISED.

14.4. setUStepLevel()

The command

```
gd.setUStepLevel(ustep_level,  grid_state, fpuset=[])
```

sets the microstepping level of the FPUs motion controller to the passed value. Only the values 1,2,4,8 are possible. The larger this value is, the better the precision of the movements. However, a higher microstepping level also requires a higher computational effort in the FPU controller, which can cause step timing errors in the execution of waveforms. Such errors cause a a movement to be aborted and set the FPU into an ABORTED state.

The command is only allowed when an FPU is in state UNINITIALISED.

15. Querying FPU state and configuration

Contents

15.1. getFirmwareVersion()

After a connection to an FPU grid is established, the command

```
gd.getFirmwareVersion(grid_state)
```

retrieves the firmware version of the motion controllers of the attached FPUs and stores them in the grid state variable. The version information is stored in the strcut for each FPU in the fields `fw_version_major`, `fw_version_minor`, `fw_version_patch`. The firmware creation date is stored in the fields `fw_date_year`, `fw_date_month`, and `fw_date_day`).

The command can be restricted to a certain subset of FPUs by passing a list of integer FPU IDs to the parameter `fpuset`, like this:

```
gd.getFirmwareVersion(grid_state, fpuset=[0,1,2,5,10])
```

15.2. printFirmwareVersion()

The method retrieves the version and creation data of each firmware on the FPUs motion controllers, and prints them in a list.

As the `getFirmwareVersion()` method, the method accepts an `fpuset` keyword parameter which restricts the output to the fpu IDs listed in the parameter.

15.3. minFirmwareVersion()

The method retrieves the firmware version from the attached FPUs and returns a three-element tuple with the smallest firmware version. The elements of the tuple are major version, minor version, and patch level.

The method is designed to be used in scripts in combination with Python's `assert()` statement, and the property that Python tuples can be compared like strings. For example, the statement

```
assert(gd.minFirmwareVersion() >= (1,3,0))
```

will either make sure that the firmware version is at least 1.3.0, or abort the running script.

15.4. readSerialNumbers()

The command

```
gd.readSerialNumbers(grid_state)
```

reads the serial numbers of the FPUs and places them into the `grid_state` variable into the field named `serial_number`. For CAN protocol version 2, the serial number can be any sequence of up to six printable ASCII chars.

15.5. printSerialNumbers()

The command

```
gd.printSerialNumbers(grid_state, fpuset=[])
```

reads and prints the serial numbers of the FPUs. The `fpuset` parameter is a list of the FPU IDs which will be printed. By default, the numbers for all FPUs will be displayed.

15.6. checkIntegrity()

The command

```
gd.checkIntegrity(grid_state, fpuset=[], verbose=True)
```

computes a firmware CRC32 check-sum on all addressed FPUs, and returns the 32-bit check-sum in the member field 'crc32' of the grid_state variable for the corresponding FPU. By default, the result is printed, when the command is issued like this:

```
gd.checkIntegrity(grid_state, fpuset=[])
```

This checksum allows to verify whether a firmware image has the intended data. For example, it can detect when the image in the motion controller flash memory was altered due to memory corruption, or if the firmware image was changed and the version number update was forgotten. The checksum is computed in a way such that it is not affected by changing the serial number.

This command is intended to be run regularly in the operational FPU grid, to detect any system failures at an early stage. The "verbose" keyword allows to suppress the printing of the CRC32 value. The following code shows how to retrieve the value for FPU number 17 as a number:

```
gd.checkIntegrity(grid_state, fpuset=[], verbose=False)
grid_state.FPU[17].crc32
```

Because it is somewhat time-consuming, the command can not be run when the FPU is moving or when it is in READY_FORWARD or READY_REVERSED state.

15.7. getGridState()

The method FpuGridDriver.getGridState() creates and returns an instance of the grid state object:

```
gd.getGridState(grid_state)
```

To update its content, it can be passed to the pingFPUs() command. Also, passing it to any other driver method will update it. After this, it will reflect the updated state and position of each FPU.

In the following descriptions, the state variable is named as grid_state, it can however have any other name which is valid for a Python variable.

Typically, the grid_state variable is updated by each call to a EtherCAN interface method. In case of movement errors such as collisions, it can however be useful to refresh its state by calling getGridState() again, because the state of the FPU grid might still be changing after the failed command has returned.

15.8. getSwitchStates()

The command

```
gd.getSwitchStates(grid_state, fpuset=[])
```

retrieves and prints the current state of the alpha limit switches and the beta limit switches of the FPUs. The fpuset parameter is a list of the FPU which will be queried. By default, the numbers for all FPUs will be displayed.

The result will look like that:

```
{0: {'beta_datum_active': True, 'alpha_limit_active': False},
  1: {'beta_datum_active': False, 'alpha_limit_active': False},
  2: {'beta_datum_active': False, 'alpha_limit_active': True},  }
```

15.9. pingFPUs()

The method `FpuGridDriver.pingFPUs(grid_state)` connects to each FPU and refreshes the content of the `grid_state` variable. If an FPU cannot be reached within a time-out period, an `ConnectionFailure` exception is raised. The sub-type of the error can be `DE_NO_CONNECTION`, if the connection to an EtherCAN gateway was lost, or `DE_CAN_COMMAND_TIMEOUT_ERROR`, if one or more FPUs failed to respond. The total count of timed out responses is available in the field `grid_state.count_timeout` field.

After the first `pingFPUs()` command, the positions of the FPUs can be displayed using the functions `list_positions()` and `list_angles()`, which are described in section 19.1 and 19.2 on page 89 ff.

15.10. countedAngles()

The driver method

```
FpuGridDriver.countedAngles(grid_state, fpuset=[], show_uninitialized=False)
```

retrieves the current step counts and shows approximate current angle of both arms, in units of degrees. The angles are computed by multiplying the step counters with the correct gear ratio. The configured offset to the alpha datum position is applied, so that the datum position serves as the coordinate reference point.

If the step counters have not been initialised, the symbolic floating point value NaN (not-a-number) is returned. If the keyword argument `show_uninitialized` is passed with a value of `True`, instead of NaN values the values derived from the current counter positions are shown. In this case, the angles may or may not correspond to the current arm position.

The differences to the utility function `list_angles()` are that the method always retrieves the current angles, and also always uses the same alpha datum offset with which the driver was configured upon initialisation. Therefore, when it is intended to evaluate angles from a previously retrieved `grid_state` structure *without* modifying it, the function `list_angles()` needs to be used.

The command accepts an `fpuset` keyword argument which, when set, causes it to retrieve and list only values for the passed FPU IDs.

15.11. trackedAngles()

The driver method

```
  FpuGridDriver.trackedAngles(grid_state, fpuset=[],
                              show_offsets=False, active=False)
```

displays the current tracked positions of FPUs, the angles derived from the step counter values, and the offset between the two. It is useful to assess whether an intended movement command will be within the safe range of movements, and to view in which position FPUs have been left when they were powered off.

In addition, the method call shows also the interval of absolute angle positions which the currently configured movement, or the last configured movement will touch. If a `reverseMotion()` command is issued, these ranges are updated accordingly, taking into account the reversed movement.

An example output of the command looks like this:

```
  >>> gd.trackedAngles(gs)
  FPU 0: angle = (-180, 0), offsets = (0, 0), stepcount angle= (-180.0,
           0.0), last_wform_range=([],[])
```

Here,the current actual angle is at position alpha=-180°, beta=0° degrees. The angles derived from the step counts are identical, and the offsets are zero. When the FPU is moved by an angle of alpha=10 degree and beta = 20° degrees, the output looks like this:

```
  >>> w = gen_wf(10, 20)
  >>> gd.configMotion(w, gs)
  fpu_driver.E_DriverErrCode.DE_OK
  >>> gd.executeMotion(gs)
  fpu_driver.E_DriverErrCode.DE_OK
  >>> gd.trackedAngles(gs)
  FPU 0: angle = (-170.009, 19.9965), offsets = (0, 0),
         stepcount angle= (-170.00, 19.99),
         last_wform_range=([-180.0, -170.00],[0.0, 19.99])
>>>
```

This indicates that the FPU is now at an angle of (-170, 20) degrees. If a `resetFPUs()` command were issued, the result would look as follows:

```
>>> gd.resetFPUs(gs)
fpu_driver.E_DriverErrCode.DE_OK
>>> gd.trackedAngles(gs)
FPU #0: angle = (-170.0, 19.9), offsets = (9.94, 19.99), \
               stepcount angle= (-180.0, 0.0), \
               last_wform_range=([-180.0, -170.0],[0.0, 19.9])
>>>
```

If the positions of FPUs become uncertain for any reason, for example due to communication failures during a movement, or a collision, then instead of a single-value position, the interval of *possible* position values will be shown. These intervals are also tracked when uncalibrated movements are made, and when the driver is exited and started again.

The command accepts an `fpuset` keyword argument which, when set, causes it to retrieve and list only values for the passed FPU IDs.

Display options The "`show_offset`" flag shows the FPU step counters and the offset between step counters and tracked angles, when true.

If the "`active`" flag is set to true, it shows only active waveforms, which will be executed when an `executeMotion()` command is issued next. This does not display waveforms which may me present on the FPUs, but which are currently not in executable state.

(It is also possible that a previous waveform is displayed but actually not any more executable, for example if the FPU has been reset, or if it had a collision).

16. Calibration commands

16.1. findDatum()

The full signature of the findDatum() method is

```
FpuGridDriver.findDatum(grid_state,
                        selected_arm=DASEL_BOTH, fpuset=[],
                        soft_protection=True, count_protection=True,
                        support_uninitialized_auto=True,
                        timeout=DATUM_ENABLE_TIMEOUT)
```

In almost all cases, the automatic datum search is sufficient. The other options will be explained below.

Automatic datum search The method

```
FpuGridDriver.findDatum(grid_state)
```

moves the arms of the FPUs to the datum position. By default, this always moves both arms of all FPUs. The FPU driver automatically selects the required movement direction of the beta arm according to the tracked position. The latter behaviour can be disabled by setting the keyword parameter support_uninitialized_auto, which is enabled by default, to False.

Datuming a subset of FPUs If the keyword parameter fpuset is passed with a non-empty list of FPU ids, only the FPUs with these IDs are datumed.

Moving alpha and beta arms independently For testing purposes, it is sometimes needed to move both arms separately. In this case, the keyword argument selected_arm can be passed to move the arms individually like this:

```
gd.findDatum(grid_state, selected_arm=DASEL_ALPHA)
```

If the value of this parameter is DASEL_BOTH, it shows the default behaviour. If it is DASEL_ALPHA, only the alpha arm is datumed, if it is DASEL_BETA, only the beta arm is datumed.

Software protection The command will check with the FPU position database whether a datum search is possible and in which direction the FPU beta arms need to be moved. If a datum operation is not safe, a `ProtectionError` exception is thrown, and the command is aborted without moving any FPU.

In some error conditions, it is possible that the arm positions end up to be ambiguous, so that the needed direction for the arm movement is not clear. In this case, it is perhaps easiest to look at the output of the `trackedAngles()` command, and move the FPU so that all possible arm positions are at the same side of the datum switch. Alternatively, it might be necessary to update the position database with the true position, as described in section 4.9 on page 22.

Defining the search direction Using the `search_modes` keyword argument, it is possible to manually define the direction in which the beta arm will move when performing the datum search. Normally, this is not necessary as the position is known from the FPU position database, so the correct direction is filled in by the driver. A datum command with manually defined direction looks like this:

```
gd.findDatum(grid_state, {0: SEARCH_CLOCKWISE}, soft_protection=False)
```

This instructs the FPU to move clock-wise, and enable the driver to do this without checking the current position. Analogously, the constant `SEARCH_ANTI_CLOCKWISE` can be passed to move the FPU into the negative direction.

Such a manual search was required to be performed in previous versions of the EtherCAN interface, earlier than version 1.1.0. For current EtherCAN interface versions, while still possible, this is not necessary any more since the position database keeps track of the actual positions, when the system is powered off.

Disabling protection flags It is possible to disable the software protection by passing the keyword parameter `soft_protection=False`. In this case, no range check will be performed, and the user is responsible to make sure that the FPUs cannot be damaged. *This option can be destructive* if the alpha arm is in the dead zone, or if the beta arm is moved into the wrong direction. It is normally not needed. *To test the proper working of the datum switch, instead use the test described in section 5 on page 27 ff.*

Manual movement operations will be checked at a lower level against the step counters to avoid destructive movements. When an FPU is not initialized, such checks may fail. This low-level checks can also be disabled by passing the keyword parameter `count_protection=False`. Using this parameter is discouraged as it is only necessary to pass in exceptional circumstances.

WARNING: Only disable protection flags for the findDatum command if this is clearly needed and only if you are sure that it will not damage the FPU.

Refer to up-to-date additional hardware and firmware information outside of this manual to determine whether operations are safe.

Disabling hardware timeouts FPU firmware with version >= 1.4.3 includes a time-out check for the datum search. This check stops a datum search if no datum switch change was detected within a certain step number. For an initialized FPU, this number is derived from the internal step counter, and otherwise it is in the range of a few degrees. For some situations, it might be desirable to disable this hardware safety check for an uninitialized FPU. This can, for example, be perform a datum scan for an FPU which is newly installed, and powered on the first time.

To disable the check, one can pass the keyword parameter

```
timeout=DATUM_TIMEOUT_DISABLE
```

This mode is only valid when no selected FPU was initialized before, otherwise it will be rejected by the EtherCAN interface.

Result state of a datum operation After a normal termination of the command, all selected FPUs will be in `AT_DATUM` state. If one or both arms have not been initialised, the final state is `UNINITIALIZED`.

If there was any collision, limit switch breach, or step timing error at firmware level, the movement of that FPU is stopped and the command returns an exception of type `FpuGridDriver.MovementError`. The sub-type of the exception indicates the cause of the error; section 21.1 on page 96 lists the name of this sub-classes. In such a case, the "initialised" flags of the arms are cleared, as the step counter information cannot be longer deemed accurate, and a new manual datum search is needed to continue normal movements. If a new datum search is not possible, the `configMotion()` command can be used with the `check_proetction` flag set to False.

The `findDatum()` command can be interrupted by the `<Control>-<C>` keyboard combination, which causes a `SIGINT` signal to be delivered to the Python process. A programmable way to do the same is to send an `abortMotion()` command to the FPUs. In both cases, FPUs which were moving will be in `ABORTED` state. As in case of a collision, the "initialised" flags of alpha and beta arms are cleared, and a new datum search will be required before normal movements can be resumed.

If in an FPU, the controller firmware cannot detect a datum switch within a maximum time, it aborts the movement and sends a time-out message, leading to a `FirmwareTimeoutError` exception. This can be caused by either issuing the datum command in the wrong place, or by a malfunction in the hardware or firmware, such as a failing datum switch. If an FPU's alpha arm is initially *on* the datum switch, the command will be rejected for safety reasons, leading to a `ProtectionError` Exception with error code `DE_ALPHA_ARM_ON_LIMIT_SWITCH`. If the state is detected by the FPU, this leads to a `HardwareProtectionError`.

17. Commands for configuring and executing FPU movements

Contents

17.1. configMotion()

The method

```
FpuGridDriver.configMotion(wavetable, grid_state, fpuset=[],
                           soft_protection=True,
                           allow_uninitialized=False,
                           ruleset_version=DEFAULT_WAVEFORM_RULESET_VERSION)
```

sends a dictionary of waveform tables to a set of FPUs.

waveform table parameter The dictionary keys of wavetable are the IDs of the FPUs which are configured. The valid format for the input data is described in section 6.5.7 on page 38, and the function `gen_wf()` can be used to generate a suitable waveform table. (The `gen_wf()` function is described in section 19.6 on page 90.)

Moving uninitialised FPUs In a number of situations, it can be necessary to move the FPU even if it was not initialised before, for example when the alpha arm is in a limit breach position, which prevents the datum operation. To do this, *two* safety controls need to be overridden.

To skip the state check in the FPU driver, the keyword argument `allow_uninitialized=True` needs to be passed. **Furthermore, in hardware protocol 2, the FPU firmware checks for a valid state, too, and this hardware check in the FPU needs to be temporarily disabled for the next configuration command to succeed.** This is done by sending each FPU which is to be moved an explicit `enableMove()` command, which sets the state to `FPT_RESTING`.

Software protection Before accepting a waveform and sending it to the FPUs, the command checks that during all movements the FPU arms are kept in a safe range of angles, so that they will not hit the hard stops, and do not run into dead zones.

To do this, the current positions are augmented by the interval within which the waveform table will move the FPUs. The resulting interval is checked against the safe ranges. If the interval exceeds the safe range of movements, a `ProtectionError` exception is raised, and the movement is not configured. The software protection can be switched off by passing the keyword parameter `soft_protection=False`. This is normally only needed when checking the working of the datum switches, as described in section 5 on page 27. *If the software protection is switched off, it is the sole responsibility of the user to make sure that the FPU is not damaged by movement operations.*

The last waveform that was successfully configured is stored. Both a `repeatMotion()` and a `reverseMotion()` command repeat these checks, using the stored waveform and the tracked position of the FPUs.

Whether the protection check was switched off or not, the `trackedAngles()` method shows the current positions of the FPU arms, and the angle ranges of the next configured movements.

Waveform validity checks If the passed waveform is not valid, an exception of type `InvalidWaveformException` is raised. In this case, the error symbol and text message provides a more detailed description of the failed check. If an `fpuid` parameter in the keys of the waveform dictionary is not valid, an `InvalidParameterError` exception is raised.

If an `fpuset` keyword parameter is passed with a list of FPU IDs, only FPUs will be configured which are both contained in the waveform table, and whose IDs are in the fpuset list. If `fpuset` is set to an empty list, all FPUs in the waveform table will be configured.

17.2. getCurrentWaveTables()

The method `getCurrentWaveTables()` retrieves the currently valid, or the last configured waveform tables as a dictionary, one entry for each FPU.

17.3. getReversed()

The method `getReversed()` retrieves for each currently configured, or last configured waveform table whether it was configured for the forward or reverse direction.

17.4. configPaths()

The method

```
FpuGridDriver.configPaths(path, grid_state, fpuset=[],
                          soft_protection=True,
                          allow_uninitialized=False,
                 ruleset_version=DEFAULT_WAVEFORM_RULESET_VERSION)
```

sends a movement path from the path generator software to the FPUs, after checking that the current position corresponds exactly to the starting point of the path. To load a path generator path from its text file representation, the utility function `wflib.load_paths()` needs to be used.

The keyword argument `ruleset_version` selects a rule set version for the checks of the dynamic properties of the generated wave forms. The keyword argument `reverse` is a Boolean value which allows to reverse the movement described by the paths. It does not otherwise change the path, which has the consequence that a reversed path can be valid or not, according to the used waveform rules. Also, a reversed path might be executed by the firmware differently from a path which was loaded, executed, and then reversed by the `reverseMotion()` command.

All other parameters are identical to the parameters `configMotion()` driver method - the `configPaths()` method calls the former after checking for the correct current position.

17.5. executeMotion()

Moving FPUs The method

```
FpuGridDriver.executeMotion(grid_state)}
```

starts the movement of all FPUs for which a valid wavetable was configured, and which are in either the state `READY_FORWARD` or `READY_REVERSE`. The command blocks until this movement has completed, or until an error occurs. Depending on the individual FPU state, the movement will be forward or backward, where “forward” means that increasing step numbers (positive differences) cause a anti-clockwise rotation.

As the `findDatum()` command, the `executeMotion()` command accepts an `fpuset` parameter, a list which will restrict the movement to only those FPUs whose IDs are elements of the list.

Software protection The `executeMotion` command does not implement further software protection checks itself, because all needed checks are done when the movement is configured using the `configMotion()`, `repeatMotion()`, or `reverseMotion()` commands. However, the `trackedAngles()` method can be used to inspect whether the movement is in a safe range before it is actually executed.

Result state If no error occurs, the grid_state parameter contains the final new positions of the FPUs.

If there is any collision, limit switch breach, or error at firmware level, the movement of that FPU is stopped and the command returns an exception of type fpu_driver.MovementError. The sub-type of the exception indicates the cause of the error; section 21.1 on page 96 lists the name of this sub-classes.

If the FpuGridDriver.abortMotion() method is called, or if the key combination <Ctrl>-<c> is pressed (which generates an abortMotion message), all FPUs are stopped and the command returns.

In the case of an error or interruption of the movement, the grid_state parameter will be partially updated with the state of one FPU which was stopped, but might not contain the final positions and states of all FPUs.

The command requires the FPUs to be in state READY_FORWARD or READY_REVERSE, and on normal completion will leave FPUs in the state RESTING. In case of an error, the FPUs will be in state ABORTED or OBSTACLE_ERROR. Section 10 on page 55 explains how such a condition is resolved.

17.6. reverseMotion()

The method FpuGridDriver.reverseMotion(fpu_state) requires that a valid waveform table is present and that FPUs are either in READY_FORWARD or RESTING state (READY_REVERSE is accepted, too, but the command will have no effect). It configures the FPU so that on the next executeMotion command, the waveforms will be executed in reverse direction, and puts the FPUs into READY_REVERSE state. This is used to return FPUs close to the datum position.

Similar to the configMotion() command, the command checks that the movement remains in a safe range of angles for each FPU.

Using the keyword parameter soft_protection=False, this check can be disabled, leaving the responsibility to avoid damage of the FPUs with the user.

Note that this command is not available after a collision, movement error, or abortMotion message – the assumption is that the calibration of the FPU is not more exact after such an event, therefore reversing the movement is ill-defined, and a new findDatum command is needed before to restore the calibrated state.

17.7. repeatMotion()

The method FpuGridDriver.repeatMotion(fpu_state) requires that a valid waveform table is present and that FPUs are either in READY_REVERSE or RESTING state (READY_FORWARD is accepted, too, but the command will have no effect). It configures the FPU so that on the next executeMotion command, the waveforms will be executed in forward direction, and puts the FPUs into READY_FORWARD state.

Similar to the configMotion() command, the command checks that the movement remains in a safe range of angles for each FPU.

Using the keyword parameter soft_protection=False, this check can be disabled, leaving the responsibility to avoid damage of the FPUs with the user.

Note that this command is, as the preceding one, not available after a collision, movement error, or abortMotion message – the assumption is that the calibration of the FPU is not more exact after such an event, and a new findDatum command is needed to restore the calibrated state.

17.8. abortMotion()

The method FpuGridDriver.abortMotion(fpu_state) terminates any movement and datum search operation, and puts any moving FPUs into ABORTED state. FPUs which are in state READY_FORWARD or READY_REVERSED, are put into state RESTING. On other FPUs which are not moving it has no effect. The abortMotion() method is also automatically called if the key combination <Control>-<c> is called during a movement operation or a datum search, or if the Python process receives a SIGINT Unix signal.

Similar to other commands, the method is thread-safe, but it has a slightly different behaviour in that it pre-empts other commands: If other driver methods are called while another operation is ongoing, they wait until the previous operation is completed. In contrary, the abortMotion() method cancels all ongoing operations, including unsent commands, waits until all FPUs have send a confirmation, and returns.

After an abortMotion() command, FPUs which are in the state ABORTED will not be able to move again before this state is resolved. In protocol version 2, this can be done more safely using the enableMove command, so that it is not necessary to reset or power-cycle the FPU.

18. Special methods for managing FPU state

Contents

18.1. lockFPU()

The `lockFPU()` command allows to disable a specific FPU so that it will not move and does not accept any new movement configuration. While locked, it will react to no commands except the `unlockFPU` and the `resetFPUs()` commands. The signature of the method is:

```
FpuGridDriver.lockFPU(fpu_id, grid_state)
```

Locking an FPU is possible in any state except when they are currently moving (in this case, it needs to be stopped first, using the `abortMotion()` command).

18.2. unlockFPU()

The `unlockFPU()` command allows to enable a specific FPU so that it becomes unlocked and moves again. The signature of the method is:

```
FpuGridDriver.unlockFPU(fpu_id, grid_state)
```

When the FPU is unlocked, it returns, as far as possible, to its previous state. If a collision or a limit switch breach has occurred while it was locked, it changes into the state `OBSTACLE_ERROR`.

18.3. enableMove()

The command

```
FpuGridDriver.enableMove(fpu_id, grid_state)
```

allows an FPU to get out of an ABORTED or UNINITIALISED state without resetting the FPU. It addresses one specific FPU whose number is passed as a parameter.

The purpose of this command is to provide the ability to bypass firmware checks of th the FPU state machine in specific situations, without circumventing the firmware protection entirely.

If an FPU is in ABORTED or in UNINITIALISED state, the state is changed to RESTING. Otherwise, the command is rejected. This can be used to move an FPU which is uninitialized. For details on this, see section 17.1 on page 79.

18.4. freeBetaCollision()

The command

```
FpuGridDriver.freeBetaCollision(fpu_id, direction, fpu_state,
                                soft_protection=True)
```

allows to conveniently resolve a collision of the beta arm without completely losing the step counter information. This is done by moving the FPU a small number of steps (about 10 steps) into a specified direction.

The parameter fpu_id identifies the FPU which shall be moved. The parameter direction can be one of FpuGridDriver.REQD_CLOCKWISE or FpuGridDriver.REQD_ANTI_CLOCKWISE, and indicates the direction in which the FPU should be moved. The grid_state variable is updated accordingly.

The FPU state parameter direction_beta can be used to determine the last direction in which the beta arm was moving before the collision. Assuming that the collision affected FPU number 5, the parameter can be printed using the following code:

```
grid_state = gd.getGridState()
grid_state.FPU[11].direction_beta
```

After resolving the collision, the command enableBetaCollisionProtection() needs to be called, to return the FPU back into a movable state. Also, the flags which indicate that the alpha and beta step counters have been zeroed are cleared, because the step counters cannot longer accurately reflect the position. Therefore, a new findDatum() operation needs to be performed, possibly after first moving the beta arm to a safe sector.

The driver counts how many times the command tries to move the FPU into the same direction, and throws a ProtectionError exception if the number exceeds a the threshold stored in the database for that FPU. This check can be disabled by passing the keyword parameter check_protection=false to the command. Disabling this protection should normally not be necessary and can lead to breaking the FPU.

18.5. enableBetaCollisionProtection()

The command `FpuGridDriver.enableBetaCollisionProtection(fpuid, grid_state)` resolves the collision state of one FPU by changing its state from `OBSTACLE_ERROR` to `RESTING`. This reactivates the collision protection, and allows to operate the FPU safely. If the collision has not been resolved successfully, another `CollisionError` exception is raised, and the FPU state is set again to `OBSTACLE_ERROR`.

18.6. freeAlphaLimitBreach()

The command

```
FpuGridDriver.freeAlphaLimitBreach(fpu_id, direction, fpu_state,
                                   soft_protection=True)
```

allows to conveniently resolve a limit switch breach of the alpha arm (also called “alpha limit breach”) without completely losing the step counter information. This is done by moving the FPU a small number of steps (about 10 steps) into a specified direction.

The FPU state parameter `direction_alpha` can be used to determine the last direction in which the alpha arm was moving before the breach. Assuming that the limit breach affected FPU number 11, the parameter can be printed using the following code:

```
grid_state = gd.getGridState()
grid_state.FPU[11].direction_alpha
```

Please note that a visual inspection of the FPU and a verification of the actual position is highly advisable.

The parameter `fpu_id` identifies the FPU which shall be moved. The parameter `direction` can be one of `FpuGridDriver.REQD_CLOCKWISE` or `FpuGridDriver.REQD_ANTI_CLOCKWISE`, and indicates the direction in which the FPU should be moved. The `grid_state` variable is updated accordingly.

After resolving the limit breach, the command `enableAlphaLimitProtection()` needs to be called, to return the FPU back into a movable state. Also, the flags which indicate that the alpha and beta step counters have been zeroed are cleared, because the step counters cannot longer accurately reflect the position. Therefore, a new `findDatum()` operation needs to be performed, possibly after first moving the beta arm to a safe sector.

The driver counts how many times the command tries to move the FPU into the same direction, and throws a `ProtectionError` exception if the number exceeds a the threshold stored in the database for that FPU. This check can be disabled by passing the keyword parameter `check_protection=false` to the command. Disabling this protection should normally not be necessary and can lead to breaking the FPU.

WARNING: Keep in mind that the alpha arm after a limit breach is close to the hard stop. The correct direction of movement should be checked and verified before making even small movements. Repeated limit breaches can indicate a failure of the alpha datum switch, which will cause damage to the FPU.

18.7. enableAlphaLimitProtection()

The command `FpuGridDriver.enableAlphaLimitProtection(fpuid, grid_state)` resolves the limit breach state of one FPU by changing its state from `OBSTACLE_ERROR` to `RESTING`. This reactivates the limit breach protection, and allows to operate the FPU safely. If the collision has not been resolved successfully, another `CollisionError` exception is raised, and the FPU state is set again to `OBSTACLE_ERROR`.

18.8. resetStepCounters()

The `resetStepCounters()` command allows to set the step counters to zero or another new value:

```
gd.resetStepCounters(new_alpha_steps, new_beta_steps, grid_state, fpuset=[])
```

Here, `alpha_steps` and `beta_steps` are the new counter values. This are signed 24-bit values. The command can be restricted to a sub-set of FPUs.

The method is primarily intended for engineering and testing purposes, because usually, the software protection layer will keep track and update any offset between the FPU step counter and the actual position.

18.9. readRegister()

The command

```
gd.readRegister(address, grid_state, fpuset=[])
```

reads one byte of data from each FPU controller at the memory location *address*. Here, *address* is a 16-bit integer value which contains the memory bank value in the upper byte, and the address byte in the lower byte.

The address value can be passed in hex notation, as in `0x0A31`. After a successful read, the value of the memory register for FPU *k* is placed in

```
grid_state.FPU[k].register_value
```

and the used address is placed in `grid_state.FPU[k].register_address`.

This value is overwritten by other operations and becomes invalid with the next method call which writes to the gs structure. If the parameter `fpuset` is set to a non-empty list of FPU IDs, the operation is restricted to these FPUs.

18.10. resetFPUs()

`FpuGridDriver.resetFPUs(fpu_state)` resets all FPUs to their power-on state. All step counters are reset to zero, the datum flag is cleared, and the FPU state is set to `UNINITIALIZED`. This command is useful to get out of specific error states.

As `findDatum()` and `executeMotion()`, the command accepts an `fpuset` key parameter which restricts the operation to the FPU IDs passed as a list.

After an `abortMotion()` command, FPUs which are in the state `ABORTED` will not be able to move again before this state is resolved. In protocol version 2, this can be done using the `enableMove` command, so that it is not necessary to reset the FPU.

19. Utility functions

Contents

The following functions are not part of the driver object, but are contained in the module `fpu_commands`, which needs to be imported separately. They are implemented in pure Python. It is encouraged to inspect their implementation and modify a copy if a somewhat different functionality is desired.

19.1. list_positions()

The function `fpu_commands.list_positions(grid_state)` returns the step counters of each FPU in a list. Each element of the list consists in a tuple which has the following four elements: `alpha_steps, beta_steps, alpha_initialized, beta_initialized`. `alpha_steps` and `beta_steps` are integer values which contain the step counters of the alpha and beta arm, and `alpha_initialized` and `beta_initialized` are each a Boolean value which indicates whether this arm was initialised by searching for the datum position.

If the keyword argument `show_zeroed` is passed with a value of `False`, the last two parameters are not shown.

19.2. list_angles()

Similar to the preceding function, the function `fpu_commands.list_angles(grid_state))` shows the approximate current angle of both arms, in units of degrees. The angles are computed by multiplying the step counters with the correct gear ratio. If the step counters have not been initialised, the symbolic floating point value NaN (not-a-number) is returned. If the keyword argument `show_uninitialized` is passed with a value of `True`, instead of NaN values the values derived from the current counter positions are shown. In this case, the angles may or may not correspond to the current arm position.

The function accepts a second keyword argument, `alpha_datum_offset`, which sets the conventional value of the alpha angle when the alpha arm is at the datum position. To match the values which are used in the EtherCAN interface logs, it should have the same value as the driver parameter with the same name described in section 13.1. The default value is $-180.0°$.

Under normal circumstances, it is preferable to use the `countedAngles()` driver method, because that method automatically updates the grid state data to the current position. Also, in difference to `list_angles()`, it is not necessary to pass the alpha datum offset if a non-standard offset was used.

19.3. list_states()

The function `list_states(grid_state)` lists the state of all FPUs. This allows, for example, to see quicker which FPUs have run into an error.

19.4. list_deviations()

This command lists the counter deviations of each FPU after the last datum search, that is, the residual value of the alpha and beta step counters when the datum switch was hit.

19.5. list_timeouts()

This command lists the time-out count for each FPU. Note that the internally used values are unsigned 16-bit values which wrap around when they overflow – you should not depend on the assumption they increase monotonically.

19.6. gen_wf()

The function

```
fpu_commands.gen_wf(delta_alpha, delta_beta, unit='degrees')
```

generates a valid waveform which moves the alpha arm by an angle of `delta_alpha` and the beta arm by `delta_beta`. A positive value means a rotation in the counter-clockwise direction. One or both parameters can be set to zero. The function makes sure that the rules described in section 6.5.7 on page 38 are followed.

If at least one of the parameters is a Python list or a numpy array, the other parameter will be augmented to an array if it is a scalar (it will be broadcasted), and the function will generate waveforms for multiple FPUs, one FPU for each element of the arrays. If both parameters are lists or arrays, their length needs to match.

The return value of the function is a normal Python data structure which can be assigned to a Python variable, and examined like other Python objects (for example, the ``type()'') builtin function can be used to examine the data structure).

Generating waveforms with constant acceleration The default form of gen_wf creates a waveform which has a constant maximum speed ratio between adjacent time segments. This means that the FPU would accelerate up to the maximum speed, with the resulting torque linearly increasing over time.

To provide for waveforms which have a constant maximum torque, and a better manoeuvrability at low speeds, there exist an alternative mode for waveform generation, with function parameters as follows:

```
fpu_commands.gen_wf(delta_alpha, delta_beta,
                    mode='limacc', unit='degrees',
                    max_acceleration_alpha=MOTOR_MAX_ACCELERATION,
                    max_deceleration_alpha=MOTOR_MAX_DECELERATION,
                    min_steps_alpha=STEPS_LOWER_LIMIT,
                    min_stop_steps_alpha=None,
                    max_steps_alpha=STEPS_UPPER_LIMIT,
                    max_acceleration_beta=MOTOR_MAX_ACCELERATION,
                    max_deceleration_beta=MOTOR_MAX_DECELERATION,
                    min_steps_beta=STEPS_LOWER_LIMIT,
                    min_stop_steps_beta=None,
                    max_steps_beta=STEPS_UPPER_LIMIT,
                    insert_rest_accelerate=True)
```

Here, the mode='limacc' keyword sets the mode to waveform generation with constant acceleration. For each arm, one can pass the minimum and maximum acceleration as a parameter, in addition to the minimum and maximum speed. If the unit parameter is changed to 'steps', the argument is integer step count, not degree.

If the parameters min_stop_steps_alpha or min_stop_steps_beta are not None, the will define the stopping speed, as long as the waveform is long enough to decelerate to that speed in the requested number of steps.

If the alpha and beta parameters should be equal, a somewhat simpler signature can be used:

```
fpu_commands.gen_wf(delta_alpha, delta_beta,
                    mode='limacc', unit='degrees',
                    max_acceleration=MOTOR_MAX_ACCELERATION,
                    max_deceleration=MOTOR_MAX_DECELERATION,
                    min_steps=STEPS_LOWER_LIMIT,
                    min_stop_steps=None,
                    max_steps=STEPS_UPPER_LIMIT,
                    insert_rest_accelerate=True)
```

If these keywords are used, they apply unless the corresponding alpha/beta keyword is defined explicitly.

If a stopping speed is defined which is smaller than the corresponding starting speed, and the requested number of steps or angles is small, the waveform may consist only consist of the deceleration stage, and start at a smaller speed than the starting speed.

The last parameter, `insert_rest_accelerate`, merits some further explanation. The waveform generation function goes inserting steps at the maximum acceleration until the maximum speed is reached, or the requested distance is reached. Usually, there will remain a small rest distance, which is between the minimum and maximum speed, and is inserted in the accelerating or decelerating ramp. By default, these extra steps will be inserted in the accelerating ramp, causing a small jerk with reduced acceleration in this phase of movement. If the parameter is set to `False`, any extra steps will be inserted in the deceleration ramp.

The `gen_wf()` function does not control for jerk. If waveforms with limited jerks are desired, they need to be defined by the user.

19.7. wflib.load_paths()

The utility function `wflib.load_paths()` is the correct way to load a path generated by the path generator prototype software so that the `configPaths()` method described in section 17.4 can execute it. As its only argument, it takes the file name of the file with the path data, for example:

```
p = wflib.load_paths("targets_7fp_case_2_1_PATHS.txt")
```

The function depends on the Python mocpath library which is part of the MOONS software library, and not included in the driver. This library needs to be installed separately and be accessible via the `PYTHONPATH` environment variable. The library itself may depend on further python modules, depending on the used environment.

The function itself does not check the path data for validity - this happens in the driver when the `configPaths()` method is called. To quickly check whether a waveform file is valid, a utility named `check_pathfile.py` is provided, which performs the same checks as the EtherCAN interface, and marks each step which does not conform to all rules, along with the unsatisfied rule.

19.8. wflib.load_waveform()

The utility function `wflib.load_waveform()` is an alternative way to load a path generated by the path generator prototype software so that the `configMotion()` method described in section 17.4 can execute it. As its only argument, it takes the file name of the file with the path data, for example:

```
p = wflib.load_paths("targets_7fp_case_2_1_PATHS.txt")
```

As the `load_path()(` function, the function depends on the Python mocpath library which is part of the MOONS software library, and not included in the driver. This library needs to be installed separately and be accessible via the `PYTHONPATH` environment variable. The library itself may depend on further python modules, depending on the used environment.

The function itself does not check the path data for validity - this happens in the EtherCAN interface when the `configMotion()` method is called.

The crucial difference between `load_paths()` and `load_waveform()` is that the latter strips away the absolute position of the movement, resulting in a relative movement. This has the consequence that the resulting movement can be applied to a different starting position, but the assurance of the path planning software that the movement will not lead to collisions is lost.

Rounding of movement angles If angle values in degrees are passed, they always need to be rounded to entire steps of the stepper motors before they can be passed to the motors. The maximum rounding error *for a single positioning operation* is half a motor step. It is easy to overlook that this rounding takes place. Calculations on relative movement angles always need to be corrected by the actual position values of the arms, which are reported by the `countedAngles()` method. Omitting this correction would lead to a erroneous accumulation of rounding errors.

If exact positioning of relative movements is required, for example for calibration measurements, the user should instead pass the integer number of steps, using the keyword argument `unit='steps'`.

Alternatively, the `configPaths()` method can be used, which allows to configure an absolute path with a maximum error of 0.5 steps.

Moving multiple FPUs at once The `delta_alpha`, `delta_beta` parameters can also be either Python lists or Numpy vectors. In this case, either one of the parameters needs to be a scalar, or they need to be lists or numpy arrays of the same length. A scalar parameter gets extended to a vector of matching length. After this, the element 0 of both vectors contain the alpha/beta movement for FPU 0, element 1 the values for FPU 1, and so on.

If only one or a few FPUs shall be moved, the waveform table should still contain data for the FPUs which are not going to be moved – the movement distance should just be set to zero for these FPUs. (Otherwise, an `executeMotion()` command could activate a waveform entry which is stale, and is not intended to be used.) Alternatively, the `fpuset` keyword of the `configMotion()` command can be used to specify a subset of FPUs which should move – see section 17.1 on page 79 for details.

Moving both arms separately The function has the optional keyword parameter `mode`, which can be one of `"fast"`, `"slow"`, or `"slowpar"`. Mode "fast" is the default mode, which uses a high acceleration and moves the arms in parallel. Mode "slow" moves one arm after the other, and uses less acceleration. Mode "slowpar" also uses less acceleration, but moves the arms in parallel.

Using the “slow” mode can be useful if spurious triggers of the alpha switch occur, for example caused by vibration.

If one of the parameters is a `NaN` value, an `assertionError` exception is raised.

20. Helper scripts

20.1. initializeGrid.py

The script `initializeGrid.py` in the "`python`" subfolder is intended as a template for more specific control scripts. It can be started with a configurable number of FPUs, by either setting the `NUM_FPUS` environment variable, or passing the number with the "`-N`" option. It should be started by using Python's interactive option, like this:

```
$ python -i initializeGrid.py -N 5
```

Started this way, the script initialises the EtherCAN interface for that number of FPUs, sends them a ping command, shows the tracked positions from the FPU database, and returns control to the command line, waiting for input. One can then inspect the tracked angles and send a `findDatum()` command, if the shown positions are correct.

Further options to this script can be listed by using the `--help` option.

21. Errors

21.1. Exception Hierarchy

The hierarchy of exceptions used by the Python module is as follows:

```
EtherCANException                    // top-level class
    MovementError                    // error during movement operations
        CollisionError               // collision detected
        LimitBreachError             // alpha limit breach detected
        AbortMotionError             // AbortMotion message was sent
        StepTimingError              // internal FPU error
       FirmwareTimeoutError        // firmware time-out, movement took too long
      HardwareProtectionError     // firmware rejected movement, or fatal failure
  InvalidStateException        // command does not match EtherCAN interface state
        ProtectionError              // unsafe command called
    SystemFailure                    // system ressource error
    InvalidParameterError            // parameters for command not valid
        SetupError                   // system set-up not allowed
        InvalidWaveformException     // waveform does not match specs
    ConnectionFailure                // connection problem
        SocketFailure                // Socket connection was lost
        CommandTimeout               // FPU is not responding in time
        CAN_BufferOverflowException  // CAN buffer overflow in FPU firmware
```

21.2. Error codes

The following error symbols are used in the EtherCAN interface. It is recommended to use only the symbols, not the actual enumeration values, which might change.

```
// everything worked
DE_OK = 0,

/*********************************/
/* non-error return codes */

// The user waited for a command completion using a time-out
// value, and the state has not been reached yet. This is a
// "user notification", not an error.
DE_WAIT_TIMEOUT = 1,

// Firmware does not implement operation for this protocol version
// - the calling code might need to check and branch according to
// the used protocol version
DE_FIRMWARE_UNIMPLEMENTED = 2,


/**********************************/
/* Fatal system failure */

// An initialization command ran out of memory, which prevents
// successful EtherCAN interface start-up.
DE_OUT_OF_MEMORY = 10,

// Some resource from the OS is not available, which leads
// to an unrecoverable situation.
DE_RESOURCE_ERROR = 11,

// A necessary assumption or check for correctness of the EtherCAN interface
// was violated.
DE_ASSERTION_FAILED = 12,


/**********************************/
/* state errors where requested operations do not match the
   current system state */

// A command was tried to send, or the EtherCAN interface was instructed to
// connect, but the EtherCAN interface was not initialized properly.  That can
// happen if the system goes out of memory, or if a logical error
// affected the initialization.
DE_INTERFACE_NOT_INITIALIZED = 101,


// EtherCAN Interface has already been correctly initialised, and another
// initialisation was tried.
DE_INTERFACE_ALREADY_INITIALIZED = 102,


// The user tried to send a high-level command while another
// high-level command was still not finished and waited for.
```

```
DE_STILL_BUSY = 103,

// The user tried to start a movement command while at least one
// FPU was in collided or aborted state - the command was rejected
// because of that.
DE_UNRESOLVED_COLLISION = 104,

// An FPU has not been initialised, so it cannot be moved
// accurately and safely.
DE_FPU_NOT_INITIALIZED = 105,

// EtherCAN Interface is already initialized.
DE_INTERFACE_ALREADY_CONNECTED = 106,

// EtherCAN Interface is still connected.
DE_INTERFACE_STILL_CONNECTED = 107,

// Waveform is not configured / not ready for movement.
DE_WAVEFORM_NOT_READY = 108,

// The addressed FPUs were not yet calibrated by a datum search.
DE_FPUS_NOT_CALIBRATED = 109,

// A motion command was issued but no FPUs are allowed to move.
DE_NO_MOVABLE_FPUS = 110,

// Command not allowed for present FPU state.
DE_INVALID_FPU_STATE = 111,

// The operation can damage hardware and protection is enabled
DE_PROTECTION_ERROR = 112,

// The EtherCAN interface state does not allows the operation
DE_INVALID_INTERFACE_STATE = 113,

// Some addressed FPUs are locked.
DE_FPUS_LOCKED = 114,

// A previous movement was aborted.
DE_IN_ABORTED_STATE = 115,

// An alpha arm is on the limit switch, and cannot
// be datumed.
DE_ALPHA_ARM_ON_LIMIT_SWITCH = 116,
/***************************************/
/* setup errors */

// Insufficient number of gateways for requested number of FPUs.
DE_INSUFFICENT_NUM_GATEWAYS = 201,

// Configuration parameters invalid, see log message.
DE_INVALID_CONFIG = 202,

/***************************************/
/* invalid command parameters*/

// An FPU id which was passed as a parameter is invalid
// because it is larger than the maximum number of FPUs.
DE_INVALID_FPU_ID = 301,

// passed parameter value is invalid
DE_INVALID_PAR_VALUE = 302,

// duplicate serial number
DE_DUPLICATE_SERIAL_NUMBER = 303,

/***************************************/
/* Connection failures */

// The maximum retry count was exceeded for command.
DE_MAX_RETRIES_EXCEEDED = 401,

// A CAN command to an FPU surpassed the maximum waiting time for
// a response.
//
// This can indicate either a connection problem, a failure of the
// FPU controller, or a failure of the FPU hardware.
DE_CAN_COMMAND_TIMEOUT_ERROR = 402,

// A command was tried to send to the FPUs but this was not
// possible because the EtherCAN interface was or became disconnected from a
// gateway. During operation, this should only happen when the
// socket connection breaks down for an extended time, as the
// socket protocol will try hard to do re-sends for several
// minutes.  Before this error happens, one will probably see
// time-outs on every single FPU command to the corresponding
// gateways as they all fail to respond.
DE_NO_CONNECTION = 403,


// A CAN buffer overflow message was received, meaning
// that more commands were received at once than the FPU firmware
// and its CAN implementation were able to process. This
// is similar to a COMMAND_TIMEOUT_ERROR, except that we
// know that the last message wasn't processed.
DE_FIRMWARE_CAN_BUFFER_OVERFLOW = 404,

/***************************************/
/* invalid waveforms */

// General error in waveform definition, see text.
//
// Also: We tried to move FPUs but some addressed FPUs still have
// invalid waveforms.
DE_INVALID_WAVEFORM = 500,


// waveform has to many steps
DE_INVALID_WAVEFORM_TOO_MANY_SECTIONS = 501,

// Number of sections different for different FPUS - this isn't
// allowed to avoid collisions.
DE_INVALID_WAVEFORM_RAGGED = 502,

// Step number in section is too high for current firmware.
DE_INVALID_WAVEFORM_STEPCOUNT_TOO_LARGE = 503,

// The change in step count per section is incorrect (e.g. too large)
DE_INVALID_WAVEFORM_CHANGE = 504,

// The tail of the waveform is incorrect.
DE_INVALID_WAVEFORM_TAIL = 505,


/***************************************/
/***************************************/
/* Errors which terminate movements  */

/***************************************/
/* Collision error */
```

```
// A collision occured, and the operation was aborted.
DE_NEW_COLLISION = 601,

/***************************************/
/* limit breach */
// An alpha limit breach occured, and the operation was aborted.
DE_NEW_LIMIT_BREACH = 602,

// At least one FPU ran into a step timing error, which means the
// FPU's motion controller was not able to compute the required
// step frequency quick enough for the configured microstepping
// level.

DE_STEP_TIMING_ERROR = 603,

/***************************************/
/* abort message */
// The movement has just been aborted.
DE_MOVEMENT_ABORTED = 604,

/***************************************/
/* Datum rejected: alpha arm on limit switch */
// The datum command was rejected.
DE_HW_ALPHA_ARM_ON_LIMIT_SWITCH = 605,

/***************************************/
/* datum time-out */
// The datum command has timed out on the FPU.
DE_DATUM_COMMAND_HW_TIMEOUT = 606,


/***************************************/
/* illegal step counter value */
// The EtherCAN interface received an illegal counter value from
// an FPU, so that it cannot correctly track the FPUs
// any more. It is required to measure the
// position and update the position database.
DE_INCONSISTENT_STEP_COUNT = 607,
```

22. Initialising a driver instance for unconfigured FPUs

When the FPU are flashed with the same firmware, they will initially have the same predefined serial number. To work correctly, a unique serial number needs to be assigned to each FPU. Actually, the `FpuGridDriver` class will throw an error if one attempts to connect it to several FPUs which have the same serial number.

This poses a kind of chicken-and-egg problem: For assigning a serial number, a driver instance needs to connect to the FPUs, which the standard driver instance will not do for safety reasons. To overcome this problem, a special EtherCAN interface driver class exists, which has the name **`UnprotectedGridDriver`**. This class does not use any protection features which are based on the unique serial number. Therefore, it can connect to unconfigured FPUs. This allows to use an instance of this class to flash FPUs with a new serial number, using the `writeSerialNumber()` method. The `UnprotectedGridDriver` also has methods for moving FPUs. However, the new positions of the FPUs are not registered in the LMDB position database, therefore it is not possible to operate FPUs safely with this class.

23. Firmware compatibility

The EtherCAN interface documented here, version 2.1.0, requires firmware version 2.0.0 or newer. The following limitations apply for older firmware versions:

firmware versions <= 2.0.0 These firmware versions are not supported by this driver version.

24. Changelog

The following list contains a chronological list of all changes which might require adaptions in scripts which use the corresponding commands, or alter otherwise visible behaviour of the driver.

Changes in version 2.1.2

- Added documentation how to properly move uninitialized FPUs using the `enableMove()` command, skipping the state check added in the FPU firmware.
- Bugfix: Fix some edge cases in the driver where some state checks were missing in the `configMotion()` method, and where the hardware simulation returned invalid codes when configuring invalid movements.

Changes in version 2.1.1

- Corrected a regression from merging changes to the protocol 1 version, which caused an error in the `findDatum()` command.
- Corrected segment length and number of waveform segments to match protocol 2 firmware.
- Added a new waveform rule set, rule set number 5, and made it the default. This rule set allows a constant acceleration, which allows for quicker movement changes a slow speeds.

Changes in version 2.1.0

- Merged a number of changes for the FPU verification system, such as optionally disabling output.
- Can now pass the FPU database as an external initialization parameter to the FPU driver class.
- Bugfix: Adjust duration of waveform segments to 125 milliseconds.
- Bugfix: Adjust maximum number of waveform segments to 256.
- Changed: `configMotion()` parameters changed to allow for faster upload of waveform data.

Changes in version 2.0.5

- First tested version of driver with CAN protocol v2 support.

Changes in version 1.5.8

- Updated installation instructions, and switched git branches. The protocol 1 driver is from now referred to as the “can1” branch, not any more as the “master” branch. The “master” branch now refers to the tested protocol 2 code.

Changes in version 1.5.7

- Add constants in radian, to assist use of uniform SI units in the software.
- Add long-form installation instructions captured from lab workstation re-install.

Changes in version 1.5.6

- Several small changes to make driver less chatty when used within the verification system.

Changes in version 1.5.5

- Fix typo in API constant `DEFAULT_WAVEFORM_RULESET_VERSION`.

Changes in version 1.5.3

- Adapt field names in FPU state struct to protocol specification (renaming `was_zeroed` to `was_referenced`).
- Added: example script `example-short-script.py` for code shown in section 2.7.
- Updated: Listing of FPU state fields in example for interactive inspection.

Changes in version 1.5.2

- Added: Use of LMDB can now be disabled by clearing the environment variable `FPU_DATABASE`. This requires to use the undocumented class `UnprotectedGridDriver` in place of `GridDriver`.
- Added: Add option to set socket write buffer size in `EtherCanInterfaceConfig` class.
- Fixed: Fixed some compiler warnings which have been added in g++ 7.2.0 (re-enabling compilation on Ubuntu 17.10)

- Added: Default file size for LMDB is now set to a supported size if a 32-bit system platform is detected. The amount of movements that can be logged is limited in this case.
- Changed: Set time-out value for CAN messages to 100 seconds for both `configMotion` and `pingFPUs` messages.
- Changed: Fix Makefile so that explicit file lists are used for source files in place of wild cards. This allows to share folders with ICS code in the MOONS ICS subversion repository.

Changes in version 1.5.1

- Added: `testGateway.py` script was added, which allows to soak-test the EtherCAN gateway by sending many CAN messages in quick succession.
- Changed: Remove small pauses which were inserted at the level of the Python wrapper, when executing several commands.

Changes in version 1.5.0

- Internal optimisation: Data structure which keeps next time-out was optimised, which reduces the CPU load significantly.
- Fixed: Add missing event signal (broadcast of condition variable) when new commands arrive to empty command queue, further speeding up sending of commands.
- Changed: Set default DummyDelay value to 2, making sure that the minimum waiting time is one millisecond.

Changes in version 1.4.1

- Fixed: A bug in version 1.4.0 was fixed which caused an error message when using the "flash" command of the fpu-admin tool. The generated error was not critical in this case, but uncovered a fault in the logging of movement counts in non-protected driver instances.
- Added: Previously, the driver did not check whether an EtherCAN gateway was exclusively accessed. Connecting with several client instances to the same gateway causes that gateway to mix up operations, which it is not designed for. This subsequently causes the driver to stop operations because they have no consistent result. This can in theory also damage FPUs.

  To address that, a check for exclusive access was added to the Python layer, making sure that no other driver instance on the same computer will access the same gateway address at the same time.

Changes in version 1.4.0

- Added: Example script named "plot-fpu-counters.py" which reads the healthlog table of the position database and produces a matplotlib plot of datum aberrations and their binned means and estimated standard deviation as a function of the count of datum operations. Additionally, it can produce a plot showing the number of movements over calendar time, ad well as a chart on CAN time-outs.
- Fixed an aliasing bug in the processing of counters such as number of datum operations, count of collisions, and so on. This caused statistical values erroneously to be shared between different FPUs, when more than one FPU were running at the same time. Instead of updating counters for each FPU separately, the same collection of counters was used in this case. As a result, all "counters" values in the protection database, and all values in the health log are possibly incorrect for versions sooner than v1.3.8, and should not relied upon. To discern correct data from compromised data, the field value was changed. Values which use the replacement "counters2" field name are stored correctly.
- Changed: The default maximum alpha range was decreased from +172.0 to +159.0°, because an alpha arm limit breach was observed at +160 degrees. The value can be individually increased for any FPU using the `fpu-admin` tool.
- Improved: Some cases where CAN time-outs were not captured in the health log were fixed.
- Fixed: The FPU error code `ER_DATUM_LIMIT`, sent when the alpha arm is found to be on the limit switch, was previously not converted correctly into an exception. This conversion is now working properly.

Changes in version 1.3.7

- Added: The driver parameters `min_bus_repeat_delay_ms` and `min_fpu_repeat_delay_ms` allow to set rate limits to CAN commands sent to the same bus, or to the same FPU, which allows to avoid timing problems. For details, see section 13.1 on page 66.
- Changed: The parameter `waveform_upload_pause_us` is set to zero by default.
- Added: The driver parameter `can_command_priority` allows to set the priority of CAN commands to the firmware. By default, version 1.3.7 and newer send commands with a lower priority than response messages from the FPUs. This helps to avoid problems with buffering and missing responses, but requires firmware version 1.6.0 or newer.
- Added: Added a diagnostic utility `get_wtable.getwt()` which can download configured waveforms from all running FPUs. This makes it possible to compare and verify the actually used waveforms in the case of movement errors.

- Changed: The default upper software protection limit for the beta arm was set to +140° after some beta arm collisions were detected when using larger ranges.

Changes in version 1.3.6

- Make re-sending `configMotion()` data more robust by checking for correct FPU state and number of waveform segments. Previously, re-sending waveforms was skipped if the final state of the FPU was correctly `READY_FORWARD`, but the number of waveform segments too low. This could cause invalid movements due to missing waveform segments. This change improves the detection of such cases.

Changes in version 1.3.5

- Activate additional retry code in `configMotion()` command, which re-sends data if a CAN time-out occurred when sending waveforms. Now, re-sending waveform data is supported for up to 50 times, before a fatal error is reported.

Changes in version 1.3.4

- Changed: Increase maximum size of LMDB database to 5 GB for life time test.
- Fixed: Syntax error in fpu-admin tool.
- Fixed: Correct error in check for firmware version 1.5.0 .

Changes in version 1.3.3

- Added: to the `gen_wf()` function in 'limited acceleration' mode, an extra parameter was added which allows to define a stopping speed which is different from the starting speed.

Changes in version 1.3.2

- Added: To the `gen_wf()` function, a mode was added which generates waveforms with constant acceleration. This is described in section 19.6 on page 91.
- Added: Waveform validity ruleset 4 was added and set as default. The detailed description of the rules can be found on page 41.
- Changed: The `check_pathfile.py` script was adapted to new capabilities of firmware version 1.5.

- Added: The life time test script was partially adapted to the new firmware capabilities and waveform ruleset version 4.

Changes in version 1.3.1

- Fixed: The documented value for indicating an overflow in the communicated step counter of the alpha arm had an off-by-one error. This was fixed and matched to communication protocol 1.1 from 2018-10-18. Also, the corresponding value in `fpu_constants` was fixed.
- Added: A explanation on the origin of rounding errors, and how to configure absolute paths manually was added in section 6.9.3.
- A number of internal classes and types in the C++ driver were renamed to match the terminology of the ESO software architecture better, and to avoid possible confusion in future discussions. Specifically, the name part "driver" was substituted by "EtherCAN Interface". This is reflected in renaming `FpuDriverException` to `EtherCANException`. Also, member names like `driver_state` were renamed to `interface_state`. The name of the internal Python extension module was changed from `fpu_driver` to `ethercanif`.

  Normal scripts should continue to work without any change.
- The documentation was adapted to reflect the new distinction between the Python driver class, and the C++ communication interface which implements most of its functionality.

Changes in version 1.3.0

- Compatibility: By default, this FPU EtherCAN interface version is compatible with firmware version 1.4.4 upwards. To use it with earlier firmware versions, the EtherCAN interface configuration parameter `firmware_version_address_offset` to the value `0`. Detection of CAN overflow conditions requires at least firmware version 1.4.4. For more detailed information on firmware compatibility, see section 23.
- Updated: The lifetime test script has been updated to match the current waveform rules. (The last version of the waveform rules are defined in the hardware communication protocol document, document number `VLT-TRE-MON-14620-3016`, issue 1.0 from 2018-08-03, section 5.3.5.8.1.1. However a part of the rules are so far not formally specified, as the hardware properties are not yet precisely known.)
- Fixed: The function `gen_wf()` has been updated to match the current waveform rules, and the firmware capabilities of firmware up to 1.4.4. Especially, when combining a short movement for one arm, and a movement with more waveform segments for the other arm, zeros are now always padded at the beginning of the shorter movement,

not at the end or both ends. This ensures that the firmware can generate small step numbers with appropriate frequency.

- Added: The `configMotion()` command has now an additional version parameter, `ruleset_version` which allows to disable the waveform validity check, or use a different version of the rule set for the check. The waveform rules defined by the RFE to Software document are the rule set version which is enabled by default. Details are described in section 6.5.8.
- Added: In addition to the `configMotion()` command, the driver now has a `configPaths()` command which allows to use paths generated the path analysis software, and loaded using the `load_paths()` utility function. For more information, see section 17.4 on page 81 ff..
- Changed: The waveform validation check was made less strict in some aspects in order to match it better with the Software to RFE ICD, and to allow to use waveforms as generated by the DNF algorithm. For more information, see section 6.5.8 on page 39 and section 17.1 on page 79.
- Improved: The error message for the case that the driver has to abort operation because the FPU step counters are inconsistent has been clarified and completed with some helpful details.
- Fixed: Handling of cases where the step counter overflows after a reset of an FPU was improved. (Correct handling of step counter overflow requires some additional firmware support, which should become available with firmware version 1.4.4.). If correctly implemented in the firmware, this situation is handled internally without generating a driver error. Otherwise, a `ProtectionError` exception can be raised.
- Added: A new driver method named `configPath()` allows to send paths which are defined using absolute angles and execute them on the FPUs with minimum quantisation error when rounding to step counts. See sections 6.9.2 on page 45 for more information, and section 17.4 for a full reference.
- Added: A `load_paths()` function was added which can load the output from the path generator software and use it as a waveform for the `configPaths()` method. For details, see section 6.9.2 on page 45.
- Added: A `load_waveform()` function was added which can load the output from the path generator software and use it as a waveform for the `configMotion()` method. This function should only be used in special circumstances, as no protection against collisions is conserved.
- Added: The EtherCAN interface checks for CAN buffer overflow error messages which have been added in firmware version `1.4.4`. The new exception type `CAN_BufferOverflowException` has been added for reporting this error. See section 21.1 for details.

- Added: The EtherCAN interface has two additional parameters. The first, named `waveform_upload_paus` allows to select a higher upload speed for the upload of waveforms to the FPUs. The second, `confirm_each_step`, allows to optionally disable most waiting for confirmations. These options are not enabled by default, because they might trigger some non-obvious problems. As of EtherCAN interface version 1.3.0, the second option is not implemented in the firmware protocol. Details are described in 13.1 on page 66.
- Fixed: Minimum and maximum speed settings from the lifetime test script are now passed correctly as instantiation parameters when the driver instance is created. In earlier driver versions, these parameters were not passed correctly, which could cause rejection of waveforms which were matching the changed limits.
- Added: operations in the driver protection layer have been added to the logging, in a new log file with the suffix `*_protection.log`.
- Fixed: A bug was fixed in the logging of the total number of steps for the alpha and beta arm in the LMDB position database.

Changes in version 1.2.0

- Compatibility: This EtherCAN interface version is not compatible with later developed firmware versions of v1.4.4 or higher. The reason is that these later version store the firmware version information in a different binary location which can't be found by the EtherCAN interface.
- Added: Life time test script which generates relatively realistic movements in a continuous loop.
- Added: Some information on why the movement configuration using `gen_wf()` needs to round angles to entire steps, and the consequences of this has been added.
- Fixed: A bug in the software protection layer was fixed which affected reversing movements with the `reverseMotion()` command when the movement was not monotonous, but going forwards and backwards. The cause for the bug was that the stored waveform was correctly checked with the reverse sign, but incorrectly with the temporal order of the forward movement. This had the effect that waveforms were rejected which were actually safe, which is now fixed.
- Added: The `gen_wf()` function now provides a keyword parameter `units='steps'` which allows to configure exact movements, without rounding errors.
- Added: The `fpu-admin` command has an additional option `--gateway_address <ipaddress>` which allows to flash FPUs which are connected to a gateway with an IP address different from the default.

- Added: The EtherCAN interface has now initialisation parameters which allow to configure the minimum and maximum speed of the stepper motors.
- Added: According to a change in the protocol and ICD, when starting a waveform, the starting steps do not need to be exactly at the minimum step count and speed, but can be within a tolerance range of about 5%. This range is configurable. This change is needed because the waveform passed from the path planning libraries need to be rounded and corrected for gearbox non-linearities in a way which causes small deviations in the step counts.
- Changed: The alpha upper limit of movement as set by fpu-admin, and tested in the CalibrationPositioning.py script, has been set to +172.0 degrees.
- Deprecated: The manual datum search has been deprecated. It has been replaced by configuring the position database with the actual angle of the FPUs, so that a subsequent datum search will automatically chose the right direction.
- Fixed: The default directory for the position database had been set to `/var/run/fpudb`. However the /var/run directory is automatically cleared on system boot, so the default was changed to `var/lib/fpudb`.

Changes in version 1.1.1

- Added: The `configMotion()`, `repeatMotion()`, and `reverseMotion()` commands all now print the number of configured FPUs if the passed wavetable data did not configure all of them. This should avoid confusion if there are old wave tables active, or if some FPUs are unintentionally not configured to move.
- Changed: In order to reduce details more to the essential, the `trackedAngles()` method now shows angles from the FPU step counters only when the keyword argument `show_offsets` is set to true. Otherwise, only current angles and movement ranges are displayed.
- Changed: The `trackedAngles()` now can show the currently active waveforms (instead of the last configured waveforms), if the keyword argument `active` is set to `True`.
- Fixed: During execution of the `findDatum()` command, a safety tolerance of 0.5° degrees had been added. The shady goal of this was to make simulations easier. This was dropped since it caused confusion and made error recovery from hardware datum time-outs more difficult than needed. Instead, simulations were fixed to return an exact datum position by default.
- Fixed: If only a subset of FPUs was configured to move directly after a `resetFPUs()` command, a following `executeMotion()` command could trigger a Python KeyError exception.

- Fixed: Value of alpha datum angle updated to -180° degrees. Version numbers in explanations updated.
- Fixed/changed: Changed `repeatMotion()` and `reverseMotion()` commands to wait for a response from the FPU. For allowing workarounds, this wasn't required in earlier EtherCAN interface versions. However, a response is now needed for the protection wrapper in order to operate reliably.
- Added: Any passed `fpuset` argument is now checked in a number of additional places, to filter out invalid parameters.

Changes in version 1.1.0

- Added: The FPU database now contains counters for the number of movements, number of datum operations, collisions, limit switch breaches, and datum time-outs. Sums and square sums of the datum counter aberrations are stored as well.
- Added: A new, second database now tracks the counters for each FPU indexed by the count of datum operations. The command

  ```
  fpu-admin healthlog {serialnumber}
  ```

  allows to print these series. See section 8 for any details.
- Added: Support for the hardware error codes `ER_DATUMTO` and `ER_DATUM_LIMIT`. These correspond to two new cases where datum operations which are rejected by the FPU with the goal to improve safety. The first case is a time-out that is triggered when a certain step count is exceeded without finding datum. The second change rejects datum commands when the alpha limit switch is active.
- Added: The `findDatum()` command now also features an extra option to disable the above time-out safety check, which is available for firmware version 1.4.3. See the reference section on the datum command.
- Added: To the FPU & EtherCAN simulation, support was added for hardware 1 protocol l.4.
- Fixed: The maximum number of FPUs on one CAN bus was changed from 67 to 76.
- Fixed: The fpu status field `timeout_count`, a wrapping 16-bit counter, is now incremented every time a CAN time-out occurs.
- Fixed: After an `resetFPUs()` command, a 2.5 sec waiting time was inserted. Without that pause, the subsequent `readSerialNumbers()` command fails because the FPU cannot respond that fast, leading to a time-out.
- Fixed: Fixed an error in state tracking in cases where a findDatum or executeMotion command was aborted. The initial states had been stored but were erroneously references to objects which were then mutated. This lead to errors in state tracking.

Changes in version 1.0.0

- Added protection layer which disables most unsafe movements unless explicitly enabled by switching off a flag. This is described in the sections 4.9 and 4.11.
- The current positions are stored in real-time in a high-performance memory-mapped key-value database (called LMDB aka “Lightning Memory-Mapped Database” which has a good amount of protection against system failures and even power failure.
- Passing explicit direction parameters to the initial `findDatum()` command after powering on an FPU is no longer required, as they can be safely derived from positions stored in LMDB.
- The `soft_protection` parameter for `findDatum()` has been split into `soft_protection`, which checks against the position database, and `count_protection` which falls back to checking the step count of an initialised FPU. For further information, see section 16.1.
- Movements configured using `configMotion()` are checked to make sure they are always within a safe range of angles.
- The `configMotion()` command has an additional parameter `allow_uninitialized`, now needed to configure movements where no datum command was completed before. Details of this are described in section 17.1.
- The `list_angles()` function has been replaced by a more powerful `countedAngles()` method. see section 15.10 for details.
- It is now possible to view the last configured waveforms and their direction, using the `getCurrentWaveTables()` and `getReversed()` commands.
- The EtherCAN simulator has been updated to include new information on safe operational ranges of the FPU hardware and current firmware.
- Fixed bug in code which should have waited for a reverse movement to complete, causing the `executeMotion()` command to return prematurely to the Python command line.

Changes in version 0.7.2

- added `getSwitchStates()` command to read out the datum switch states.

Changes in version 0.7.1

- fixed spelling error in name of command `writeSerialNumber()`

Changes in version 0.7.0

- The readSerialNumbers(), writeSerialNumber(), and printSerialNumbers() methods have been added.
- The EtherCAN interface now logs alpha arm angles with a configurable offset parameter for the datum position, named alpha_datum_offset. The default value is $-180.0°$. The same parameter can be set in the list_angles() function.
- Some information and warnings about hardware safety precautions have been updated.
- The check_protection keyword has been replaced with the soft_protection keyword, to clarify the difference between driver-level (software) and hardware protection.
- The EtherCAN and FPU simulator has been updated to CAN protocol 1.3.0, allowing to store FPU serial numbers in a file, and accepting a further keyword argument alpha_datum_offset.
- The CalibrationPositioning.py script has been updated to use the new default range of alpha arm angle values between -180 and 180.

Changes in version 0.6.0

- The findDatum() method now automatic moves the beta arm into the correct direction if the FPU has been initialised before.
- All FPU commands which concern multiple FPUs, such as findDatum(), executeMotion(), resetFPUs(), and so on, now have an additional fpuset keyword parameter. It can be set to a list of FPU IDs to which the called command will be restricted. For details, see section 6.4.3 on page 34, and section 12.2 on page 62.
- The EtherCAN interface methods getFirmwareVersion(), printFirmwareVersion(), and minFirmwareVersion() allow to retrieve, display, and check the version numbers of the FPU firmware. For details, see sections 15.1 – 15.3.
- As a low-level debugging tool, the readRegister() command allows to read bytes from the FPU controller memory. It is described in section 18.9.

Changes in version 0.5.4

- Fixed bug in C++ EtherCAN interface which caused OBSTACLE_ERROR state to be overwritten by ABORTED message after an FPU detected a collision.
- Corrections to the documentation of keyword arguments findDatum().
- Collision or limit switch breach messages received are now additionally logged to standard error, so that they are not overlooked.

Changes in version 0.5.3

- Added optional logging of commands and CAN messages, which is enabled by default (described in section 9).

Changes in version 0.5.2

- the `findDatum()` method has an additional keyword argument which allows to select only the alpha or only the beta arm for the datum operation. Using this feature requires firmware for CAN protocol version 1.1
- The `list_positions()` function will now, by default, show not only a tuple containing the step counters for each FPU, but additionally also two Boolean values which indicate whether each of alpha and beta arm have performed a valid datum search.
- Several small errors in the CalibrationPositioning test script have been fixed.
- The `connect()` method will now cause a time-out and return of a ConnectionFailure if establishing the connection takes longer than 20 seconds.
- The `list_angles()` function has now an option which causes it to display angles for uninitialised step counters.
- The numerical error codes have been changed.
- The EtherCAN simulator has an option to simulate protocol version 1.1

Part IV.

Appendix

A. Additional examples

The python directory in the driver package contains two sub-directories with further examples and small test scripts, `python/test_mock` and `python/test_fpu`. The scripts in the former are written to send commands to the mock EtherCAN gateway, and the scripts in the latter send commands to physical FPUs. They can be modified and adapted as needed. Here is an overview what each script does:

test_pingFPU.py issues a pingFPU command and reports the positions

test_printSerialNumbers.py retrieves and prints the serial numbers of all FPUs.

testGateway.py test the EtherCAN gateway by sending a large number of CAN commands, checking that all commands are confirmed correctly, and computing the average rate of CAN messages per seconds.

test_configMotion.py configures a movement by defining a wavetable as a Python literal

test_executeMotion.py configures and executes a movement

test_findDatum.py performs a datum search

test_findDatumAlpha.py performs a datum search only for the alpha arm

test_findDatumBeta.py performs a datum search only for the beta arm

test_repeatMotion.py uses the repeatMotion command to perform a movement repeatedly

test_warnCollisionAlpha.py generates a alpha limit breach message

test_warnCollisionBeta.py generates a beta collision message

test_detectAlphaLowerLimit.py drives the FPU into the limit switch by issuing repeated negative movement commands.

test_detectAlphaUpperLimit.py drives the FPU into the limit switch by issuing repeated positive movement commands.

test_detectBetaCollision.py provokes a beta arm collision

test_warnStepTimingError.py generates a step timing error in the mock-up EtherCAN gateway by setting the uStepLevel to a high value

test_abortMotion.py configures a movement which can be aborted manually using the `<Ctrl>-<c>` key combination.

B. Dependency requirements and installation

Contents

B.1. Platform requirements

The EtherCAN interface and Python layer has been tested on Scientific Linux 2017, Ubuntu 17.10 (codename Artful Aardvark), Debian 9 (codename Stretch), and Arch Linux (2018.08.01). As a minimum, it requires a Linux kernel with version 2.6 or newer.

The Python module is configured and tested to use Python 2.7. No major adaptations, but some minor changes in the configuration of `boost::python` are expected for running it in Python 3.

It is highly recommended to use a 64-bit system installation. Otherwise the logging capacity in LMDB is limited by the file size limit of the system kernel, and file sizes larger than 2 GiB require a kernel compiled with large file support (LFS), which is not enabled by default in 32-bit Linux.

B.2. Prerequisites

1. For the C++ EtherCAN interface and static library
   - gcc-4.9 (essentially, with support for C++11)
   - Linux-2.6 or newer (requiring support for eventfd)
   - glibc-2.3.2 (epoll support)

- liblmdb0 (Lightning Memory database, aka LMDB)
- lmdb-utils (contains backup utility for LMDB)

These should be available on all current Linux systems.

2. For the Python module
   - Python-2.7
   - boost-1-66 , which can be downloaded from `http://www.boost.org/users/download/`
   - python-lmdb
3. For loading paths from the path generator software
   - For this functionality, the `mocpath` python library module needs to be installed, which is available via the MOONS SVN repository at `http://svnhq1.hq.eso.org/p1/trunk/Instruments/Paranal/MOONS/COMMON/mocpath`. This dependency is only required when the `wflib` library module is loaded. Installing `mocpath` possibly requires installation of further commonly available dependencies, namely `numpy`, `scipy`, and `matplotlib`.

Note that Python might need to be called with the command name `python2` on newer systems.

B.3. Installation of FPU EtherCAN interface module

B.3.1. Installing boost libraries

Recent Boost libraries are required to build the Python module, which is needed in the verification system (but not in the final ICS software).

Please keep in mind that any installed earlier version of boost libraries is very likely an essential part of your Linux system, so do **not** uninstall older versions from your system!

- for security, verify boost package signature using gnupg
- unpack boost package:

```
$ tar xzf boost_1_66_0.tar.gz
```

- build package:

```
$ cd boost_1_66_0/
$ ./bootstrap.sh
$ ./b2
```

- install package into /usr/local directory:

```
$ su
# ./b2 install
```

In the case that the package should *not* be installed into the /us/local mount point, this can be done by passing a prefix parameter to the `bootstrap.sh` script:

```
$ cd boost_1_66_0/
$ ./bootstrap.sh --prefix=$HOME
$ ./b2
```

and then proceeding as described above.

To make the boost libraries accessible when the final program is run, you need to add the command

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
```

to your shell configuration (usually, `$HOME/.bashrc`). Also, for building the Python module, it might be needed to add something like

```
export LIBRARY_PATH=$LIBRARY_PATH:/usr/local/lib
```

(if this is missing, the “ld” command will fail during building).

If the boost libraries are installed to a non-default location, you should add the directory with the boost header files (by default `/usr/local/include`) to the path configured in the environment variable `CPLUS_INCLUDE_PATH`, so that gcc can find them when building the module.

The boost::python library may require re-compiling when system upgrades are performed.

B.3.2. Installing the python-gevent module

The Python mock-up gateway uses the gevent library which is used to serve parallel requests using asynchronous code. In difference to Python3’s asyncio, it is available both in Python2 and Python3. The present version (1.6.0) runs with Python 2. It can emulate several generations of firmware versions.

On Scientific Linux / Fedora / RHEL, it can be installed as follows by the root user:

```
root$ yum install python-gevent
```

On Ubuntu or Debian, install it as follows:

```
$ pip install --user gevent
```

(This command needs to be run as root, and without the `--user` option, in order to install under `/usr/local`).

B.3.3. Building the C++ EtherCAN interface library

The FPU EtherCAN interface is a static library which becomes included in the Python module. Build the FPU driver library as follows:

```
$ cd fpu_driver
$ make lib
```

This places the static library libethercan.a into the subdirectory "/lib". It can be used in any C++ program.

B.3.4. Building the python module

Run

```
$ make wrapper
```

This builds the binary extension module `fpu_driver.so` and places it into the subdirectory `/python`. In the same subdirectory, there are some test scripts which can be used to test the module.

After installing the dependencies, several configuration parameters need to be set. These are explained in section 4.6 on page 20 ff.

B.3.5. Generating the documentation

The documentation source is managed along with the EtherCAN interface sources. To generate the documentation for a specific earlier version of the EtherCAN interface, a LaTeX distribution including the `minted` package and the Python pygments software needs to be installed (package name `python-pygments` or `pygments`). This is typically included in the `tex-live` package. Also, the `inkscape` package needs to be installed in order to generate the PDF version of the FPU state diagram.

From the folder `fpu_driver`, run the command

```
make manual
```

to generate the documentation[1].

The generated document has the path name `python/doc/manual.pdf`.

[1]If the document is generated from scratch, it is necessary to repeat this command three times to update the list of contents.

C. Running the test gateway simulator

C.1. Running the gateway simulator normally

In the sub-directory /test/HardwareSimulation, the package contains a mock-up test gateway which responds to messages as specified by the hardware protocol. To run it, change into a new terminal window and issue

```
$ cd test/HardwareSimulation
$ python2 mock_gateway.py
```

This test code opens three socket connections on localhost and will display received messages and commands. It will not necessarily run with the same speed as the hardware but should match the hardware protocol specification faithfully. Run the script with the option "-h" to see useful options.

Most test scripts will set suitable port numbers and IP addresses when run with the "–mockup" option.

C.2. Running the test gateway with debug output

In normal operation, the test gateway shows for each FPU the main commands it receives. In some situations, such as when debugging the CAN message encoding, it might be of interest which binary messages are actually send and responded. To run the mock-up gateway in this mode, set the environment variable DEBUG to the value "1" when running it. This can be done quickly with the command

```
$ DEBUG=1 python2 mock_gateway.py
```

Use the command line option "–help" to check for additional features, such as simulating communication failures etc.

In another terminal window, you can run the tests:

```
$ python2 -i test_mock/test_findDatum.py
```

and so on.

Apart from DEBUG, it is also possible to set the environment variable NUM_FPUS to a smaller values (the default is 1000). The same is achieved using the option "-N". Usually, this makes little visible difference. Keep in mind, however, that surplus simulated FPUs respond to broadcast commands, which is reported by the EtherCAN interface, and recorded in the logs. Conversely, simulating too few FPUs will result in CAN timeout errors when any commands are sent to the missing FPUs.

C.3. Command-line options

The full command-line options of the gateway simulator are:

```
usage: mock_gateway.py [-h] [-d] [-v VERBOSITY] [-V PROTOCOL_VERSION]
                       [-t DATUM_ALPHA_TIMEOUT_STEPS]
                       [-b DATUM_BETA_TIMEOUT_STEPS] [-D FIRMWARE_DATE]
                       [-N NUM_FPUS] [-O ALPHA_DATUM_OFFSET] [-A ALPHA_START]
                       [-B BETA_START]
                       [p [p ...]]

Start EtherCAN gateway simulation

positional arguments:
  p                     ports which will listen to a connection

optional arguments:
  -h, --help            show this help message and exit
  -d, --debug           print received binary commands and responses
  -v VERBOSITY, --verbosity VERBOSITY
                        verbosity: 0 - no extra output ... 5 - print extensive
                        debug output
  -V PROTOCOL_VERSION, --protocol_version PROTOCOL_VERSION
                        CAN protocol version
  -t DATUM_ALPHA_TIMEOUT_STEPS, --datum_alpha_timeout_steps DATUM_ALPHA_TIMEOUT_STEPS
                        timeout limit for alpha arm, in steps
  -b DATUM_BETA_TIMEOUT_STEPS, --datum_beta_timeout_steps DATUM_BETA_TIMEOUT_STEPS
                        timeout limit for beta arm, in steps
  -D FIRMWARE_DATE, --firmware_date FIRMWARE_DATE
                        ISO timestamp with firmware date (format yy-mm-dd as
                        "18-12-31")
  -N NUM_FPUS, --NUM_FPUS NUM_FPUS
                        number of simulated FPUs
  -O ALPHA_DATUM_OFFSET, --alpha-datum-offset ALPHA_DATUM_OFFSET
                        Conventional angle of datum position.
  -A ALPHA_START, --alpha-start ALPHA_START
                        simulated offset of alpha arm at start, when the step
                        count is 0. This can be used to simulate conditions
```

```
                      like a power failure
-B BETA_START, --beta-start BETA_START
                      simulated offset of beta arm at start
```

D. Glossary

Angle The absolute position of a component, measured in a physical unit (usually degrees)

CAN ID A number which a piece of hardware has which is connected to a CAN bus.

Collision The event that the beta arm of a fibre positioner unit touches another beta arm or an end stop, causing it to stop and generate an electrical collision signal.

Datum search Movement of a piece of hardware to a special switch which tells it about the exact position when it is activated.

Driver A software component which has an API and accepts high-level commands, transforming them into low-level control commands for a piece of hardware or I/O device

EtherCAN A special protocol and hardware stack which allows to send commands to a CAN (controller area network) bus via a UNIX socket connection.

Firmware That piece of embedded software that runs the motion controller of an individual fibre positioner unit.

Frequency In the context of this document, the frequency with which an FPU motion controller generates pulses which move an stepper motor forward by one step.

Fibre Positioner Unit A small robotic unit which can move a optical fibre with high precision in two dimensions

FPU Grid An assembly of up to 1000 fibre positioner units

Limit Breach The event that the alpha arm activates a switch which signals that the arm has passed the physical safe range of operation.

Microstepping The ability of stepper motor control hardware to generate "partial steps" which allows to make smoother movements.

Hardware protection A set of switches and safety checks which have the goal to ensure that hardware failures or system errors can at most cause limited damage.

State Symbolic operational state of a fibre positioner, a discrete value which defines which commands it will accept and execute.

State Machine Finite State Machine, an abstract description which defines the relationship between commands, events, and operational states of a fibre positioner unit.

Steps discrete movements in a stepper motor, which can be counted to derive the relative position

Waveform A data structure which describes how many steps the two motors of a fibre positioner unit should move in a given interval of time. This is also used for a tabled structure which describes waveforms for many fibre positioner units at once.

Path or Movement Path A data structure which describes absolute angles to which the motors of a fibre positioner unit should move at a given relative point in time.

Power-cycle switching something on and off again, with the intention to bring it into a well-defined starting state.

Socket An abstract communication connection which safely transports binary data between two computers.

Unix Time Number of seconds since January 1, 1970.

Software protection A set of rules and checks which has the goal to ensure that wrong or erroneous commands cannot damage the addressed hardware.

E. Abbreviations

CAN ID	See glossary.
CAN	Controller Area Network, an industrial bus system
FPU	Fibre positioner unit
FSM	Finite State Machine
ICS	Instrument Control Software with in the Very Large Telescope software framework
LMDB	Lightning Memory Database, a library which is used to store data quickly in files
MOONS	Multi-Object Optical and Near-Infrared Spectrograph, a large astronomic instrument

Index