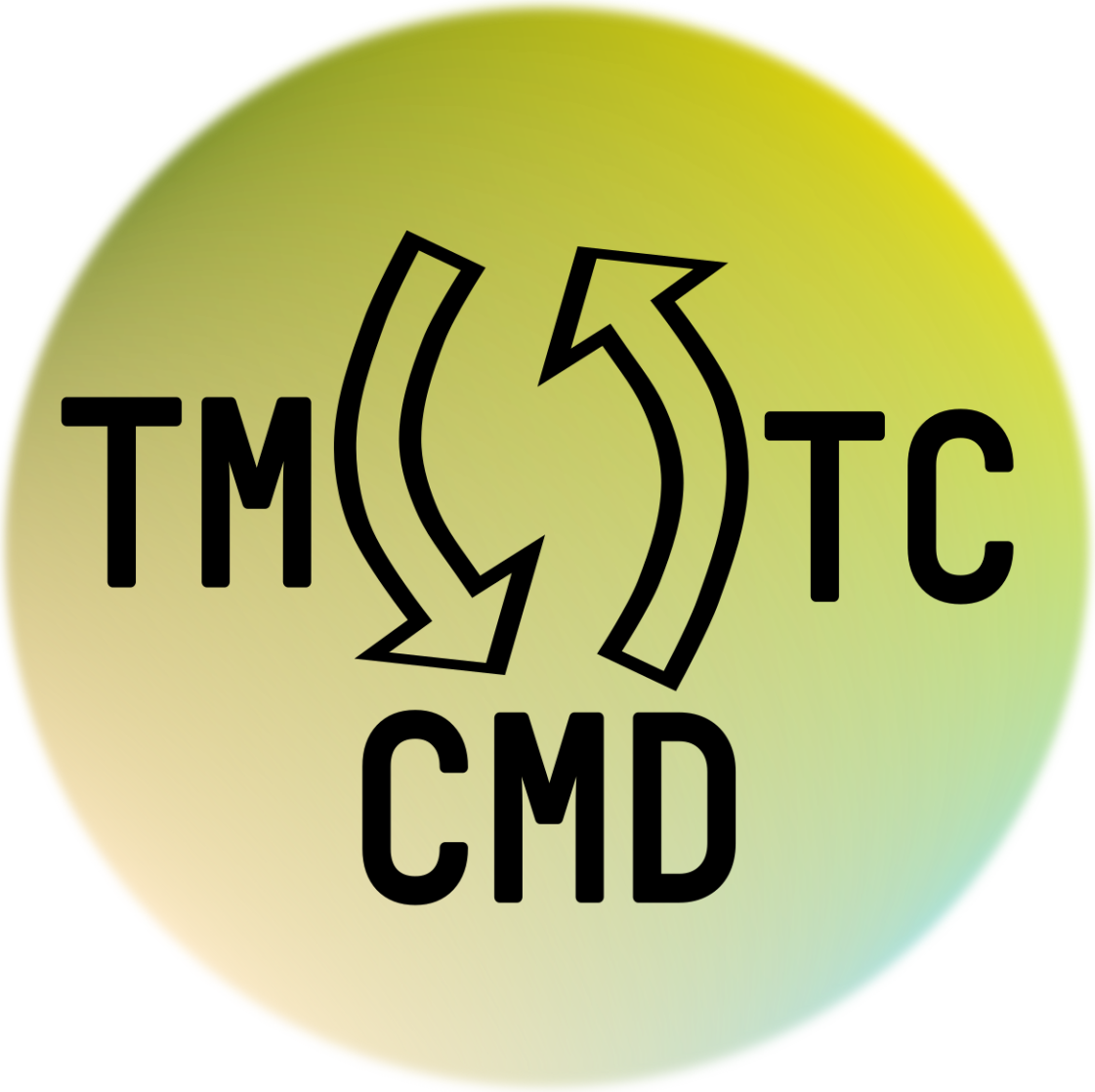


tmtccmd

Release 8.0.1

Robin Mueller

May 16, 2024

CONTENTS

This is a small Python framework targeted towards the testing of remote systems like satellites and rovers. It simplifies sending and receiving TMTCs (Telemetry and Telecommands) and testing via different communication interfaces. This tool can be used either as a command line tool or as a GUI tool which requires a PyQt5 installation. This package also has dedicated support to send and receive ECSS PUS packets or other generic CCSDS packets.

This framework started as a small application for the SOURCE project to test the on-board software but has evolved to be more generic and easily adaptable to new projects.

Other pages (online)

- project page on GitHub
- This page, when viewed online is at https://tmtccmd.readthedocs.io/en/latest/

CHAPTER

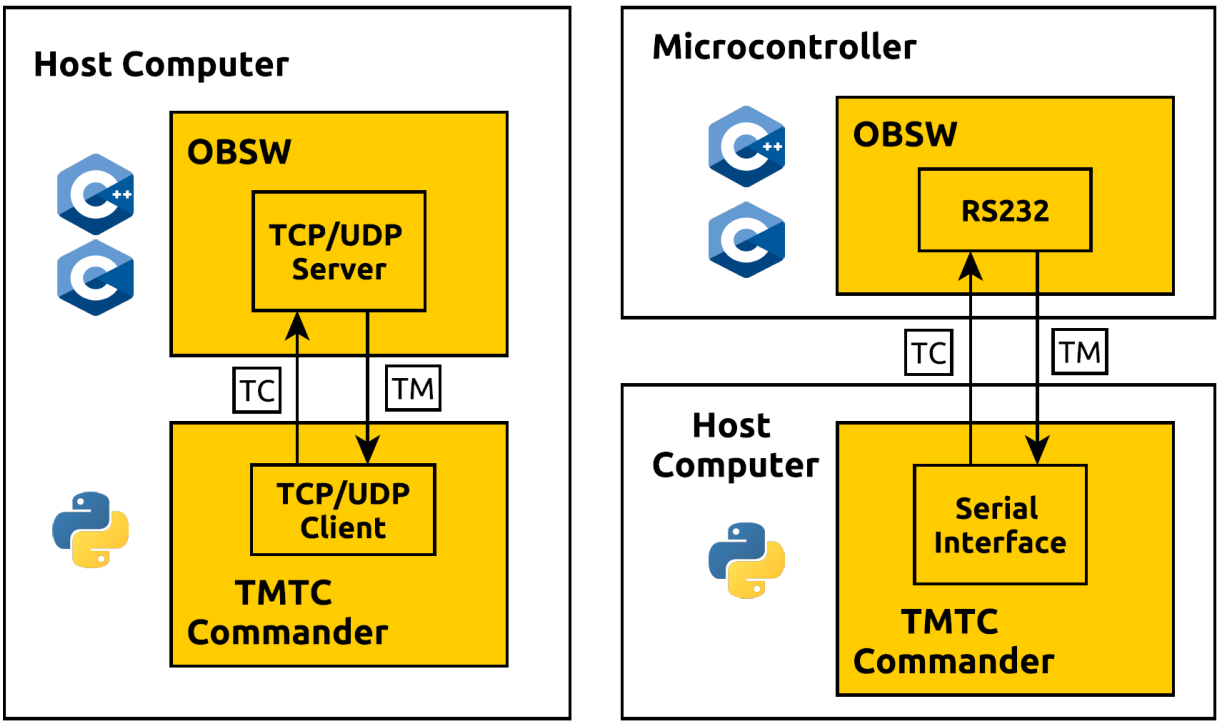
ONE

CONTENTS

1.1 Introduction

1.1.1 Overview

The goal of this framework is to make it as easy to send telecommands (TCs) to the On-Board Software (OBSW) running on an external On-Board Computer (OBC) and to analyse the telemetry (TMs) coming back. The following graph shows two possible ways to use the TMTC commander



The first way assumes that the OBSW can be run on a host computer and starts a TPC/IP server internally. The TMTC commander can then be used to send telecommands via the TCP/IP interface. The second way assumes that the OBSW is run on an external microcontroller. Here, the serial interface is used to send telecommands. Other ways like sending TMTCs via Ethernet to a microcontroller running a TCP/IP server are possible as well.

1.1.2 Features

- Generic communication interface abstraction in form of the *tmtccmd.com.ComInterface*. This abstraction could also be used without the other components of the library if the goal is to separate the packet logic from the communication interface. The *Communication Abstraction* chapter contains a more information and examples.
- Special support for Packet Utilisation Standard (PUS) packets and CCSDS Space Packets. This library uses the spacepackets library for most packet implementations.
- Support for both CLI and GUI usage.
- Flexibility in the way to specify telecommands to send and how to handle incoming telemetry. This is done by requiring the user to specify callbacks for both TC specification and TM handling.
- One-Queue Mode for simple command sequences and Multi-Queue for more complex command sequences.
- Listener mode to only listen to incoming telemetry.
- Some components are tailored towards usage with the Flight Software Framework (FSFW) and the sat-rs library.

This framework also has a communication interface abstraction which allows to exchange TMTC through different channels. The framework currently supports (among others) the following communication interfaces:

1. TCP/IP with the *tmtccmd.com.udp.UdpClient* and *tmtccmd.com.tcp.TcpSpacepacketsClient*.
2. Serial Communication with COBS encoded packets by using the *tmtccmd.com.serial_cobs.SerialCobsComIF*.

It is also possible to supply custom interfaces.

1.2 Getting Started

1.2.1 Example Project

The example application is the best way to learn how this framework works and to get started. It shows how to set up handler classes for TC and TM handling and then ties together all components. You can also run this application in GUI mode by passing the `-g` GUI flag to the example application.

Some explanation of classes and modules inside the example are given here.

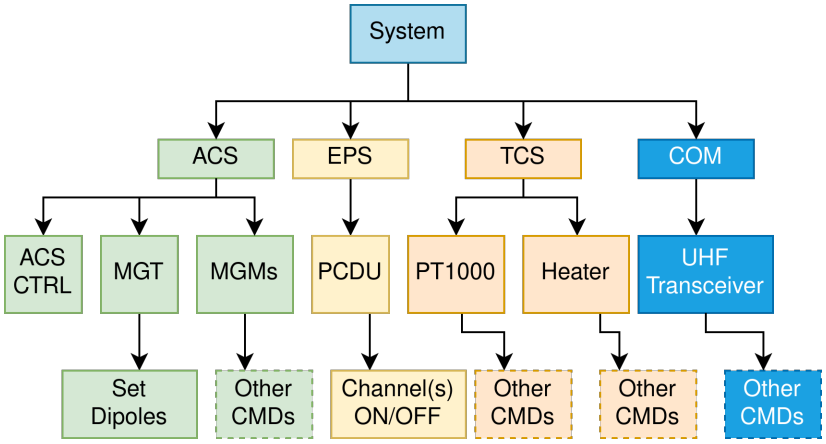
1.2.2 The Configuration Hook Class

The class *ExampleHookClass* is the example configuration class implementing the *tmtccmd.config.hook.HookBase*. There are two functions which must be impelemented by a user:

1. The *tmtccmd.config.hook.HookBase.get_communication_interface()* method is used to return a communication interface given a string identifier. You can read more about the communication abstraction in the *Communication Abstraction* chapter.
2. The *tmtccmd.config.hook.HookBase.get_command_definitions()* returns a tree of command definition which can be used by both users and developers to specify the available commands and procedures.

TC Command Definition Specification

The command tree mechanism provides a flexible mechanism to also model the command definitions for more complex systems. These systems are oftentimes structured into dedicated modules. For exampe, the command tree for a satellite system might look like this:



For a system like this, it makes a lot of sense to also model the command definitions similarly to the system. An example tree modelling the system shown in the example above would look like this when printing one built using the *tmtccmd.config.tmtc.CmdTreeNode*:

```
/                                   [ Root Node ]
├── ping                            [ Send PUS ping command ]
├── test                            [ Test Node ]
│   └── event                       [ Send PUS event test command ]
├── system                          [ System Commands ]
├── acs                             [ ACS Subsystem ]
│   ├── acs_ctrl                    [ ACS Controller ]
│   ├── mgt                         [ Magnetorquer ]
│   │   └── set_dipoles             [ Set MGT Dipoles ]
│   ├── mgm0                        [ Magnetometer 0 ]
│   │   └── other cmds              [ Other MGM commands ]
│   └── mgm1                        [ Magnetometer 1 ]
│       └── other cmds              [ Other MGM commands ]
├── tcs                             [ TCS Subsystem ]
│   ├── tcs_ctrl                    [ TCS Controller ]
│   ├── pt1000                      [ Temperature Sensor ]
│   └── heater                      [ Heater ]
├── com                             [ COM Subsystem ]
│   └── uhf_transceiver             [ UHF Transceiver ]
└── eps                             [ EPS Subsystem ]
    └── pcdu                        [ PCDU ]
        ├── channel_0_on            [ Channel 0 on ]
        ├── channel_0_off           [ Channel 0 off ]
        ├── channel_1_on            [ Channel 1 on ]
        └── channel_1_off           [ Channel 1 off ]
```

and the code to create this tree would look like this:

```
def get_command_definitions(self) -> CmdTreeNode:
    root_node = CmdTreeNode.root_node()
```

(continues on next page)

(continued from previous page)

```
    root_node.add_child(CmdTreeNode("ping", "Send PUS ping command"))
    root_node.add_child(CmdTreeNode("test", "Test Node"))
    root_node.children["test"].add_child(
        CmdTreeNode("event", "Send PUS event test command")
    )
    root_node.add_child(CmdTreeNode("system", "System Commands"))
    root_node.add_child(CmdTreeNode("acs", "ACS Subsystem"))
    root_node["acs"].add_child(CmdTreeNode("acs_ctrl", "ACS Controller"))
    root_node["acs"].add_child(CmdTreeNode("mgt", "Magnetorquer"))
    root_node["acs"]["mgt"].add_child(CmdTreeNode("set_dipoles", "Set MGT Dipoles"))
    root_node["acs"].add_child(CmdTreeNode("mgm0", "Magnetometer 0"))
    root_node["acs"].add_child(CmdTreeNode("mgm1", "Magnetometer 1"))
    mgm_node = CmdTreeNode("other cmds", "Other MGM commands")
    root_node["acs"]["mgm0"].add_child(mgm_node)
    root_node["acs"]["mgm1"].add_child(mgm_node)
    root_node.add_child(CmdTreeNode("tcs", "TCS Subsystem"))
    root_node["tcs"].add_child(CmdTreeNode("tcs_ctrl", "TCS Controller"))
    root_node["tcs"].add_child(CmdTreeNode("pt1000", "Temperature Sensor"))
    root_node["tcs"].add_child(CmdTreeNode("heater", "Heater"))
    root_node.add_child(CmdTreeNode("com", "COM Subsystem"))
    root_node["com"].add_child(CmdTreeNode("uhf_transceiver", "UHF Transceiver"))
    root_node.add_child(CmdTreeNode("eps", "EPS Subsystem"))
    root_node["eps"].add_child(CmdTreeNode("pcdu", "PCDU"))
    root_node["eps"]["pcdu"].add_child(CmdTreeNode("channel_0_on", "Channel 0 on"))
    root_node["eps"]["pcdu"].add_child(
        CmdTreeNode("channel_0_off", "Channel 0 off")
    )
    root_node["eps"]["pcdu"].add_child(CmdTreeNode("channel_1_on", "Channel 1 on"))
    root_node["eps"]["pcdu"].add_child(
        CmdTreeNode("channel_1_off", "Channel 1 off")
    )
    return root_node
```

You can now specify your commands as command paths, which will then serve as identifier for single command or command stacks and procedures. The command path will be passed on as the *cmd_path* parameter of the `tmtccmd.tmtc.procedure.DefaultProcedureInfo` which is passed to the *tmtccmd.tmtc.handler.TcHandlerBase* implementation of the user.

It is also possible to pass the command path as a CLI argument. For example, you can use `./tmtcc.py -p /test` to send a ping command with the example application. Passing the command tree definition to the framework also allows it to provide a GUI command path selector for the GUI mode.

It is optionally possible to pass a command history to the framework by implementing the *tmtccmd.config.hook.HookBase.get_cmd_history()* function. An example implementation using the `prompt_toolkit.history.FileHistory` class would look like this:

```
def get_cmd_history(self) -> Optional[History]:
    """Optionlly return a history class for the past command paths which will be used
    when prompting a commad path from the user in CLI mode."""
    return FileHistory(".tmtc-cli-history.txt")
```

1.2.3 The TC handler

This object is responsible for the telecommand handling. Therefore this object implements the *tmtccmd.tmtc.handler.TcHandlerBase*.

In the example case, the handler object is reponsible for returning telecommand queues based on input information. This task is done by the *tmtccmd.tmtc.handler.TcHandlerBase.feed_cb()* callback method.

The actual handling of telecommand queue entries is done in the *tmtccmd.tmtc.handler.TcHandlerBase.send_cb()* method implementation. One thing to note here is that a queue entry does not necessarily have to be a command to be sent. For example, the queue can also contain something like log requests or delay requests, or even complete custom requests. These requests can then be handled by the user.

1.2.4 The PUS TM handler

This object is responsible for the handling of PUS telemetry. In the example case, the handler object is responsible space packets with a certain application process identifier (APID). Therefore, this object implements the *tmtccmd.tmtc.common.SpecificApidHandlerBase*.

The *handle_tm* method implementation is the primary functions where incoming PUS packets are handled. This can something like prinouts or logging, either to a file or to a database.

1.2.5 Other example applications

The EIVE and SOURCE project implementation of the TMTC commander provide more complex implementations.

1.3 Communication Abstraction

This library contains a generic communication abstraction specifically targeted towards the exchange of binary data like CCSDS packets.

The core of this abstraction is the *tmtccmd.com.ComInterface* class. An implementation of this class can be passed to other library components and makes the commanding logic independent of the used interface. It is also possible to use this abstraction independenctly of the other library components and implement custom interfaces.

The use of a communication abstraction allows to use the same TMTC specification independently of the used communication interface. For example, it might be possible to build an on-board software both for a remote MCU and for a hosted system. The MCU might expect the same command data to be exchanged via a specific transport layer, while the hosted version might just use a UDP socket.

The following example shows how to use the *tmtccmd.com.udp.UdpClient* to send PUS packets (a subtype of CCSDS space packets) to a UDP server

```
import socket

from tmtccmd.com import ComInterface
from tmtccmd.com.udp import UdpClient, EthAddr
from spacepackets.ecss.tc import PusTelecommand

def send_my_telecommand(tc: PusTelecommand, com_if: ComInterface):
    com_if.send(tc.pack())

simulated_udp_server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

(continues on next page)

(continued from previous page)

```
# Let the OS assign an appropriate port.
addr = ("127.0.0.1", 0)
simulated_udp_server.bind(addr)
addr = simulated_udp_server.getsockname()

udp_client = UdpClient("udp", send_address=EthAddr.from_tuple(addr))
udp_client.initialize()
udp_client.open()

ping_tc = PusTelecommand(service=17, subservice=1, apid=0x020)
send_my_telecommand(ping_tc, udp_client)

recvd_data, sender_addr = simulated_udp_server.recvfrom(4096)
recvd_tc = PusTelecommand.unpack(recvd_data)

print(f"Sent TC: {ping_tc}")
print(f"Received TC: {recvd_tc}")
udp_client.close()
```

Output:

```
Sent TC: PUS TC[17, 1] with Request ID 0x1820c000, APID 0x020, SSC 0
Received TC: PUS TC[17, 1] with Request ID 0x1820c000, APID 0x020, SSC 0
```

It should be noted that the *tmtccmd.com.ComInterface.receive* function should never block and returns a list of received packets. For many concrete implementations, this means that a separate receiver thread is required to poll for packets periodically and fill them into a packet list, which is then returned on the receive call.

The receiver thread may then also implement the logic required for some transport layers using blocking API. For example, the serial COBS interface will perform a blocking `serial.Serial.read()` to look for the start marker 0, and then read until the end marker 0 has been read.

Here is another example where a the same packet is sent via a serial interface. This example only runs of Unix systems because it simulates a serial port using pty.

```
import os
import pty
import time
from cobs import cobs

from tmtccmd.com import ComInterface
from tmtccmd.com.serial_cobs import SerialCfg, SerialCobsComIF

sim_ser_device, pty_slave = pty.openpty()
sim_serial_port = os.ttyname(pty_slave)
ser_cfg = SerialCfg(
    "serial_cobs", serial_port=sim_serial_port, baud_rate=9600, serial_timeout=1.0
)
cobs_com_if = SerialCobsComIF(ser_cfg)
cobs_com_if.initialize()
cobs_com_if.open()

test_data = bytes([0x01, 0x02, 0x03])
```

(continues on next page)

(continued from previous page)

```
# Will be used later and to determine how much to read.
encoded_data = cobs.encode(test_data)

# Data will be COBS encoded internally, with the 0 frame delimiter inserted at the start␣
↪and
# end
print(f"Sending raw data: 0x[{test_data.hex(sep=',')}]")
cobs_com_if.send(test_data)

# Other side receives COBS encoded packet
encoded_packet = os.read(sim_ser_device, len(encoded_data) + 2)
decoded_packet = cobs.decode(encoded_packet[1:-1])
print(f"Encoded packet received at simulated serial device: 0x[{encoded_packet.hex(sep=',
↪')}]")
print(f"Decoded packet: 0x[{decoded_packet.hex(sep=',')}]")

# Now send COBS encoded data back
data_sent_back = bytes([0x01, 0x02, 0x03])
# 0 start marker
cobs_encoded_data = bytearray([0])
cobs_encoded_data.extend(encoded_data)
# 0 end marker
cobs_encoded_data.append(0)
os.write(sim_ser_device, cobs_encoded_data)
# Receiver thread might take some time
time.sleep(0.1)

packet_list = cobs_com_if.receive()
print(f"Data received from simulated serial device: 0x[{packet_list[0].hex(sep=',')}]")
cobs_com_if.close()
```

Output:

```
Sending raw data: 0x[01,02,03]
Encoded packet received at simulated serial device: 0x[00,04,01,02,03,00]
Decoded packet: 0x[01,02,03]
Data received from simulated serial device: 0x[01,02,03]
```

This interface could of course also exchange a higher level protocol like PUS packets, but this example was kept more simple to also show how a communication interface can also provide a transport layer.

1.4 CCSDS File Delivery Protocol (CFDP)

The majority of the CFDP components were moved to the dedicated cfdp-py library. It is recommended to read its dedicated documentation.

1.5 API

1.5.1 Communication Package

Communication module. Provides generic abstraction for communication and commonly used concrete implementations.

Communication Package

Communication module. Provides generic abstraction for communication and commonly used concrete implementations.

class tmtccmd.com.**ComInterface**

Bases: ABC

Generic form of a communication interface to separate communication logic from the underlying interface.

abstract close(*args: Any = 0*)

Closes the ComIF and releases any held resources (for example a Communication Port).

Returns

abstract data_available(*timeout: float*, *parameters: Any = 0*) → int

Check whether TM packets are available.

Parameters

- **timeout** – Can be used to block on available data if supported by the specific communication interface.
- **parameters** – Can be an arbitrary parameter.

Raises

ReceptionDecodeError – If the underlying COM interface uses encoding and decoding when determining the number of available packets, this exception can be thrown on decoding errors.

Returns

0 if no data is available, number of packets otherwise.

abstract property id: str

abstract initialize(*args: Any = 0*) → Any

Perform initializations step which can not be done in constructor or which require returnvalues.

abstract is_open() → bool

Can be used to check whether the communication interface is open. This is useful if opening a COM interface takes a longer time and is non-blocking

abstract open(*args: Any = 0*)

Opens the communication interface to allow communication.

Returns

abstract receive(*parameters: Any = 0*) → List[bytes]

Returns a list of received packets. The child class can use a separate thread to poll for the packets or use some other mechanism and container like a deque to store packets to be returned here.

Parameters
parameters

Raises
ReceptionDecodeError – If the underlying COM interface uses encoding and decoding and the decoding fails, this exception will be returned.

Returns

abstract send(*data: bytes*)

Send raw data.

Raises
SendError – Sending failed for some reason.

exception tmtccmd.com.**ReceptionDecodeError**(*msg: str*, *custom_exception: Exception | None*)

Bases: Exception

Generic decode error which can also wrap the exception thrown by other libraries.

exception tmtccmd.com.**SendError**(*msg: str*, *custom_exception: Exception | None*)

Bases: Exception

Generic send error which can also wrap the exception thrown by other libraries.

TCP Client Module

TCP communication interface

class tmtccmd.com.tcp.**TcpCommunicationType**(*value*)

Bases: Enum

Parse for space packets in the TCP stream, using the space packet header.

SPACE_PACKETS = 0

class tmtccmd.com.tcp.**TcpSpacepacketsClient**(*com_if_id: str*, *space_packet_ids: Sequence[PacketId]*, *inner_thread_delay: float*, *target_address:* EthAddr, *max_packets_stored: int | None = None*)

Bases: *ComInterface*

Communication interface for TCP communication. This particular interface expects raw space packets to be sent via TCP and uses a list of passed packet IDs to parse for them.

close(*args: any | None = None*) → None

Closes the ComIF and releases any held resources (for example a Communication Port).

Returns

data_available(*timeout: float = 0*, *parameters: any = 0*) → int

Check whether TM packets are available.

Parameters

- **timeout** – Can be used to block on available data if supported by the specific communication interface.
- **parameters** – Can be an arbitrary parameter.

Raises

ReceptionDecodeError – If the underlying COM interface uses encoding and decoding when determining the number of available packets, this exception can be thrown on decoding errors.

Returns

0 if no data is available, number of packets otherwise.

property id: str

initialize(*args: Any | None = None*)

Perform initializations step which can not be done in constructor or which require returnvalues.

is_open() → bool

Can be used to check whether the communication interface is open. This is useful if opening a COM interface takes a longer time and is non-blocking

open(*args: Any | None = None*)

Opens the communication interface to allow communication.

Returns

receive(*poll_timeout: float = 0*) → List[bytes]

Returns a list of received packets. The child class can use a separate thread to poll for the packets or use some other mechanism and container like a deque to store packets to be returned here.

Parameters

parameters

Raises

ReceptionDecodeError – If the underlying COM interface uses encoding and decoding and the decoding fails, this exception will be returned.

Returns

send(*data: bytes*)

Send raw data.

Raises

SendError – Sending failed for some reason.

class tmtccmd.com.tcpip_utils.**EthAddr**(*ip_addr: 'str'*, *port: 'int'*)

Bases: object

classmethod from_tuple(*addr: Tuple[str, int]*) → *EthAddr*

ip_addr: str

port: int

property to_tuple: Tuple[str, int]

class tmtccmd.com.tcpip_utils.**TcpIpConfigIds**(*value*)

Bases: Enum

An enumeration.

RECV_ADDRESS = 2

RECV_MAX_SIZE = 3

SEND_ADDRESS = 1

SPACE_PACKET_ID = 4

auto = <class 'enum.auto'>

class tmtccmd.com.tcpip_utils.**TcpIpType**(*value*)

Bases: Enum

An enumeration.

TCP = 1

UDP = 2

UDP_RECV = 3

tmtccmd.com.tcpip_utils.**determine_recv_buffer_len**(*json_cfg_path: str*, *tcpip_type:* TcpIpType)

tmtccmd.com.tcpip_utils.**determine_tcp_send_address**(*json_cfg_path: str*) → *EthAddr*

tmtccmd.com.tcpip_utils.**determine_tcpip_address**(*tcpip_type:* TcpIpType, *json_cfg_path: str*) → *EthAddr*

tmtccmd.com.tcpip_utils.**determine_udp_recv_address**(*json_cfg_path: str*) → *EthAddr*

tmtccmd.com.tcpip_utils.**determine_udp_send_address**(*json_cfg_path: str*) → *EthAddr*

tmtccmd.com.tcpip_utils.**prompt_ip_address**(*type_str: str*) → *EthAddr*

Prompt a valid IP address from the user

tmtccmd.com.tcpip_utils.**prompt_recv_buffer_len**(*tcpip_type:* TcpIpType) → int

UDP Client Module

UDP Communication Interface

class tmtccmd.com.udp.**UdpClient**(*com_if_id: str*, *send_address:* EthAddr, *recv_addr:* EthAddr | *None* = *None*)

Bases: *ComInterface*

Communication interface for UDP communication

close(*args: any* | *None* = *None*) → None

Closes the ComIF and releases any held resources (for example a Communication Port).

Returns

data_available(*timeout: float = 0*, *parameters: any = 0*) → bool

Check whether TM packets are available.

Parameters

- **timeout** – Can be used to block on available data if supported by the specific communication interface.
- **parameters** – Can be an arbitrary parameter.

Raises

ReceptionDecodeError – If the underlying COM interface uses encoding and decoding when determining the number of available packets, this exception can be thrown on decoding errors.

Returns

0 if no data is available, number of packets otherwise.

property id: str

initialize(*args: any | None = None*) → any

Perform initializations step which can not be done in constructor or which require returnvalues.

is_open() → bool

Can be used to check whether the communication interface is open. This is useful if opening a COM interface takes a longer time and is non-blocking

open(*args: any | None = None*)

Opens the communication interface to allow communication.

Returns

receive(*poll_timeout: float = 0*) → List[bytes]

Returns a list of received packets. The child class can use a separate thread to poll for the packets or use some other mechanism and container like a deque to store packets to be returned here.

Parameters

parameters

Raises

ReceptionDecodeError – If the underlying COM interface uses encoding and decoding and the decoding fails, this exception will be returned.

Returns

send(*data: bytes*)

Send raw data.

Raises

SendError – Sending failed for some reason.

Serial Communication with COBS Module

class tmtccmd.com.serial_cobs.**SerialCobsComIF**(*ser_cfg:* SerialCfg)

Bases: *SerialComBase*, *ComInterface*

Serial communication interface which uses the COBS protocol to encode and decode packets.

This class will spin up a receiver thread on the *open()* call to poll for COBS encoded packets. This means that the *close()* call might block until the receiver thread has shut down.

close(*args: any | None = None*) → None

Closes the ComIF and releases any held resources (for example a Communication Port).

Returns

data_available(*timeout: float*, *parameters: any = 0*) → int

Check whether TM packets are available.

Parameters

- **timeout** – Can be used to block on available data if supported by the specific communication interface.
- **parameters** – Can be an arbitrary parameter.

Raises

ReceptionDecodeError – If the underlying COM interface uses encoding and decoding when determining the number of available packets, this exception can be thrown on decoding errors.

Returns

0 if no data is available, number of packets otherwise.

property id: str

initialize(*args: any | None = None*) → any

Perform initializations step which can not be done in constructor or which require returnvalues.

is_open() → bool

Can be used to check whether the communication interface is open. This is useful if opening a COM interface takes a longer time and is non-blocking

open(*args: any | None = None*) → None

Spins up a receiver thread to permanently check for new COBS encoded packets.

receive(*parameters: any = 0*) → List[bytes]

Returns a list of received packets. The child class can use a separate thread to poll for the packets or use some other mechanism and container like a deque to store packets to be returned here.

Parameters

parameters

Raises

ReceptionDecodeError – If the underlying COM interface uses encoding and decoding and the decoding fails, this exception will be returned.

Returns

send(*data: bytes*)

Send raw data.

Raises
SendError – Sending failed for some reason.

Serial Communication with DLE Module

class tmtccmd.com.serial_dle.**DleCfg**(*dle_queue_len: int | None = None*, *dle_max_frame: int | None = None*, *encode_cr: bool = True*)

Bases: object

dle_max_frame: int | None = None

dle_queue_len: int | None = None

encode_cr: bool = True

class tmtccmd.com.serial_dle.**SerialDleComIF**(*ser_cfg:* SerialCfg, *dle_cfg:* DleCfg | *None*)

Bases: *SerialComBase*, *ComInterface*

Serial communication interface which uses the DLE protocol to encode and decode packets.

This class will spin up a receiver thread on the *open()* call to poll for DLE encoded packets. This means that the *close()* call might block until the receiver thread has shut down.

close(*args: any | None = None*) → None

Closes the ComIF and releases any held resources (for example a Communication Port).

Returns

data_available(*timeout: float*, *parameters: any = 0*) → int

Check whether TM packets are available.

Parameters

- **timeout** – Can be used to block on available data if supported by the specific communication interface.
- **parameters** – Can be an arbitrary parameter.

Raises
ReceptionDecodeError – If the underlying COM interface uses encoding and decoding when determining the number of available packets, this exception can be thrown on decoding errors.

Returns
0 if no data is available, number of packets otherwise.

property id: str

initialize(*args: any | None = None*) → any

Perform initializations step which can not be done in constructor or which require returnvalues.

is_open() → bool

Can be used to check whether the communication interface is open. This is useful if opening a COM interface takes a longer time and is non-blocking

open(*args: any | None = None*) → None

Spins up a receiver thread to permanently check for new DLE encoded packets.

receive(*parameters: any = 0*) → List[bytes]

Returns a list of received packets. The child class can use a separate thread to poll for the packets or use some other mechanism and container like a deque to store packets to be returned here.

Parameters
parameters

Raises
ReceptionDecodeError – If the underlying COM interface uses encoding and decoding and the decoding fails, this exception will be returned.

Returns

send(*data: bytes*)

Send raw data.

Raises
SendError – Sending failed for some reason.

Generic Serial Modules

class tmtccmd.com.serial_base.**SerialCfg**(*com_if_id: str*, *serial_port: str*, *baud_rate: int*, *serial_timeout: float*)

Bases: object

baud_rate: int

com_if_id: str

serial_port: str

serial_timeout: float

class tmtccmd.com.serial_base.**SerialComBase**(*logger: Logger*, *ser_cfg:* SerialCfg, *ser_com_type:* SerialCommunicationType)

Bases: object

close_port()

static data_available_from_queue(*timeout: float*, *reception_buffer: deque*)

is_port_open() → bool

open_port()

class tmtccmd.com.serial_base.**SerialCommunicationType**(*value*)

Bases: Enum

Right now, two serial communication methods are supported. One uses frames with a fixed size containing PUS packets and the other uses a simple ASCII based transport layer called DLE. If DLE is used, it is expected that the sender side encoded the packets with the DLE protocol. Any packets sent will also be encoded.

COBS = 0

DLE_ENCODING = 2

class tmtccmd.com.serial_base.**SerialConfigIds**(*value*)

Bases: Enum

An enumeration.

SERIAL_BAUD_RATE = 2

SERIAL_COMM_TYPE = 4

SERIAL_DLE_MAX_FRAME_SIZE = 7

SERIAL_DLE_QUEUE_LEN = 6

SERIAL_FRAME_SIZE = 5

SERIAL_PORT = 1

SERIAL_TIMEOUT = 3

tmtccmd.com.ser_utils.**check_port_validity**(*com_port_to_check: str*) → bool

tmtccmd.com.ser_utils.**determine_baud_rate**(*json_cfg_path: str*) → int

Determine baud rate. Tries to read from JSON first. If the baud rate is not contained in the config JSON, prompt it from user instead with the option to store value in JSON file.

Returns

Determined baud rate

tmtccmd.com.ser_utils.**determine_com_port**(*json_cfg_path: str*) → str

Determine serial port. Tries to read from JSON first. If the serial port is not contained in the config JSON, prompt it from user instead with the option to store value in JSON file.

Returns

Determined serial port

tmtccmd.com.ser_utils.**find_com_port_from_hint**(*hint: str*) → Tuple[bool, str]

Find a COM port based on a hint string

tmtccmd.com.ser_utils.**prompt_com_port**() → str

Dummy Module

Dummy Virtual Communication Interface. Currently serves to use the TMTC program without needing external hardware or an extra socket

class tmtccmd.com.dummy.**DummyComIF**

Bases: *ComInterface*

close(*args: any | None = None*) → None

Closes the ComIF and releases any held resources (for example a Communication Port).

Returns

data_available(*timeout: float = 0*, *parameters: any = 0*)

Check whether TM packets are available.

Parameters

- **timeout** – Can be used to block on available data if supported by the specific communication interface.

- **parameters** – Can be an arbitrary parameter.

Raises

ReceptionDecodeError – If the underlying COM interface uses encoding and decoding when determining the number of available packets, this exception can be thrown on decoding errors.

Returns

0 if no data is available, number of packets otherwise.

property id: str

initialize(*args: any | None = None*) → any

Perform initializations step which can not be done in constructor or which require returnvalues.

is_open() → bool

Can be used to check whether the communication interface is open. This is useful if opening a COM interface takes a longer time and is non-blocking

open(*args: any | None = None*) → None

Opens the communication interface to allow communication.

Returns

receive(*parameters: any = 0*) → List[bytes]

Returns a list of received packets. The child class can use a separate thread to poll for the packets or use some other mechanism and container like a deque to store packets to be returned here.

Parameters

parameters

Raises

ReceptionDecodeError – If the underlying COM interface uses encoding and decoding and the decoding fails, this exception will be returned.

Returns

send(*data: bytes*)

Send raw data.

Raises

SendError – Sending failed for some reason.

class tmtccmd.com.dummy.**DummyHandler**

Bases: object

generate_reply_package()

Generate a reply package. Currently, this only generates a reply for a ping telecommand.

insert_telecommand(*data: bytes*)

pass_telecommand(*data: bytearray*)

Deprecated since version 6.0.0: Use insert_telecommand instead

receive_reply_package() → List[bytes]

Utilities Module

tmtccmd.com.utils.**determine_com_if**(*com_if_dict: Mapping[str, Tuple[str, Any]]*, *json_cfg_path: str*, *use_prompts: bool*) → str

tmtccmd.com.utils.**prompt_com_if**(*com_if_dict: Mapping[str, Tuple[str, Any]]*) → str

tmtccmd.com.utils.**store_com_if_json**(*com_if_string: str*, *json_cfg_path: str*)

QEMU Module

QEMU_SERIAL Communication Interface to communicate with emulated QEMU_SERIAL hardware via the UART interface.

It utilizes the the asyncio library.

Requirements:
Python >= 3.7 (asyncio support)

Instructions:
Run QEMU_SERIAL (modified for OBSW) via

qemu-system-arm -M isis-obc -monitor stdio -bios path/to/sourceobsw-at91sam9g20_ek-sdram.bin -qmp unix:/tmp/qemu,server -S

Then run the telecommand script with -c 2

class tmtccmd.com.qemu.**DataFrame**(*seq*, *cat*, *frame_id*, *data=None*)

Bases: object

Basic protocol unit for communication via the IOX API introduced for external device emulation

bytes()

Convert this protocol unit to raw bytes

class tmtccmd.com.qemu.**QEMUComIF**(*serial_cfg: SerialCfg*, *ser_com_type: SerialCommunicationType = SerialCommunicationType.DLE_ENCODING*)

Bases: *ComInterface*

Specific Communication Interface implementation of the QEMU_SERIAL USART protocol for the TMTC software

close(*_=None*) → None

Closes the ComIF and releases any held resources (for example a Communication Port).

Returns

data_available(*timeout: float = 0*, *_=0*) → int

Check whether TM packets are available.

Parameters

- **timeout** – Can be used to block on available data if supported by the specific communication interface.
- **parameters** – Can be an arbitrary parameter.

Raises
ReceptionDecodeError – If the underlying COM interface uses encoding and decoding

when determining the number of available packets, this exception can be thrown on decoding errors.

Returns
0 if no data is available, number of packets otherwise.

data_available_dle(*timeout: float = 0*) → int

data_available_fixed_frame(*timeout: float = 0*) → int

property id: str

initialize(_=*None*)

Needs to be called by application code once for DLE mode!

is_open() → bool

Can be used to check whether the communication interface is open. This is useful if opening a COM interface takes a longer time and is non-blocking

open(_=*None*) → None

Opens the communication interface to allow communication.

Returns

async poll_dle_packets()

receive(_) → List[bytes]

Returns a list of received packets. The child class can use a separate thread to poll for the packets or use some other mechanism and container like a deque to store packets to be returned here.

Parameters
parameters

Raises
ReceptionDecodeError – If the underlying COM interface uses encoding and decoding and the decoding fails, this exception will be returned.

Returns

send(*data: bytearray*)

Send raw data.

Raises
SendError – Sending failed for some reason.

send_data(*data: bytearray*)

async send_data_async(*data*)

set_dle_settings(*dle_queue_len: int*, *dle_max_frame: int*, *dle_timeout: float*)

set_fixed_frame_settings(*serial_frame_size: int*)

async start_dle_polling()

class tmtccmd.com.qemu.**QmpConnection**(*addr='/tmp/qemu'*)

Bases: object

A connection to a QEMU_SERIAL machine via QMP

close()

Close this connection

async cont()

Continue machine execution if it has been paused

async open()

Open this connection. Connect to the machine ensure that the connection is ready to use after this call.

async quit()

Quit the emulation. This causes the emulator to (non-gracefully) shut down and close.

async stop()

Stop/pause machine execution

exception tmtccmd.com.qemu.QmpException(*ret*, **args*, ***kwargs*)

Bases: Exception

An exception caused by the QML/QEMU_SERIAL as response to a failed command

class tmtccmd.com.qemu.QmpProtocol(*conn*)

Bases: Protocol

The QMP transport protocoll implementation

connection_lost(*exc*)

Called when the connection is lost or closed.

The argument is an exception object or None (the latter meaning a regular EOF is received or the connection was aborted or closed).

connection_made(*transport*)

Called when a connection is made.

The argument is the transport representing the pipe connection. To receive data, wait for data_received() calls. When the connection is closed, connection_lost() is called.

data_received(*data*)

Called when some data is received.

The argument is a bytes object.

class tmtccmd.com.qemu.Usart(*addr*)

Bases: object

close()

Close this connection

async static create_async(*addr*)

flush()

get_data_in_waiting() → int

inject_frame_error()

Inject a frame error (set CSR_FRAME)

inject_overrun_error()

Inject an overrun error (set CSR_OVRE)

inject_parity_error()

Inject a parity error (set CSR_PARE)

inject_timeout_error()

Inject a timeout (set CSR_TIMEOUT)

new_data_available() → bool

async open()

Open this connection

read(*n*)

Wait for 'n' bytes to be received from the USART timeout in seconds

async read_async(*n*, *timeout=None*)

Wait for 'n' bytes to be received from the USART.

This function will return early if the specified timeout (in seconds) is exceeded. In this case, only the data received up to that point will be returned. If timeout is None, no timeout will be set.

async read_until_async(*expected*, *size=None*, *timeout=None*)

Read data until either the expected byte sequence has been found, the specified number of bytes has been received, or the timeout has occured.

This function will return whatever data has been received up until the first termination condition has been met. In case size is None, there will be no size limit. In case timeout is None, ther will be no timeout.

async write(*data*)

Write data (bytes) to the USART device

class tmtccmd.com.qemu.**UsartProtocol**(*conn*)

Bases: Protocol

The USART transport protocoll implementation

connection_lost(*exc*)

Called when the connection is lost or closed.

The argument is an exception object or None (the latter meaning a regular EOF is received or the connection was aborted or closed).

connection_made(*transport*)

Called when a connection is made.

The argument is the transport representing the pipe connection. To receive data, wait for data_received() calls. When the connection is closed, connection_lost() is called.

data_received(*data*)

Called when some data is received.

The argument is a bytes object.

exception tmtccmd.com.qemu.**UsartStatusException**(*errn*, **args*, ***kwargs*)

Bases: Exception

An exception returned by the USART send command

tmtccmd.com.qemu.**parse_dataframes**(*buf*)

Parse a variable number of DataFrames from the given byte buffer

tmtccmd.com.qemu.**start_background_loop**(*loop: AbstractEventLoop*) → None

1.5.2 ECSS & CCSDS Package

This module contains PUS data structures and helpers common for both PUS telemetry and telecommands.

Content:

- *tmtccmd.pus.VerificationWrapper* helper class

PUS Package

Submodules

Module contents

This module contains PUS data structures and helpers common for both PUS telemetry and telecommands.

Content:

- *tmtccmd.pus.VerificationWrapper* helper class

class tmtccmd.pus.**CustomFsfwPusService**(*value*)

Bases: IntEnum

An enumeration.

SERVICE_200_MODE = 200

class tmtccmd.pus.**VerificationWrapper**(*pus_verificator: PusVerificator*, *console_logger: Logger | None*, *file_logger: Logger | None*)

Bases: object

add_tc(*pus_tc: PusTc*) → bool

add_tm(*srv_1_tm: Service1Tm*) → TmCheckResult

dlog(*log_str: str*, *level: int = 20*)

log_progress_to_console_from_status(*status: VerificationStatus*, *req_id: RequestId*, *subservice: Subservice | None = None*)

log_to_console(*srv_1_tm: Service1Tm*, *res: TmCheckResult*)

log_to_console_from_req_id(*req_id: RequestId*, *res: TmCheckResult*, *subservice: Subservice | None = None*)

log_to_file(*srv_1_tm: Service1Tm*, *res: TmCheckResult*)

log_to_file_from_req_id(*req_id: RequestId*, *res: TmCheckResult*, *subservice: Subservice | None = None*)

log_to_file_from_status(*status: VerificationStatus*, *req_id: RequestId*, *subservice: Subservice | None = None*)

static step_num(*status: VerificationStatus*)

property verificator: PusVerificator

tmtccmd.pus.**cross_mark_unicode**(*with_color: bool*) → str

tmtccmd.pus.**dash_unicode**(*with_color: bool*) → str

tmtccmd.pus.**gen_console_char_from_status**(*status: StatusField*, *with_color: bool*)

tmtccmd.pus.**gen_file_char_from_status**(*status: StatusField*)

tmtccmd.pus.**tick_mark_unicode**(*with_color: bool*) → str

Service 1 Telecommand Verification Module

class tmtccmd.pus.tm.s1_verification.**Service1FsfwWrapper**(*tm: Service1Tm*)

Bases: object

Service 2 Raw Commanding Telemetry Module

Service 3 Housekeeping Modules

PUS Service 3 components

class tmtccmd.pus.tm.s3_hk_base.**HkContentType**(*value*)

Bases: Enum

An enumeration.

DEFINITIONS = <class 'enum.auto'>

HK = <class 'enum.auto'>

class tmtccmd.pus.tm.s3_hk_base.**Service3Base**(*object_id: int*, *custom_hk_handling: bool = False*)

Bases: object

Base class. The TMTC core provides a Service 3 implementation which is intended to be used with the FSFW. However, users can define an own Service 3 implementation.

The TMTC printer utility uses the fields defined in this base class to perform prinouts so if a custom class is defined, the user should implement this class and fill the fields in the TM handling hook if printout of the HK field and validity checking is desired.

property has_custom_hk_handling: bool

property hk_definitions_list: Tuple[List, List]

Can be implemented by a child class to print definitions lists. The first list should contain a header with parameter names, and the second list shall contain the corresponding set IDs

property object_id: *ComponentIdU32*

property set_id: int

Contains definitions and functions related to PUS Service 3 Telecommands.

tmtccmd.pus.tc.s3_fsfw_hk.**create_disable_periodic_hk_command**(*sid: bytes*) → PusTc

tmtccmd.pus.tc.s3_fsfw_hk.**create_disable_periodic_hk_command_with_diag**(*diag: bool*, *sid: bytes*) → PusTc

Deprecated since version v6.0.0rc0: use diagnostic agnostic API if possible

tmtccmd.pus.tc.s3_fsfw_hk.**create_enable_periodic_hk_command**(*sid: bytes*) → PusTc

tmtccmd.pus.tc.s3_fsfw_hk.**create_enable_periodic_hk_command_with_diag**(*diag: bool*, *sid: bytes*) → PusTc

Deprecated since version v6.0.0rc0: use diagnostic agnostic API if possible

tmtccmd.pus.tc.s3_fsfw_hk.**create_enable_periodic_hk_command_with_interval**(*sid: bytes*, *interval_seconds: float*) → Tuple[PusTc, PusTc]

tmtccmd.pus.tc.s3_fsfw_hk.**create_enable_periodic_hk_command_with_interval_with_diag**(*diag: bool*, *sid: bytes*, *interval_seconds: float*) → Tuple[PusTc, PusTc]

Deprecated since version v6.0.0rc0: use diagnostic agnostic API if possible

tmtccmd.pus.tc.s3_fsfw_hk.**create_modify_collection_interval_cmd**(*sid: bytes*, *interval_seconds: float*) → PusTc

tmtccmd.pus.tc.s3_fsfw_hk.**create_modify_collection_interval_cmd_with_diag**(*diag: bool*, *sid: bytes*, *interval_seconds: float*) → PusTc

Deprecated since version v6.0.0rc0: use diagnostic agnostic API if possible

tmtccmd.pus.tc.s3_fsfw_hk.**create_request_one_diag_command**(*sid: bytes*) → PusTc

tmtccmd.pus.tc.s3_fsfw_hk.**create_request_one_hk_command**(*sid: bytes*) → PusTc

tmtccmd.pus.tc.s3_fsfw_hk.**disable_periodic_hk_command**(*diag: bool*, *sid: bytes*) → PusTc

Deprecated since version v4.0.0a2: use create… API instead

tmtccmd.pus.tc.s3_fsfw_hk.**enable_periodic_hk_command**(*diag: bool*, *sid: bytes*) → PusTc

Deprecated since version v4.0.0a2: use create… API instead

tmtccmd.pus.tc.s3_fsfw_hk.**enable_periodic_hk_command_with_interval**(*diag: bool*, *sid: bytes*, *interval_seconds: float*) → Tuple[PusTc, PusTc]

Deprecated since version v4.0.0a2: use create… API instead

tmtccmd.pus.tc.s3_fsfw_hk.**generate_one_diag_command**(*sid: bytes*) → PusTc

Deprecated since version v4.0.0a2: use create… API instead

tmtccmd.pus.tc.s3_fsfw_hk.**generate_one_hk_command**(*sid: bytes*) → PusTc

Deprecated since version v4.0.0a2: use create… API instead

tmtccmd.pus.tc.s3_fsfw_hk.**make_interval**(*interval_seconds: float*) → bytearray

tmtccmd.pus.tc.s3_fsfw_hk.**make_sid**(*object_id: bytes*, *set_id: int*) → bytearray

tmtccmd.pus.tc.s3_fsfw_hk.**modify_collection_interval**(*diag: bool*, *sid: bytes*, *interval_seconds: float*) → PusTc

Deprecated since version v4.0.0a2: use create… API instead

Service 5 Event Module

Contains definitions and functions related to PUS Service 5 Telecommands.

tmtccmd.pus.tc.s5_event.**create_disable_event_reporting_command**(*apid: int = 0*, *seq_count: int = 0*) → PusTc

tmtccmd.pus.tc.s5_event.**create_enable_event_reporting_command**(*apid: int = 0*, *seq_count: int = 0*) → PusTc

tmtccmd.pus.tc.s5_event.**pack_disable_event_reporting_command**(*apid: int = 0*, *seq_count: int = 0*) → PusTc

Deprecated since version v4.0.0a0: use create… API instead

tmtccmd.pus.tc.s5_event.**pack_enable_event_reporting_command**(*apid: int = 0*, *seq_count: int = 0*) → PusTc

Deprecated since version v4.0.0a0: use create… API instead

Service 5 Event Module - sat-rs support

sat-rs specific PUS event support.

class tmtccmd.pus.s5_satrs_event_defs.**EventSeverity**(*value*)

Bases: IntEnum

An enumeration.

HIGH = 3

INFO = 0

LOW = 1

MEDIUM = 2

class tmtccmd.pus.s5_satrs_event_defs.**EventU32**(*severity: 'EventSeverity'*, *group_id: 'int'*, *unique_id: 'int'*)

Bases: object

group_id: int

severity: *EventSeverity*

unique_id: int

classmethod unpack(*data: bytes*) → *EventU32*

Service 5 Event Module - FSFW support

FSFW specific PUS event support

class tmtccmd.pus.s5_fsfw_event_defs.**EventInfo**(*id: int = 0*, *name: str = ''*, *severity: str = ''*, *info: str = ''*, *file_location: str = ''*)

Bases: object

file_location: str = ''

id: int = 0

info: str = ''

name: str = ''

severity: str = ''

class tmtccmd.pus.s5_fsfw_event_defs.**Severity**(*value*)

Bases: IntEnum

An enumeration.

HIGH = 4

INFO = 1

LOW = 2

MEDIUM = 3

tmtccmd.pus.s5_fsfw_event_defs.**str_to_severity**(*string: str*) → *Severity* | None

Contains classes and functions to deserialize PUS Service 5 Telemetry

class tmtccmd.pus.tm.s5_fsfw_event.**EventDefinition**(*event_id: 'int'*, *reporter_id: 'bytes'*, *param1: 'int'*, *param2: 'int'*)

Bases: object

classmethod empty() → *EventDefinition*

event_id: int

classmethod from_bytes(*data: bytes*) → *EventDefinition*

pack() → bytes

param1: int

param2: int

reporter_id: bytes

class tmtccmd.pus.tm.s5_fsfw_event.**Service5Tm**(*apid: int*, *subservice: Subservice*, *event: EventDefinition*, *timestamp: bytes*, *ssc: int = 0*, *packet_version: int = 0*, *space_time_ref: int = 0*, *destination_id: int = 0*)

Bases: AbstractPusTm

property ccsds_version: int

property event_definition: *EventDefinition*

classmethod from_tm(*pus_tm: PusTm*) → *Service5Tm*

pack() → bytearray

property packet_id: PacketId

property packet_seq_control: PacketSeqCtrl

property service: int

property severity: *Severity*

property source_data: bytes

property sp_header: SpacePacketHeader

property subservice: int

property timestamp: bytes

classmethod unpack(*data: bytes*, *timestamp_len: int*) → *Service5Tm*

Service 8 Action Commanding Module

FSFW specific PUS actions support

class tmtccmd.pus.s8_fsfw_action_defs.**CustomSubservice**(*value*)

Bases: IntEnum

An enumeration.

TC_FUNCTIONAL_CMD = 128

TM_DATA_REPLY = 130

Contains classes and functions to handle PUS Service 8 telemetry.

tmtccmd.pus.tc.s8_fsfw_action.**create_action_cmd**(*object_id: bytes*, *action_id: int*, *user_data: bytes = b''*, *apid: int = 0*, *seq_count: int = 0*) → PusTc

tmtccmd.pus.tc.s8_fsfw_action.**make_action_id**(*action_id: int*) → bytearray

tmtccmd.pus.tc.s8_fsfw_action.**make_fsfw_action_cmd**(*object_id: bytes*, *action_id: int*, *user_data: bytes = b''*, *apid: int = 0*, *seq_count: int = 0*) → PusTc

Deprecated since version v4.0.0a2: use create… API instead

Service 11 Telecommand Scheduling Module

class tmtccmd.pus.s11_tc_sched_defs.**Subservice**(*value*)

Bases: IntEnum

Unless specified, TCs and TMs are related to a request ID

TC_DELETE = 5

TC_DELETE_WITH_FILTER = 6

TC_DETAIL_REPORT_FILTER_BASED = 11

TC_DETAIL_REPORT_TIME_BASED = 9

TC_DISABLE = 2

TC_ENABLE = 1

TC_INSERT = 4

TC_RESET = 3

TC_TIMESHIFT = 7

TC_TIMESHIFT_ALL = 15

TC_TIMESHIFT_WITH_FILTER = 8

TM_DETAIL_REPORT_FILTER_BASED = 12

TM_DETAIL_REPORT_TIME_BASED = 10

class tmtccmd.pus.s11_tc_sched_defs.**TcSchedReqId**(*apid: int*, *seq_cnt: int*, *src_id: int*)

Bases: object

classmethod **build_from_tc**(*tc: PusTc*) → *TcSchedReqId*

property **id_u64**

pack() → bytes

class tmtccmd.pus.s11_tc_sched_defs.**TypeOfTimeWindow**(*value*)

Bases: IntEnum

An enumeration.

FROM_TIMETAG = 2

FROM_TIMETAG_TO_TIMETAG = 1

SELECT_ALL = 0

TO_TIMETAG = 3

Service 17 Test Module

class tmtccmd.pus.s17_test_defs.**CustomSubservice**(*value*)

Bases: IntEnum

An enumeration.

TC_GEN_EVENT = 128

Service 20 Parameter Module - FSFW support

class tmtccmd.pus.s20_fsfw_param_defs.**CustomSubservice**(*value*)

Bases: IntEnum

An enumeration.

TC_DUMP = 129

TC_LOAD = 128

TM_DUMP_REPLY = 130

class tmtccmd.pus.s20_fsfw_param_defs.**FsfwParamId**(*object_id: bytes*, *param_id:* ParameterId, *ptc: Ptc*, *pfc: int*, *rows: int*, *columns: int*)

Bases: object

Wrapper for the whole FSFW specific parameter data. It contains the ECSS PTC and PFC numbers and the number of columns and rows in the parameter. See https://ecss.nl/standard/ecss-e-st-70-41c-space-engineering-telemetry-and-telecommand-packet-utilization-15-april-2016/ p.428 for more information.

Parameters

- **ptc** – ECSS PTC number
- **pfc** – ECSS PFC number
- **rows** – Number of rows in parameter (for matrix entries, 1 for vector entries, 1 for scalar entries)
- **columns** – Number of columns in parameter (for matrix or vector entries, 1 for scalar entries)

Returns

Parameter information field as 4 byte bytearray

columns: int

object_id: bytes

pack() → bytearray

Convert the wrapper to the raw byte format expected for PUS TC or PUS TM creation.

param_id: *ParameterId*

pfc: int

ptc: Ptc

rows: int

classmethod unpack(*data: bytes*) → *FsfwParamId*

class tmtccmd.pus.s20_fsfw_param_defs.**Parameter**(*fsfw_param_id: 'FsfwParamId'*, *param_raw: 'bytes'*)

Bases: object

property columns

classmethod empty()

fsfw_param_id: *FsfwParamId*

property object_id

pack() → bytearray

Convert the wrapper to the raw byte format expected for PUS TC or PUS TM creation.

property param_id

param_raw: bytes

parse_scalar_param() → int | float

property pfc

property ptc

property rows

classmethod unpack(*data: bytes*) → *Parameter*

class tmtccmd.pus.s20_fsfw_param_defs.**ParameterId**(*domain_id: 'int'*, *unique_id: 'int'*, *linear_index: 'int'*)

Bases: object

as_u32() → int

domain_id: int

classmethod empty() → *ParameterId*

linear_index: int

pack() → bytes

unique_id: int

classmethod unpack(*data: bytes*) → *ParameterId*

tmtccmd.pus.s20_fsfw_param_defs.**create_matrix_double_parameter**(*object_id: bytes*, *domain_id: int*, *unique_id: int*, *parameters: Sequence[Sequence[float]]*) → *Parameter*

tmtccmd.pus.s20_fsfw_param_defs.**create_matrix_float_parameter**(*object_id: bytes*, *domain_id: int*, *unique_id: int*, *parameters: Sequence[Sequence[float]]*) → *Parameter*

tmtccmd.pus.s20_fsfw_param_defs.**create_scalar_boolean_parameter**(*object_id: bytes*, *domain_id: int*, *unique_id: int*, *parameter: bool*) → *Parameter*

tmtccmd.pus.s20_fsfw_param_defs.**create_scalar_double_parameter**(*object_id: bytes*, *domain_id: int*, *unique_id: int*, *parameter: float*) → *Parameter*

tmtccmd.pus.s20_fsfw_param_defs.**create_scalar_float_parameter**(*object_id: bytes*, *domain_id: int*, *unique_id: int*, *parameter: float*) → *Parameter*

tmtccmd.pus.s20_fsfw_param_defs.**create_scalar_i16_parameter**(*object_id: bytes*, *domain_id: int*, *unique_id: int*, *parameter: int*) → *Parameter*

tmtccmd.pus.s20_fsfw_param_defs.**create_scalar_i32_parameter**(*object_id: bytes*, *domain_id: int*, *unique_id: int*, *parameter: int*) → *Parameter*

tmtccmd.pus.s20_fsfw_param_defs.**create_scalar_i8_parameter**(*object_id: bytes*, *domain_id: int*, *unique_id: int*, *parameter: int*) → *Parameter*

tmtccmd.pus.s20_fsfw_param_defs.**create_scalar_u16_parameter**(*object_id: bytes*, *domain_id: int*, *unique_id: int*, *parameter: int*) → *Parameter*

tmtccmd.pus.s20_fsfw_param_defs.**create_scalar_u32_parameter**(*object_id: bytes*, *domain_id: int*, *unique_id: int*, *parameter: int*) → *Parameter*

tmtccmd.pus.s20_fsfw_param_defs.**create_scalar_u8_parameter**(*object_id: bytes*, *domain_id: int*, *unique_id: int*, *parameter: int*) → *Parameter*

tmtccmd.pus.s20_fsfw_param_defs.**create_vector_double_parameter**(*object_id: bytes*, *domain_id: int*, *unique_id: int*, *parameters: Sequence[float]*) → *Parameter*

tmtccmd.pus.s20_fsfw_param_defs.**create_vector_float_parameter**(*object_id: bytes*, *domain_id: int*, *unique_id: int*, *parameters: Sequence[float]*)

tmtccmd.pus.s20_fsfw_param_defs.**deserialize_scalar_entry**(*ptc: int*, *pfc: int*, *param_data: bytes*) → int | float

Try to deserialize a scalar parameter entry (row = 1 and column = 1).

Raises

- **ValueError** – Passed parameter data length invalid.
- **NotImplementedError** – Parsing of parameter data not implemented for given PTC and PFC.

tmtccmd.pus.s20_fsfw_param_defs.**parse_scalar_param**(*wrapper:* Parameter) → int | float

Contains definitions and functions related to PUS Service 20 Telecommands.

tmtccmd.pus.tc.s20_fsfw_param.**create_dump_param_cmd**(*param_fsfw_id:* FsfwParamId, *apid: int = 0*) → PusTc

tmtccmd.pus.tc.s20_fsfw_param.**create_load_param_cmd**(*parameter:* Parameter, *apid: int = 0*) → PusTc

tmtccmd.pus.tc.s20_fsfw_param.**create_load_param_cmd_from_raw**(*parameter_raw: bytes*, *apid: int = 0*) → PusTc

tmtccmd.pus.tc.s20_fsfw_param.**pack_boolean_parameter_app_data**(*object_id: bytes*, *domain_id: int*, *unique_id: int*, *parameter: bool*) → bytearray | None

Deprecated since version 3.1.0: use create_scalar_boolean_parameter instead

tmtccmd.pus.tc.s20_fsfw_param.**pack_parameter_id**(*domain_id: int*, *unique_id: int*, *linear_index: int*) → bytearray

Packs the Parameter ID (bytearray with 4 bytes) which is part of the service 20 packets. The first byte of the parameter ID is the domain ID, the second byte is a unique ID and the last two bytes are a linear index if a parameter is not loaded from index 0. :param domain_id: One byte domain ID :param unique_id: One byte unique ID :param linear_index: Two byte linear index.

Deprecated since version 4.0.0a2: use ParameterId helper class with pack() instead

tmtccmd.pus.tc.s20_fsfw_param.**pack_scalar_boolean_parameter_app_data**(*object_id: bytes*, *domain_id: int*, *unique_id: int*, *parameter: bool*) → bytearray | None

Tailored towards FSFW applications.

Parameters

- **object_id**
- **domain_id**
- **unique_id**
- **parameter**

Returns
Application data

Deprecated since version 4.0.0a2: Please use create_scalar_boolean_parameter instead

tmtccmd.pus.tc.s20_fsfw_param.**pack_scalar_double_param_app_data**(*object_id: bytes*, *domain_id: int*, *unique_id: int*, *parameter: float*) → bytearray | None

Deprecated since version 4.0.0a2: Please use create_scalar_double_parameter instead

tmtccmd.pus.tc.s20_fsfw_param.**pack_scalar_float_param_app_data**(*object_id: bytes*, *domain_id: int*, *unique_id: int*, *parameter: float*) → bytearray | None

Deprecated since version 4.0.0a2: use create_scalar_float_parameter instead

tmtccmd.pus.tc.s20_fsfw_param.**pack_scalar_u8_parameter_app_data**(*object_id: bytes*, *domain_id: int*, *unique_id: int*, *parameter: int*) → bytearray | None

Tailored towards FSFW applications.

Parameters

- **object_id**
- **domain_id**
- **unique_id**
- **parameter**

Returns
Application data

Deprecated since version 4.0.0a2: Please use create_scalar_u8_parameter instead

tmtccmd.pus.tc.s20_fsfw_param.**pack_type_and_matrix_data**(*ptc: int*, *pfc: int*, *rows: int*, *columns: int*) → bytearray

Deprecated since version 4.0.0a2: use ParamWrapper helper class with pack() instead

tmtccmd.pus.tc.s20_fsfw_param.**prepare_param_packet_header**(*object_id: bytes*, *domain_id: int*, *unique_id: int*, *ptc: Ptc*, *pfc: int*, *rows: int*, *columns: int*, *start_at_idx: int = 0*) → bytearray | None

Deprecated since version 4.0.0a2: use ParamWrapper helper class with pack() instead

class tmtccmd.pus.tm.s20_fsfw_param.**Service20FsfwTm**(*subservice: int*, *source_data: bytes*, *timestamp: bytes*, *apid: int = 0*)

Bases: AbstractPusTm

property ccsds_version: int

classmethod empty() → *Service20FsfwTm*

classmethod from_tm(*tm: PusTm*)

property object_id: bytes

pack() → bytearray

property packet_id: PacketId

property packet_seq_control: PacketSeqCtrl

property service: int

property source_data: bytes

property sp_header: SpacePacketHeader

property subservice: int

property timestamp: bytes

classmethod unpack(*raw_telemetry: bytes*, *timestamp_len: int*) → *Service20FsfwTm*

class tmtccmd.pus.tm.s20_fsfw_param.**Service20ParamDumpWrapper**(*param_tm:* Service20FsfwTm)

Bases: object

property base_tm: PusTm

get_param() → *Parameter*

Tries to build a Parameter from the own raw telemetry data.

Raises

ValueError – Telemetry source data too short.

Service 200 Mode Commanding Module - FSFW support

class tmtccmd.pus.s200_fsfw_mode_defs.**Subservice**(*value*)

Bases: IntEnum

An enumeration.

TC_MODE_ANNOUNCE = 4

TC_MODE_ANNOUNCE_RECURSIVE = 5

TC_MODE_COMMAND = 1

TC_MODE_COMMAND_FORCES = 2

TC_MODE_READ = 3

TM_CANT_REACH_MODE = 7

TM_MODE_REPLY = 6

TM_WRONG_MODE_REPLY = 8

Core components for mode commanding (custom PUS service).

class tmtccmd.pus.tc.s200_fsfw_mode.**Mode**(*value*)

Bases: IntEnum

Standard modes when commanding objects. These mode IDs are reserved by the FSFW, so it is recommended to avoid these numbers for custom modes.

NORMAL = 2

OFF = 0

ON = 1

RAW = 3

tmtccmd.pus.tc.s200_fsfw_mode.**create_announce_mode_command**(*object_id: bytes*) → PusTc

tmtccmd.pus.tc.s200_fsfw_mode.**create_announce_mode_recursive_command**(*object_id: bytes*) → PusTc

tmtccmd.pus.tc.s200_fsfw_mode.**create_mode_command**(*object_id: bytes*, *mode: int | Mode*, *submode: int*) → PusTc

tmtccmd.pus.tc.s200_fsfw_mode.**create_read_mode_command**(*object_id: bytes*) → PusTc

tmtccmd.pus.tc.s200_fsfw_mode.**pack_mode_command**(*object_id: bytes*, *mode: int | Mode*, *submode: int*) → PusTc

Deprecated since version v4.0.0a2: use create… API instead

tmtccmd.pus.tc.s200_fsfw_mode.**pack_mode_data**(*object_id: bytes*, *mode: Mode | int*, *submode: int*) → bytearray

FSFW modes: Mode 0: Off, Mode 1: Mode On, Mode 2: Mode Normal, Mode 3: Mode Raw

Base class for Service 200 mode commanding reply handling.

class tmtccmd.pus.tm.s200_fsfw_mode.**Service200FsfwReader**(*tm: PusTm*)

Bases: object

contains_mode() → bool

is_cant_reach_mode_reply() → bool

class tmtccmd.pus.tm.s200_fsfw_mode.**Service200FsfwTm**(**args*, ***kwargs*)

Bases: *PusTmBase*, *PusTmInfoBase*

Deprecated since version 7.0.0: use Service200FsfwReader instead

append_telemetry_column_headers(*header_list: list*)

Default implementation adds the PUS header content header (confusing, I know) to the list which can then be printed with a simple print() command. To add additional headers, override this method. Any child class should call this function as well if header information is required.

Parameters
header_list – Header content will be appended to this list

Returns

append_telemetry_content(*content_list: list*)

Default implementation adds the PUS header content to the list which can then be printed with a simple print() command. To add additional content, override this method. Any child class should call this function as well if header information is required.

Parameters
content_list – Header content will be appended to this list

Returns

classmethod unpack(*raw_telemetry: bytes*, *time_reader: CcsdsTimeProvider | None*) → *Service200FsfwTm*

Service 201 Health Commanding Module - FSFW support

class tmtccmd.pus.s201_fsfw_health_defs.**Subservice**(*value*)

Bases: IntEnum

An enumeration.

TC_ANNOUNCE_HEALTH = 3

TC_ANNOUNCE_HEALTH_ALL = 4

TC_SET_HEALTH = 1

TM_HEALTH_SET = 2

CFDP Module

Request Submodule

class tmtccmd.cfdp.request.**CfdpRequestBase**(*req_type:* *CfdpRequestType*)

Bases: object

class tmtccmd.cfdp.request.**CfdpRequestWrapper**(*base:* CfdpRequestBase | *None*)

Bases: object

property request: CfdpRequestType

property request_type: CfdpRequestType

to_put_request() → *PutRequestCfgWrapper*

class tmtccmd.cfdp.request.**PutRequestCfgWrapper**(*cfg:* CfdpParams)

Bases: *CfdpRequestBase*

1.5.3 Application Package

Module

Contains core methods called by entry point files to setup and start a tmtccmd application. It also re-exports commonly used classes and functions.

Core Package

Contains core methods called by entry point files to setup and start a tmtccmd application. It also re-exports commonly used classes and functions.

tmtccmd.**create_default_tmtc_backend**(*setup_wrapper:* SetupWrapper, *tm_handler:* TmHandlerBase, *tc_handler:* TcHandlerBase, *init_procedure:* ProcedureWrapper | *None*) → BackendBase

Creates a default TMTC backend instance which can be passed to the tmtccmd runner

Parameters

- **init_procedure**
- **tc_handler**
- **setup_wrapper**
- **tm_handler**

Returns

tmtccmd.**get_lib_logger**() → Logger

Get the library logger. Please note that this logger can be configured by calling *init_logger()*. This might make sense depending on how the library logger is used.

tmtccmd.**init_logger**(*propagate:* *bool* = *False*, *log_level:* *int* = *20*)

Initiate the library internal logger. There are various ways how to use the logging support of tmtccmd in an application.

1. Usage of a custom application logger, possibly a root logger created with `logging.getLogger()`. In that case, this function does not have to be called if the goal is to have the library logs be emitted by the custom logger. It might still make sense to apply the library console logging format to the application logger using `tmtccmd.logging.add_colorlog_console_logger()`.
2. Usage of a custom application logger, but the library logs should not be emitted by that logger. In that case, the propagation can be disabled but this function can be used to still set up the library logger.
3. No usage of logging in the application but the logs of the library should still be emitted. In that case, this function should still be called to set the log level and set up formatting. The propagation flag does not matter and can be left at the default level.

Parameters

- **propagate** – Specifies whether logs are propagated. If the user wants to use an own (root) logger and does not wish the logs of the library to be propagated to that logger, this should be set to False, which is the default.
- **log_level** – Sets the log level of the library logger

tmtccmd.**init_printout**(*use_gui: bool*)

tmtccmd.**setup**(*setup_args:* SetupWrapper)

This function needs to be called first before running the TMTC commander core. The setup arguments encapsulate all required arguments for the TMTC commander.

Parameters

setup_args – Setup arguments

tmtccmd.**setup_backend_def_procedure**(*backend:* CcsdsTmtcBackend, *tmtc_params:* TreeCommandingProcedure)

tmtccmd.**start**(*tmtc_backend: BackendBase*, *hook_obj:* HookBase, *tmtc_frontend:* FrontendBase | *None* = *None*, *app_name: str* = *'TMTC Commander'*)

This is the primary function to run the TMTC commander. Users should call this function to start the TMTC commander. Please note that *setup()* needs to be called before this function. You also need to build a TMTC backend instance and pass it to this call. You can use *create_default_tmtc_backend()* to create a generic backend.

Parameters

- **tmtc_backend** – Custom backend can be passed here. Otherwise, a default backend will be created
- **hook_obj**
- **tmtc_frontend** – Custom frontend can be passed here. Otherwise, a default frontend will be created
- **app_name** – Name of application. Will be displayed in GUI

Raises

RunTimeError – if *setup()* was not called before

Returns

Core Package

Module contents

Submodules

tmtccmd.core.ccsds_backend module

class tmtccmd.core.ccsds_backend.**CcsdsTmtcBackend**(*tc_mode:* TcMode, *tm_mode:* TmMode, *com_if:* ComInterface, *tm_listener:* CcsdsTmListener, *tc_handler:* TcHandlerBase)

Bases: BackendBase

This is the primary class which handles TMTC reception and sending

close_com_if()

Closes the TM listener and the communication interface :return:

property com_if: *ComInterface*

com_if_active()

property com_if_id

property current_procedure: *ProcedureWrapper*

default_operation()

Command handling. This is a convenience function to call the TM and the TC operation and then auto-determine the internal mode with the *mode_to_req()* method.

Raises

NoValidProcedureSet – No valid procedure set to be passed to the feed callback of the TC handler

property inter_cmd_delay

mode_to_req()

This function will convert the internal state of the backend to a backend *request*, which can be used to determine the next operation. These requests can be treated like recommendations. For example, for if both the TC and the TM mode are IDLE, the request will be set to BackendRequest.DELAY_IDLE field.

open_com_if()

Start the backend. Raise RuntimeError on failure

periodic_op(*_args: any | None = None*) → BackendState

Periodic operation. Simply calls the *default_operation()* function. :raises KeyboardInterrupt: Yields info output and then propagates the exception :raises IOError: Yields informative output and propagates exception :

poll_tm()

Poll TM, irrespective of current TM mode

register_keyboard_interrupt_handler()

Register a keyboard interrupt handler which closes the COM interface and prints a small message

property request

start()

property state

property tc_mode

tc_operation()

This function will handle consuming the current TC queue if one is available, or attempting to fetch a new one if it is not. This function will only do something if the *tc_mode* is set to a non IDLE value.

It is necessary to set a valid procedure before calling this by using the `current_proc_info` setter function.

Raises

NoValidProcedureSet – No valid procedure set to be passed to the feed callback of the TC handler

property tm_listener

property tm_mode

tm_operation()

This function will fetch and forward TM data from the current communication interface to the user TM handler. It only does so if the *tm_mode* is set to the LISTENER mode

try_set_com_if(*com_if:* ComInterface) → bool

exception tmtccmd.core.ccsds_backend.**NoValidProcedureSet**

Bases: Exception

tmtccmd.core.base module

class tmtccmd.core.base.**BackendRequest**(*value*)

Bases: IntEnum

These requests can be treated like recommendations on what to do after calling the backend handler functions and the `BackendState.mode_to_req()` function.

Brief explanation of fields:

1. NONE: No special recommendation
2. TERMINATION_NO_ERROR: Will be returned for the One Queue mode after finishing queue handling.
3. DELAY_IDLE: TC and TM mode are idle, so there is nothing to do
4. DELAY_LISTENER: TC handling is not active but TM listening is active. Delay to wait for new TM packets
5. CALL_NEXT: It is recommended to call the handler functions immediately, for example to handle the next entry in the TC queue

CALL_NEXT = 5

DELAY_CUSTOM = 4

DELAY_IDLE = 2

DELAY_LISTENER = 3

NONE = 0

TERMINATION_NO_ERROR = 1

class tmtccmd.core.base.**FrontendBase**

Bases: object

abstract start(*args: Any*)

Start the frontend. :return:

class tmtccmd.core.base.**ModeWrapper**

Bases: object

class tmtccmd.core.base.**TcMode**(*value*)

Bases: IntEnum

An enumeration.

IDLE = 0

MULTI_QUEUE = 2

ONE_QUEUE = 1

class tmtccmd.core.base.**TmMode**(*value*)

Bases: IntEnum

An enumeration.

IDLE = 0

LISTENER = 1

tmtccmd.core.globals_manager module

tmtccmd.core.globals_manager.**get_global**(*global_param_id: int*, *lock: bool = False*)

tmtccmd.core.globals_manager.**lock_global_pool**(*blocking: bool | None = None*, *timeout: float | None = None*) → bool

Lock the global objects. This is important if the values are changed. Don't forget to unlock the pool after finishing work with the globals! :param timeout_seconds: Attempt to lock for this many second. Default value -1 blocks permanently until lock is released. :return: Returns whether lock was locked or not.

tmtccmd.core.globals_manager.**set_lock_timeout**(*timeout: float*)

tmtccmd.core.globals_manager.**unlock_global_pool**()

tmtccmd.core.globals_manager.**update_global**(*global_param_id: int*, *parameter: any*, *lock: bool = False*)

Telemetry Handling Package

Telemetry Handling Package

CCSDS TM Listener Module

Contains the TmListener which can be used to listen to Telemetry in the background

class tmtccmd.tmtc.ccsds_tm_listener.**CcsdsTmListener**(*tm_handler:* CcsdsTmHandler)

Bases: object

Simple helper object which can be used for retrieving and routing CCSDS packets. It can be used to poll CCSDS packets from a provided `tmtccmd.com_if.ComInterface` and then route them using a provided CCSDS TM handler.

operation(*com_if:* ComInterface) → int

Core operation to route packet to the provided handler.

Parameters
com_if

Raises
PacketsTooSmallForCcsds – If any of the received packets are too small. The internal handler will still continue to process the remaining packet list retrieved from the COM interface.

Returns

exception tmtccmd.tmtc.ccsds_tm_listener.**PacketsTooSmallForCcsds**(*packets: List[bytes]*)

Bases: Exception

TM Common Module

class tmtccmd.tmtc.common.**CcsdsTmHandler**(*generic_handler:* GenericApidHandlerBase | *None*)

Bases: *TmHandlerBase*

Generic CCSDS handler class. The user can create an instance of this class to handle CCSDS packets by adding dedicated APID handlers or a generic handler for all APIDs with no dedicated handler

add_apid_handler(*handler:* SpecificApidHandlerBase)

Add a TM handler for a certain APID. The handler is a callback function which will be called if telemetry with that APID arrives.

Parameters
handler – Handler class instance

Returns

handle_packet(*apid: int*, *packet: bytes*) → bool

Handle a packet with an APID. If a handler exists for the given APID, it is used to handle the packet. If not, a dedicated handler for unknown APIDs is called.

Parameters

- **apid**
- **packet**

Returns
True if the packet was passed to as dedicated APID handler, False otherwise

has_apid(*apid: int*) → bool

user_hook(*apid: int*, *packet: bytes*)
Can be overriden to trace all packets received.

class tmtccmd.tmtc.common.**DefaultApidHandler**(*user_args: Any*)
Bases: *GenericApidHandlerBase*

handle_tm(*apid: int*, *_packet: bytes*, *_user_args: Any*)

class tmtccmd.tmtc.common.**GenericApidHandlerBase**(*user_args: Any*)
Bases: ABC

This class is similar to the *SpecificApidHandlerBase* but it is not specific for an APID and the found APID will be passed to the callback

abstract handle_tm(*apid: int*, *_packet: bytes*, *_user_args: Any*)

class tmtccmd.tmtc.common.**SpecificApidHandlerBase**(*apid: int*, *user_args: Any*)
Bases: ABC

Abstract base class for an CCSDS APID specific handler. The user can implement a TM handler by implementing this class and then adding it to the *CcsdsTmHandler*. If a CCSDS space packet with a specific APID is received, it will be routed to this handler using the *handle_tm()* callback function

abstract handle_tm(*_packet: bytes*, *_user_args: Any*)

class tmtccmd.tmtc.common.**TmHandlerBase**(*tm_type:* TmTypes)
Bases: object

get_type()

class tmtccmd.tmtc.common.**TmTypes**(*value*)
Bases: Enum

An enumeration.

CCSDS_SPACE_PACKETS = <class 'enum.auto'>

NONE = <class 'enum.auto'>

TM Base Module (deprecated)

class tmtccmd.tmtc.tm_base.**PusTmBase**(*pus_tm: PusTm*)
Bases: *PusTmInterface*

property apid: int

pack() → bytearray

property service: int

property ssc: int

property subservice: int

property tm_data: bytes

class tmtccmd.tmtc.tm_base.PusTmInfoBase(*pus_tm: PusTm*)

Bases: *PusTmInfoInterface*

append_packet_info(*info: str*)

append_telemetry_column_headers(*header_list: list*)

Default implementation adds the PUS header content header (confusing, I know) to the list which can then be printed with a simple print() command. To add additional headers, override this method. Any child class should call this function as well if header information is required.

Parameters
header_list – Header content will be appended to this list

Returns

append_telemetry_content(*content_list: list*)

Default implementation adds the PUS header content to the list which can then be printed with a simple print() command. To add additional content, override this method. Any child class should call this function as well if header information is required.

Parameters
content_list – Header content will be appended to this list

Returns

get_custom_printout() → str

get_print_info() → str

get_source_data_string(*print_format: PrintFormats = PrintFormats.HEX*) → str

set_custom_printout(*custom_string: str*)

set_packet_info(*print_info: str*)

class tmtccmd.tmtc.tm_base.PusTmInfoInterface

Bases: object

abstract append_packet_info(*info: str*)

abstract append_telemetry_column_headers(*header_list: list*)

abstract append_telemetry_content(*content_list: list*)

abstract get_custom_printout() → str

abstract get_print_info() → str

abstract get_source_data_string() → str

abstract set_packet_info(*print_info: str*)

class tmtccmd.tmtc.tm_base.PusTmInterface

Bases: object

abstract property apid: int

abstract pack() → bytearray

abstract property service: int

abstract property ssc: int

abstract property subservice: int

abstract property tm_data: bytes

abstract property valid: bool

Telecommand Handling Package

Telecommand Handling Package

TC Handler Submodule

class tmtccmd.tmtc.handler.**FeedWrapper**(*queue_wrapper:* QueueWrapper, *auto_dispatch: bool*)

Bases: object

This class wraps the queue and some additional information and useful fields which can be set by the user.

Variables

- **queue_helper** – Can be used to simplify insertion of queue entries like telecommands into the queue
- **dispatch_next_queue** – Can be used to prevent the dispatch of the queue
- **modes** – Currently contains the current TC Mode and TM mode of the calling handler class

class tmtccmd.tmtc.handler.**SendCbParams**(*info:* ProcedureWrapper, *entry:* QueueEntryHelper, *com_if:* ComInterface)

Bases: object

Wrapper for all important parameters passed to the TC send callback.

Variables

- **info** – Procedure info about the procedure this queue entry is related too
- **entry** – Queue entry base type. The user can cast this back to the concrete type or just use duck typing if the concrete type is known
- **com_if** – Communication interface. Will generally be used to send the packet, using the `tmtccmd.com_if.ComInterface.send()` method

class tmtccmd.tmtc.handler.**TcHandlerBase**

Bases: ABC

Generic abstract class for a TC handler object. Should be implemented by the user. This object then takes care of sending packets by providing the *send_cb()* send-callback. It also provides telecommand queues by providing the *feed_cb()* queue feeder callback.

abstract feed_cb(*info:* ProcedureWrapper, *wrapper:* FeedWrapper)

This function will be called to retrieve a telecommand queue from the user code, based on a procedure. The passed feed wrapper can be used to set the TC queue or other parameter like the inter-packet delay.

Parameters

- **info** – Generic base class for a procedure. For example, the py:class:*tmtccmd.tmtc.DefaultProcedureInfo* class uses a service string and op code string which can be used in the user code to select between different telecommand queues being packed
- **wrapper** – Wrapper type around the queue. It also contains a queue helper class to simplify adding entries to the telecommand queue

Returns

abstract queue_finished_cb(*info:* ProcedureWrapper)

abstract send_cb(*send_params:* SendCbParams)

This function callback will be called for each queue entry. This also includes miscellaneous queue entries, for example the ones used to log additional information. It is up to the user code implementation to determine the concrete queue entry and what to do with it.

In general, an implementation will perform the following steps:

1. Determine the queue entry and what to do with it
2. If applicable, retrieve the raw data to send from the queue entry and send it using the generic communication interface

All delay related entries will generally be handled by the send queue consumer so there is no need to manually delay the application in this callback. However, the queue consumer will not handle log entries so the user needs to take care of handling these entries and log the content to a console, file logger or any other system used to log something.

Parameters

send_params

TC Queue Submodule

class tmtccmd.tmtc.queue.**DefaultPusQueueHelper**(*queue_wrapper:* QueueWrapper, *tc_sched_timestamp_len: int*, *seq_cnt_provider: ProvidesSeqCount | None*, *pus_verificator: PusVerificator | None*, *default_pus_apid: int | None*)

Bases: *QueueHelperBase*

Default PUS Queue Helper which simplifies inserting PUS telecommands into the queue. It also provides a way to optionally stamp common PUS TC fields which would otherwise add boilerplate code during the packet creation process. This includes the following packet properties and it is also able to add the telecommand into a provided PUS verificator.

This queue helper also has special support for PUS 11 time tagged PUS telecommands and will perform its core functionality for the time-tagged telecommands as well.

add_ccsds_tc(*space_packet: SpacePacket*)

add_pus_tc(*pus_tc: PusTc*)

pre_add_cb(*entry:* TcQueueEntryBase)

class tmtccmd.tmtc.queue.**LogQueueEntry**(*log_str: str*)

Bases: *TcQueueEntryBase*

class tmtccmd.tmtc.queue.**PacketDelayEntry**(*delay_time: timedelta*)

Bases: *TcQueueEntryBase*

classmethod from_millis(*millis: int*) → *PacketDelayEntry*

class tmtccmd.tmtc.queue.**PusTcEntry**(*pus_tc: PusTc*)

Bases: *TcQueueEntryBase*

class tmtccmd.tmtc.queue.**QueueEntryHelper**(*base:* TcQueueEntryBase | *None*)

Bases: object

property entry_type: *TcQueueEntryType*

property is_tc: bool

to_log_entry() → *LogQueueEntry*

to_packet_delay_entry() → *PacketDelayEntry*

to_pus_tc_entry() → *PusTcEntry*

to_raw_tc_entry() → *RawTcEntry*

to_space_packet_entry() → *SpacePacketEntry*

to_wait_entry() → *WaitEntry*

class tmtccmd.tmtc.queue.**QueueHelperBase**(*queue_wrapper:* QueueWrapper)

Bases: ABC

add_log_cmd(*print_str: str*)

add_packet_delay(*delay: timedelta*)

add_packet_delay_ms(*delay_ms: int*)

add_raw_tc(*tc: bytes*)

add_wait(*wait_time: timedelta*)

add_wait_ms(*wait_ms: int*)

add_wait_seconds(*wait_seconds: float*)

empty() → bool

abstract pre_add_cb(*entry:* TcQueueEntryBase)

class tmtccmd.tmtc.queue.**QueueWrapper**(*info:* TcProcedureBase, *queue: Deque[*TcQueueEntryBase*]*, *inter_cmd_delay: timedelta = datetime.timedelta(0)*)

Bases: object

classmethod empty()

class tmtccmd.tmtc.queue.**RawTcEntry**(*tc: bytes*)

Bases: *TcQueueEntryBase*

class tmtccmd.tmtc.queue.**SpacePacketEntry**(*space_packet: SpacePacket*)

Bases: *TcQueueEntryBase*

class tmtccmd.tmtc.queue.**TcQueueEntryBase**(*etype:* TcQueueEntryType)

Bases: object

Generic TC queue entry abstraction. This allows filling the TC queue with custom objects

is_tc() → bool

Check whether concrete object is an actual telecommand

class tmtccmd.tmtc.queue.**TcQueueEntryType**(*value*)

Bases: Enum

An enumeration.

CCSDS_TC = 'ccsds-tc'

CUSTOM = 'custom'

LOG = 'log'

PACKET_DELAY = 'set-delay'

PUS_TC = 'pus-tc'

RAW_TC = 'raw-tc'

WAIT = 'wait'

class tmtccmd.tmtc.queue.**WaitEntry**(*wait_time: timedelta*)

Bases: *TcQueueEntryBase*

classmethod from_millis(*millis: int*) → *WaitEntry*

TC Procedure Submodule

class tmtccmd.tmtc.procedure.**CfdpProcedure**

Bases: *TcProcedureBase*

property cfdp_request_type

class tmtccmd.tmtc.procedure.**CustomProcedureInfo**(*procedure: Any*)

Bases: *TcProcedureBase*

class tmtccmd.tmtc.procedure.**ProcedureWrapper**(*procedure:* TcProcedureBase | *None*)

Bases: object

Procedure helper class. It wraps the concrete procedure object but allows easily casting it to concrete types supported by the framework.

property proc_type

to_cfdp_procedure() → *CfdpProcedure*

to_custom_procedure() → *CustomProcedureInfo*

to_tree_commanding_procedure() → *TreeCommandingProcedure*

class tmtccmd.tmtc.procedure.**TcProcedureBase**(*ptype:* TcProcedureType)

Bases: object

class tmtccmd.tmtc.procedure.**TcProcedureType**(*value*)

Bases: Enum

An enumeration.

CFDP = 1

CUSTOM = 2

TREE_COMMANDING = 0

class tmtccmd.tmtc.procedure.**TreeCommandingProcedure**(*cmd_path: str | None*)

Bases: *TcProcedureBase*

Generic abstraction for procedures. A procedure can be a single command or a sequence of commands. Generally, one procedure is mapped to a specific TC queue which is packed during run-time

classmethod empty()

Sequential CCSDS Sender Submodule

Used to send multiple TCs in sequence

class tmtccmd.tmtc.ccsds_seq_sender.**SenderMode**(*value*)

Bases: IntEnum

An enumeration.

BUSY = 0

DONE = 1

class tmtccmd.tmtc.ccsds_seq_sender.**SeqResultWrapper**(*mode:* SenderMode)

Bases: object

class tmtccmd.tmtc.ccsds_seq_sender.**SequentialCcsdsSender**(*queue_wrapper:* QueueWrapper, *tc_handler:* TcHandlerBase)

Bases: object

Specific implementation of CommandSenderReceiver to send multiple telecommands in sequence

handle_new_queue_forced(*queue_wrapper:* QueueWrapper)

handle_non_tc_entry(*queue_entry:* TcQueueEntryBase) → bool

Checks whether the entry in the pus_tc queue is a telecommand. :param queue_entry: Generic queue entry :return: True if queue entry is telecommand, False if it is not

property mode

no_delay_remaining() → bool

operation(*com_if:* ComInterface) → *SeqResultWrapper*

Primary function which should be called periodically to consume a TC queue.

Parameters

com_if – Communication interface used to send telecommands. Will be passed to the user send function

property queue_wrapper

resume()

Can be used to resume a finished sequential sender it the provided queue is not empty anymore

1.5.4 Application Configuration Package

Configuration helpers and definitions.

Submodules:

- *tmtccmd.config.hook* - Base hook class which should be implemented by user and is used by the framework to retrieve certain configuration from the user.
- *tmtccmd.config.args* - Various helper methods and classes to create the argument parsers and arguments converts to create the data structures expected by this library from passed CLI arguments.

Configuration Package

Module

Configuration helpers and definitions.

Submodules:

- *tmtccmd.config.hook* - Base hook class which should be implemented by user and is used by the framework to retrieve certain configuration from the user.
- *tmtccmd.config.args* - Various helper methods and classes to create the argument parsers and arguments converts to create the data structures expected by this library from passed CLI arguments.

class tmtccmd.config.**SetupWrapper**(*hook_obj:* HookBase, *setup_params:* SetupParams, *proc_param_wrapper:* ProcedureParamsWrapper, *json_cfg_path: str | None = None*)

Bases: object

This class encapsulates various important setup parameters required by tmtccmd components

property params

tmtccmd.config.**backend_mode_conversion**(*mode: str*, *mode_wrapper:* ModeWrapper)

tmtccmd.config.**cfdp_put_req_params_to_procedure**(*params:* CfdpParams) → *CfdpProcedure*

tmtccmd.config.**params_to_procedure_conversion**(*param_wrapper:* ProcedureParamsWrapper) → *ProcedureWrapper*

tmtccmd.config.**tmtc_params_to_procedure**(*params:* TreeCommandingParams) → *TreeCommandingProcedure*

Configuration Hook Submodule

class tmtccmd.config.hook.**HookBase**(*cfg_file_path: str | None = None*)

Bases: ABC

This hook allows users to adapt the TMTC commander core to the unique mission requirements. It is used by implementing all abstract functions and then passing the instance to the TMTC commander core.

assign_communication_interface(*com_if_key: str*) → *ComInterface* | None

Assign the communication interface used by the TMTC commander to send and receive TMTC with.

Parameters

com_if_key – String key of the communication interface to be created.

Deprecated since version 8.0.0rc0: implement and use get_communication_interface instead

get_cmd_history() → History | None

Optionlly return a history class for the past command paths which will be used when prompting a command path from the user in CLI mode.

get_com_if_dict() → Mapping[str, Tuple[str, Any]]

abstract get_command_definitions() → *CmdTreeNode*

This function should return the root node of the command definition tree.

abstract get_communication_interface(*com_if_key: str*) → *ComInterface* | None

get_object_ids() → Mapping[bytes, *ComponentIdBase*]

Deprecated since version 8.0.0: application specific code

get_tmtc_definitions() → *TmtcDefinitionWrapper*

This is a dicitonary mapping services represented by strings to an operation code dictionary.

Returns

Deprecated since version 8.0.0rc0: implement and use get_command_definitions instead

perform_mode_operation(*tmtc_backend: BackendBase*, *mode: int*)

Perform custom mode operations.

Parameters

- **tmtc_backend**
- **mode**

Returns

Argument Parsing Submodule

Argument parser module.

class tmtccmd.config.args.**AppParams**(*use_gui: 'bool' = False*, *reduced_printout: 'bool' = False*, *use_ansi_colors: 'bool' = True*, *compl_style: 'CompleteStyle' = <CompleteStyle.READLINE_LIKE: 'READLINE_LIKE'>*)

Bases: object

compl_style: CompleteStyle = 'READLINE_LIKE'

reduced_printout: bool = False

use_ansi_colors: bool = True

use_gui: bool = False

class tmtccmd.config.args.**BackendParams**(*mode: 'str' = ''*, *com_if_id: 'str' = ''*, *listener: 'bool' = False*, *interactive: 'bool' = False*)

Bases: object

com_if_id: str = ''

interactive: bool = False

listener: bool = False

mode: str = ''

class tmtccmd.config.args.**CommandingParams**(*delay: 'float' = 0.0*, *apid: 'int' = 0*, *print_tree: 'bool' = False*, *tree_print_with_description: 'bool' = True*, *tree_print_max_depth: 'Optional[int]' = None*)

Bases: object

apid: int = 0

delay: float = 0.0

print_tree: bool = False

tree_print_max_depth: int | None = None

tree_print_with_description: bool = True

class tmtccmd.config.args.**PostArgsParsingWrapper**(*args_raw: Namespace*, *unknown_args: List[str]*, *params:* SetupParams, *hook_obj:* HookBase)

Bases: object

This helper class helps with the internalization of the parse arguments into the format expected by tmtccmd.

Variables

assign_com_if – If this is set to True (default), the wrapper will try to create a *tmtccmd.com.ComInterface* on the conversion methods.

request_type_from_args() → *TcProcedureType*

set_cfdp_params_with_prompts(*cfdp_params:* CfdpParams)

set_cfdp_params_without_prompts(*cfdp_params:* CfdpParams)

set_params_with_prompts(*proc_base:* ProcedureParamsWrapper)

set_params_without_prompts(*proc_wrapper:* ProcedureParamsWrapper)

set_tmtc_params_with_prompts(*tmtc_params:* TreeCommandingParams)

set_tmtc_params_without_prompts(*tmtc_params:* TreeCommandingParams)

Set up the parameter object from the parsed arguments. This call auto-determines whether prompts should be used depending on whether the GUI flag was passed or not.

Raises

Value Error – Parse function call missing

property use_gui

This only yields valid values if parse() was called once

class tmtccmd.config.args.**PreArgsParsingWrapper**

Bases: object

This class can be used to simplify parsing all tmtccmd CLI arguments.

It wraps a parent parser and an argument parser but is also able to create default parsers. The *parse()* method can be used to convert this parse the CLI arguments and then create a *PostArgsParsingWrapper* to process these arguments.

Please note that the parent parser and argument parser field have to be set or created first after creating this wrapper. They can be created by calling *create_default_parent_parser()* and the *create_default_parser()*, but the second function requires the parent parser to be set or created first.

add_cfdp_args()

Add the default CFDP procedure parameters to the default parser.

add_def_proc_and_cfdp_as_subparsers() → Tuple[ArgumentParser, ArgumentParser]

Add the default tmtc and cfdp procedure as subparsers.

add_def_proc_args()

Add the default tmtc procedure parameters to the default parser. This includes the service and operation code flags.

create_default_parent_parser()

Create a default parent parser, which contains common flags for all possible tmtccmd submodules. For example, both the cfdp and default tmtc submodule could contain these common flags. The user can extend or modify the parent parser after it was created.

create_default_parser()

Create the default parser. Requires a valid parent parser containing common flags, The user can create or modify the parser after it was created. This function requires a valid parent parser to be set or created via *create_default_parent_parser()*.

parse(*hook_obj:* HookBase, *setup_params:* SetupParams) → *PostArgsParsingWrapper*

Parses the set parser by calling the `argparse.ArgumentParser.parse_known_args()` method internally and returns the *PostArgsParsingWrapper* to simplify processing the arguments.

class tmtccmd.config.args.**ProcedureParamsWrapper**

Bases: `object`

cfdp_params() → *CfdpParams* | None

property ptype

set_params(*params:* TreeCommandingParams | CfdpParams)

tree_commanding_params() → *TreeCommandingParams* | None

class tmtccmd.config.args.**SetupParams**(*com_if:* ComInterface | *None* = *None*, *cmd_params:* CommandingParams | *None* = *None*, *backend_params:* BackendParams | *None* = *None*, *app_params:* AppParams | *None* = *None*)

Bases: `object`

property apid

property com_if_id

property mode

property use_gui

tmtccmd.config.args.**add_cfdp_procedure_arguments**(*parser_or_subparser: ArgumentParser*)

TODO: Could be extended to support the various types of CFDP user primitives. Right now, the first thing to be implemented will be the put request

tmtccmd.config.args.**add_default_com_if_arguments**(*arg_parser: ArgumentParser*)

tmtccmd.config.args.**add_default_tmtccmd_args**(*parser: ArgumentParser*)

tmtccmd.config.args.**add_ethernet_arguments**(*arg_parser: ArgumentParser*)

tmtccmd.config.args.**add_generic_arguments**(*arg_parser: ArgumentParser*)

tmtccmd.config.args.**add_tmtc_listener_arg**(*arg_parser: ArgumentParser*)

tmtccmd.config.args.**add_tmtc_mode_arguments**(*arg_parser: ArgumentParser*)

tmtccmd.config.args.**add_tree_commanding_arguments**(*parser_or_subparser: ArgumentParser*)

tmtccmd.config.args.**args_to_all_params_for_cfdp**(*pargs: Namespace*, *params:* SetupParams, *cfdp_params:* CfdpParams, *hook_obj:* HookBase, *use_prompts: bool*, *assign_com_if: bool*)

Helper function to convert CFDP command line arguments to the setup parameters.

tmtccmd.config.args.**args_to_all_params_tmtc**(*pargs: Namespace*, *params:* SetupParams, *def_tmtc_params:* TreeCommandingParams, *hook_obj:* HookBase, *use_prompts: bool*, *assign_com_if: bool = True*)

This function converts command line arguments to the internalized setup parameters.

It is recommended to use the `PostArgsParsingHelper` class to do this instead of calling this function directly.

If some arguments are unspecified, they are set here with (variable) default values.

Parameters

- **pargs** – Parsed arguments from calling parse method.
- **params** – Setup parameter object which will be set by this function.
- **hook_obj**
- **def_tmtc_params**
- **use_prompts** – Specify whether terminal prompts are allowed to retrieve unspecified arguments. For something like a GUI, it might make sense to disable this
- **assign_com_if** – Specifies whether this function should try to determine the COM interface from the specified key.

Returns

None

tmtccmd.config.args.**args_to_params_generic**(*pargs: Namespace*, *params:* SetupParams, *hook_obj:* HookBase, *use_prompts: bool*, *assign_com_if: bool*)

tmtccmd.config.args.**cfdp_args_to_cfdp_params**(*pargs: Namespace*, *cfdp_params:* CfdpParams)

Convert the argument parser CFDP arguments provided by this library to the internalized *tmtccmd.config.defs.CfdpParams* dataclass.

tmtccmd.config.args.**create_default_args_parser**(*parent_parser: ArgumentParser*, *descript_txt: str | None = None*) → ArgumentParser

tmtccmd.config.args.**determine_cmd_path**(*params:* SetupParams, *def_params:* TreeCommandingParams, *hook_obj:* HookBase, *pargs: Namespace*, *use_prompts: bool*)

tmtccmd.config.args.**get_default_descript_txt**() → str

tmtccmd.config.args.**parse_default_tmtccmd_input_arguments**(*args: Sequence[str]*, *parser: ArgumentParser*, *print_known_args: bool = False*, *print_unknown_args: bool = False*) → Tuple[Namespace, List[str]]

Parses all input arguments by calling `argparse.ArgumentParser.parse_known_args()`. It is recommended to use the *PreArgsParsingWrapper* instead of using this function directly.

Parameters

- **args** – The actual full list of parse CLI arguments
- **parser** – The parser to be used.
- **print_known_args** – Debugging function to print all known arguments.
- **print_unknown_args**

Returns

Input arguments contained in a special namespace and accessable by args.<variable>

tmtccmd.config.args.**perform_tree_printout**(*cmd_params: CommandingParams*, *cmd_def_tree: CmdTreeNode*)

TMTC Configuration Submodule

class tmtccmd.config.tmtc.**CmdTreeNode**(*name: str*, *description: str*, *parent: CmdTreeNode | None = None*, *hide_children_for_print: bool = False*, *hide_children_which_are_leaves: bool = False*)

Bases: `object`

The command tree node is the primary data structure used to specify the command structure in a way it can be used by framework components.

The node class provides an API which allows to build a tree of command nodes. Generally, a full path from the root node to a leaf will be the command identifier or command path for executing a certain command or procedure.

You can create the root node using the *CmdTreeNode.root_node()* class method. After that children can be appended to the nodes using the *CmdTreeNode.add_child()* method.

You can use square bracket operator to access the children of a node directly. For example, if a node with the name `test_node` has the child `event`, you could use `test_node["event"]` to access the child node.

add_child(*child: CmdTreeNode*)

Add a child to the node. This will also assign the parent class of the child to the current node.

contains_path(*path: str*) → bool

Check whether a full slash separated command path is contained within the command tree.

contains_path_from_node_list(*node_name_list: List[str]*) → bool

Check whether the given list of nodes are contained within the command tree.

extract_subnode(*path: str*) → *CmdTreeNode* | None

Extract a subnode given a relative path.

extract_subnode_by_node_list(*node_list: List[str]*) → *CmdTreeNode* | None

Extract a subnode given a list which would form a relative path if it were joined using slashes.

is_leaf() → bool

A leaf is a node which has no children.

property name_dict: Dict[str, Dict[str, Any] | None]

Returns a nested dictionary where the key is always the name of the node, and the value is one nested name dictionary for each child node.

classmethod root_node() → *CmdTreeNode*

str_for_tree(*with_description: bool*, *max_depth: int | None = None*, *show_hidden_elements: bool = False*) → str

Retrieve the a human readable printout of the tree.

Parameters

- **with_description** – Display descriptions right to the tree.
- **max_depth** – Entries will be cut-off at the specified depth. None can be specified to print all depths.
- **show_hidden_elements** – Overrides the hide argument of command tree nodes.

class tmtccmd.config.tmtc.**DepthInfo**(*depth: int*, *last_child: bool*, *max_depth: int | None = None*, *layer_is_last_set: Set[int] | None = None*)

Bases: object

clear_layer_is_last_child(*depth: int*)

is_layer_for_last_child(*depth: int*) → bool

set_layer_is_last_child(*depth: int*)

class tmtccmd.config.tmtc.**OpCodeEntry**(**args*, ***kwargs*)

Bases: object

Deprecated since version 8.0.0: use the new command definition tree instead

add(*keys: str | ~typing.List[str], info: str, options: ~tmtccmd.config.tmtc.OpCodeOptionBase = <tmtccmd.config.tmtc.OpCodeOptionBase object>*)

info(*op_code: str*) → str | None

property op_code_dict_num_keys

property op_code_dict_str_keys

sort_num_key_dict()

sort_text_key_dict()

class tmtccmd.config.tmtc.**OpCodeOptionBase**(**args*, ***kwargs*)

Bases: object

Deprecated since version 8.0.0: use the new command definition tree instead

class tmtccmd.config.tmtc.**TmtcDefinitionWrapper**(**args*, ***kwargs*)

Bases: object

Deprecated since version 8.0.0: use the new command definition tree instead

add_service(*name: str*, *info: str*, *op_code_entry:* OpCodeEntry)

op_code_entry(*service_name: str*) → *OpCodeEntry* | None

sort()

class tmtccmd.config.tmtc.**TreePart**(*value*)

Bases: Enum

An enumeration.

BLANK = ' '

CORNER = '└──'

EDGE = '├──'

LINE = '│ '

tmtccmd.config.tmtc.**call_all_definitions_providers**(*defs: TmtcDefinitionWrapper*, **args*, ***kwargs*)

Deprecated since version 8.0.0: use the new command definition tree instead

tmtccmd.config.tmtc.**tmtc_definitions_provider**(*adder_func*)

Function decorator which registers the decorated function to be a TMTC definition provider. The `execute_tmtc_def_functions()` function can be used to call all functions.

It is expected that the decorated function takes the py:class:*TmTcDefWrapper* as the first argument. The user can pass any additional arguments as positional and keyword arguments.

Deprecated since version 8.0.0: use the new command definition tree instead

CFDP Configuration Submodule

tmtccmd.config.cfdp.**cfdp_req_to_put_req_get_req**(*params: CfdpParams*, *local_id: UnsignedByteField*, *remote_id: UnsignedByteField*) → PutRequest | None

This function converts the internalized CFDP parameters to the get request variant of the `tmtccmd.cfdp.request.PutRequest` class. Please note that the local ID refers to the receiver of the target of the file copy operation for a get request while the remote ID refers to the sender component for the file copy operation.

tmtccmd.config.cfdp.**cfdp_req_to_put_req_proxy_put_req**(*params: CfdpParams*, *dest_id_put_request: UnsignedByteField*, *dest_id_proxy_put_req: UnsignedByteField*) → PutRequest | None

Generic function to convert the internalized CFDP parameters to a proxy put request.

Parameters

- **params** – CFDP parameters
- **dest_id_put_request** – Recipient of the put request.
- **dest_id_proxy_put_req** – Recipient of the proxy put operation. For a get request, this should the the ID of the sender.

tmtccmd.config.cfdp.**cfdp_req_to_put_req_regular**(*params: CfdpParams*, *dest_id: UnsignedByteField*) → PutRequest | None

tmtccmd.config.cfdp.**generic_cfdp_params_to_put_request**(*params: CfdpParams*, *local_id: UnsignedByteField*, *remote_id: UnsignedByteField*, *dest_id_proxy_put_req: UnsignedByteField*) → PutRequest | None

Please note that this function currently only has the following functionality. It might be extended in the future to have more functionality, or be converted to a factory class.

1. Create a regular put request for a file copy operation.
2. Create a proxy put request.

Communication Configuration Submodule

class tmtccmd.config.com.**ComCfgBase**(*com_if_key: str*, *json_cfg_path: str*)

Bases: object

class tmtccmd.config.com.**SerialCfgWrapper**(*com_if_key: str*, *json_cfg_path: str*, *serial_cfg:* SerialCfg)

Bases: *ComCfgBase*

class tmtccmd.config.com.**TcpipCfg**(*if_type:* TcpIpType, *com_if_key: str*, *json_cfg_path: str*, *send_addr:* EthAddr, *space_packet_ids: Sequence[PacketId]* | *None*, *recv_addr:* EthAddr | *None* = *None*)

Bases: *ComCfgBase*

tmtccmd.config.com.**create_com_interface_cfg_default**(*com_if_key: str*, *json_cfg_path: str*, *space_packet_ids: Sequence[PacketId]* | *None*) → *ComCfgBase* | None

tmtccmd.config.com.**create_com_interface_default**(*cfg:* ComCfgBase) → *ComInterface* | None

Return the desired communication interface object

Parameters
cfg – Generic configuration

Returns

tmtccmd.config.com.**create_default_serial_interface**(*com_if_key: str*, *json_cfg_path: str*, *serial_cfg:* SerialCfg) → *ComInterface* | None

Create a default serial interface. Requires a certain set of global variables set up. See `set_up_serial_cfg()` for more details.

Parameters

- **com_if_key**
- **json_cfg_path**
- **serial_cfg** – Generic serial configuration parameters

Returns

tmtccmd.config.com.**create_default_tcpip_interface**(*tcpip_cfg:* TcpipCfg) → *ComInterface* | None

Create a default serial interface. Requires a certain set of global variables set up. See *default_tcpip_cfg_setup()* for more details.

Parameters
tcpip_cfg – Configuration parameters

Returns

tmtccmd.config.com.**default_serial_cfg_baud_and_port_setup**(*json_cfg_path: str*, *cfg:* SerialCfg)

Default setup for serial interfaces.

Parameters

- **json_cfg_path**
- **cfg** – The baud and serial port parameter will be set in this dataclass

Returns

tmtccmd.config.com.**default_tcpip_cfg_setup**(*com_if_key: str*, *tcpip_type:* TcpIpType, *json_cfg_path: str*, *space_packet_ids: Sequence[PacketId] | None*) → *TcpipCfg*

Default setup for TCP/IP communication interfaces. This intantiates all required data in the globals manager so a TCP/IP communication interface can be built with *create_default_tcpip_interface()*

Parameters

- **com_if_key**
- **tcpip_type**
- **json_cfg_path**
- **space_packet_ids** – Required if the TCP com interface needs to parse space packets

Returns

Configuration Definitions Submodule

class tmtccmd.config.defs.**CfdpParams**(*source_file: str = ''*, *dest_file: str = ''*, *closure_requested: bool = False*, *transmission_mode: TransmissionMode = TransmissionMode.UNACKNOWLEDGED*, *proxy_op: bool = False*)

Bases: object

Simplified dataclass to model the most important CFDP parameters. This can for example be used to internalize CFDP CLI or GUI options.

closure_requested: bool = False

dest_file: str = ''

proxy_op: bool = False

source_file: str = ''

transmission_mode: TransmissionMode = 1

class tmtccmd.config.defs.**CoreComInterfaces**(*value*)

Bases: str, Enum

An enumeration.

DUMMY = 'dummy'

SERIAL_COBS = 'serial_cobs'

SERIAL_DLE = 'serial_dle'

SERIAL_QEMU = 'serial_qemu'

TCP = 'tcp'

UDP = 'udp'

UNSPECIFIED = 'unspec'

class tmtccmd.config.defs.**CoreModeConverter**

Bases: object

static get_str(*mode:* CoreModeList | *int*) → str

class tmtccmd.config.defs.**CoreModeList**(*value*)

Bases: IntEnum

These are the core modes which will be translated to different TC and TM modes for the CCSDS backend

1. ONE_QUEUE_MODE: This mode is optimized to handle one queue. It will configure the backend to request program termination upon finishing the queue handling. This is also the appropriate solution for single commands where the queue only consists of one telecommand.
2. LISTENER_MODE: Only listen to TM
3. MULTI_INTERACTIVE_QUEUE_MODE:

IDLE = 5

LISTENER_MODE = 1

MULTI_INTERACTIVE_QUEUE_MODE = 3

ONE_QUEUE_MODE = 0

class tmtccmd.config.defs.**TreeCommandingParams**(*cmd_path: str* | *None*)

Bases: object

cmd_path: str | None

tmtccmd.config.defs.**default_json_path**() → str

tmtccmd.config.defs.**default_toml_path**() → str

Objects Submodule

tmtccmd.config.objects.**get_base_component_id_mapping**() → Mapping[bytes, *ComponentIdBase*]

These are the object IDs for the tmtccmd core. The core will usually take care of inserting these into the object manager during the program initialization.

:return Dictionary of the core object IDs

tmtccmd.config.objects.**get_core_object_ids**() → Mapping[bytes, *ComponentIdBase*]

1.5.5 Miscellaneous Modules

Logging Module

Logging Package

Submodules

tmtccmd.logging.pus

class tmtccmd.logging.pus.**RawTmtcLogBase**(*logger: Logger*, *log_repr: bool = True*, *log_raw_repr: bool = True*)

Bases: object

log_bytes_readable(*prefix: str*, *packet: bytes*)

log_bytes_repr(*prefix: str*, *packet: bytes*)

log_repr(*prefix: str*, *packet: PusTc | PusTm*)

log_tc(*packet: PusTc*)

Default log function which logs the Python packet representation and raw bytes

log_tm(*packet: PusTm*)

Default log function which logs the Python packet representation and raw bytes

static tc_prefix(*packet: PusTc*, *counter: int*)

static tm_prefix(*packet: PusTm*, *counter: int*)

class tmtccmd.logging.pus.**RawTmtcRotatingLogWrapper**(*max_bytes: int*, *backup_count: int*, *logger: Logger | None = None*, *file_name: Path = PosixPath('log/tmtccmd_raw_pus')*, *suffix: str | None = '2024-05-16.log'*)

Bases: *RawTmtcLogBase*

class tmtccmd.logging.pus.**RawTmtcTimedLogWrapper**(*when: TimedLogWhen*, *interval: int*, *logger: Logger | None = None*, *file_name: Path = PosixPath('log/tmtccmd_raw_pus')*, *suffix: str | None = '2024-05-16.log'*)

Bases: *RawTmtcLogBase*

class tmtccmd.logging.pus.**RegularTmtcLogWrapper**(*file_name: Path | None = None*, *logger: Logger | None = None*)

Bases: object

classmethod get_current_tmtc_file_name() → Path

class tmtccmd.logging.pus.**TimedLogWhen**(*value*)

Bases: Enum

An enumeration.

PER_DAY = 'D'

PER_HOUR = 'h'

PER_MINUTE = 'M'

PER_SECOND = 'S'

tmtccmd.logging.pus.**date_suffix**() → str

Version Module

Version Module

tmtccmd.version.**get_version**() → str

Retrieve the package version using the importlib.metadata API.

Other Modules

Utility Package

Submodules

JSON Module

class tmtccmd.util.json.**JsonKeyNames**(*value*)

Bases: Enum

An enumeration.

COM_IF = 'com_if'

SERIAL_BAUDRATE = 'serial_baudrate'

SERIAL_HINT = 'serial_hint'

SERIAL_PORT = 'serial_port'

TCPIP_TCP_DEST_IP_ADDRESS = 'tcpip_tcp_ip_addr'

TCPIP_TCP_DEST_PORT = 'tcpip_tcp_port'

TCPIP_TCP_RECV_MAX_SIZE = 'tcpip_tcp_recv_max_size'

TCPIP_UDP_DEST_IP_ADDRESS = 'tcpip_udp_ip_addr'

TCPIP_UDP_DEST_PORT = 'tcpip_udp_port'

TCPIP_UDP_RECV_IP_ADDRESS = 'tcpip_udp_recv_addr'

TCPIP_UDP_RECV_MAX_SIZE = 'tcpip_udp_recv_max_size'

TCPIP_UDP_RECV_PORT = 'tcpip_udp_recv_port'

tmtccmd.util.json.**check_json_file**(*json_cfg_path: str*) → bool

The check JSON file and return whether it was valid or not. A JSON file is invalid if it does not exist or the format ins invalid. :return: True if JSON file is valid, False if not and a new one was created at the specified path

tmtccmd.util.json.**save_to_json_with_prompt**(*key: str*, *value: Any*, *name: str*, *json_cfg_path: str*, *json_obj: Any*) → bool

Object ID Module

class tmtccmd.util.obj_id.**ComponentIdBase**(*obj_id: int*, *byte_len: int*, *name: str | None = None*)

Bases: UnsignedByteField

Base class for unsigned object IDs with different byte sizes

property as_hex_string: str

property obj_id: int

class tmtccmd.util.obj_id.**ComponentIdU16**(*obj_id: int*, *name: str | None = None*)

Bases: *ComponentIdBase*

A helper object for a unique object identifier which has a raw unsigned 16-bit representation.

classmethod from_bytes_typed(*obj_id_as_bytes: bytes*) → *ComponentIdU16*

class tmtccmd.util.obj_id.**ComponentIdU32**(*obj_id: int*, *name: str | None = None*)

Bases: *ComponentIdBase*

A helper object for a unique object identifier which has a raw unsigned 32-bit representation.

```
>>> obj_id = ComponentIdU32(42, "Object with the answer to everything")
>>> int(obj_id)
42
>>> obj_id.name
'Object with the answer to everything'
>>> obj_id.as_bytes.hex(sep=",")
'00,00,00,2a'
>>> obj_id.as_hex_string
'0x0000002a'
```

classmethod from_bytes_typed(*obj_id_as_bytes: bytes*) → *ComponentIdU32*

class tmtccmd.util.obj_id.**ComponentIdU8**(*obj_id: int*, *name: str | None = None*)

Bases: *ComponentIdBase*

A helper object for a unique object identifier which has a raw unsigned 8-bit representation.

classmethod from_bytes_typed(*obj_id_as_bytes: bytes*) → *ComponentIdU8*

tmtccmd.util.obj_id.**ObjectIdBase**

Deprecated type defintion for *ComponentIdBase*

tmtccmd.util.obj_id.**ObjectIdDictT**

Deprecated type defintion for *ObjectIdMapping*

alias of Mapping[bytes, *ComponentIdBase*]

tmtccmd.util.obj_id.**ObjectIdMapping**

Deprecated type defintion for ComponentIdMapping

alias of Mapping[bytes, *ComponentIdBase*]

tmtccmd.util.obj_id.**ObjectIdU16**

Deprecated type defintion for *ComponentIdU16*

tmtccmd.util.obj_id.**ObjectIdU32**

Deprecated type defintion for *ComponentIdU32*

tmtccmd.util.obj_id.**ObjectIdU8**

Deprecated type defintion for *ComponentIdU8*

Exit Module

class tmtccmd.util.exit.**GracefulKiller**

Bases: object

exit_gracefully()

kill_now = False

tmtccmd.util.exit.**keyboard_interrupt_handler**(*tmtc_backend: BackendBase*)

Hamming-Code Module

Hamming Code Implementation Hamming codes belong to the family of linear error correcting codes. Documentation: https://en.wikipedia.org/wiki/Hamming_code They can be used to identify up to two bit errors and correct one bit error per 256 byte block.

class tmtccmd.util.hammingcode.**HammingReturnCodes**(*value*)

Bases: Enum

An enumeration.

CODE_OKAY = 0

ERROR_ECC = 2

ERROR_MULTI_BIT = 3

ERROR_SINGLE_BIT = 1

OTHER_ERROR = 4

tmtccmd.util.hammingcode.**hamming_compute_256**(*data: bytearray*) → bytearray

Takes a bytearray with the size of 256 bytes and calculates the 22 parity bits for the hamming code which will be returned as a three byte bytearray.

Parameters
data

Returns

tmtccmd.util.hammingcode.**hamming_compute_256x**(*data: bytearray*) → bytearray

Computes 3-bytes hamming codes for a data block whose size is multiple of 256 bytes. Each 256 bytes block gets its own code.

Parameters
data – Data to compute code for. Should be a multiple of 256 bytes, pad data with 0 if necessary!

Returns
bytearray of hamming codes with the size (3 / 256 * size). Empty bytearray if input is invalid.

tmtccmd.util.hammingcode.**hamming_test**()

Algorithm was verified with this simple test.

tmtccmd.util.hammingcode.**hamming_verify_256**(*data: bytearray*, *original_hamming_code: bytearray*) → *HammingReturnCodes*

Verifies and corrects a 256-bytes block of data using the given 22-bits hamming code. Returns 0 if there is no error, otherwise returns a HAMMING_ERROR code.

Parameters

- **data** – 256 code block to verify
- **original_hamming_code** – Original 3 byte hamming code with 22 parity bits

Returns

See HammingReturnCodes enums. - -1 for invalid input - 0 if there are no errors. - 1 if there is a single bit error which has been corrected - 2 if the hamming code has been corrupted - 3 if there was a multi bit error which can not be corrected

tmtccmd.util.hammingcode.**hamming_verify_256x**(*data: bytearray*, *original_hamming_code: bytearray*) → *HammingReturnCodes*

TMTC Printer (FSFW) Module

Contains classes and functions that perform all printing functionalities.

class tmtccmd.fsfw.tmtc_printer.**DisplayMode**(*value*)

Bases: Enum

List of display modes

LONG = <class 'enum.auto'>

SHORT = <class 'enum.auto'>

class tmtccmd.fsfw.tmtc_printer.**FsfwTmTcPrinter**(*file_logger: Logger | None*, *display_mode: DisplayMode = DisplayMode.SHORT*)

Bases: object

This class handles printing to the command line and to files

static bit_extractor(*byte: int*, *position: int*)

Parameters

- **byte**
- **position**

Returns

static chunks(*lst: List*, *n*) → Generator[List[List], None, None]

Yield successive n-sized chunks from lst.

generic_hk_tm_print(*content_type: HkContentType*, *object_id: ComponentIdU32*, *set_id: int*, *hk_data: bytes*)

This function pretty prints HK packets with a given header and content list :param content_type: Type of content for HK packet :param object_id: Object ID of the HK source :param set_id: Unique set ID for the HK packet :param hk_data: User defined HK data :return:

static generic_short_string(*packet_if:* PusTmInterface) → str

static get_validity_buffer_str(*validity_buffer:* *bytes*, *num_vars:* *int*) → str

Parameters

- **validity_buffer** – Validity buffer in bytes format
- **num_vars** – Number of variables

Returns

handle_long_tm_print(*packet_if:* PusTmInterface, *info_if:* PusTmInfoInterface)

Main function to print the most important information inside the telemetry :param packet_if: Core packet interface :param info_if: Information interface :return:

static print_data(*data:* *bytes*)

Parameters
data – Data to print

Returns
None

print_validity_buffer(*validity_buffer:* *bytes*, *num_vars:* *int*)

tmtccmd.fsfw.tmtc_printer.**get_validity_buffer_str**(*validity_buffer:* *bytes*, *num_vars:* *int*) → str

Parameters

- **validity_buffer** – Validity buffer in bytes format
- **num_vars** – Number of variables

Returns

Configuration Utility Module

class tmtccmd.util.conf_util.**AnsiColors**

Bases: object

BLUE = '\x1b[34m'

BOLD = '\x1b[1m'

CYAN = '\x1b[36m'

GREEN = '\x1b[32m'

MAGNETA = '\x1b[35m'

RED = '\x1b[31m'

RESET = '\x1b[0m'

YELLOW = '\x1b[33m'

tmtccmd.util.conf_util.**acquire_timeout**(*lock*, *timeout*)

Helper functions which allows to check result of the acquire operation while also using the context manager. :param lock: :param timeout: :return:

tmtccmd.util.conf_util.**check_args_in_dict**(*param: any*, *iterable: Iterable | dict*, *warning_hint: str*) → Tuple[bool, int]

This functions checks whether the integer representation of a given parameter in contained within the passed collections, for example an (integer) enumeration. Please note that if the passed parameter has a string representation but is a digit, this function will attempt to check whether the integer representation is contained inside the passed enumeration. :param param: Value to be checked :param iterable: Enumeration, for example a enum.Enum or enum.IntEnum implementation :param warning_hint: :return:

tmtccmd.util.conf_util.**wrapped_prompt**(*text: str*)

Module contents

Countdown Module

The countdown module was moved to the *spacepackets* library. Use the `spacepackets.countdown` module.

Sequence Count Module

The sequence count module was moved to the *spacepackets* library. Use the `spacepackets.seqcount` module.

Flight Software Framework (FSFW) Support Package

Submodules

Module contents

tmtccmd.fsfw.**bit_extractor**(*byte: int*, *position: int*)

Parameters

- **byte**
- **position**

Returns

tmtccmd.fsfw.**parse_fsfw_events_csv**(*csv_file: str*) → Dict[int, *EventInfo*] | None

tmtccmd.fsfw.**parse_fsfw_objects_csv**(*csv_file: str*) → Mapping[bytes, *ComponentIdBase*] | None

tmtccmd.fsfw.**parse_fsfw_returnvalues_csv**(*csv_file: str*) → Mapping[int, RetvalInfo] | None

tmtccmd.fsfw.**validity_buffer_list**(*validity_buffer: bytes*, *num_vars: int*) → List[bool]

Parameters

- **validity_buffer** – Validity buffer in bytes format
- **num_vars** – Number of variables

Returns

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

A

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y