MIB Generator

Release 0.1

Josef Vácha

Sep 15, 2023

DOCUMENTATION:

MIB Generator is a set of Python scripts that serve the puspose of generating MIB databases (Used in ESA's MCS SCOS 2000) from a set of header (and others) files written in C which define the structure of TM and TC packets. It is concieved mostly as a single script which the user runs from the terminal rather than a series of methods, but the structure of the project should hopefully allow easy modification of the generation process, which in the current implementation assumes one specific formatting of the inputted C-files (see *here* for more info).

Note: This project is under active development.

CHAPTER

ONE

ABOUT THE SCRIPT

The workings of the MIB generator can be divided into multiple steps:

1. Interaction with the user and configuration. These matters are handled by the initial Python script in the *mib_generator.main* package and various utilities in the *mib_generator.utilities* package.
2. Parsing of the C-file into a format which is easily manipulated in Python. This is mainly the job of the *mib_generator.parsing* package.
3. Construction of Python objects which represent TM packets, TC commands, various calibrations, etc. from the parsed representations of C-files. This is done by modules in the *mib_generator.construction* package.
4. Generation of MIB tables, their checking and saving to appropriate location. This is done by the *mib_generator.generation* package.

For a further general understanding of how this code functions, I recommend you to start by reading the documentation for the *mib_generator.main.main* module and then continue from there on.

Currently, four C files are expected as a basis for the generation process:

1. Tm `.h` file. This should include information about the structure of Tm packets, their entries, associated calibrations, etc.
2. Tm `.c` file. This should include mainly list of all Tm packet types and some of their meta-characteristics.
3. Tc `.h` file. This should include information about the structure of Tc commands, their entries, headers of Tc packets and associated decalibrations and verifications.
4. TcTm `.h` file. This should include information common to both Tm and Tc packets like the common id structures or various enums.

For the expected content of these files in more detail see *code examples here*.

CHAPTER

TWO

INSTALLATION AND BUILDING

This is a Python project so in any case you have to have a Python interpreter installed on your machine.

2.1 Installation

The releases are published to PyPI and so the project can be most easily installed with:

```
$ pip install mib-generator
```

Or alternatively:

```
$ python3 -m pip install mib-generator
```

2.2 Building from source

In case you want to build the project from the source yourself:

Clone the source repository using:

```
$ git clone https://github.com/vachaj11/MIB-Generator.git
```

Navigate to the cloned directory:

```
$ cd MIB-Generator
```

The building process requires the Python `build` package. Install it (or check the installation) with:

```
$ python3 -m pip install --upgrade build
```

Build the MIB Generator package with:

```
$ python3 -m build
```

If this finishes successfully, navigate to the build directory:

```
$ cd dist
```

And install the built package with:

```
$ python3 -m pip install *.whl
```

(Replace the `*.whl` with the full name in case you have multiple versions built.)

2.3 Running without building

The set of scripts can also be used directly without any building and packet installation. In such case, clone the source code into desired directory with:

```
$ git clone https://github.com/vachaj11/MIB-Generator.git
```

Before running the cloned code, you will have to manually install the prerequisite python libraries, which are `json5`, `python-docx` and `PySide6` (optional, used only for the GUI). Install them using (assuming a standard Python3 environment):

```
$ python3 -m pip install json5 python-docx PySide6
```

You should then be able to directly run any script found in the package, subpackages and modules in the `./src` directory.

CHAPTER

THREE

USAGE

Finding and running the main script:

3.1 With the package installed

If you have the project installed as a package (either from PyPI or built yourself.), running is should be as easy as typing:

```
$ mib-gen
```

or in case of a more general installation:

```
$ python3 -m mib-gen
```

if even this doesn't work than you can also run directly the Python file `mib-gen` found in the `.../bin` folder of your Python installation.

3.2 Running without building

Navigate to the directory where you unpacked or cloned the project and run the program with:

```
$ python3 src/main/cli.py
```

This does the same as running `$ mib-gen` would with the project installed as a package.

3.3 First time

If you're running the program for the first time, you should start by specifying the input/output files. This can be either done manually by modifying the file `data/paths.json5` (found in `.../src//data/` with respect to the rest of the package) or by running the script as:

```
$ mib-gen -p
```

3.4 Flags and options

The CLI of the main script also accepts a few additional flags, to see all of the options and their consequences, run:

```
$ mib-gen --help
```

3.5 GUI

A simple GUI is provided for the program. Run it with:

```
$ mib-gen-gui
```

more generatlly:

```
$ python3 -m mib-gen-gui
```

or without installation:

```
$ python3 src/mib_generator/gui/gui.py
```

CHAPTER

FOUR

I NEED TO CHANGE...

What entries are put in a MIB table which is associated to a...

- TM packet -> Edit appropriate attribute of *mib_generator.construction.TM_packet.TM_packet*
- TC command -> Edit appropriate attribute of *mib_generator.construction.TC_packet.TC_packet*
- TC header -> Edit appropriate attribute of *mib_generator.construction.TC_packet.TC_header*
- TM calibration -> Edit the corresponding class in *mib_generator.construction.calib*
- TC decalibration/verification -> Edit the corresponding class in *mib_generator.construction.calib*

How a given file is parsed. This file has an ending...

- `.h` -> Edit appropriate class/parsing method in *mib_generator.parsing.par_header*
- `.c` -> Edit appropriate class/parsing method in *mib_generator.parsing.par_cfile*

Something about how the actual output MIB files are generated and checked...

- -> Edit some method in *mib_generator.generation.gener* or *mib_generator.generation.gener_methods*
- -> For the generated `.docx` document: *mib_generator.generation.gener_doc*

Something about the general working of the script...

- -> for general functioning of the script you want to change either *mib_generator.main.main.main* or *mib_generator.parsing.load*
- -> for the UI/cli interface of the script you want to change either something in general in *mib_generator.main.cli* or some module in *mib_generator.utilities*
- -> for the GUI interface you want to change something in *mib_generator.gui*

Some incorrect specification about the structure of MIB tables...

- -> Edit some attribute in *mib_generator.data.longdata*

The default config files...

- -> Edit the `.json5` files in *mib_generator.data*
- -> For operations with them see *mib_generator.temp*

CHAPTER

FIVE

C-CODE SAMPLES

Here you will find examples of how various objects which are recognised by this program can be stated in the inputted C code.

5.1 TM packets

This program first looks for the TM `.c` file, in which it expects to see two things (in this order):

1. An array defining all the apids to be used:

```
const uint16_t apidNum[APID_LAST] =
{
    [APID_DAPU] = 1265,
    [APID_FGM] = 1231,
    [APID_DISC] = 1232,
    [APID_COMPLIMENT] = 1233,
    [APID_LEES] = 1234,
    [APID_SCIENA] = 1235,
};
```

   This is then used similarly to an `enum` to match apid numbers to packets.

2. An array of structures defining all TM packets types to be interpreted by their name, apid, type and subtype. For e.g. two packet types:

```
const struct TmProperties tmProperties[TM_LAST] =
{
    {
        .type = TM_ACK1,
        .apid = APID_DAPU,
        .serviceType = 1,
        .serviceSubType = 1
    },
    {
        .type = TM_NACK2,
        .apid = APID_DAPU,
        .serviceType = 1,
        .serviceSubType = 2
    }
};
```

The program then turns to the TM `.h` file. There it first resolves parts of pre-processor logic to figure out what parts of the code to omit and then starts looking at what C-objects are included and what they represent. This is a multi step process so I won't go into full detail here, hopefully it will be sufficient to provide some examples.

1. All `enum` objects and `#define` macros are added to one giant dictionary which is then used to evaluate stuff. E.g. if the following lines are in the file:

```
#define CRC_SIZE 2

// Telemetry packet definition
enum TmType
{
    TM_ACK1,
    TM_NACK2,
    TM_ACK7,
    TM_NACK8
};
```

   the result will be the following Python evaluation dictionary:

```
{"CRC_SIZE": 2, "TM_ACK1": 0, "TM_NACK2": 1, "TM_ACK7": 2, "TM_NACK8": 3}
```

2. All instances of `struct` preceded by an appropriate comment which includes json5 syntax with the `"pack_type"` entry can be interpreted as a declaration of a packet structure. For instance here is defined a packet of type `"TM_DISC"`:

```
/*{ packet: "TM", pack_type: "TM_DISC", "base_par_index": 59200, prefix: "LWT",
    text_id: "LWY_SCI0002", spid: 75032, desc: "TM Acceptance success", Mnemonic:
↪"ACK_jez", use_structure: "TmDiscFull" }*/
struct PACKED TmDiscFull
{
    uint16_t sid; /*{ sid: true, enum: "sci_sids", desc: "DISC_B2_FULL",  const_
↪value: "SID_SCI_DISC_FULL" }*/
    uint8_t time[7]; /*{ "desc": "Acquisition time", "type": "CUCTIME4_3" }*/
    uint8_t compression:5; ///< 0 not compression
    uint8_t numBlocks; /*{ "desc": "Number of full blocks", vpd: "count" }*/
    // NOTE: Here you need to create a nested VPD block
    struct DiscFullBlock dataBlocks[4]; /*{ "desc": "DISC data", vpd: "data" , cal:
↪"voltage_cal"}*/;
};
```

   Lots of things can be seen here. First, any comment which is of the type `/*{ ... }*/` is taken as an interpretable comment and will be assumed to contain a json5 dictionary (where as the comment `// NOTE...` will be ignored). In terms of the comment directly preceding the `struct`, this is taken to provide additional information about the packet. Here:

   1. `packet` - This describes whether the packet is telemetry or telecommand. It is not used anywhere.
   2. `pack_type` - This describes packet type of this packet. It is used to match the packet to appropriate type, subtype and apid numbers as declared in the TM `.c` file.
   3. `base_par_index` and `prefix` - This is a basis on which the parameters inside the packet will be assigned unique names. E.g. here the parameter named in C as `compression` will be assigned name `"LWT59202"`.
   4. `text_id` - A name that will be used for this packet in the MIB database.
   5. `spid` - Spid of this packet.

6. `desc` - Description of the purpose/task of this packet.
7. `Mnemonic` - Used for more systematic naming scheme. So far not implemented.
8. `use_structure` - Allows for explicit specification of the `struct` to be used as a structure declaration for this packet. It is redundant here since the `struct "TmDiscFull"` directly follows the comment.

Comments are also used for each of the entries/parameters to states their additional properties. Here:

1. `sid` - This means that this entry is an additional identification field for this packet, with its value defined in the `const_value` entry.
2. `type` - This defines the type of the parameter in addition to its type in C and information on whether it is an array.
3. `desc` - Description of the entry/parameter.
4. `vpd` - This describes that the parameter is part of variable packet definition. Three values are possible here:
   - `"fixed"` - in which case the parameter is taken to repeat a fixed amount of times (this amount being described by its array).
   - `"count"` - in which case the parameter defines the number of times the parameter/groups of parameters following it will be repeated.
   - `"data"` - in which case the parameter is the one being repeated (or part of such group).
5. `cal` - This states whether and what calibration should be used for this parameter.

Also notice here that one of the entries is itself a `struct`. The code will "unpack" this `struct` (if it finds its declaration) by including all its entries inside this packet (in this case marking them as a repeating vpd group).

Custom bit sizes are also implemented so the entry `"compression"` here will only be taken to have length of 5 bits.

3. Calibration of TM parameters can be defined in both the Tm and TcTm `.h` files and can be stated in various ways:

   Polynomial calibrations are expected to have the following format:

```
/*{
    "cal_def": "second_cal", "cal_ident": "CAL00004", "desc": "Time to seconds␣
↪calibration",
    "mcf": {"a0": 0, "a1": 0.1, "a2": 0, "a3": 0}
}*/
```

   where the nature of the calibration is recognised by the `"mcf"` key and the entries in the sub-dictionary define the polynomial coefficients.

   Logarithmic calibrations are expected to have the following format:

```
/*{
    "cal_def": brightness_cal", "cal_ident": "CAL00008", "desc": "Brightness␣
↪calibration",
    "lgf": {"a0": 1, "a1": 2.1, "a2": 0, "a3": 6.2}
}*/
```

   where the nature of the calibration is recognised by the `"lgf"` key and the entries in the sub-dictionary define the calibration coefficients.

   Textual calibrations can be formatted in two ways, either wholly through a comment or through an `enum`:

```
/*{
    cal_def: "load_levels", cal_ident: "CAL00003", "desc": "Voltage calibration",
    text_cal: { min: 0, max: 10, lookup: [
        { val: 1, text: "Low" },
        { val: 2, text: "Less low" },
        { from: 3, to: 9, text: "Higher" },
    ]}
}*/
```

or

```
/*{ "enum": "error_codes", "cal_ident": "CAL00002", "desc": "Error code lookup" }*/
enum ErrorCode {
    DAPU_ERROR_1 = 1, /*{ text: "DPU Error 1"}*/
    DAPU_ERROR_2 = 2, /*{ text: "DPU Error 2"}*/
    DAPU_ERROR_3 = 3  /*{ text: "DPU Error 3"}*/
};
```

In the former case the calibration type is recognised by the "text_cal", in the latter by the "enum".

Numerical calibrations should be of the format:

```
/*{
    cal_def: "sinus", cal_ident: "CAL00007", "desc": "Sinus curve",
    num_cal: [[0.0, 0.0], [0.44, 0.43], [0.89, 0.78], [1.34, 0.97], [1.79, 0.97],␣
↪[2.24, 0.78], [2.69, 0.43]],
}*/
```

Recognised by the "num_cal" key.

5.2 TC packets

The case with TC packets/commands is mostly similar to TM packets with the main difference being that in their case, there is no Tc `.c` file, so the command definitions are searched for directly in the Tc `.h` file. Three various things (apart from enums and preprocessor stuff which is analogous to TM packets) can be recognised in this file:

1. TC header/s. Are recognised by their names, etc. Only basic analysis is performed on them and overall this side of things is not very much implemented. Example header (not much interesting to see here):

```
struct PACKED TcHead
{
    struct Id id;
    struct Sequence sequence;
    uint16_t length;
    // End of Packet primary header, start of packet secondary header
    unsigned int version :4; ///< 1 for ECSS-E-70-41A, 2 for ECSS-E-ST-70-41C
    unsigned int ackFlags:4; ///< B3 acceptence, B2 execution, B1 progress, B0␣
↪completion acknowledgment requested
    uint8_t serviceType; ///< Service type
    uint8_t subType; ///< Service subtype
    uint16_t sourceID; ///< TC source
    uint8_t spare; /*{ desc: "spare", const_value: 0 }*/
};
```

2. Command definition. This mostly conforms to what was said about the TM packet definitions above. Example Tc command defined with `struct` can be:

```
/*{ packet: "TC", service: 3, sub: 6, "base_par_index": 66400, prefix: "LWP", use_
↪structure: "TcHkDisable",
    text_id: "LWC00306", desc: "TC Disable HK", Mnemonic: "ACK_jez", cvs: [17001] }
↪*/
struct TcHkDisable
{
    struct SpwHead spwHead; // ignore this for MIB - common SpW header
    struct TcHead tcHead;   // ignore this for MIB - common PUS header
    uint16_t numPars;   /*{ cdf: "count", desc : "Number of Sids", max : "MAX_
↪BLOCKS_IN_PUS", min : 1}*/
    uint16_t hkSid[1];  /*{ cdf: "data", enum: "hk_sids", desc : "HK SID", min :
↪"SID_HK_REPORT_DAPU",
                            max: "SID_HK_REPORT_PSU",  default: "SID_HK_REPORT_DAPU
↪"}*/
    uint8_t spare; /*{ desc: "spare", const_value: 0 }*/
};
```

Unlike with with TM however:

1. `sub` and `service` - state what is the service type and subtype of the packet that this command is send by
2. `cvs` - defines what verifications should be used for this command (if this is not stated, then the defaults are used)

And for parameters:

1. `cdf` - defines repetition of some parameter in this command in a way analogous to `vpd` in TM packets.
2. `min` and `max` - define a range check to be applied to the parameter.
3. `default` - define a default value to be used for the parameter.
4. `const_value` - used like this with the `spare` entry, it defines an entry which is a fixed area in the command.
5. `enum` - here means that decalibration of the specified name should be used for the parameter.

3. Decalibrations and verifications. **Decalibrations** in case of Tc commands are analogous to textual calibrations for Tm packets defined through `enum`. They can look e.g. like:

```
/*{ "enum": "on_off", "dec_ident": "DEC00005", "cal_ident": "CAL00005", "cal_desc":
↪"on/off" }*/
enum OnOff {
    ONOFF_OFF = 0, /*{ text: "Off"}*/
    ONOFF_ON = 1  /*{ text: "On"}*/
};
```

The decalibration is recognised through the `"dec_ident"` key in the comment. Since `"cal_ident"` is also a key, this structure also defines a Tm textual calibration at the same time.

Verifications are defined through a single isolated comments. They have e.g. the following form:

```
// Verification definitions to generate CVS file: CVS_SOURCE is always R, CVS start␣
↪is 0
/*{ cvs_def: 17001, cvs_type: "A", cvs_interval: 60, default: true }*/
/*{ cvs_def: 17002, cvs_type: "C", cvs_interval: 60, default: true }*/
```

Here the parameters inside are passed to the cvs mib table apart from `default` which is used to determine whether the given verification should be applied by default automatically to commands which do not have specific set of verifications assigned to them.

CHAPTER

SIX

MIB_GENERATOR.MAIN PACKAGE

This subpackage does a sort of administrative work around the other sub-packages. It doesn't contain any program logic itself but it calls it from other parts of the code and provides an joint interface for it that the user can interact with.

Included submodules:

- *main* - Module that contains all the logic communicating and passing data between components from other subpackages.
- *cli* - Module that provides a simple CLI interface that can be called from the terminal and which the user can interact with.

6.1 Submodules

6.2 mib_generator.main.cli module

Provides a simple CLI that can be accessed by the user.

If ran directly as a Python script, this module provides the user with a simple CLI which allows for specification of various options for runtime of the script itself (whether the MIB databases should be constructed, whether saved, whether the parsed file should be visualised, etc..). It also provides a quick `--help` summary and an option to run scripts for updating the config and paths files before running the generating script itself.

mib_generator.main.cli.**cli_run**()

Manages the CLI and runs the program.

This is the main method that is run by external calls that want to run the whole program. It takes no arguments but assuming it is run as a bash/terminal program, it tries to parse and interpret passed flags and options and creates a small CLI interface which can be presented to the user with the `--help` flag.

mib_generator.main.cli.**directory**(*string*)

6.3 mib_generator.main.main module

This module works as a startpoint and a junction for creation of the MIB databases.

This module through its *main* method serves the role of a logical centre of the whole program. It interchanges appropriate files between sub-packages, calls appropriate methods for each task, etc. See *main* for more info.

mib_generator.main.main.**main**(*visual=False*, *generate=True*, *parseonly=False*, *paths=False*, *config=False*, *generate_t=False*, *custom_dir=None*)

Run this whole hellish thing.

This method holds the main logic of the whole program. Roughly it sequentially does this:

1. If the appropriate option is raised, run script to update the input/output paths in default/specified directory.
2. If the appropriate option is raised, run script to update the config file in default/specified directory.
3. Load the configuration file into the runtime directory (*mib_generator.temp*).
4. Initialise the *mib_generator.parsing.load* module which automatically parses the files at the specified paths.
5. Unless construction is disabled, call appropriate construction scripts and receive Python representation of all the calibrations, TM-packets, TC-commands, etc. that occur in the parsed files.
6. Perform some checks that these objects (mainly TM-packets and calibrations) are linked in a correct way.
7. Unless generation is disabled, call the generation script which turns all previously constructed objects into MIB tables and saves them.
8. If the appropriate option is raised, generate a document summing up the interpreted TM/TC packages.
9. If the appropriate option is raised, show the parsed files' contents in a GUI visualisation.

Parameters

- **visual** (*bool*) – `True` if the GUI visualisation of the parsed files should be shown, `False` otherwise (and by default).
- **generate** (*bool*) – `True` (by default) if the MIB tables should be generated and saved from the constructed Python representations, `False` otherwise.
- **parseonly** (*bool*) – `True` if only parsing of the C-files should be done and none of the subsequent steps. `False` otherwise and by default.
- **paths** (*bool*) – `True` if the script to update the input/output paths should be run, `False` otherwise (and by default).
- **config** (*bool*) – `True` if the script to update the config file should be run, `False` otherwise (and by default).
- **generate_t** (*bool*) – `True` if the `.docx` document summing up the processed TM and TC packets is to be generated, `False` otherwise.
- **custom_dir** (*str*) – A string specifying the path to a directory where the configuration files on basis of which this program runs, are located.

Returns

`True` if the script finished successfully, `None` otherwise.

Return type

bool

CHAPTER

SEVEN

MIB_GENERATOR.PARSING PACKAGE

These modules serve the purpose of parsing the inputted C/header files into a variety Python objects, that can be then easily accessed at the later construction stage of the process. The resulting product with parsed objects should contain all information in the input files, which requires the representation to be a bit convoluted from place to place, but still understandable I hope.

The submodules here are:

- *load* - Module that allows for interaction between the parser and rest of the code.
- *parser_main* - Module that initialises the parsing process for a given file and holds class that represents the result.
- *par_cfile* - Module that holds the methods and classes for parsing `.c` files.
- *par_header* - Module that holds the methods and classes for parsing header `.h` files.
- *par_header* - Module holding various methods that are used by other modules throughout the parsing process.

7.1 Submodules

7.2 mib_generator.parsing.load module

Central location from which other modules can source data from the parsed C-files.

This module is initialised and its methods are called in the first step of the MIB construction process. At this point the C-files are loaded from the defined paths, are parsed using the *mib_generator.parsing.parser_main.main* method and are subsequently available for other scripts throughout the rest of the MIB construction process. The only other method here is *extr_values* which serves the purpose creating a global evaluation dictionary holding values from all header enums, macros, etc…

`mib_generator.parsing.load.`**`TmH`**

Holds the Python parsed representation of the Tm-header C-files. Each of type *mib_generator.parsing.parser_main.file*

Type
list

`mib_generator.parsing.load.`**`TcH`**

Holds the Python parsed representation of the Tc-header C-files. Each of type *mib_generator.parsing.parser_main.file*

Type
list

mib_generator.parsing.load.**TcTmH**

Holds the Python parsed representation of the TcTm-header C-files. Each of type *mib_generator.parsing.parser_main.file*

Type
list

mib_generator.parsing.load.**TmC**

Holds the Python parsed representation of the Tm (normal) C-files. Each of type *mib_generator.parsing.parser_main.file*

Type
list

mib_generator.parsing.load.**enumerations**

A dictionary containing all possible usable evaluations/enumerations sourced form enums, macros, etc… found in all 3 of the parsed header files.

Type
dict

mib_generator.parsing.load.**enum_stuff**()

Create a dictionary that includes all the possible evaluations from the input files.

For each of the inputted (and processed) C-files, this method extracts an evaluation (dictionary including all possible variable substitutions, from e.g. *enum* objects.) and then joins all of these dictionaries to one and assigns it to a global variable so it can be easily accessed.

mib_generator.parsing.load.**extr_values**(*files*)

Create a dictionary from constants, `enum` correspondences, etc… in the given files.

Goes through all objects in the parsed files and for each `enum` and macro, appends the name-value pairs present to a dictionary (which represents the "global" evaluation in the file).

Parameters
files (*list*) – Files from which the evaluation dictionary is to be extracted. Each of type *mib_generator.parsing.parser_main.file*

Returns
A dictionary with all possible "global" evaluation found in the given file.

Return type
dict

mib_generator.parsing.load.**get_conf**()

mib_generator.parsing.load.**get_paths**()

Load paths from the temporary config files.

This method looks up paths to each of the input files in the config files in the runtime config directory. It then saves these paths as global attributes of the *mib_generator.parsing.load* module, so they can be easily accessed to other methods in the whole program.

mib_generator.parsing.load.**load_all**()

Run all initialisation and parsing methods.

This method (replacing previous simple initialisation of this module) runs other methods, which

1. Load the paths to input C-files.

2. Load the configuration settings. 2. Parse files at these paths. 3. Create evaluation dictionary from these parsed files.

mib_generator.parsing.load.**parse_all**()

Parse the inputted C-files and save the output.

This method uses the *mib_generator.parsing.parser_main* module to parse contents of the files specified at the paths given by global attributes of this module. It then assigns the outputs of the parsing process to various other global attributes of this module, so they can be easily accessed.

7.3 mib_generator.parsing.par_cfile module

Parsing and structural interpretation module for C files.

This module holds functions and classes serving the purpose of parsing C files containing information about the structure of packets and commands. The parsing in general happens mostly by recognising (a subset of) standard C-syntax and the system of Python representations of C-objects tries to include all possible information present in the C file, which makes it perhaps a bit too convoluted.

class mib_generator.parsing.par_cfile.**instance**(*typ*, *inds*, *inde*, *cont*)

Bases: *instance_og*

Class of "first order" C structure/object found in the body of the file.

This class is a Python representation of an instance of a defined "top-level" object in the C-file. It does correspond to any specific type of such object (like `struct`, `uint8`, ...) unlike its child-classes, but rather holds general properties and methods that all of these classes need, like parsing into array of elements.

All **parameters** are passed into the *instance_og* class from which this structure also inherits all its **attributes**.

name

The name of the instance of a C-object.

Type
str

array

Equals to "-1" if the object isn't an array and otherwise denotes the number of elements found in the array. Either as an int number or some abstract expression to be evaluated later.

Type
str

elements

List of elements present if the structure is an array. Each of the elements is either of class *struct_r* or *misc_r*.

Type
list

reposition()

Use classic ordering if no specific position of each element of a structure is declared.

The C-code at the level of instances of the structure in the array can contain information about a custom ordering inside this array. This is found while parsing the structure-instances, however if this information is not present, the default ordering has to be used instead which is what this function does by rewriting positions of the elements in the array in such case.

str_parse(*cont*)

Parse the content of the C-object into its syntactic/semantic components.

By first looking for whitespace characters and various types of brackets and then calling classes *struct_r* and *misc_r*, this function parses the text of the C-object into its name, array size and entries present.

Parameters
cont (*str*) – The raw text of the C-object as found in the C file.

Returns

A tuple containing:

- *str* - The name of the C-structure.
- *str* - See the attribute *array* above.
- *list* - See the attribute *elements* above.

Return type
tuple

class mib_generator.parsing.par_cfile.**instance_og**(*typ*, *inds*, *inde*, *cont*)

Bases: object

Parent class of all representations of the C-objects.

This class holds properties which are shared by all Python representations of C-objects found in the analysed file.

Parameters

- **typ** (*str*) – The type of the C-object (e.g. struct or enum).
- **inds** (*int*) – Starting index of the C-object.
- **inde** (*int*) – End index of the C-object.
- **cont** (*str*) – Raw text of the C-object as in the source file.

type

The type of the C-object (e.g. struct or enum).

Type
str

start

Starting index of the C-object.

Type
int

end

End index of the C-object.

Type
int

text

Raw text of the C-object as in the source file.

Type
str

comment

List holding all found interpretable comments associated to this C-object.

Type
list

class mib_generator.parsing.par_cfile.**misc_r**(*typ*, *inds*, *inde*, *cont*)

Bases: *instance_og*

Class of a singular instance of a constant within an array.

This class represents a specific instance of a constant (e.g. `unit32`, ...) inside a defined array. The contents of this instance are parsed (based on whitespace characters and "=") such that the extracted information are the position of this instance inside the array and the defined value of this instance.

All **parameters** are passed into the *instance_og* class from which this structure also inherits all its **attributes**.

position

Position of the instance inside an array. Can be either a specific integer or an abstract expression to be interpreted later.

Type
str

value

A value of this instance of the constant. Can be either a specific integer or an abstract expression to be interpreted later.

Type
str

mis_parse(*cont*)

Parse the information about the constant instance.

Based on the presence of "=" and whitespace characters this function recognises the presence of an identifier of the position of the constant instance (inside the array) and interprets the rest as the expression/value assigned to this instance.

Parameters
cont (*str*) – Raw text of the C-code defining the object instance.

Returns

A tuple containing:

- *str* - Position of the instance. See *position*.
- *str* - A value of this instance of constant. See *value*.

Return type
tuple

class mib_generator.parsing.par_cfile.**miscal**(*typ*, *inds*, *inde*, *cont*)

Bases: *instance*

Class of an instance of an array of constants.

This class represents an instance of definition of a constant (e.g. `unsigned integer`, `uint16`, etc...) in the C-file. Being possibly an array of such objects, it inherits from *instance*, which already does most of the parsing, so at creation of this child-class, only some already parsed information are rewritten and corrected rather than the parsing happening from scratch.

All **parameters** are passed into the *instance* class from which this structure also inherits all its **attributes**.

name

Corrected (from *instance*) name of the object.

Type
str

array

Corrected (from *instance*) information about the array of the objects.

Type
str

add_parse(*name*)

Correct information about the object's name.

Looks again at the string identified by *instance* as the object's name, correct it (e.g. delete whitespace characters from it) and also save this name as a property `misc_r.flav` of each instance of the object in the array.

Parameters
name (*str*) – The possibly wrong name of the structure as recognised by the parser in *instance*

Returns
The actual name of the structure.

Return type
str

mib_generator.parsing.par_cfile.**str_parse**(*stri*)

Parse the given string into blocks of "top-level" C-structures.

By going iteratively through the string and recognising the features of standard C-syntax, this function first marks the positions of all C-objects (including `#define` macro declaration) and then creates an appropriate Python representation-object of them, which then at its initialisation further parses the inner structure of these objects. After going through the whole string, this function then returns all found C-objects in a list.

Parameters
stri (*str*) – A string to be parsed/analysed into the C-objects.

Returns
List of Python representations of found C-objects, each represented by an object of child-class of *instance*

Return type
list

class mib_generator.parsing.par_cfile.**struct**(*typ*, *inds*, *inde*, *cont*)

Bases: *instance*

Class of an instance of an array of structs.

This class represents an instance of definition of a C-structure `struct` in the C-file. Being possibly an array of such structures, it inherits from *instance*, which already does most of the parsing, so at creation of this child-class only, some already parsed information are rewritten and corrected rather than the parsing happening from scratch.

All **parameters** are passed into the *instance* class from which this structure also inherits all its **attributes**.

flav

The type of the `struct`, i.e. name of the `struct` declared possibly in the header of which this structure/array of structures is an instance.

Type
str

name

Corrected (from *instance*) name of the structure.

Type
str

add_parse(*name*)

Correct information about the object's name and type.

Looks again at the string identified by *instance* as the object's name and if it seems wrong (e.g. whitespace characters occur in it) further parse it and extract the actual *name* and *flav* of the structure from it.

Parameters
name (*str*) – The possibly wrong name of the structure as recognised by the parser in *instance*

Returns

A tuple containing:

- *str* - The specific type of the structure. See *flav*.
- *str* - The actual name of the structure.

Return type
tuple

class mib_generator.parsing.par_cfile.**struct_r**(*typ*, *inds*, *inde*, *cont*)

Bases: *instance_og*

Class of a singular instance of `struct` within an array.

This class represents a specific instance of a C-structure inside a defined array. The contents of this instance are parsed (based on brackets and commas) such that the extracted information are the position of this instance inside the array and the contents of the defined array represented by a Python dictionary.

All **parameters** are passed into the *instance_og* class from which this structure also inherits all its **attributes**.

position

Position of the instance inside an array. Can be either a specific integer or an abstract expression to be interpreted later.

Type
str

entries

A dictionary representing the defined contents of the instance of the `struct`.

Type
dict

srr_parse(*cont*)

Parse the `struct` instance into its entries.

Based on the presence of "=", brackets and commas, this function recognises the presence of an identifier of the position of the structure instance (inside the array) and each of the entries inside the structure definitions. It then further interprets the former and creates a dictionary from the latter.

Parameters
cont (*str*) – Raw text of the C-code defining the object.

Returns

A tuple containing:

- *str* - Position of the instance. See *position*.
- *dict* - A dictionary with the entries found in the `struct` instance.

Return type
tuple

7.4 mib_generator.parsing.par_header module

Parsing and structural interpretation module for C-header files.

This module holds functions and classes serving the purpose of parsing C-header files containing information about the structure of packets and commands. The parsing in general happens mostly by recognising (a subset of) standard C-syntax and the system of Python representations of C-objects tries to include all possible information present in the C-header file, which makes it perhaps a bit too convoluted.

class mib_generator.parsing.par_header.**define**(*typ*, *inds*, *inde*, *cont*)

Bases: *structure*

Class of a `#define` makro.

This class is a Python representation of an occurrence of `#define` macro in the C-header file. The inner structure of the macro is parsed and represented as a name and an expression.

All **parameters** are passed into the *structure* class from which this structure also inherits all its **attributes**.

name

The name of the definition in the macro.

Type
str

expression

A string with the expression or value that the macro points to.

Type
str

def_parse(*cont*)

Parse the `#define` macro into its syntactic/semantic components.

By looking for whitespace characters, parses the text of the macro into its name and the expression it points to.

Parameters
cont (*str*) – The text of the macro.

Returns

A tuple containing:

- *str* - The name of the macro.
- *str* - The expression the macro points to.

Return type
tuple

class mib_generator.parsing.par_header.**enum**(*typ*, *inds*, *inde*, *cont*)

Bases: *structure*

Class of an enum C-object.

This class is a Python representation of an occurrence of enum in the C-header file. The inner structure of the enum is parsed and represented as a Python dictionary.

All **parameters** are passed into the *structure* class from which this structure also inherits all its **attributes**.

name

Name of the enum object.

Type
str

entries

Dictionary containing key-value pairs found in the enum.

Type
dict

ele_parse(*cont*)

Parse enum into its syntactic/semantic components.

By recognising and parsing the inner syntax of enum this function identifies all elements in it and the values assigned to them (be it directly or implicitly by the order of the elements enum). Subsequently it represents these key-value pairs by a Python dictionary.

Parameters
cont (*str*) – Text of the whole enum object.

Returns

A tuple containing:

- *str* - The name of the enum.
- *dict* - A dictionary representing the entries in the enum and the values/expressions assigned to them.

Return type
tuple

class mib_generator.parsing.par_header.**enum_r**(*typ*, *inds*, *inde*, *cont*)

Bases: *structure*

Class of a reference to an enum inside a C-structure.

This class is used to represent occurrence of reference to enum inside a C-structure (a struct object). It can be of two types:

1. It is only a reference to an enum defined elsewhere. In this case, the *form* attribute holds the name of this reference.

2. There is a nested definition of the enum inside this C-structure. In this case, an object of class *enum* is used to represent the included enum and it is equated to *form*.

All **parameters** are passed into the *structure* class from which this structure also inherits all its **attributes**.

name

The name of the referenced enum.

Type
str

form

Either name of the referenced enum or directly the object of the referenced enum.

Type
str or *enum*

bites

Number of bites that the value of this enum is represented by.

Type
int

array

An expression set to "-1" if the enum reference is not an array and to its number of elements (either integer value or an text expression to be later evaluated) in case it is an array.

Type
str

enr_parse(*cont*)

Parse the reference to its syntactic/semantic components.

By looking for "{" and "}" this function first decides whether the references includes a nested enum or only and external reference. Depending on the result it either calls the class *enum* or leaves the reference as a text. It then continues by looking for the name, array and bites information, etc by looking at the whitespaces and brackets.

Parameters
cont (*str*) – The text of the enum reference to be analysed.

Returns

A tuple consisting of:

- *str* - The name of the referenced enum.
- *str* or *struct* - See the attribute *form* above.
- *int* - See the atribute bite above.
- *str* - See the attribute *array* above.

Return type
tuple

class mib_generator.parsing.par_header.**extern**(*typ*, *inds*, *inde*, *cont*)

Bases: *structure*

Class of external declaration of a constant.

This class is a Python representation of an occurrence of external constant declaration in the C-header file. The inner structure of the expression is parsed and represented as its name, type (e.g. uint8) and an information on whether it is an array.

All **parameters** are passed into the *structure* class from which this structure also inherits all its **attributes**.

name

The name of the external declaration.

Type
str

flav

The data type of the declaration (e.g. "uint8")

Type
str

array

An expression set to "-1" if the declared constant is not an array and to its number of elements (either integer value or an text expression to be later evaluated) in case it is an array.

Type
str

ext_parse(*cont*)

Parse the external declaration of a constant into its syntactic/semantic components.

By looking for whitespace characters and square brackets, parses the text of the declaration into its name, data type and n information on whether it is an array.

Parameters
cont (*str*) – The text of the declaration to be parsed.

Returns

A tuple containing:

- *str* - The data type of the declaration (e.g. "unsigned integer").
- *str* - The name of the declaration.
- *str* - The information on its array size (see above for more detail).

Return type
tuple

class mib_generator.parsing.par_header.**misc_r**(*typ*, *inds*, *inde*, *cont*)

Bases: *structure*

Class of a declaration of a constant inside a C-structure.

This class is a Python representation of an occurrence of constant value declaration (e.g. uint8) inside of some C-structure . The inner structure of the expression is parsed and represented as its name, bite length and an information on whether it is an array.

All **parameters** are passed into the *structure* class from which this structure also inherits all its **attributes**.

name

The name of the declared constant.

Type
str

bites

Number of bites that the value of this constant is represented by.

Type
int

array

An expression set to "-1" if the constant is not an array and to its number of elements (either integer value or an text expression to be later evaluated) in case it is an array.

Type
str

mir_parse(*cont*)

Parse the declaration into its syntactic/semantic components.

By looking at whitspaces and brackets in the string this function parses the text of the declaration into the name of the constant, and information about its bite-length and array-size.

Parameters
cont (*str*) – The text of the constant declaration.

Returns

A tuple containing:

- *str* - The name of the constant.
- *int* - The declared bite-length of the constant.
- *str* - See the attribute *array* above.

Return type
tuple

mib_generator.parsing.par_header.**str_parse**(*stri*)

Parse the given string into blocks of "top-level" C-structures.

By going iteratively through the string and recognising the features of standard C-syntax, this function first marks the positions of all C-objects (including `#define` macro declaration) and then creates an appropriate Python representation-object of them, which then at its initialisation further parses the inner structure of these objects. After going through the whole string, this function then returns all found C-objects in a list.

Parameters
stri (*str*) – A string to be parsed/analysed into the C-objects.

Returns
List of Python representations of found C-objects, each represented by an object of child-class of *structure*

Return type
list

mib_generator.parsing.par_header.**str_parse_r**(*offset*, *stri*)

Parse the given string into "2nd-order" blocks of C-structures.

Unlike for *str_parse* this time "second-order" strings (that is strings inside other already recognised C-objects; i.e. e.g. inner content of `struct` declaration) are expected and the parsing itslef is recursive (in order to account e.g. for multiple nested occurences of `struct`).

Parameters

- **offset** (*int*) – Offset of this string from the start of the original file. I.e. original start index of its first character.
- **stri** (*str*) – The string to be parsed/interpreted into its contents.

Returns
List of identified and interpreted C-object inside the original string. Each is represented by an object which is some child-class of the class *structure*.

Return type
list

class mib_generator.parsing.par_header.**struct**(*typ*, *inds*, *inde*, *cont*)

Bases: *structure*

Class of declaration of a C-structure `struct ...`.

This class is a Python representation of an occurrence of `struct` structure in the C-header file. The inner structure of this ... structure is represented by its name, information on whether it is packed and its contents/elements (which are themselves other similar Python objects).

All **parameters** are passed into the *structure* class from which this structure also inherits all its **attributes**.

name

The name of the structure.

Type
str

packed

True if the structure is packed, false otherwise.

Type
bool

elements

A list of elements inside this structure. Each is represented by a Python object which is some child-class of *structure*

Type
list

stc_parse(*cont*)

Parse the `struct` C-structure into its syntactic/semantic components.

By first looking for whitespace characters and "{" and then calling *str_parse_r*, this function parses the text of the `struct` object into its name, entries, etc.

Parameters
cont (*str*) – The raw text of the `struct` C-structure in the C-header file.

Returns

A tuple containing:

- *str* - The name of the C-structure.
- *bool* - `True` if the structure is packed, `False` otherwise.
- *list* - List of elements inside the structure. Each represented by an object of child-class of *structure*.

Return type
tuple

class mib_generator.parsing.par_header.**struct_r**(*typ*, *inds*, *inde*, *cont*)

Bases: *structure*

Class of a reference to a C-structure `struct` inside another.

This class is used to represent occurrence of declaration of one structure inside another. It can be of two types:

1. It is only a reference to a structure defined elsewhere. In this case, the *form* attribute holds the name of this reference.
2. There is a nested definition of one structure inside another. In this case, an object of class *struct* is used to represent the included structure and it is equated to *form*.

All **parameters** are passed into the *structure* class from which this structure also inherits all its **attributes**.

name

The name of the referenced C-structure.

Type
str

form

Either name of the referenced structure or directly the object of the referenced structure.

Type
str or *struct*

array

An expression set to "-1" if the `struct` reference is not an array and to its number of elements (either integer value or an text expression to be later evaluated) in case it is an array.

Type
str

str_parse(*cont*)

Parse the reference to its syntactic/semantic components.

By looking for "{" and "}" this function first decides whether the references includes a nested structure or only and external reference. Depending on the result it either calls the class *struct* or leaves the reference as a text. It then continues by looking for the name, array information, etc by looking at the whitespaces and brackets.

Parameters
cont (*str*) – The text of the structure reference to be analysed.

Returns

A tuple consisting of:

- *str* - The name of the referenced structure.
- *str* or *struct* - See the attribute *form* above.
- *str* - See the attribute *array* above.

Return type
tuple

class mib_generator.parsing.par_header.**structure**(*typ*, *inds*, *inde*, *cont*)

Bases: `object`

Parent class of all representations of the C-objects.

This class holds properties which are shared by all Python representations of C-objects found in the analysed file.

Parameters

- **typ** (*str*) – The type of the C-object (e.g. `struct` or `enum`).
- **inds** (*int*) – Starting index of the C-object.
- **inde** (*int*) – End index of the C-object.
- **cont** (*str*) – Raw text of the C-object as in the source file.

`type`

The type of the C-object (e.g. `struct` or `enum`).

Type
str

`start`

Starting index of the C-object.

Type
int

`end`

End index of the C-object.

Type
int

`text`

Raw text of the C-object as in the source file.

Type
str

`comment`

List holding all found interpretable comments associated to this C-object.

Type
list

7.5 mib_generator.parsing.par_methods module

Depository of parsing methods.

This module holds various methods used for parsing of the header and normal C files and their subsequent interpretation into corresponding python objects.

`mib_generator.parsing.par_methods.clean(stri, tokens)`

Replace tokens with spaces.

Takes a string and a list of tokens and replaces all occurences of the tokens in the string with an appropriate number of space so the absolute positions of unaffected parts of the string stay the same.

Parameters

- **stri** (*str*) – A string to be “cleaned”.
- **tokens** (*set*) – A set of tokens to be replaced.

Returns
The resulting “cleaned” string.

Return type
str

mib_generator.parsing.par_methods.**com_parse**(*comm*)

Parse the comments in the string into blocks corresponding to singular comments.

Looks at each passed comment and evaluates whether it should be interpreted or not. If yes, it then calls the class *comment* from it and adds this object to the list of interpreted comments.

Parameters
comm (*list*) – List of strings each representing the content of a single comment.

Returns
List of found interpreted comments, each represented by an object of the *comment* class.

Return type
list

class mib_generator.parsing.par_methods.**comment**(*start*, *end*, *text*)

Bases: object

Class representing an occurrence of an interpretable comment in the file, entries found in it, etc.

Saves the comment text and its start/end indexes and then it tries to interpret the comment as a dictionary represented in the json5 format.

Parameters

- **start** (*int*) – Start index of the comment in the original file.
- **end** (*int*) – End index of the comment in the original file.
- **text** (*str*) – Original text of the comment.

start

Start index of the comment in the original file.

Type
int

end

End index of the comment in the original file.

Type
int

text

Original text of the comment.

Type
str

entries

Dictionary containing the content of the comment found in the json5 string.

Type
dict

mib_generator.parsing.par_methods.**erase_text**(*stri*)

Erase all text inside quotation marks in the given string.

Looks for first order quotation marks (that is, not inside another quotation marks) and replaces their content with matching amount of spaces so that the total length of the string remains the same.

Parameters
stri (*str*) – The string in which the quotation marks are to be replaced.

Returns
The string with the quotation marks contents replaced.

Return type
str

mib_generator.parsing.par_methods.**line_index**(*stri*)

Give list of indexes of line starts of the given string.

Based on the newline character, looks for indexes of new lines and adds them to a list.

Parameters
stri (*str*) – The string to be analysed.

Returns
List of indexes at which new lines start.

Return type
list

mib_generator.parsing.par_methods.**preproc_eval**(*variab*)

Evaluate whether a pre-processing condition holds.

Looks into config file whether the passed pre-processing condition holds. If the condition name does not occur in the config file, asks the user what boolean value it should assign to it.

Parameters
variab (*str*) – Name of the pre-processing variable/condition to be evaluated.

Returns
Truth-value of the condition.

Return type
bool

mib_generator.parsing.par_methods.**preproc_filter**(*stri_o*, *stri_c*)

Filter out what parts of the C-code should be further considered based on preprocessor directives.

First, this function looks for blocks pre-processor logic using the *preproc_parse* function. It then evaluates the condition associated with this logic with *preproc_eval* and depending on the result, decides what parts of the string should be deleted (replaced by equal number of spaces). Finally it performs this replacement.

The preprocessor logic is determined based on the passed string with the code (`stri_o`), but the deliting is applied also to the string with comment (`stri_c`).

Parameters

- **stri_o** (*str*) – The text with C-code to be filtered on the basis of pre-processor logic.
- **stri_c** (*str*) – The text with comments to be filtered on the basis of pre-processor logic.

Returns

A tuple containing:

- *str* - The string with C-code with the pre-processor logic applied.
- *str* - The string with comments with the pre-processor logic applied.

Return type
tuple

mib_generator.parsing.par_methods.**preproc_parse**(*stri*)

Parse C-preprocessor conditional syntax into logic blocks.

Looks for logical C-preprocessor directives (specifically `#ifdef`, `#ifndef`, etc.), and marks the start/end indices of the corresponding "logic blocks" (i.e. sections beween `#ifdef` and `#else`). Then returns these block indexes in a list. Also performs some checks along the way.

Parameters
stri (*str*) – The string to be analysed/parsed.

Returns
List of list of indexes denoting start/end positions of the logic blocks.

Return type
list

mib_generator.parsing.par_methods.**split_comment**(*stri*)

Split the given string into a section with C comments and a section without them.

Goes iteratively through the string and based on single/multi-line C-comment syntax, recognised blocks of comments and marks their start/end positions. It then constructs versions of the passed string with the comment/code blocks ommited.

Parameters
stri (*str*) – The string to be split into comments/code.

Returns

A tuple consisting of:

- *str* - The original string with comments omitted.
- *str* - The original string with C-code omitted.

Return type
tuple

7.6 mib_generator.parsing.parser_main module

The main parsing module.

This module is a starting point for parsing of C-files and holds class which plays the role of their interpretation.

class mib_generator.parsing.parser_main.**file**(*stri*, *header*)

Bases: `object`

Class representing the file being analysed and its internal structure in an easily Python-accessible format.

During creation this class goes through multiple steps of parsing/interpretation using methods in *mib_generator.parsing.par_header*, *mib_generator.parsing.par_cfile* and `parsing.par_methods`. The steps are as follows:

1. Get general information about the file (length, line indexes)
2. Separate text in comment and the code (while keeping the absolute positions in the file).
3. Process any C-preprocessor logic that's in the code.
4. Parse the resulting code into Python representations of the corresponding C-objects.
5. Parse the comments into intepretable sections (written in json5) and interpret their content.
6. Create links between the found C-objects and interterpretable comments

Parameters

- **stri** (*str*) – The content of the file to be parsed/analysed.
- **header** (*bool*) – Parameter denoting whether file is a header or normal C file.

text

Text of the file.

Type
str

max_position

Length of the file.

Type
int

lines

List of positions of line starts.

Type
list

text_o

File text with only code (without comments).

Type
str

text_c

File text with only comments (without code).

Type
str

text_of

File (with code) after sorting out the pre-processor logic.

Type
str

text_cf

File (with comments) after sorting out the pre-processor logic.

Type
str

structures

List of Python representations of C-objects found in the file.

The Python representations are child classes of either *mib_generator.parsing.par_cfile.instance_og* or *mib_generator.parsing.par_header.structure*

Type
list

comments

List of found interpretable comments which are represented using *mib_generator.parsing.par_methods.comment*

Type
list

link

List of pairs of comments and the Python representation of C-objects they belong to.

Type
list

linker()

Link each comment to a corresponding C-object in the file

Goes one by one through comments and searches for their corresponding structure. For the closes found match, creates link from the comment to the structure and from the structure to the comment (by generating attributes for their respectives classes) and returns them in a list of pairs.

Returns
Pairs of comments and their corresponding structures.

Return type
list of pairs

mib_generator.parsing.parser_main.**main**(*name='/home/vachaj11/Documents/MIB/start/src/PUS_TmDefs.h'*)

Create a parsed python representation of a file.

Opens file, depending on its ending chooses correct parser and lets it run.

Parameters
name (*str*) – Path to file.

Returns
File object

Return type
file

CHAPTER

EIGHT

MIB_GENERATOR.CONSTRUCTION PACKAGE

The modules here serve the purpose of constructing all the various types of MIB tables. They generally take the Python representation of the C-files generated as the result of the parsing process, and by looking what objects are included, in what order, of what types, with what comments, etc... They generate the appropriate entries in MIB tables for them. For conciseness sake, these entries are here generated as dictionaries which are attributes of some constructed object - TM packet (*TM_packet.TM_packet*), TC packet (*TC_packet.TC_packet*), calibration/decalibration (*calib.calib*), verification (*calib.verification*), etc... - which are in a sense abstract representation of the underlying logical structure of the packets/communication standards.

The submodules here are:

- *TM_packet* - Module that holds classes that represent the TM header and each TM packet including all the MIB tables that are connected to each of them.
- *TM_packet_methods* - Module that holds various methods that are used when constructing the MIB tables which are attributes of the *TM_packet.TM_packet* class.
- *TC_packet* - Module that holds classes that represent the TC header and each TC packet including all the MIB tables that are connected to each of them.
- *TC_packet_methods* - Module that holds various methods that are used when constructing the MIB tables which are attributes of the *TC_packet.TC_packet* and *TC_packet.TC_header* classes.
- *calib* - Module which holds classes that represent various types of calibrations/decalibrations/verifications which can occur in the C-files. It also holds methods that are used in the construction of these classes.

8.1 Submodules

8.2 mib_generator.construction.TC_packet module

Module holding Python representations of TC packets, TC header and the corresponding MIB tables.

This module takes care of creating a representation of TC commands from parsed sections of C-files and extracting entries for MIB databases. These classes represent the TC commands only in the sense that they hold in a structured form any possible information that could be found relating to that command including the entries in various MIB tables that correspond to it.

class mib_generator.construction.TC_packet.**TC_header**(*structure*)

Bases: object

Class representing the TC header common to all TC-packets.

This class is an abstract representation of a TC packet header with all of its properties, entries and corresponding MIB tables. It is created from passed *mib_generator.parsing.par_header.struct* object and and subsequently analysed using included methods into the entries in various TC-side MIB tables.

Parameters
structure (parsing.par_cfile.struct) – An object corresponding to a description of this packet-header found in the *mib_generator.parsing.load.TcH* file (i.e. the TC .h file).

structure

An object corresponding to a description of this packet found in the *mib_generator.parsing.load.TcH* file (i.e. the TC .h file).

Type
parsing.par_cfile.struct

entries

List of entries found inside the TC-header. Each is an instance of *mib_generator.parsing.par_header.misc_r*.

Type
list

size

Size of the packet header (joint size of all its entries) in bytes.

Type
int

positions

List of starting positions of entries in the TC-header packet. Each entry is an integer representing an offset from the header start.

Type
list

tcp

Dictionary corresponding to one line in MIB tcp table.

Type
dict

pcpc

List of dictionaries each one corresponding to one line in MIB pcpc table.

Type
list

pcdf

List of dictionaries each one corresponding to one line in MIB pcdf table.

Type
list

pcdf_listdict()

Define elements for entries in pcdf table.

Creates a list of dictionaries in each of which a key-value pair corresponds to entry in one column of the pcdf table (with the key being the name of the column and value the entry to be filled in). Here for each parameter in the header, one row (i.e. one entry in the list) is created, entries in which are mostly deduced from the parameter's name or position/width.

Returns
List of dictionaries which are to be lines in the MIB table. Assigned to *pcdf*.

Return type
list

`pcpc_listdict()`

Define elements for entries in pcpc table.

Creates a list of dictionaries in each of which a key-value pair corresponds to entry in one column of the pcpc table (with the key being the name of the column and value the entry to be filled in). Here for each parameter in the header, one row (i.e. one entry in the list) is created, entries in which are extracted from comments around the given parameter, calculated from type/general information, etc...

Returns
List of dictionaries which are to be lines in the MIB table. Assigned to *pcpc*.

Return type
list

`tcp_dictionary()`

Define elements for entry in tcp table.

Creates a dictionary where each key-value pair corresponds to an entry in one column of the tcp table (with the key being the name of the column and value the entry to be filled in). Here the values are extracted from the comment preceding the TC-header definitions.

Returns
Dictionary which is one line in the MIB table. Assigned to *tcp*.

Return type
dict

`class mib_generator.construction.TC_packet.TC_packet(h_structure, header, h_comment)`

Bases: `object`

Class representing a TC-packet/command and its various properties.

This class is an abstract representation of a TC command with all of its properties, entries and corresponding MIB tables. It is created from passed *mib_generator.parsing.par_header.struct* and *TC_header* objects and and subsequently analysed using included methods into the entries in various TC-side MIB tables.

Parameters

- **h_structure** (parsing.par_header.struct) – An object corresponding to a description of this command found in the *mib_generator.parsing.load.TcH* file (i.e. the TC `.h` file).
- **header** (TC_header) – A TC-header included at the start of the packet in which the command in question is send.
- **h_comment** (parsing.par_methods.comment) – A comment found in the `parsing.parsing.load.TcH` file which holds meta-information about the command.

`h_structure`

An object corresponding to a description of this command found in the *mib_generator.parsing.load.TcH* file (i.e. the TC `.h` file).

Type
parsing.par_header.struct

h_comment

A comment found in the `parsing.parsing.load.TcH` file which holds meta-information about the command.

Type
parsing.par_methods.comment

header

A TC-header included at the start of the packet in which the command in question is send.

Type
TC_header

h_entries

List of entries that relate to the header found inside the command definition. Each is an instance of *mib_generator.parsing.par_header.misc_r*.

Type
list

entries

List of entries that do not relate to the headerfound inside the command definition. Each is an instance of *mib_generator.parsing.par_header.misc_r*.

Type
list

size

Size of the command definition (joint size of all parameters in *entries*) in bytes.

Type
int

positions

List of starting positions of *entries* in the command definition. Each entry is an integer representing an offset from the start.

Type
list

h_size

Size of the command definition (joint size of all parameters in *h_entries*) in bytes.

Type
int

h_positions

List of starting positions of *h_entries* in the command definition. Each entry is an integer representing an offset from the start.

Type
list

parameters

List of indexes of all parameters in *entries* (i.e. those that aren't fixed areas). If entry is a parameter, the corresponding field in this list has its index value, otherwise it is assigned -1.

Type
list

ccf

Dictionary corresponding to one line in MIB ccf table.

Type
dict

cpc

List of dictionaries each one corresponding to one line in MIB cpc table.

Type
list

cdf

List of dictionaries each one corresponding to one line in MIB cdf table.

Type
list

prf

List of dictionaries each one corresponding to one line in MIB prf table.

Type
list

prv

List of dictionaries each one corresponding to one line in MIB prv table.

Type
list

cvp

List of dictionaries each one corresponding to one line in MIB cvp table.

Type
list

ccf_dictionary()

Define elements for entry in ccf table.

Creates a dictionary where each key-value pair corresponds to an entry in one column of the ccf table (with the key being the name of the column and value the entry to be filled in). Here the values are mostly general information extracted from the comment in from of the command definition in the C-header file and such.

Returns
Dictionary which is one line in the MIB table. Assigned to *ccf*.

Return type
dict

cdf_listdict()

Define elements for entries in cdf table.

Creates a list of dictionaries in each of which a key-value pair corresponds to entry in one column of the cdf table (with the key being the name of the column and value the entry to be filled in). Here for each parameter in the header, one row (i.e. one entry in the list) is created, entries in which are mostly deduced from the parameter's name or position/width.

Returns
List of dictionaries which are to be lines in the MIB table. Assigned to *cdf*.

Return type
list

cpc_listdict()

Define elements for entries in cpc table.

Creates a list of dictionaries in each of which a key-value pair corresponds to entry in one column of the cpc table (with the key being the name of the column and value the entry to be filled in). Here for each parameter in the header, one row (i.e. one entry in the list) is created, entries in which are extracted from comments around the given parameter, calculated from type/general information, etc... It should be mentioned that not all entries in `etnries` are parameters since they can be also fixed areas. Hence first, before the dictionary is created, a check is run for this.

Unusual case here is the range check `"CPC_PRFREF"`. Normal here would be to directly associate it to some externally defined calibration, but since that would be quite convoluted and ineffective for such simple task, it was decided to generate a random placeholder name (rather than predefined one) to be put into the `"CPC_PRFREF"` entry and then generate the range check tables as an attribute of this command class *prf* and *prv* (rather then as an external object as is the case with all other calibrations).

Also, before entries for any parameter are calculated, a simple check is run inspecting whether the `"base_par_index"` defined in the packet-level comment corresponds in length to the length of the numerical part of the parameter names to be generated (this is defined in the config file). If not, a warning is raised.

Returns
List of dictionaries which are to be lines in the MIB table. Assigned to *cpc*.

Return type
list

cvp_listdict()

Define elements for entries in cvp table.

Creates a list of dictionaries in each of which a key-value pair corresponds to entry in one column of the cvp table (with the key being the name of the column and value the entry to be filled in). First checks whether there are verifications associated to the given command and if so, creates an entry row to each one of them.

There are default values for verification which are applied automatically later for each command if no verification are specified here (in which case the value of *cvp* will be `None` for now).

Returns
List of dictionaries which are to be lines in the MIB table. Assigned to *cvp*.

Return type
list

prf_listdict()

Define elements for entries in prf table.

Creates a list of dictionaries in each of which a key-value pair corresponds to entry in one column of the prf table (with the key being the name of the column and value the entry to be filled in). Here it is first checked whether the given parameter has a range associated to it and if so appropriate entries for name of the parameter and ad-hoc calibration are generated.

For a reason why this table is here see *cpc_listdict*.

Returns
List of dictionaries which are to be lines in the MIB table. Assigned to *prf*.

Return type
list

prv_listdict()

Define elements for entries in prv table.

Creates a list of dictionaries in each of which a key-value pair corresponds to entry in one column of the prv table (with the key being the name of the column and value the entry to be filled in). Here it is first checked whether the given parameter has a range associated to it and if so, the entries for the corresponding ad-hoc calibration are created.

For a reason why this table is here see *cpc_listdict*.

Returns
List of dictionaries which are to be lines in the MIB table. Assigned to *prv*.

Return type
list

8.3 mib_generator.construction.TC_packet_methods module

Methods for creation of TC MIB tables and internal representation of TC command packets in general.

This module holds methods that help with formatting of parsed data into TC packet and commands characteristics. They are usually concerned with value evaluation, bite counting, type identification etc. I.e. mostly kind of housekeeping jobs.

mib_generator.construction.TC_packet_methods.**find_header**(*files*)

Find structure in the passed files which describes a TC header.

Based on its name, find a structure which corresponds to a TC header. Otherwise return a warning.

Parameters

- **files** (*list*) – List of files (in Python representation of type *mib_generator.parsing.parser_main.file*
- **header.**() *which is searched for the*) –

Returns
The structure identifies as the header.

Return type
parsing.par_cfile.struct

mib_generator.construction.TC_packet_methods.**get_gr_sizes**(*entries*)

Get sizes of (repeting) groups of command entries.

Based on comments attached to each entry/parameter this method decides whether it is part of a repeating group or not and if so whether it is a "data" or "count" parameter. It then marks positions of such groups and creates a list of corresponding group-sizes where position of each counter corresponds to a group size and the remaining entries are left 0.

Parameters
entries (*list*) – List of entries which are to be analysed for command groups. Each of type *mib_generator.parsing.par_header.misc_r*.

Returns
List parallel to the inputted one with "counter" parameters having the value of their group-sizes and all other values are **0**.

Return type
list

mib_generator.construction.TC_packet_methods.**h_analysis**(*h_struct*)

Make a list of all individual (all structs unpacked) entries in the command.

This method goes iteratively through all entries/parameters inside the given structure. If it finds a normal entry, it adds it to the list, if it finds a `struct` or reference to one, it expands it by recursively calling itself and adds all elements in the expansion to the list.

Furthermore here, two output lists are created, one including all parameters from the TC-packet header and the other one including all parameters relating to the actual command.

Parameters
h_struct (parsing.par_header.struct) – The packet structure from which the parameters are to be expanded.

Returns

A tuple consisting of:

- *list* - List of parameters found inside the structure relating to the packet header. Each is of type *mib_generator.parsing.par_header.misc_r*.
- *list* - List of parameters found inside the structure relating to the command content. Each is of type *mib_generator.parsing.par_header.misc_r*.

Return type
tuple

mib_generator.construction.TC_packet_methods.**packet_search**(*files*)

Extract individual TC-command definitions from TC-header files.

Search the inputted files for TC-command definition using their preceding description comments and return them in a list.

This is needed because unlike for TM-packets where the list of all packets is predefined in the TM `.c` file, here all packets' definitions have to be found in the first place.

Parameters

- **files** (*list*) – List of files (in Python representation of type *mib_generator.parsing.parser_main.file*)
- **definitions.** (*which are searched for the command*) –

Returns
List of pairs of command definition [comments, structures] found in the file. Each an instance of [*mib_generator.parsing.par_methods.comment*, *mib_generator.parsing.par_header.struct*].

Return type
list

mib_generator.construction.TC_packet_methods.**param_list**(*entries*)

Get list of parameters in command entries and their positions in the cpc table.

Based on their name (if it is not `"spare"`) this method decides whether a given entry in a command definition is a parameter or a fixed area. It then creates a list in which parameters are marked by their original index and fixed areas by `-1`.

Parameters
entries (*list*) – List of entries which are to be analysed. Each of type *mib_generator.parsing.par_header.misc_r*.

Returns
List parallel to the inputted one with parameters having the value of their index and fixed areas a value of -1.

Return type
list

8.4 mib_generator.construction.TM_packet module

Module holding Python representations of TM packets, TM header and the corresponding MIB tables.

This module takes care of creating a representation of monitoring packets from parsed sections of C-files and extracting entries for MIB databases. These classes represent the monitoring packets only in the sense that they hold in a structured form any possible information that could be found relating to that packet including the entries in various MIB tables that correspond to it.

class mib_generator.construction.TM_packet.**TM_header**(*files*)

Bases: object

Class representing the TM header common to all TM-packets.

This class holds information about the TM header which is common to all TM packets. It is first found in the parsed C-code using the *find_header* method and then using methods from TM_packet_methods its entries are recognised and analysed:

Parameters

- **files** (*list*) – List of parsed representation of files (each of type mib_generation.parsing.parser_main.file)
- **found.** (*in which the header is to be*) –

structure

The structure found in the header file (in its Python representation) which describes the common TM header.

Type
parsing.par_header.struct

entries

List of entries found inside the TM header. Each is an instance of *mib_generator.parsing.par_header.misc_r*.

Type
list

size

Size of the whole TM header in bytes.

Type
int

positions

List of starting positions of entries in the TM header. Each entry is an integer representing an offset from the header start.

Type
list

find_header(*files*)

Find structure which describes the TM header.

Based on its name, find a structure which corresponds to the TM header. Otherwise return a warning.

Parameters

- **files** (*list*) – Files (in Python representation of type *mib_generator.parsing.parser_main.file*)
- **header.** (*which are searched for the*) –

Returns
The structure identified as the header.

Return type
mib_generator.parsing.par_header.struct

class mib_generator.construction.TM_packet.**TM_packet**(*structure*, *h_structure*, *header*, *h_comment*)

Bases: object

Class representing a TM packet type and its various properties.

This class is an abstract representation of a TM packet with all of its properties, entries and corresponding MIB tables. It is created from passed structures found in the *mib_generator.parsing.load.TmH* and *mib_generator.parsing.load.TmC* (Python representations of the two files describing telemetry packets) and subsequently using included methods analysed into the entries in various telemetry-side MIB tables.

Parameters

- **structure** (parsing.par_cfile.struct) – An object corresponding to a description of this packet found in the *mib_generator.parsing.load.TmC* file (i.e. the telemetry .c file).
- **h_structure** (parsing.par_header.struct) – An object corresponding to a description of this packet found in the *mib_generator.parsing.load.TmH* file (i.e. the telemetry .h file).
- **header** (TM_header) – The header structure included in the packet.
- **h_comment** (parsing.par_methods.comment) – A comment found in the parsing.parsing.load.TmH file which holds meta-information about the packet.

structure

An object corresponding to a description of this packet found in the *mib_generator.parsing.load.TmC* file (i.e. the telemetry .c file).

Type
parsing.par_cfile.struct

h_structure

An object corresponding to the structure of this packet found in the *mib_generator.parsing.load.TmH* file (i.e. the telemetry .h file).

Type
parsing.par_header.struct

h_comment

A comment found in the parsing.parsing.load.TmH file which holds meta-information about the packet.

Type
parsing.par_methods.comment

header

The header structure included in the packet.

Type
TM_header

entries

List of entries found inside the TM packet. Each is an instance of *mib_generator.parsing.par_header.misc_r*.

Type
list

var_entries

List of entries which are subject to variable packet definition. For every such entry, the index of this entry (w.r.t. *entries*) is added to this list (or its negated value in case it is a fixed repetition).

Type
list

size

Size of the packet (joint size of all its entries) in bytes.

Type
int

positions

List of starting positions of entries in the TM packet. Each entry is an integer representing an offset from the header start.

Type
list

pid

Dictionary corresponding to one line in MIB pid table.

Type
dict

pic

Dictionary corresponding to one line in MIB pic table.

Type
dict

tpcf

Dictionary corresponding to one line in MIB tpcf table.

Type
dict

pcf

List of dictionaries each one corresponding to one line in MIB pcf table.

Type
list

plf

List of dictionaries each one corresponding to one line in MIB plf table.

Type
list

`cur`

List of dictionaries each one corresponding to one line in MIB cur table.

Type
list

`vpd`

List of dictionaries each one corresponding to one line in MIB vpd table.

Type
list

`cur_listdict()`

Define elements for entries in cur table.

Creates a list of dictionaries in each of which a key-value pair corresponds to entry in one column of the cur table (with the key being the name of the column and value the entry to be filled in). Here for each parameters it is looked up whether any calibrations is required (if it is defined in the corresponding comment), and if so the appropriate entry is created.

As of now, this supports only a single calibration assigned to each parameter.

Returns
List of dictionaries which are to be lines in the MIB table. Assigned to *cur*.

Return type
list

`pcf_listdict()`

Define elements for entries in pcf table.

Creates a list of dictionaries in each of which a key-value pair corresponds to entry in one column of the pcf table (with the key being the name of the column and value the entry to be filled in). Here one "MIB row" is created for each entry/parameter in *entries* and for each row most of the information is taken from the comment attached to the line at which the C-object describing the entry is found in the original C-files.

Also, before entries for any parameter are calculated, a simple check is run inspecting whether the `"base_par_index"` defined in the packet-level comment corresponds in length to the length of the numerical part of the parameter names to be generated (this is defined in the config file). If not, a warning is raised.

Returns
List of dictionaries which are to be lines in the MIB table. Assigned to *pcf*.

Return type
list

`pic_dictionary()`

Define elements for entry in pic table.

Creates a dictionary where each key-value pair corresponds to an entry in one column of the pic table (with the key being the name of the column and value the entry to be filled in). Here most of the work is done at the search of the `sid` entry of the packet (which is done using *mib_generator.construction.TM_packet_methods.pi_sid* method) and extracting information about additional packet identifiers from it.

Because of how these tables are created. I can't initially respect the requirement that there should be only one unique entry in the `"PIC_TYPE"` and `"PIC_STYPE"` columns. Because of this, there has to be a pruning method applied later at the generation step which deletes such repeating entries.

Returns
Dictionary which is one line in the MIB table. Assigned to *pic*.

Return type
dict

`pid_dictionary()`

Define elements for entry in pid table.

Creates a dictionary where each key-value pair corresponds to an entry in one column of the pid table (with the key being the name of the column and value the entry to be filled in). The entries here are extracted mostly from information in the telemetry `.c` file or the comment preceding the structure in the `.h` file.

Returns
Dictionary which is one line in the MIB table. Assigned to *pid*.

Return type
dict

`plf_listdict()`

Define elements for entries in plf table.

Creates a list of dictionaries in each of which a key-value pair corresponds to entry in one column of the plf table (with the key being the name of the column and value the entry to be filled in). Here the information in *positions* are employed in order to calculates the offsets of each parameter (corresponding to an element in *entries*) from the start of the packet and its width in bites.

Returns
List of dictionaries which are to be lines in the MIB table. Assigned to *plf*.

Return type
list

`tpcf_dictionary()`

Define elements for entry in tpcf table.

Creates a dictionary where each key-value pair corresponds to an entry in one column of the tpcf table (with the key being the name of the column and value the entry to be filled in). Here only text id of the packet is extracted from the comment preceding the packet structure in the header file.

Returns
Dictionary which is one line in the MIB table. Assigned to *tpcf*.

Return type
dict

`vpd_listdict()`

Define elements for entries in vpd table.

Creates a list of dictionaries in each of which a key-value pair corresponds to entry in one column of the vpd table (with the key being the name of the column and value the entry to be filled in). As the first step here, the group sizes for each repetition of parameters are calculated using the information in *var_entries*. Then, based on these information, for each of entries identified as repeting/counters in *var_entries*, a row in the vpd table is created.

A weird subcase are fixed repetitions. Here since there has to be some counter for them to be included in the vpd table, an additional "virtual" entry is added to the pcf table which represents this counter but isn't included in any packet.

Returns
List of dictionaries which are to be lines in the MIB table. Assigned to *vpd*.

Return type
list

8.5 mib_generator.construction.TM_packet_methods module

Methods for creation of telemetry MIB tables and internal representation of TM packets in general.

This module holds methods that help with formatting of parsed data into monitoring packet characteristics. They are usually concerned with value evaluation, bite counting, type identification etc. I.e. mostly kind of housekeeping jobs.

mib_generator.construction.TM_packet_methods.**apidnum**(*name*)

Find the value of apid from evaluation of references, etc.

This method is probably unnecessarily complicated because I tried to follow a logic of C here, which seemed to have a missing link anyways. But what it does is simply looking up the value of apid based on its name. This value is stored in an array in the `.c` telemetry file and hence isn't included in the standard *mib_generator.parsing.load.enumerations* and hence has to be evaluated this more complicated way.

Parameters
name (*str*) – Name of the apid reference.

Returns
Value of the apid reference.

Return type
int

mib_generator.construction.TM_packet_methods.**categfromptc**(*ptc*)

Get category from ptc value of the entry.

Extracts the value of category entry in pcf table from the parameter's ptc type.

This is automatically changed later if the parameter is subject to textual calibration.

Parameters
ptc (*int*) –

Returns
The letter identifying the category.

Return type
str

mib_generator.construction.TM_packet_methods.**count_size**(*entries*)

Counts bit size of the packet and bit position of entries within it.

Goes through the given list of parameters, for each one evaluates its bite-size and from it counts the total byte-size of the whole list and bite-positions of start of each parameter within it.

Parameters
entries (*list*) – List of parameters who's length is to be determined. Each of type *mib_generator.parsing.par_header.misc_r*.

Returns

A tuple consisting of:

- *int* - Byte-length of the list of parameters.
- *list* - List o offset positions (in bites) of the parameters from the list start.

Return type
tuple

mib_generator.construction.TM_packet_methods.**evalu**(*string*)

Evaluate the given expression using all known substitutions, macros, etc.

This methods tries to evaluate the passed expression using various methods. First it tries a simple integer conversion, then it tries to evaluate it using the dictionary *mib_generator.parsing.load.enumerations* which stores all the possible substitutions found across the code. Then even if this doesn't work (e.g. the string contains algebraic expressions), tries to use Python's `eval()` function. If even this fails, it evaluates the expression as `-1`.

Parameters
string (*str*) – A string which is to be evaluated.

Returns
The evaluated value or `-1` if the evaluation failed.

Return type
int

mib_generator.construction.TM_packet_methods.**getptcpcf**(*entry*, *size*)

Get ptc and pfc values from the size and nature of the given entry.

From data type (in C) of the entry and information in its comments, tries to deduce what SCOS data type given by the ptc|pcf combination it should be assigned. Uses data from *mib_generator.data.longdata.time_pfc* and *mib_generator.data.longdata.uint_pfc*.

Parameters

- **entry** (parsing.par_header.misc_r) – The entry who's value is to be evaluated.
- **size** (*int*) – Bite-size of the entry.

Returns

A tuple consisting of:

- *int* - Value of ptc of the parameter.
- *int* - Value of pfc of the parameter.

Return type
tuple

mib_generator.construction.TM_packet_methods.**h_analysis**(*h_struct*)

Make a list of all individual (all structs unpacked) entries in the packet.

This method goes iteratively through all entries/packet parameters inside the given structure. If it finds a normal entry, it adds it to the list, if it finds a `struct` or reference to one, it expands it by recursively calling itself and adds all elements in the expansion to the list.

This method is also a precursor to the construction of the vpd tables, since it add to every entry it encounters an attribute `is_vpd`, which identifies whether the given parameter/set of parameters is subject to variable packet definition or not. I also identifies fixed vpd repetitions from the arrays of the parameters and marks them as such.

Parameters
h_struct (parsing.par_header.struct) – The packet structure from which the parameters are to be expanded.

Returns
List of parameters found inside the structure. Each is of type *mib_generator.parsing.par_header.misc_r*.

Return type
list

mib_generator.construction.TM_packet_methods.**header_search**(*typ*, *files*)

Search for corresponding header structures of the packet based on information in the comments.

Given the name of the packet as it appears in the TM `.c` file, this method searches among comments in *mib_generator.parsing.load.TmH* (the TM `.h` files) for a corresponding packet/packets description (list of parameters, etc). More packet definitions can correspond to a single type (they then differ in additional packet identifiers) and hence more than one such structures can be found sometimes.

The comment with the header declaration does not also have to always have a structure directly associated to it (i.e. bellow it), in which it is expected that it includes a `"use_structure"` key which refers to some C-structure which is such structure.

Parameters

- **typ** (*str*) – The "type" entry of the packet definition in the `.c` TM file.
- **files** (*list*) – A list of files (of type `mib_generator.parsing.par_main.file`) in which the header structure
- **for.** (*is to be searched*) –

Returns
A list of pairs of [comments, structures] which are packet descriptions for the given packet "type".

Return type
list

mib_generator.construction.TM_packet_methods.**pi_sid**(*entries*, *positions*)

Extract position of additional identification field from packet entries.

This method looks the presence/position of the first additional identification field (which is recognised by the entry `"sid"` in its comments) in the packet, and if it finds it, calculates its width (in bites) and offset position (in bytes) from the start of the packet.

Parameters

- **entries** (*list*) – List of entries (of type *mib_generator.parsing.par_header.misc_r*) among which the additional identification field is searched for.
- **positions** (*list*) – List of position (bite-offests from the start of the packet) of the parameters in the packet.

Returns

A tuple consisting of:

- *int* - Start position (in bytes from the start of the packet) of the additional identification field.
- *int* - Bite-width of the additional identification field.

Return type
tuple

mib_generator.construction.TM_packet_methods.**var_get**(*entries*)

Extract variable parameters from list of all parameters.

This method goes through all entries in the given list and marks position of the entries which are subject to variable packet definition. If it encounters a case of a fixed repetition, it marks its negation (index of) position instead.

Parameters
entries (*list*) – List of parameters to be searched for vpd. Each of type *mib_generator.parsing.par_header.misc_r*.

Returns
List of indices (or their negations in case of fixed repetitions) at which vpd parameters were found.

Return type
list

8.6 mib_generator.construction.calib module

Methods and classes for construction, representation and calibrating of calibrations/decalibrations/verifications.

This module puts together all classes and function that represent the calibration and verification parts of MIB databases. These representations only collect all information about the structure of the calibrations and its corresponding MIB tables in a structured form (but are not functional themselves).

class mib_generator.construction.calib.**caf_calib**(*comment*)

Bases: *calib*

Class of a numerical calibration.

This class represents a numerical calibration and its corresponding caf and cap tables. It is created from an interpreted comment and is a child-class of *calib* who's initialisation **arguments** and **attributes** it shares.

It hasn't been fully decided yet what the syntax for declaring these calibrations should be, so some of the extraction/construction steps here might not be entirely correct. As of now it is assumed that numerical calibration definition would look analogously to textual calibration (the non-`enum` one).

type

Type of the data (defined through two strings, hence the list) this calibration is used on. Corresponding to the `"CAF_RAWFMT"` and `"CAF_ENGFMT"` MIB entries.

Type
list

length

Number of entries (in integer) that this calibration should be defined by.

Type
str

caf

Dictionary corresponding to one line in MIB caf table.

Type
dict

cap

List of dictionaries each one corresponding to one line in MIB cap table.

Type
list

`caf_data()`

Get information about the nature of data calibrated.

From the entries in the comment, extracts information about the nature of the data to be calibrated and the length of the numerical calibration (number of its entries).

Returns

A tuple consisting of:

- *list* - The type of the data for calibration. See *type*.
- *str* - The number of entries in this calibration. See *length*.

Return type
tuple

`caf_dictionary()`

Define elements for entry in caf table.

Creates a dictionary where each key-value pair corresponds to an entry in one column of the caf table (with the key being the name of the column and value the entry to be filled in). Here the MIB table entries are mostly straightforwardly assigned already identified values.

Returns
Dictionary which is one line in the MIB table. Assigned to *caf*.

Return type
dict

`cap_listdict()`

Define elements for entries in cap table.

Creates a list of dictionaries in each of which a key-value pair corresponds to entry in one column of the cap table (with the key being the name of the column and value the entry to be filled in). The construction here assumes the X-Y data being stored in list of `[X,Y]` pairs under the key `"num_cal"` in the comment declaring this calibration.

Returns
List of dictionaries which are to be lines in the MIB table. Assigned to *cap*.

Return type
list

`class mib_generator.construction.calib.calib(comment)`

Bases: `object`

General class of calibrations/decalibrations.

This class is a parent class of all the various types of interpreted calibrations/decalibrations. It hence holds only very general attributes that all of them require. It is created based on intepreted comment passed at creation.

Parameters
comment (*parsing.par_methods.comment*) – A comment from which the calibration is to be created.

`comment`

The comment from which this calibration was created.

Type
parsing.par_methods.comment

name

The name of the calibration (that is, its internal name, not the identifier name passed into the MIB table).

Type
str

mib_generator.construction.calib.**calib_extract**(*comments*)

Extract declaration of various calibrations if they occur in the comments.

This function goes through inputted list of interpreted comments and looks (based on the entries in them) for ones which hold definition of calibrations. Based on such definition it first decides what calibrations these are and subsequently interprets them through appropriate calibration classes. From all of such interpreted calibration it then creates a dictionary where they are joined in lists by their types.

Parameters
comments (*list*) – List of interpreted comments. Each of type *mib_generator.parsing.par_methods.comment*.

Returns

A dictionary holding lists of interpreted calibrations sorted by their types. Each calibration is of some
child-class of *calib*.

Return type
dict

mib_generator.construction.calib.**cpc_update**(*command*, *dec*)

Check whether decalibration exists for parameter which require it and if yes, change cpc entry correspondingly.

For a given TC command, goes through all of its parameters and for each checks whether its corresponding decalibration (if declared) exists. If yes, then it updates the name of the decalibration in the command parameter to its correct referential value.

Parameters

- **command** (TC_packet.TC_packet) – Command, who's parameters' decalibrations are to be checked and updated.
- **dec** (*list*) – List of all available decalibrations. Each of type *decalib*.

mib_generator.construction.calib.**cur_update**(*packet*, *cal*)

Check whether calibration exists for parameters which require it and if yes, change cur entries correspondingly.

For a given TM packet goes through all of its entries/parameters and for each checks whether its corresponding calibration (if stated) exists. If yes, then it updates the name of the calibration in the packet to its correct referential value.

Initially this method did this for both cases of single and multiple calibration associated to a parameter. However later I focused only on the single case (the "PCF_CURTX" entry of the parameter) so the other case (the entries in cur table associated to the parameter) might not work anymore.

Parameters

- **packet** (TM_packet.TM_packet) – Packet, who's entries' calibrations are to be checked and updated.
- **cal** (*dict*) – A dictionary holding lists of various types of calibration. Each is a child-class of *calib*.

mib_generator.construction.calib.**cvs_update**(*command*, *verifs*)

Check whether verification exists for a given command and if yes, change cvs entry correspondingly.

For a given TC command checks whether its corresponding verification (if declared) exists. If yes, then it updates the name of the verification in the command parameter to its correct referential value, otherwise warning is raised. If no verifications were declared for this command (excluding the case where explicitely no verification were declared, i.e. `[]` was given as the list of verifications), then default selection of verifications is used and corresponding entries in cvp and cvs tables are created.

Parameters

- **command** (TC_packet.TC_packet) – Command, who's verification entries are to be checked and updated.
- **verifs** (*list*) – List of all available verifications. Each of type *verification*.

mib_generator.construction.calib.**decal_extract**(*comments*)

Extract declarations of decalibrations from comments.

This function goes through inputted list of interpreted comments and looks (based on the entries in them) for ones which hold definition of decalibrations. It then interprets these with an appropriate class (*decalib*) and construct a list of them.

Parameters

comments (*list*) – List of interpreted comments. Each of type *mib_generator.parsing.par_methods.comment*.

Returns

A list of all found decalibrations. Each of the type *decalib*.

Return type

list

class mib_generator.construction.calib.**decalib**(*comment*)

Bases: *calib*

Class of a textual decalibration for TC commands.

This class represents a textual decalibration used for TC commands and its corresponding paf and pas tables. It is created from an interpreted comment and is a child-class of *calib* who's initialisation **arguments** and **attributes** it shares.

type

Type of the data this decalibration is used on. Corresponding to the "PAF_RAWFMT" MIB entry.

Type

str

length

Number of entries (in integer) that this calibration should be defined by.

Type

str

paf

Dictionary corresponding to one line in MIB paf table.

Type

dict

pas

List of dictionaries each one corresponding to one line in MIB pas table.

Type
list

paf_data()

Get information about the nature of data calibrated.

From the entries in the comment, extracts information about the nature of the data to be de/calibrated and the length of the textual decalibration (number of its entries).

Returns

A tuple consisting of:

- *str* - The type of the data for calibration. See *type*.
- *str* - The number of entries in this calibration. See *length*.

Return type
tuple

paf_dictionary()

Define elements for entry in paf table.

Creates a dictionary where each key-value pair corresponds to an entry in one column of the paf table (with the key being the name of the column and value the entry to be filled in). Here the MIB table entries are mostly straightforwardly assigned already identified values.

Returns
Dictionary which is one line in the MIB table. Assigned to *paf*.

Return type
dict

pas_listdict()

Define elements for entries in pas table.

Creates a list of dictionaries in each of which a key-value pair corresponds to entry in one column of the pas table (with the key being the name of the column and value the entry to be filled in). Here, since the Python description of `enum` in this code does not support comment associated to its specific entry, it has to be in a way "guessed" what this correspondence is, which is something to keep an eye on.

Returns
List of dictionaries which are to be lines in the MIB table. Assigned to *pas*.

Return type
list

class mib_generator.construction.calib.**lgf_calib**(*comment*)

Bases: *calib*

Class of a logarithmic lgf calibration.

This class represents a logarithmic calibration and its corresponding lgf table. It is created from an interpreted comment and is a child-class of *calib* who's initialisation **arguments** and **attributes** it shares.

lgf

Dictionary corresponding to one line in MIB lgf table.

Type
dict

lgf_dictionary()

Define elements for entry in lgf table.

Creates a dictionary where each key-value pair corresponds to an entry in one column of the lgf table (with the key being the name of the column and value the entry to be filled in). Here the values are extracted from the values in the starting comment with the logarithmic coefficients being extracted in the `for...` loop.

Returns
Dictionary which is one line in the MIB table. Assigned to *lgf*.

Return type
dict

class mib_generator.construction.calib.**mcf_calib**(*comment*)

Bases: *calib*

Class of a polynomial mcf calibration.

This class represents a polynomial calibration and its corresponding mcf table. It is created from an interpreted comment and is a child-class of *calib* who's initialisation **arguments** and **attributes** it shares.

mcf

Dictionary corresponding to one line in MIB mcf table.

Type
dict

mcf_dictionary()

Define elements for entry in mcf table.

Creates a dictionary where each key-value pair corresponds to an entry in one column of the mcf table (with the key being the name of the column and value the entry to be filled in). Here the values are extracted from the values in the starting comment with the polynomial coefficients being extracted in the `for...` loop.

Returns
Dictionary which is one line in the MIB table. Assigned to *mcf*.

Return type
dict

class mib_generator.construction.calib.**txf_calib**(*comment*)

Bases: *calib*

Class of a textual calibration.

This class represents a textual calibration and its corresponding txf and txp tables. It is created from an interpreted comment and is a child-class of *calib* who's initialisation **arguments** and **attributes** it shares.

The unusual thing here is that textual calibration can be declared in the starting C-files in two ways - either wholly in the text of a comment or through a combination of an `enum` and comments. Because of that the creation of representation here is maybe a bit convoluted.

enum

`True` if the calibration is defined through an `enum` and `False` if wholly in one comment.

Type
bool

type

Type of the data this calibration is used on. Corresponding to the `"TXF_RAWFMT"` MIB entry.

Type
str

`length`

Number of entries (in integer) that this calibration should be defined by.

Type
str

`txf`

Dictionary corresponding to one line in MIB txf table.

Type
dict

`txp`

List of dictionaries each one corresponding to one line in MIB txp table.

Type
list

`is_enum()`

Check whether the calibration is saved in the header file as enum.

Based on the content of the initialisation content, checks whether this calibration is defined through `enum` or not.

Returns
`True` if it is defined through an `enum`, `False` if it is defined wholly in a comment.

Return type
bool

`txf_data()`

Get information about the nature of data calibrated.

From the entries either in an `enum` or in the comment, extracts information about the nature of the data to be calibrated and the length of the textual calibration (number of its entries).

Returns

A tuple consisting of:

- *str* - The type of the data for calibration. See *type*.
- *str* - The number of entries in this calibration. See *length*.

Return type
tuple

`txf_dictionary()`

Define elements for entry in txf table.

Creates a dictionary where each key-value pair corresponds to an entry in one column of the txf table (with the key being the name of the column and value the entry to be filled in). Here the MIB table entries are mostly straightforwardly assigned already identified values.

Returns
Dictionary which is one line in the MIB table. Assigned to *txf*.

Return type
dict

txp_listdict()

Define elements for entries in txp table.

Creates a list of dictionaries in each of which a key-value pair corresponds to entry in one column of the txp table (with the key being the name of the column and value the entry to be filled in). Here the process is again complicated by the fact that the there are two ways in which textual calibration can be defined. In case of `enum` it has to be further "guessed" what text-values correspond to which numerical value, which might not be entirely reliable (it is done like this because the Python representation of `enum` that I created does not define the `enum` entries as individual objects).

Returns
List of dictionaries which are to be lines in the MIB table. Assigned to *txp*.

Return type
list

mib_generator.construction.calib.**verif_extract**(*comments*)

Extract declarations of verifications from comments.

This function goes through inputted list of interpreted comments and looks (based on the entries in them) for ones which hold definition of verifications. It then interprets these with an appropriate class (*verification*) and construct a list of them.

Parameters
comments (*list*) – List of interpreted comments. Each of type *mib_generator.parsing.par_methods.comment*.

Returns
A list of all found verifications. Each of the type *verification*.

Return type
list

class mib_generator.construction.calib.**verification**(*comment*)

Bases: `object`

Class of a TC command verification.

This class represents a verification of a TC command and its corresponding cvs table. It is created from an interpreted comment. It is further also identified whether the verification should be applied as a default or not, which is later used when checking/correcting verifications declared for specific commands.

comment

The comment from which this verification was created.

Type
parsing.par_methods.comment

default

`True` if this verification is to be applied as default, `False` otherwise.

Type
bool

cvs

Dictionary corresponding to one line in MIB cvs table.

Type
dict

`cvs_dictionary()`

Define elements for entry in cvs table.

Creates a dictionary where each key-value pair corresponds to an entry in one column of the cvs table (with the key being the name of the column and value the entry to be filled in). Here values from the initialisation comment are straightforwardly assigned.

Returns
Dictionary which is one line in the MIB table. Assigned to *cvs*.

Return type
dict

`is_default()`

Decide whether this verification should be applied as a default.

Checks the verification declaration for an information on whether it should be applied as a default or not. If no such information is found, returns that it shouldn't, otherwise returns the information (as a boolean).

Returns
`True` if this verification is to be applied as default, otherwise `False`.

Return type
bool

CHAPTER

NINE

MIB_GENERATOR.GENERATION PACKAGE

This sub-package and the modules within it serve the purpose of turning previously interpreted and comprehended representations of Tc/Tm packets/etc. and their sketched out MIB tables into propertly constructed and verified (against criteria of type, uniqueness, etc…) MIB tables and finally saving these tables in the correct format into a desired location and documenting the output.

The submodules here are:

- *gener* - The main generation module, chooses what should be generated, what tools should be used for that purpose and executes the generation.
- *gener_methods* - Module holding methods used by the main generation module. These methods do all from checking and converting MIB tables to saving them in the correct location.
- *gener_doc* - Module with methods that create a docx file with the list of entries in Tm and Tc packets from which the MIB tables were generated.

9.1 Submodules

9.2 mib_generator.generation.gener module

This module serves the purpose of generating MIB databases from inputed objects and checking their validity.

Methods in this module generate MIB databases from their representations previously generated by the *mib_generator.construction* package/modules and run finals checks on the results to ensure that the outputted tables/MIB files adhere to the requirements on the type, length, uniqueness and "mandatoriness".

mib_generator.generation.gener.**Tch_gen**(*typ*, *Tc_head*)

Choose how the given Tc-header MIB table should be constructed and construct it.

Based on its name, this function chooses what method to call in order to generate and check the given table. It then calls this method and returns the result.

Parameters

- **typ** (*str*) – Name of the MIB table to be generated.
- **Tc_head** (*list*) – List of Tc-headers from which the table is to be constructed. Each of type *mib_generator.construction.TC_packet.TC_header*.

Returns

A 2D list (list of lists) representing the outputted MIB table (still in Python format).

Return type

list

mib_generator.generation.gener.**Tcp_gen**(*typ*, *Tc_packets*)

Choose how the given Tc-packet MIB table should be constructed and construct it.

Based on its name, this function chooses what method to call in order to generate and check the given table. It then calls this method and returns the result.

Parameters

- **typ** (*str*) – Name of the MIB table to be generated.
- **Tc_packets** (*list*) – List of Tc-packets from which the table is to be constructed. Each of type *mib_generator.construction.TC_packet.TC_packet*.

Returns
A 2D list (list of lists) representing the outputted MIB table (still in Python format).

Return type
list

mib_generator.generation.gener.**Tmp_gen**(*typ*, *Tm_packets*)

Choose how the given Tm-packet MIB table should be constructed and construct it.

Based on its name, this function chooses what method to call in order to generate and check the given table. It then calls this method and returns the result.

The special case here is with respect to the pic table in case of which, first a filter on the packet list have to be run since the entries in the pic table are a subset of those in pid/list of packages and hence otherwise false warning would be raised later (due to lack of uniqueness).

Parameters

- **typ** (*str*) – Name of the MIB table to be generated.
- **Tm_packets** (*list*) – List of Tm-packets from which the table is to be constructed. Each of type *mib_generator.construction.TM_packet.TM_packet*.

Returns
A 2D list (list of lists) representing the outputted MIB table (still in Python format).

Return type
list

mib_generator.generation.gener.**cal_gen**(*typ*, *calibrations*)

Choose how the given calibration MIB table should be constructed and construct it.

Based on its name, this function chooses what method to call in order to generate and check the given table. It then calls this method and returns the result.

Parameters

- **typ** (*str*) – Name of the MIB table to be generated.
- **calibrations** (*list*) – List of dictionaries of calibrations from which the table is to be constructed. Each calibration is of some type which is a child-class of *mib_generator.construction.calib.calib*.

Returns
A 2D list (list of lists) representing the outputted MIB table (still in Python format).

Return type
list

mib_generator.generation.gener.**dec_gen**(*typ*, *decalibrations*)

Choose how the given decalibration MIB table should be constructed and construct it.

Based on its name, this function chooses what method to call in order to generate and check the given table. It then calls this method and returns the result.

Parameters

- **typ** (*str*) – Name of the MIB table to be generated.
- **decalibrations** (*list*) – List of decalibrations from which the table is to be constructed. Each of type *mib_generator.construction.calib.decalib*.

Returns

A 2D list (list of lists) representing the outputted MIB table (still in Python format).

Return type

list

mib_generator.generation.gener.**generation_hub**(*Tm_packets=None*, *Tc_packets=None*, *calibrations=None*, *decalibrations=None*, *verifications=None*, *Tc_head=None*, *cfg=None*)

Take all constructed packets/calibrations/commands and call generation scripts for each table.

This method is the "main interface" for the generation process. It is passed various objects which have MIB tables associated to them and for each of them calls the appropriate generation methods (either `one_generate` for objects which have only one row of the given MIB table associated to them or `two_generate` in case of possibly multiple rows) with the correct parameters.

What tables are generated is determined by the contents of the `../data/config.json5` config file, or it can be overridden using the `cfg` argument. Because of this optionality, none of the arguments of this method are compulsory, since they are not needed for generation of some tables, which is what this method could be asked to do. However they have to be present if a table which is based on them is to be generated.

Parameters

- **Tm_packets** (*list*) – List of TM-packets (their Python representations), each represented by an object of type *mib_generator.construction.TM_packet.TM_packet*.
- **Tc_packets** (*list*) – List of TC-commands (their Python representations), each represented by an object of type *mib_generator.construction.TC_packet.TC_packet*.
- **calibrations** (*dict*) – Dictionary holding lists various calibrations (their Python representations), each calibration being an object of some child-class of *mib_generator.construction.calib.calib*.
- **decalibrations** (*list*) – List of textual decalibrations (their Python representations), each represented by an object of type *mib_generator.construction.calib.decalib*.
- **verifications** (*list*) – List of TC-command verifications (their Python representations), each represented by an object of type *mib_generator.construction.calib.verification*.
- **Tc_head** (construction.TC_packet.TC_header) – A header included in all TC-commands.
- **cfg** (*list*) – List of names of the MIB tables to be generated. Otherwise a default list stated in the config file is used.

Returns

A dictionary holding all the generated mib tables (in form of Python 2D lists) with their names as keys.

Return type
dict

mib_generator.generation.gener.**save_tables**(*tables*)

Reformat and save mib tables in the passed dictionary.

Goes through the passed dictionary holding mib tables and for each first reformats it into an appropriate ASCII string and then saves it to the specified folder with all mib tables.

Parameters
tables (*dict*) – A dictionary containing mib tables in the form of 2D Python lists as values with their names as keys.

mib_generator.generation.gener.**ver_gen**(*typ*, *verifications*)

Choose how the given verification MIB table should be constructed and construct it.

Based on its name, this function chooses what method to call in order to generate and check the given table. It then calls this method and returns the result.

Parameters

- **typ** (*str*) – Name of the MIB table to be generated.
- **verifications** (*list*) – List of verifications from which the table is to be constructed. Each of type *mib_generator.construction.calib.verification*.

Returns
A 2D list (list of lists) representing the outputted MIB table (still in Python format).

Return type
list

9.3 mib_generator.generation.gener_doc module

This module enables generation of a `.docx` file which documents in a human-readable way what was put in MIB tables for a given set of Tm and Tc packets.

The only (important) method in this module builds up the `.docx` file using the not so powerful `python-docx` library.

mib_generator.generation.gener_doc.**gen_doc**(*Tm_packets=[]*, *Tc_packets=[]*)

Create a `.docx` document which sketches out what entries are in each of the Tm and Tc packets in the passed list.

This method goes through all passed Tm and Tc packets and for each adds an entry to the created `.docx` document including basic info about the packet and a table of entries within it. The format of the output tries to adhere to a standard scheme of an ICD document used normally for these purposes.

Parameters

- **Tm_packets** (*list*) – List of Tm-packets, each of type *mib_generator.construction.TM_packet.TM_packet*, from which the document is to be generated.
- **Tc_packets** (*list*) – List of Tc-packets, each of type *mib_generator.construction.TC_packet.TC_packet*, from which the document is to be generated.

mib_generator.generation.gener_doc.**to_bytes**(*bits*)

Give the number of bits in bytes and in some meaning full format.

Calculates how many bytes are in the passed bits, how many residual bits, and puts together this information into a meaningful-looking string.

Parameters
bits (*int*) – Numeber of bits to be "translated".

Returns
The number of bits in bytes and residual bits in understandable form.

Return type
str

9.4 mib_generator.generation.gener_methods module

Methods for generation of MIB tables and checking of their validity.

This module holds methods that help with generation, checking and saving of MIB tables from previously created Python representations of Tm/Tc packets, calibrations, etc..

mib_generator.generation.gener_methods.**check**(*value*, *typ*)

Check whether the given value matches the given type.

This method checks whether the passed value matches the passed type (either integer number or ASCII string without a few characters).

Parameters

- **value** (*str*) – Value who's type is to be checked.
- **typ** (*str*) – The type the value should have.

Returns
`True` if the value matches the type, `False` otherwise.

Return type
bool

mib_generator.generation.gener_methods.**delete_empty**(*table*)

Delete empty entries in arrays.

This function goes through list of dictionaries to be made into MIB tables, and removes every empty (equal to "") entry it finds. This is done so that the later tests trigger appropriate warnings.

Parameters
table (*list*) – List of dictionaries, each one representing a row of an MIB table, which is to be cleaned from empty entries.

mib_generator.generation.gener_methods.**exclude_repetition**(*table*, *typ*)

Exclude repetition in columns where there should be unique entries.

This function checks whether there is any repetition in columns where entries should be unique in the passed MIB table and if so, deletes the additional rows and raises a warning. (the required uniqueness of entries for a given MIB table is defined in *mib_generator.data.longdata.unique_entries*)

Parameters

- **table** (*list*) – List of dictionaries which represents the MIB database to be checked for repetition.
- **typ** (*str*) – The name of the MIB database the passed table adheres to.

mib_generator.generation.gener_methods.**generate**(*table_type*, *source*)

From a list of dictionaries (per rows) generate a MIB table of a given type.

This function generates and saves the MIB database in the following steps:

1. Based on the passed MIB table name, looks up what entries/columns the generated table should have (it takes this information from *mib_generator.data.longdata.tables_format*).
2. From the passed list of dictionaries it creates a 2D list (list of lists) mirroring the structure of the MIB table.
3. While doing that it runs some checks of type and uniqueness of the entries.

Parameters

- **table_type** (*str*) – The name of the MIB database the passed table adheres to.
- **table** (*list*) – List of dictionaries which correspond to the rows of the MIB table with the appropriate entries.

Returns
A 2D list (list of lists) representing the generated MIB table

Return type
list

mib_generator.generation.gener_methods.**list_to_mib**(*lis*)

Convert a 2D list of values into a string formatted as MIB ASCII file.

From a 2D list (list in list) this functions creates one string by joining all the entries with a junction \n for entries in the "1D" list and with \t for the "nested" entries.

Parameters
lis (*list*) – A 2D list (list of lists) to be joined into one string.

Returns
A string which is created from the 2D list and is formatted as an MIB database.

Return type
str

mib_generator.generation.gener_methods.**mnemon_check**(*packets*, *typ*)

Check whether mnemonics for the set of passed packets and the given table are unique and existent.

This follows additional mib specifications by OHB Italia, which requires special mnemonic parameter to be saved into the description field for a few mib tables. The mnemonics are specified to be unique and present, which is what this method checks and raises warning if they aren't.

Parameters

- **packets** (*list*) – List of packets/commands. Each either of type *mib_generator.construction.TM_packet.TM_packet* or *mib_generator.construction.TC_packet.TC_packet*.
- **typ** (*str*) – The name of the MIB table for which this is checked.

mib_generator.generation.gener_methods.**one_generate**(*lists*, *name*)

Generate MIB database of the given name from the passed list of objects.

This method does a few precursor steps to the generation process. For each of the objects in the passed list, it extracts the appropriate dictionary corresponding to the row in the MIB table associated to the object and appends this dictionary (alias row) to a list of all rows for this table. It then passes the resulting list through a "filter" which erases all entries left empty (method *delete_empty*) and calls the main generation method *generate* with this list.

Parameters

- **lists** (*list*) – List of the objects (these can be of many types) the rows of the MIB tables (as dictionaries) are associated to.
- **name** (*str*) – Name of the MIB table that is to be generated from these objects.

Returns
A 2D list (list of lists) representing the generated MIB table

Return type
list

mib_generator.generation.gener_methods.**pic_filter**(*packets*)

Check whether the information saved in the list of Tm-packet is self-consistent and prune it for generation of pic.

This function is an ad-hoc filter, that accounts for the fact that pic table has to have fewer entries than pid or other similar tables because of the fact that one entry in it corresponds only to one combination of packet type/subtype. However for a set of packets to be reducible to this single entry in pic, it has to be the case that they all share the same declaration of additional identification fields which set them apart but have the same position relative to the packet structure. This is hence also something that this filter checks and raises a warning whenever it encounters something suspicious. Finally it deletes all (valid) repetition in types/subtypes it finds, so that pic table can be directly generated from the output list of packets without raising further exceptions.

Parameters
packets (*list*) – List of representations of Tm-packets, each of type *mib_generator.construction.TM_packet.TM_packet*. This list corresponds to the pid table.

Returns
List of representations of Tm-packets, each of type *mib_generator.construction.TM_packet.TM_packet* This list corresponds to the pic table.

Return type
list

mib_generator.generation.gener_methods.**save_mib**(*mib*, *name*)

Save the string-MIB database into the specified folder.

This function saves the passed string into a file named with the passed name (+.dat ending) at the location specified in `mib_generator.parsing.load.out_dir`.

Parameters

- **mib** (*str*) – The string to be saved into a file
- **name** (*str*) – Name of the file (+.dat) the string will be saved into.

mib_generator.generation.gener_methods.**two_generate**(*lists*, *name*)

Generate MIB database of the given name from the passed list of objects.

This method does a few precursor steps to the generation process. For each of the objects in the passed list, it extracts the appropriate dictionaries (in a list) corresponding to the rows in the MIB table associated to the object and appends these dictionaries (alias rows) to a list of all rows for this table. It then passes the resulting list through a "filter" which erases all entries left empty (method *delete_empty*) and calls the main generation method *generate* with this list.

Parameters

- **lists** (*list*) – List of the objects (these can be of many types) the rows of the MIB tables (as dictionaries) are associated to.
- **name** (*str*) – Name of the MIB table that is to be generated from these objects.

Returns
A 2D list (list of lists) representing the generated MIB table

Return type
list

CHAPTER

TEN

MIB_GENERATOR.DATA PACKAGE

The files and submodules in this package/folder serve as a storage of data and global-level variables which are mostly stored here in order not to cluster the code somewhere else and in order to be easily accessible by any other file.

The files/submodules here are:

- *longdata* - This giant module stores mostly all information about the structures of MIB tables and what checks should be performed by each entry in them.
- *warn* - Module storing definition of various warnings and errors that can be raised by the program.
- `paths.json5` - This file stores default information about paths to the input/output files in a json5 format.
- `config.json5` - This file stores default config information which mostly include what pre-processor macros should be taken as defined when parsing the inputted files and what mib tables should be generated.

10.1 Submodules

10.2 mib_generator.data.longdata module

Here various data that would otherwise cluster the code elsewhere are stored.

This module includes mostly information about the structures of the MIB tables and characteristics that are specified for each entry in them. For each MIB table, there is an module-wide attribute here that through a list of dictionaries specifies, what the names and characteristics (type, length, whether mandatory) entries in each column of this table should have. All of these lists are then joined in one dictionary *tables_format* from which any characteristics of MIB tables can be extracted.

`mib_generator.data.longdata.`**`pid`**

List of dictionaries each entry of which corresponds to information about an entry in one column in the pid table. Lists like this are stored in this module for every MIB table.

Type
list

`mib_generator.data.longdata.`**`sizes`**

A lookup dictionary which links every number constant type with its bite-size.

Type
dict

`mib_generator.data.longdata.`**`tables_format`**

A dictionary created from list like *pid* above specified for each MIB tables. Allows for easy access to these information.

Type
dict

mib_generator.data.longdata.**unique_entries**

A dictionary which for each MIB table (specified by its name) states what colums or groups of columns should hole unique/non-repeating entries (e.g. there should be only one entry in pid column "PID_TYPE" for a packet of given type.).

Type
dict

mib_generator.data.longdata.**uint_pfc**

List that states what pfc type (given by the order in the list) should be linked to a integer constant of a given C-type.

Type
list

mib_generator.data.longdata.**time_pfc**

List that states what pfc type (given by the order in the list) should be linked to a time-describing constant of a given allocation of bites for fine/course time.

Type
list

mib_generator.data.longdata.**translation**

A lookup dictionary stating what descriptions should be given to each attribute of the Python representations of C-objects in order to make the entries more human-readable in the GUI visualisation.

Type
dict

10.3 mib_generator.data.warn module

Takes care of warning and errors that can be raised by the rest of the program.

This module holds both definition of all the errors and warnings that can occur throughout and a method that can be used to raise them. (This is made this centralised so that the raising method can be easily changed later if e.g. the UI changes.)

mib_generator.data.warn.**warnings**

A dictionary containing the text definitions of all the possible warnings.

Type
dict

mib_generator.data.warn.**errors**

A dictionary containing the text definitions of all the possible errors.

Type
dict

mib_generator.data.warn.**complete**

A dictionary containing the text definitions of all the possible completion messages.

Type
dict

mib_generator.data.warn.**display**

Defines (or holds the object) where the warnings should be raise.

Type
None or some object

mib_generator.data.warn.**disp_update**(*var*)

Update the *display* global variable in this module to the specified value.

This quite ugly method (or approach) serves the purpose of specifying to the method that takes care of raising warnings, that the warnings shouldn't be printed to terminal but rather added to the object's text (the passed object is expected to be some GUI console).

Parameters
var (*object such as PySide6.QtWidgets.QTextEdit*) – The object to be assigned to the *display* global variable.

mib_generator.data.warn.**raises**(*ID*, **data*)

Raise warning/error with the given ID.

This method looks up a warning/error text for a given passed ID, formats it with other passed parameters and then displays it - either to terminal if the global attribute *display* is `None` or to the object that this attribute holds..

Parameters

- **ID** (*str*) – The ID identifying the warning/error of format 3 capital letters + one-digit number.
- ***data** (**args*) – Additional parameters which are to be formatted into the warning/error text.

CHAPTER

ELEVEN

MIB_GENERATOR.TEMP PACKAGE

Sub-package taking care of runtime configuration.

This sub-package/folder serves the role of a location where config files are stored at the runtime and from where all other parts of the program ask for them. It also holds module which takes care of transfer of these files.

The files/sub-modules here are:

- *temp* - Module holding methods which manipulate the config files and transfer them here if needed.
- `paths.json5` - This file stores runtime information about paths to the input/output files in a json5 format.
- `config.json5` - This file stores runtime config information which mostly include what pre-processor macros should be taken as defined when parsing the inputted files and what mib tables should be generated.

11.1 Submodules

11.2 mib_generator.temp.temp module

This module holds method/s which take care of transferring the config files between directories and getting their data.

At runtime, the whole program works with config files which are stored in the *mib_generator.temp* sub-package/file, which means that before all of the generation processes happen, the config files from the default directory (*mib_generator.data*) have to be moved there, which is one of the jobs of this module, the others being the same inversed, loading of config data, etc.

`mib_generator.temp.temp.`**`evom_conf`**`(`*dire*`)`

Cope the runtime config files to the specified directory.

This module is passed a directory location, it check whether it exists and if yes, copies the current runtime config files to there.

Parameters
dire (*str*) – A path to directory where the config files are to be copied.

`mib_generator.temp.temp.`**`fetch_paths`**`()`

Get paths dictionary from the runtime config file.

Loads the paths config file in the runtime directory (the *mib_generator.temp* sub-package) and extracts the paths to input/output files from it as a dictionary.

Returns
A dictionary containing the paths to input/output files/directories.

Return type
dict

mib_generator.temp.temp.**move_conf**(*dire=None*)

Transfer the config files from the specified directory to runtime location.

This method looks at the passed directory, checks what config files are inside it and if such are found, moves them to the package runtime directory (*mib_generator.temp*), rewriting config files which might have been there already. If some config file is not found in the specified directory, transfers config from the default directory (*mib_generator.data*) instead.

Parameters
dire (*str*) – A string specifying the directory from which the config files are to be taken.

mib_generator.temp.temp.**update_json**(*typ*, *data*)

Update the runtime config file with the passed parameter.

This method takes a config parameter with value `data` and of type/name `typ`, loads the runtime config file, incorporates this parameter to it and then saves it again.

Parameters

- **typ** (*str*) – The type/name of the config parameter to be saved.
- **data** (*str*) – The contents of the parameter to be saved.

mib_generator.temp.temp.**update_paths**(*diction*)

Update paths in the "`paths.json5`" config file to the passed values.

Goes through each entry in the passed dictionary and if it recognises it as valid configuration path, it saves it to the runtime config file. If it is given a path to file which doesn't exists, it raises a warning but saves it nonetheless.

Parameters
diction (*dict*) – A dictionary containing paths to the input/output files/directories.

Returns
A dictionary of the values saved. (currently the same as the inputted dictionary)

Return type
dict

CHAPTER

TWELVE

MIB_GENERATOR.UTILITIES PACKAGE

These modules serve various side roles which are not essential to the functioning of the main generation scipt but make user interaction with it more straightforward, automatic and pleasant.

The submodules here are:

- *data_gen* - Module that helped automate inputting all MIB tables parameters into the *mib_generator.data.longdata* file/method. Unused now.
- *update* - Module holding methods that create a small CLI interface (accessible from terminal if the main script is run with appropriate flags) that allows the user to update the configuration options and input/output paths.
- *visualiser* - Module that provides a GUI representation of the Python objects created by parsing the inputted files. Useful mostly for checking the correct functioning of the parser.

12.1 Submodules

12.2 mib_generator.utilities.data_gen module

This is just a quick script that helped me automate translating the information about structure and characteristics of MIB tables into Python (now stored in *mib_generator.data.longdata*).

mib_generator.utilities.data_gen.**gen_data**()

This method is an endless loop that asks the user to input various information which are then formatted into a string which is a literal representation of a Python dictionary and this string is then appended to a text file `out.txt`.

12.3 mib_generator.utilities.update module

Various CLI scripts that update various files that configure the generation process.

This module holds methods that allow the user to easily modify various files that affects the behaviour of the main Python script. They can be called by simply importing and calling them or through appropriate flags build into the main CLI of the MIB generator.

mib_generator.utilities.update.**update_config_d**(*directory=None*)

Run a series of queries asking user to specify parsing configuration parameters.

This method allows the user to specify configuration parameters to be used by the parsing pre-processor (i.e. say, whether the macro is defined or not). First, the already saved parameters are loaded and the user is asked

whether he wants to keep the current value, change it or delete the parameter altogether. Then, the user is given the option to create a new parameter. The only valid accepted of parameters are boolean or a string.

All of this happens with respect to either a config file specified through the passed `directory` parameter or w.r.t. the default config file which is located in *mib_generator.data*.

Parameters
directory (*str*) – String with the location of the directory in which the config file to be modified is located.

mib_generator.utilities.update.**update_config_m**(*directory=None*)

Run a series of queries asking user to specify generation configuration parameters.

This method allows the user to specify configuration parameters to be used at the generation step (i.e. the list of the MIB databases to be generated). First, the already saved parameters are loaded and the user is asked whether he wants to keep, change or delete them. Then, the user is given the option to create a new parameter. The only valid accepted of parameters are strins.

All of this happens with respect to either a config file specified through the passed `directory` parameter or w.r.t. the default config file which is located in *mib_generator.data*.

Parameters
directory (*str*) – String with the location of the directory in which the config file to be modified is located.

mib_generator.utilities.update.**update_config_n**(*directory=None*)

Run a series of queries asking user to specify parameter name creation configuration parameters.

This method allows the user to specify configuration parameters to be used at the parameter construction step (i.e. what form should the names of the parameters take. There are four parameters which define these settings and the user is asked about the value of each of them which he can either change or leave the previous.

All of this happens with respect to either a config file specified through the passed `directory` parameter or w.r.t. the default config file which is located in *mib_generator.data*.

Parameters
directory (*str*) – String with the location of the directory in which the config file to be modified is located.

mib_generator.utilities.update.**update_path**(*directory=None*)

Run a series of queries asking user to specify valid paths to input files.

This method allows the user to specify paths to the 4 input files and 1 output directory that the MIB generator requires. The previously stored values are shown to the user and he can leave them be or choose to modify them stating the location of the target files either in terms of absolute or relative path. The inputted location is then checked and if it exists, then the path is saved (in absolute form). Only existence is checked, not that the file is valid for the given purpose.

All of this happens with respect to either a paths config file specified through the passed `directory` parameter or w.r.t. the default paths config file which is located in *mib_generator.data*.

Parameters
directory (*str*) – String with the location of the directory in which the paths config file to be modified is located.

12.4 mib_generator.utilities.visualiser module

Allow a GUI representation of the parsed data.

This module allows the user to see the output of the parsing process, i.e. the tree of the Python objects that were created as a representation of the original C-files. For the creation of the GUI the Qt framework is used in connection with its Python toolkit/bindings PySide-6.

class mib_generator.utilities.visualiser.**CommentWindow**(*comment*)

Bases: QWidget

staticMetaObject = PySide6.QtCore.QMetaObject("CommentWindow" inherits "QWidget":)

view_text(*text*)

Show a window with full text.

This methods when called opens a small window showing the passed text.

Parameters
text (*str*) – The text to be shown.

class mib_generator.utilities.visualiser.**MainWindow**(*struct*)

Bases: QMainWindow

staticMetaObject = PySide6.QtCore.QMetaObject("MainWindow" inherits "QMainWindow":)

class mib_generator.utilities.visualiser.**TextWindow**(*text*)

Bases: QWidget

staticMetaObject = PySide6.QtCore.QMetaObject("TextWindow" inherits "QWidget":)

class mib_generator.utilities.visualiser.**Ui_MainWindow**

Bases: object

This class holds functions that build up geometry and objects in the main window.

In case of this application, the UI has to be dynamically created, i.e. the number of entries in a toolbox has to change depending on the number of structures that need to be displayed. Because of this, this class which holds the description of the UI of the main visualisation window is perhaps more complicated than usual. Its base layout is created in *setupUi*, but this includes a recursive reference to *inter*, which always creates a visualisation of contents of some object (i.e. shows a list of some objects as entries in a toolbox). Since structures can be members of structures, *inter* can also call itself, hence the recursivity.

inter(*struct*)

Recursively create entries for each C-object in the parsed file.

For any passed parsed object, creates an appropriate representation through a Qt Widget. Since such object can also be a `struct`, this function is sometimes called recursively. For some objects it also creates buttons which can open other windows which visualise associated comments or e.g. long section of the original C-file.

Parameters
struct (*child-class of* parsing.par_header.structure *or* parsing.par_cfile.instance_og) – The object to be visualised by the custom-created Qt Widget.

Returns
Qt widget representing the passed object.

Return type
QFrame

setupUi(*MainWindow*, *struct*)

Builds up the UI of the Main window.

This is done dynamically based on the nature of the passed file representation. See *Ui_MainWindow* for more info.

Parameters

- **MainWindow** (*QMainWindow*) – Base Qt object in which this UI is build.
- **struct** (parsing.parser_main.file) – The Python representation of a file on basis of which the UI will be constructed.

view_comments(*comment*)

Show a window with comments.

This method when called opens a window showing the passed comment.

Parameters
comment (*list*) – List of comments to be shown in the new window. Each of type *mib_generator.parsing.par_methods.comment*.

view_text(*text*)

Show a window with full text.

This method when called opens a small window showing the passed text.

Parameters
text (*str*) – The text to be shown.

mib_generator.utilities.visualiser.**getdata**(*struct*)

Extract all posible showable (str, int) data about an object

From a list of attributes that the passed object has, extracts the interesting ones (which are not buildins for any Python object) and if they have showable value (their value is either `str` or `int` and it isn't the whole text of the corresponding C code found in the original C-file), adds this attribute-value pair to a list. It also tries to substitute the name of the attribute with a more human-readable name by looking it up in *mib_generator.data.longdata.translation*.

Parameters
struct (*any Python object really*) – The object who's attributes are to be extracted.

Returns
A list holding all "interesting" attributes of the original object jointly with their values.

Return type
list

mib_generator.utilities.visualiser.**intable**(*table*, *data*)

Add the given data to the given table.

This method takes a table (its representation as a Qt widget) and inputs the passed data into its rows and columns. It expects the inputted data to be a list of pairs.

Parameters

- **table** (*QTableWidget*) – The Qt table to which the data is to be added.
- **data** (*list*) – List of pairs representing the data to be added to the table.

mib_generator.utilities.visualiser.**main**(*struct=[None]*)

For each file create a process and run the visualiser in it.

This method creates (or rather starts the process of creating) a visualisation window for each of the files in the passed list. In order to do this it has to create a separate new process for each of the files since otherwise closing the windows would exit the whole Python runtime.

mib_generator.utilities.visualiser.**runt**(*struct*)

Run the app in one process.

This method first creates the UI based on the passed file representation and then runs it as a Qt app.

At the exit, this method kills the whole process so it should be isolated into separate process if it is only called by some additional code which is supposed to continue running.

Parameters

struct (parsing.parser_main.file) – The Python representation of the file on basis of which the visualisation will be constructed.

CHAPTER

THIRTEEN

MIB_GENERATOR.GUI PACKAGE

This sub-package provides a simple GUI for the MIB Generator.

The GUI is utilised through the Qt6 framework joint with its Python bindings PySide6. The files in this folder/sub-package hence follow the format of Qt Creator project. The GUI is however not completely isolated from the rest of the `mib_generator` package and parts of the logic unique to the GUI are also found e.g. in the *mib_generator.data.warn* or *mib_generator.temp.temp* modules.

The modules/files here are:

- *gui* - A module holding the declaration and logic of the GUI as well as its initialisation function.
- `form.ui` - An XML file holding description of all objects in the GUI, their layout, connections, etc.
- *ui_form* - A module holding description of the GUI as Python objects. Automatically generated from the `form.ui` file.
- `mib_gui.pyproject` and `mib_gui.pyproject.user` - Files associated with the Qt Creator project which makes up the GUI.

13.1 Submodules

13.2 mib_generator.gui.gui module

The main GUI module.

This module initialises and runs the GUI for the program which uses the Qt framework. The majority of the GUI structure is declared in the *mib_generator.gui.ui_form* module and here the main-window class *MainWindow* contains mostly the logic interconnecting various parts of the GUI and their connections to the main program itself (the documentation of *MainWindow* is not in Sphinx right now because of a bug in PySide6).

class mib_generator.gui.gui.**MainWindow**(*parent=None*)

Bases: `QMainWindow`

Cfile(*field*)

Open a dialog for choosing files and use the output.

Opens a system dialog for choosing files and lets the user choose an input `.c/.h` files through it. It then assigns the paths to the chosen locations to the passed text field.

Parameters

field (*PySide6.QtWidgets.QPlainTextEdit*) – A text window to which the outputted location is to be filled.

DOCgen()

Generates a `.docx` document summing up the results of the construction process.

Takes the list of representations of TC and TM packets and generates an entry in a `.docx` document based on each of them. It then assigns this value to the `docum` attribute.

DOCsave()

Save the generated `.docx` document .

Looks up the location where the document should be saved to and saves it ther.

Dfile()

Open a dialog for choosing a file and use the output.

Opens a system dialog for choosing file and lets the user choose an input `.docx` file through it. It then assigns the path to the chosen location to the `.docx` path text field.

MIBgen()

Generates MIB tables based on all computation done so far.

With all so far constructed representations (Tm packets, Tc commands, calibrations, etc…) taken into account as well as used config settings, generates the corresponding MIB tables. Also adjusts the GUI appropriately based on the result and saves the generation output to the `tables` attribute.

MIBsave()

Save the generated MIB tables to `.dat` files.

Looks up the directory where the tables should be saved to in the runtime config files and saves them there.

TcHshow()

Show GUI representations of parsed `Tc .h` file.

Opens new window with visual representation of what the parser found/interpreted inside the `Tc .h` file.

TcTmHshow()

Show GUI representations of parsed `TcTm .h` file.

Opens new window with visual representation of what the parser found/interpreted inside the `TcTm .h` file.

Tcinter()

Interpret the Tc packets/commands found in the parsed files.

This method runs the code from the *mib_generator.construction* module which interprets the parsed files relating to TC packets and constructs appropriate Python representations of the included objects from them. It updates the list of decalibrations `dec`, verifications `ver`, the list of TC packets/commands `tcs` and the TC header `TcHead`. It also enables buttons for subsequent computation if everything runs correctly.

TmCshow()

Show GUI representations of parsed `Tm .c` file.

Opens new window with visual representation of what the parser found/interpreted inside the `Tm .c` file.

TmHshow()

Show GUI representations of parsed `Tm .h` file.

Opens new window with visual representation of what the parser found/interpreted inside the `Tm .h` file.

Tminter()

Interpret the Tm packets found in the parsed files.

This method runs the code from the *mib_generator.construction* module which interprets the parsed files relating to TM packets and constructs appropriate Python representations of the included objects from

them. It updates the list of calibrations `cal`, list of TM packets `tms` and the TM header `TmHead`. It also enables buttons for subsequent computation if everything runs correctly.

clean(*stri*, *tok={'\t', '\n', ' ', '"', '"""}*)

Clean the passed string of the passed tokens.

Replaces all tokens in the passed strings with nothing (deletes them). By default with no tokens passed, deletes all whitespace characters in the string.

Parameters

- **stri** (*str*) – String which is to be cleaned.
- **tok** (*set*) – Set of tokens to be deleted from the string.

Returns
The string after cleaning.

Return type
str

compute()

Click all computation buttons in sequential order.

This links all the computation to one button, be clicking of which the user effectively clicks all of the computational buttons (if they are disabled, then nothing happens) sequentially.

def_config()

Fill in the path to default config directory to the config directory field.

Gets path to the default config location (it is the *mib_generator.data* sub-package/directory) and fills it to the config directory input text field.

direc(*field*)

Open a dialog for choosing a directory and use the output.

Opens a system dialog for choosing a directory and lets the user choose an output directory through it. It then assigns the path to the chosen location to the passed text field.

Parameters
field (*PySide6.QtWidgets.QPlainTextEdit*) – A text window to which the outputted location is to be filled.

dist_paths(*dic*)

Distribute input/output paths across various GUI fields.

This method takes the paths specified in the passed dictionary and based on their keys, assigns them to appropriate input text across the GUI.

evalu(*stri*)

Evaluates string for boolean values.

Just simple case matching of string to a boolean values.

Parameters
stri (*str*) – The string to be evaluated.

Returns
The result of evaluation.

Return type
bool

load_config()

Load all data from config files in the specified directory.

This method first looks for config files in the specified directory, if it finds any, it copies them to the runtime config directory and from there loads their content to all the appropriate text fields in the GUI.

mib_use()

Use the config settings for mib tables generation.

This method saves the config setting for mib tables generation (filled out in the GUI input field) by copying them to the runtime config files.

nam_use()

Use the config settings for parameter name creation.

This method saves the config setting for parameter name creation (filled out in the GUI input field) by copying them to the runtime config files.

parse()

Runs the parsing logic and modifies the GUI correspondingly.

This module runs all the parsing from the *mib_generator.parsing.load* module and then depending on the results for each file either enables or disables GUI buttons that would run subsequent computation or show the parsing results. Also raises warnings/errors if anything fails.

pre_use()

Use the config settings for pre-processor parsing.

This method saves the config setting for per-processor parsing (filled out in the GUI input field) by copying them to the runtime config files.

save_config()

Saves all the currently filled out information to a directory.

This method first applies all the present (filled out in various GUI fields) information by saving them to the runtime config files and them copies these files to the directory specified in the config directory field.

```
staticMetaObject = PySide6.QtCore.QMetaObject("MainWindow" inherits "QMainWindow":
Methods:  #40 type=Slot, signature=compute() #41 type=Slot, signature=parse() #42
type=Slot, signature=TmHshow() #43 type=Slot, signature=TcHshow() #44 type=Slot,
signature=TmCshow() #45 type=Slot, signature=TcTmHshow() #46 type=Slot,
signature=Tminter() #47 type=Slot, signature=Tcinter() #48 type=Slot,
signature=MIBgen() #49 type=Slot, signature=DOCgen() #50 type=Slot,
signature=MIBsave() #51 type=Slot, signature=DOCsave() #52 type=Slot,
signature=Cfile() #53 type=Slot, signature=direc() #54 type=Slot, signature=Dfile()
#55 type=Slot, signature=use_paths() #56 type=Slot, signature=def_config() #57
type=Slot, signature=load_config() #58 type=Slot, signature=save_config() #59
type=Slot, signature=mib_use() #60 type=Slot, signature=pre_use() #61 type=Slot,
signature=nam_use() )
```

to_field(*stri*)

Convert the passed string/list to an easily readable format.

Takes an inputted string/list and based on its outlook formats it so that it is easily readable/editable by the user (cleans it of redundant whitespaces, adds separators, new lines).

Parameters

stri (*str or object*) – An object/string to be reformatted.

Returns
The reformatted string.

Return type
str

use_paths()

Use the paths specified in the input field.

Gets all the contents of the paths input fields, saves them to the runtime config files found in the *mib_generator.temp* sub-package/directory and loads them. Raises various warnings if anything goes wrong.

mib_generator.gui.gui.**gui_run**()

Run the GUI.

This method runs the GUI as a Qt application, which means that it is "self-contained" and upon closing it also ends the thread in which it was run (hence multiprocessing has to be used in order to run it as part of some larger program).

13.3 mib_generator.gui.gui_methods module

13.4 mib_generator.gui.ui_form module

class mib_generator.gui.ui_form.**Ui_MainWindow**

Bases: object

retranslateUi(*MainWindow*)

setupUi(*MainWindow*)

CHAPTER

FOURTEEN

INDICES

- genindex
- modindex

PYTHON MODULE INDEX

m

INDEX

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W