Analysis Suite go2signals

Manual

Python Decoder Development

Product Version v25.2

July 4, 2025

PROCITEC®
HOUSE OF SIGNALS

Imprint

PROCITEC GmbH
Rastatter Straße 41
D-75179 Pforzheim
Germany

Phone: +49 7231 15561 0
Fax: +49 7231 15561 11
Email: service@procitec.com
Web: www.procitec.com

CEO:
Dipl.-Kaufmann / M. Sc. Stefan Haase

Registration Court:
HRB 504702 Amtsgericht Mannheim
Tax ID:
DE 203 881 534

Document ID:
PROCITEC-IMA-pyDDL_E-45ba033524

Contents

1. Manual **1**
1.1. Overview 1
1.2. Tutorial for Writing Decoders 2
1.2.1. Support Module `hng_fec_alphabet.py` 2
1.2.2. Main Decoder Module `hng_fec_dec.py` 3
1.2.2.1. Preface and General Structure 3
1.2.2.2. Decoder Main Function 4
1.2.2.3. Code Structure Alternatives 6
1.3. Decoder Development Using Spyder 7
1.3.1. Creating and Using a New Decoder 7
1.3.2. Editing an Existing Decoder 8
1.3.3. Running a Decoder 9
1.3.3.1. Execution Mode *Decoder only* 10
1.3.3.2. Execution Mode *Signal processing and decoder* 11
1.3.3.3. Decoder Parameters 12
1.3.4. Examining and Saving Decoder's Output 12
1.3.5. Debugging a Decoder 14
1.3.5.1. Examining variable values 16
1.3.5.2. Graphical Display of a BitBuffer 18
1.3.6. Profiling a Decoder 18
1.3.7. Known Spyder Issues 19
1.3.7.1. OpenGL Error 19
1.4. Using Alternative IDEs 21
1.5. Customizing Python environments 22
1.5.1. Adding new packages 22
1.5.2. Non-priviliged package installation in Spyder 23
1.5.3. Decoder module/package inspection in Spyder 24
1.6. Using DLLs/Shared Libraries 25
1.6.1. Creating Shared Libraries 25
1.6.2. Loading External Libraries 26
1.6.3. Calling Function from Loaded Libraries 26
1.6.3.1. Passing Scalar Values by Value and by Pointer 26
1.6.3.2. Passing Arrays 27
1.6.3.3. Passing an array of pointers (multidimensional arrays) 28
1.6.3.4. Using Structures 29
1.6.3.5. Retaining Independent States and Using Classes 30
1.6.4. Using Libraries Created for Non-Python DDL 31

1.7. Tools for testing, executing and packaging of decoders . . . 32
1.7.1. Executing a decoder . . . 32
1.7.1.1. Using a Python script . . . 32
1.7.1.2. Using the command line . . . 33
1.7.2. Bundle a decoder into a decoder package . . . 33
1.7.2.1. Using a Python script . . . 33
1.7.2.2. Using the command line . . . 34

2. Reference . . . 35
2.1. Decoder Runtime . . . 35
2.1.1. Top-Level API . . . 35
2.1.2. Input . . . 41
2.1.3. Output . . . 43
2.1.4. State and Parameters . . . 46
2.1.5. File Output . . . 60
2.1.6. Audio Output . . . 63
2.1.7. Standalone runtime . . . 64
2.2. BitBuffer . . . 66
2.2.1. BitBuffer . . . 66
2.2.2. BitStream . . . 80
2.2.3. Helper Functions . . . 83
2.2.4. Shift Operations . . . 87
2.3. Decoding Library . . . 88
2.3.1. Synchronisation and Search . . . 89
2.3.2. Error Correction and Detection . . . 97
2.3.3. Burst Operations . . . 115
2.3.4. Pre-Processing . . . 118
2.3.4.1. Bit-Level Pre-Processing . . . 118
2.3.4.2. Symbol-Level Pre-Processing . . . 119
2.3.4.3. Utilities . . . 120
2.3.5. Alphabets . . . 120
2.3.6. Utilities . . . 129
2.3.7. File Output Helpers . . . 139
2.3.8. Bit Formatting . . . 140
2.3.9. Encoding . . . 142
2.3.10. Miscellaneous . . . 145
2.4. Miscellaneous . . . 146

3. Porting Existing DDL Decoders to pyDDL . . . 150
3.1. Automatic Conversion . . . 150
3.2. Quick Reference . . . 150
3.2.1. Basic Programming and Language Elements . . . 152
3.2.2. Pre-Processing Functions . . . 152
3.2.3. Search Functions . . . 153
3.2.4. Read Functions . . . 153
3.2.5. Read Pointer Functions . . . 154

3.2.6. Fractionize Frame . . . 154
3.2.7. Reformat and Process Data Packages . . . 155
3.2.8. Arrays . . . 155
3.2.9. Analyzing, Code Checking and FEC . . . 156
3.2.10. Universal Decoding of Block Codes . . . 156
3.2.11. Functions for Bit Manipulation . . . 157
3.2.12. Operators . . . 157
3.2.13. Branch Commands . . . 158
3.2.14. Output Functions . . . 158
3.2.15. System Functions . . . 159
3.2.16. System Variables . . . 159
3.2.17. Symbol Tables . . . 160
3.2.18. Commands to Control Demodulator Parameters . . . 160
3.2.19. Knowledge-Base Demodulator Settings . . . 160
3.2.20. Special Commands for Morse Post-Processing . . . 161
3.2.21. Measurement Functions . . . 161
3.2.22. External Program Control and Interfacing . . . 161
3.2.23. Dynamic Link Libraries . . . 161
3.2.24. CVSD Speech Decoding . . . 162
3.2.25. Special Commands for Data Base Support . . . 162
3.2.26. Soft Decision Decoding . . . 162
3.3. Migration Guide . . . 163
3.3.1. Search pattern . . . 163
3.3.2. Performing Tasks on Decoder Exit . . . 163
3.3.3. Using Golay Block Decoder . . . 163
3.3.4. Using Hamming Block Decoder . . . 164
3.3.5. Converting Alphabets . . . 164

A. Support . . . 167

List of Figures . . . 168

List of Tables . . . 169

1. Manual

1.1. Overview

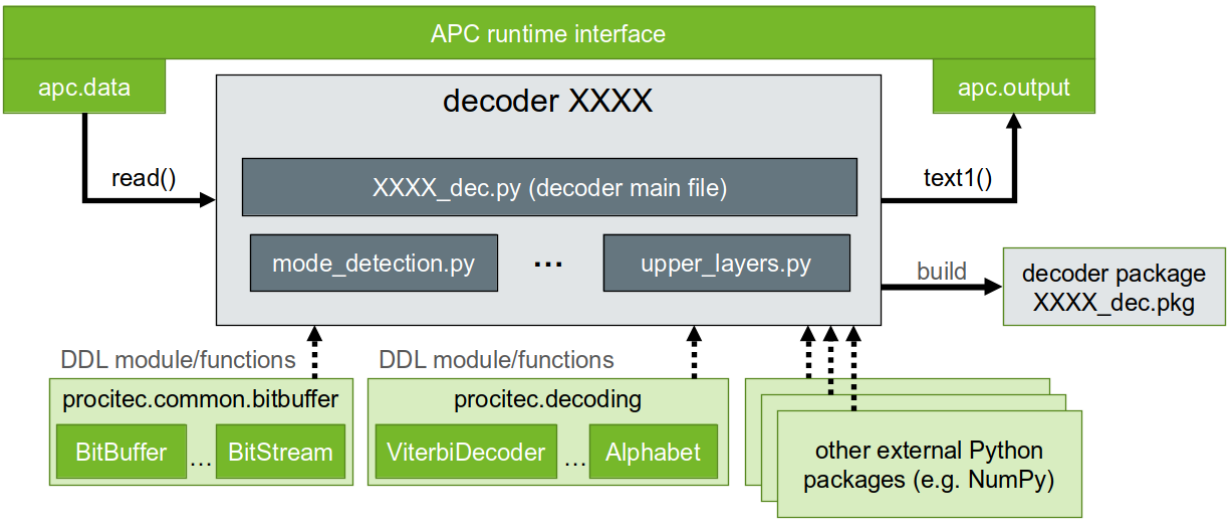
Modem definitions used in the Automatic Production Channel (APC) are comprised of a demodulator and a decoder. While signal demodulation can often be achieved by using a generic demodulator, the subsequent decoding is usually highly specific and requires more logic.

The Python programming language provides an easy to learn way to describe the decoding process programmatically. It provides structural elements for e.g. conditional execution and a rich set of builtin data types. Code can be organized into functions and classes, which can also be imported from other files.

The functions and classes described in this document provide functionality specific for developing decoders in Python:

- an interface to communicate with the *APC at runtime* to retrieve *input data*, set the modem state or *output the decoding result* as text or binary data. For development purposes there is a *standalone runtime*, that allows feeding recorded data from .rec-file into the decoder and provides the same interface as an APC.
- a data type for *processing bits* and associated metadata. This includes e.g. timestamps, demodulation quality, soft bits.
- a library of commonly used *decoding functions* including pattern search, error correction codes, etc.

This is an overview of the aforementioned architecture



As shown in the diagram a Python Decoder may be split into multiple Python modules. Each module corresponds to a .py-file and may contain only a part of a decoder or functions that are shared between different decoders. To distinguish between decoder and support modules, the decoder main files have to end in *_dec.py*.

Decoders and their support modules can also be packaged into a portable decoder package file with the suffix *_dec.pkg*.

1.2. Tutorial for Writing Decoders

In this tutorial a decoder for *HNG-FEC* is discussed. The focus here is not on the protocol itself but on how the decoder is implemented in Python and pyDDL. In short, HNG-FEC is full duplex transmission system using FSK with 100.05 Bd and 500 Hz shift formerly used by the Ministry of Foreign Affairs in Hungary. It uses an ITA-2 alphabet encoded in 15 bit codewords and interleaving. Received words can be decoded by finding the closest matching valid codeword.

The decoder receives a stream of bits from the *Decoder Runtime* and passes back the decoded message. The beginning of codewords and interleaving blocks must first be identified to then extract the transmitted characters. During execution the decoder also reports to the runtime, when synchronisation was achieved and when it is lost.

For the purposes of this tutorial the implementation of the decoder is split into two modules. That means we have two files which make up the decoder:

- the main decoder module, `hng_fec_dec.py` and
- a support module, `hng_fec_alphabet.py`.

Neither of these is too big in size not be be merged into a single file. However, it shows the potential of splitting large code files and sharing of code between decoders.

1.2.1. Support Module `hng_fec_alphabet.py`

In this module the mapping between codewords and characters is defined using the class *Alphabet* from *procitec.decoding*. For this type of encoding there are actually two different characters per infoword and therefore also per codeword. The resulting character is selected by special codewords that signal the level to be used for the following words.

```
from procitec.decoding import Alphabet, LEVEL as L

ita2_coded = Alphabet(
    {
        0b011111010010001: ["", ""],
        0b1001000011000001: ["T", "5"],
        0b0000101101111001: ["", ""],  # CR
        0b1110011111101001: ["O", "9"],
        0b1010101010101: [" ", " "],
        0b0100011000000101: ["H", "<>"],
        0b1101110011111101: ["N", ","],
        0b0011000000101101: ["M", "."],
        0b0001011011100011: ["\n", "\n"],
        0b1111101001000011: ["L", ")"],
        0b0110000010111011: ["R", "4"],
        0b1000110000011011: ["G", "&"],
        0b1100000101101111: ["I", "8"],
        0b0010110111001111: ["P", "0"],
        0b1011011100111111: ["C", ":"],
        0b0101101110011111: ["V", "="],
        0b1010010000110000: ["E", "3"],
        0b0100100001100000: ["Z", "+"],
        0b1101001000110000: ["D", "<WRU>"],
        0b0011111010010000: ["B", "?"],
        0b0111001111101000: ["S", "'"],
        0b1001111101001000: ["Y", "6"],
        0b0000010110111000: ["F", "<>"],
        0b1110100100011000: ["X", "/"],
```

(continues on next page)

(continued from previous page)

```
29          0b110011111010010: ["A", "-"],
30          0b001000110000010: ["W", "2"],
31          0b101110011111010: ["J", "<BEL>"],
32          0b010101010101010: [L[1], L[1]],
33          0b000110000010110: ["U", "7"],
34          0b111101001000110: ["Q", "1"],
35          0b011011100111110: ["K", "("],
36          0b100000101101110: [L[0], L[0]],
37          0b110010011001011: ["<P>", "<P>"],
38          0b001101100110100: ["<N>", "<N>"],
39      },
40      codeword_length=15,
41      replacement_character="_",
42      msb_first=False,
43  )
```

First, the required objects are imported. Aside from the *Alphabet* class and the *LEVEL* object is required to mark codewords as level changing.

The *Alphabet* instance `ita2_coded` is created by passing a mapping and some additional parameters. The mapping uses `int` literals in binary notation for the codewords and a list of characters, one per level, as values. Here, there are two levels. For example line 6 means codeword `0b100100011000001` maps to the value `"T"` for level 0 and to `"5"` in level 1.

Lines 32 and 36 contain the special, level-changing codewords described above. Instead of single character `str`, the values `L[0]` or `L[1]` are passed to signify a level-change 0 or 1, respectively.

1.2.2. Main Decoder Module `hng_fec_dec.py`

This module is were the actual decoding is performed. There are several ways how the code can be structured: a linear script, a set of functions or classes or a mix thereof. The runtime supports all of these and it usually depends on personal preference and the complexity of the protocol which is most suitable. It is expected, however, to try to continue execution until there is no more data to be processed. Once the module terminates, the production is stopped.

Most decoder implement a main entry point in form of a function or instance method. That entry point triggers synchronisation and decoding and may return on failure or loss of synchronisation. By looping over that call the decoding routine is then restarted to process potentially following parts of a transmission.

1.2.2.1. Preface and General Structure

In this decoder the main module has three sections: first are imports and globals, then there is a `main()` function, followed by some commands to setup control and data flow:

```
1  import procitec.decoding as ddl
2
3  from hng_fec_alphabet import ita2_coded
4
5  ita2_decoder = ddl.AlphabetDecoder(ita2_coded, initial_level=0, force_level=False)
6
7
8  def main():
9      """synchronize and decode"""
10     ...  # see below
11
```

(continues on next page)

(continued from previous page)

```
12
13 if __name__ == "__decoder__":
14     apc = ddl.runtime.APC(decoder_version="2.0.0")
15     while True:
16         main()
```

In the first section, lines 1 to 5, the *decoding* module is imported using the abbreviated name `ddl`. Instead of listing all the required objects on the import, it is oftentimes easier to import the module as a whole and access object with the "." operator.

Next is the import of the *Alphabet* `ita2_coded` from our support module shown above. From this an instance of *AlphabetDecoder* is created. An *Alphabet* itself is stateless. It defines the structure and properties of the code. The decoding of a stream of codewords with multiple levels requires not only finding the matching entry in `ita2_coded`, but also keeping track of the current level. This can, of course, be done manually. However, it is more convenient to use built-in functionality: *AlphabetDecoder* adds the described stateful decoding capabilities.

The details of the function `main()` are discussed below. For now, note the code at the end, starting from line 13: This is where an interface to the runtime, the `apc` object, is obtained by calling *ddl.runtime.APC()*. Using the variable name `apc` here is not a requirement, but convention. The Automatic Production Channel (APC) acts as host application for decoders and coordinates data flow between the data source, the demodulator and the decoder.

The final two lines of code call the function `main()` in endless loop. Inside `main()` data is read from *apc.data* until the production is terminated by the runtime. At this point the input stream is *closed*. However, data may be read until the input buffers are drained. As soon as a *BitStream* call requests more bits than available, an *EndOfDataError* is raised to terminate execution. Clean-up tasks can be performed by placing a `try ... except` statement which catches *EndOfDataError* around the loop in line 15.

The `if`-clause in line 13, is not strictly required. It is, however, useful to switch behavior depending on how the main module is used. All Python modules have a global variable __name__ containing their *import name* (filename without suffix preceded by the parent package name, if any). The main module, however, has the value "__main__" set as name instead as it is not being imported by another module, but acting as the main script file.

Decoder modules are also main modules: it is were the decoder execution starts. But decoders are not standalone Python programs or scripts. They require a host application, a runtime component to provide the data to be processed. The availability of the runtime is signaled with the main module name "__decoder__". Using the `if`-clause enables the decoder the be run outside of the APC host-application by the addition of an `else`-clause which may load the standalone runtime component to read data from pre-recorded file or to run tests.

1.2.2.2. Decoder Main Function

In this section the content of the `main()` function is discussed. This is were the actual decoding operations are performed:

```
1  def main():
2      """synchronize and decode"""
3      char_size = ita2_coded.codeword_length
4      bit_dist = 64  # interleaving
5      block_len = bit_dist * char_size
6
7      search_result = ddl.search_alphabet(apc.data, ita2_coded, repetitions=3,
8                                          max_bit_errors=0, max_offset=500,
9                                          interleaving=(bit_dist, char_size))
10     if search_result:
```

(continues on next page)

(continued from previous page)

```
11          apc.modem.ident()
12      else:
13          apc.modem.no_sync()
14          return
15
16      ita2_decoder.reset()
17      while True:
18          block = apc.data.peek(block_len)
19          codewords = ddl.extract_interleaved(block, bit_dist=bit_dist,
20                                              char_dist=char_size, char_size=char_size,
21                                              char_ix=0, block_len=block_len)
22          apc.data.consume(codewords.size)
23
24          if dec_result := ita2_decoder.decode(codewords, max_errors=3):
25              apc.modem.sync()
26              apc.output.text1(dec_result)
27          else:
28              apc.modem.no_sync()
29              return
```

Initially, there is an unknown offset between the bits in `apc.data` and the start of the next codeword and interleaving block. Only after this offset is found, deinterleaving and decoding of the resulting codewords can be performed. In many cases transmissions include special bit sequences, or preambles, to mark the start of interleaving blocks. Here we simply brute-force search the incoming bits for a sequence of valid interleaved symbols from our alphabet `ita2_coded`. This method allows synchronisation even if the start of the transmission is missing.

First, the interleaving parameters `char_size`, `bit_dist` and `block_len` are defined as local variables. This information is not only required for the synchronisation, but also for subsequent the data extraction. The actual search for the offset could be implemented using a loop over possible offsets, the deinterleaving index sequence followed by checking the resulting codewords against those in `ita2_coded`. This is a pretty common task and therefore already implemented as part of the Decoding Library.

The function *search_alphabet()* is called in line 7. The parameters passed are

- `apc.data`, the *BitStream* to be searched,
- `ita2_coded`, the alphabet to be found,
- `repetitions`, the number of valid codewords required for a result to be considered successful
- `max_bit_errors`, the number of bit errors tolerated in a successful search
- `max_offset`, when to stop searching and report failure
- `interleaving`, parameters to re-order bits before comparison

Details on these parameters and their meaning can be found in the documentation of *search_alphabet()*. The important aspect here is that this function will read and consume bits from `apc.data` until it either finds valid codewords or `max_offset` is reached. That second condition is quite important as it terminates the search to allow the decoder to report its status: *no synchronisation (yet)*.

The object returned by *search_alphabet()* is stored as `search_result` and is an instance of *SearchResult*. This object contains information about the result: success or failure as well as, in case of success, the offset and number of errors in the found pattern. For convenience, it may be used as a boolean, returning the value of *SearchResult.found* via `__bool__()`.

This is used in line 10: in case of a successful search *apc.modem.ident()* is called to report that transmission has been identified as *HNG-FEC*. Otherwise, if no valid sequence of codewords was found within `max_offset`, the decoder informs the runtime about this by calling *apc.modem.no_sync()*. It then returns from `main()`, only to be called again to search the next chunk of bits for valid codewords.

Next is the decoding phase: first the `ita2_decoder` object is reset to clear internal state (see *AlphabetDecoder.reset()* for details). Note, an alternative to using a global object for *AlphabetDecoder* and calling `reset()` would be to create a fresh instance each time this point is reached. Finally, the loop starting in line 17 reads consecutive interleaving blocks from `apc.data` and decodes them:

- line 18: here *apc.data.peek()* is used instead of *read()*. The difference is that the number of requested bits, `block_len`, are returned as a *BitBuffer*, but the current *position* within the stream is not modified. Calling that function again would return the same data. Only after calling *apc.data.consume()* in line 22 the *position* is advanced.
- line 19: *ddl.extract_interleaved()* is called, to perform deinterleaving on the current `block` of bits. The interleaving parameters defined earlier are passed and the result is returned as *BitBuffer* object and stored in `codewords`.
- line 24: the sequence of `codewords` is processed in the *AlphabetDecoder* instance `ita2_decoder` which returns a *DecodeAlphabetResult*, `dec_result`. This object contains the extracted sequence of characters, the number of potential codeword errors and the total number of faults. Its `__bool__()` operator checks if all codewords could be decoded.
- line 25: `codewords` was decoded successfully, which is reported via *apc.modem.sync()*. The decoded part of the message is sent to the default output channel `text1` using *apc.output.text1()*. Note, *DecodeAlphabetResult* defines `__str__()` which returns *dec_result.message* if used in a `str`-context.
- line 28: decoding failed, the decoder state is changed back to *no_sync()*. Then `main()` returns, gets called again and tries to re-acquire synchronisation.

1.2.2.3. Code Structure Alternatives

The example code shown above uses a single function to perform synchronisation and decoding. These two operations could be extracted into separate functions, leaving only the higher-level code in `main()`:

```
1  def main():
2      """synchronize and decode"""
3      if find_coded_ita2_alphabet(apc.data):
4          apc.modem.ident()
5          ita2_decoder.reset()
6          while dec_result := read_and_decode(apc.data):
7              apc.modem.sync()
8              apc.output.text1(dec_result)
9      apc.modem.no_sync()
10
11
12 def find_coded_ita2_alphabet(stream):
13     return ddl.search_alphabet(stream, ita2_coded, repetitions=3,
14                                max_bit_errors=0, max_offset=500,
15                                interleaving=(64, 15))
16
17
18 def read_and_decode(stream):
19     block = stream.peek(block_len)
20     codewords = ddl.extract_interleaved(block, bit_dist=64,
21                                         char_dist=15, char_size=15,
22                                         char_ix=0, block_len=64 * 15)
23     stream.consume(codewords.size)
24     return ita2_decoder.decode(codewords, max_errors=3)
```

Such a split is useful if the two operations are more complex as it separates and abstracts them cleanly and avoids deeply nested structures. Here only the required parts of the runtime, namely `apc.data` was passed to the sub-functions. The object `apc` itself is, however, defined in the module global scope and therefore accessible everywhere.

For other protocols a pure function-based structure may not work well. Oftentimes parameters and state need to be preserved and passed, e.g. the decoding function can depend on parameters determined in the synchronisation function. Here a class-based approach can be used:

```
class HNGFEC:
    def __init__(self):
        self.some_state = ...

    def find_coded_ita2_alphabet(stream):
        ...
        self.some_state = "found"


    def read_and_decode(stream):
        ...
        if self.some_state == "found":
            ...
```

The `main()` function itself can either remain a free function that creates an instance of `HNGFEC` or be part of the class itself. For complex protocols a set of classes may be used to handle different modes or messages types separately – each keeping the current state localized within each instance.

1.3. Decoder Development Using Spyder

Development of decoders is done in an *Integrated Development Environment (IDE)* based on Spyder. Use the `Extras → Launch Decoder Development (pyDDL)` menu entry in *Modem Lab* or the start menu of your operating system to start the IDE. The main window of the IDE is shown in Fig. 1.

For a tutorial how to write a decoder see *Tutorial for Writing Decoders*.

1.3.1. Creating and Using a New Decoder

To create a new decoder, use the `File → New file...` menu entry, press `Ctrl+N` or use the `New file` button in the toolbar. Save the decoder by using the `File → Save` menu entry, pressing `Ctrl+S` or using the `Save file` button in the toolbar. The file can be be saved in any folder. The name of the main file of the decoder must end in `_dec`. Additional modules for a decoder (see *Tutorial for Writing Decoders*) can be created in the same manner. They must be placed alongside the decoder's main file.

A decoder package must be created in order to use the decoder in other PROCITEC products. A decoder package bundles all modules of a decoder into a single file with the file extension `.pkg`. This simplifies the distribution of decoders. Use the `File → Export decoder package` menu entry to create a decoder package; it will be saved in in `<user directory>/procitec/common/modems`. The decoder can now be used in *Modem Lab* by selecting it from the list of decoders (see chapter 3.14.4 in the *Modem Lab* manual).

Decoder packages are marked with the  icon.

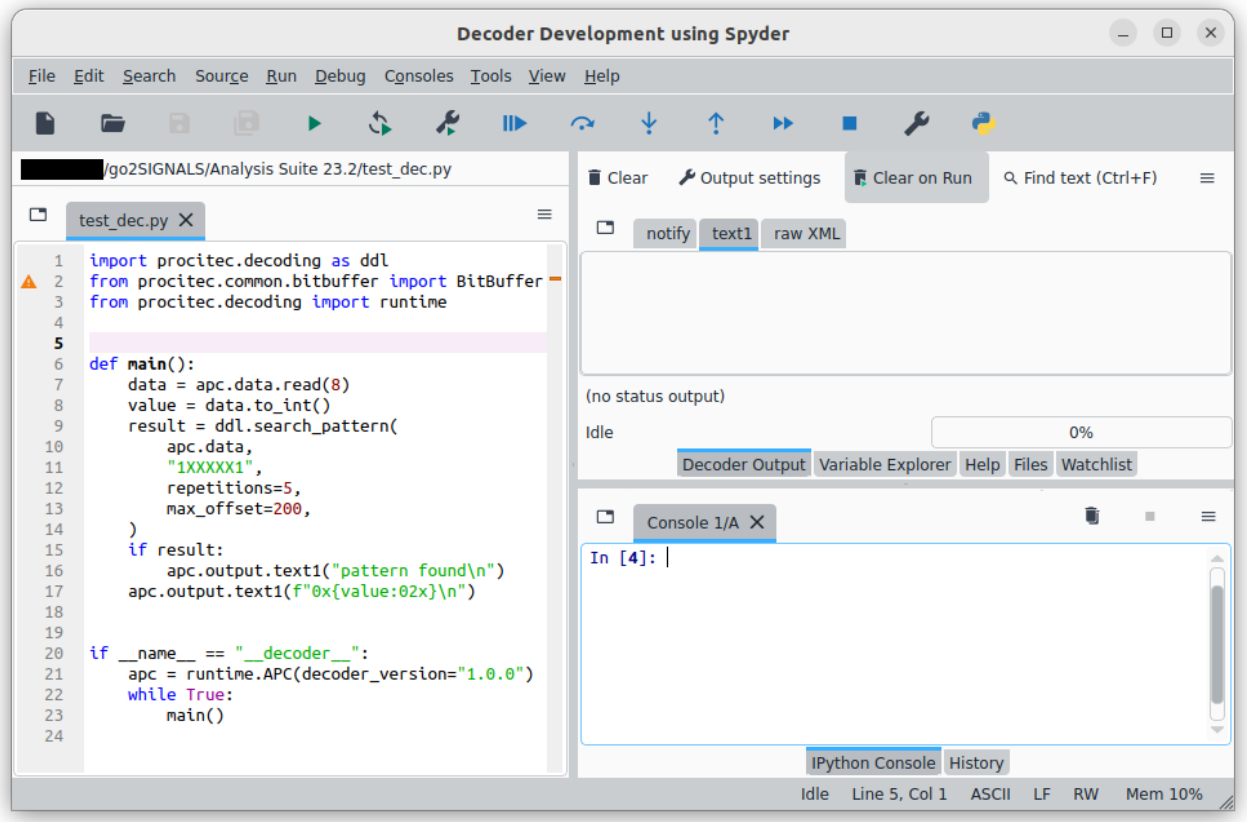


Figure 1: Spyder main window

> **Note**
>
> Changes or enhancements to the Decoder Description Language are made with backward compatibility in mind. Thus, once created decoder packages (pkg) should still work when the go2signals software is upgraded to a newer version.
>
> However, there still might be edge cases where backward compatibility is not possible (study release notes and manual carefully) or there are new DDL or Python commands which you want to apply in your decoder. Then it is recommeded to recreate the decoder package file.

1.3.2. Editing an Existing Decoder

Use the `Edit descripton` button in *Modem Lab* (see chapter 3.14.4 in the *Modem Lab* manual) to edit an existing decoder package. The decoder is looked for in the user directory for modems (`<user directory>/procitec/common/modems`). If not found there, the decoder is looked for in the common installation directory for modems[1]. Decoder packages provided by PROCITEC can only be opened if they are not subject to license restrictions.

The package will be extracted to `<user directory>/procitec/common`. This will show a dialog as depicted in Fig. 2 if the package has been opened before. Overwriting existing files is only safe if `Export decoder package` has been used in the previous edit session. If in doubt, abort by using the `No` button in the dialog and backup existing files (or do so *before* using the `Yes` button in the dialog).

You can now edit the decoder.

[1] Please note that this search order also applies to the usage of decoders by the APC: If a decoder with the same name exists both in the user directory for modems and the common installation directory for modems, the one in the user directory will be used.

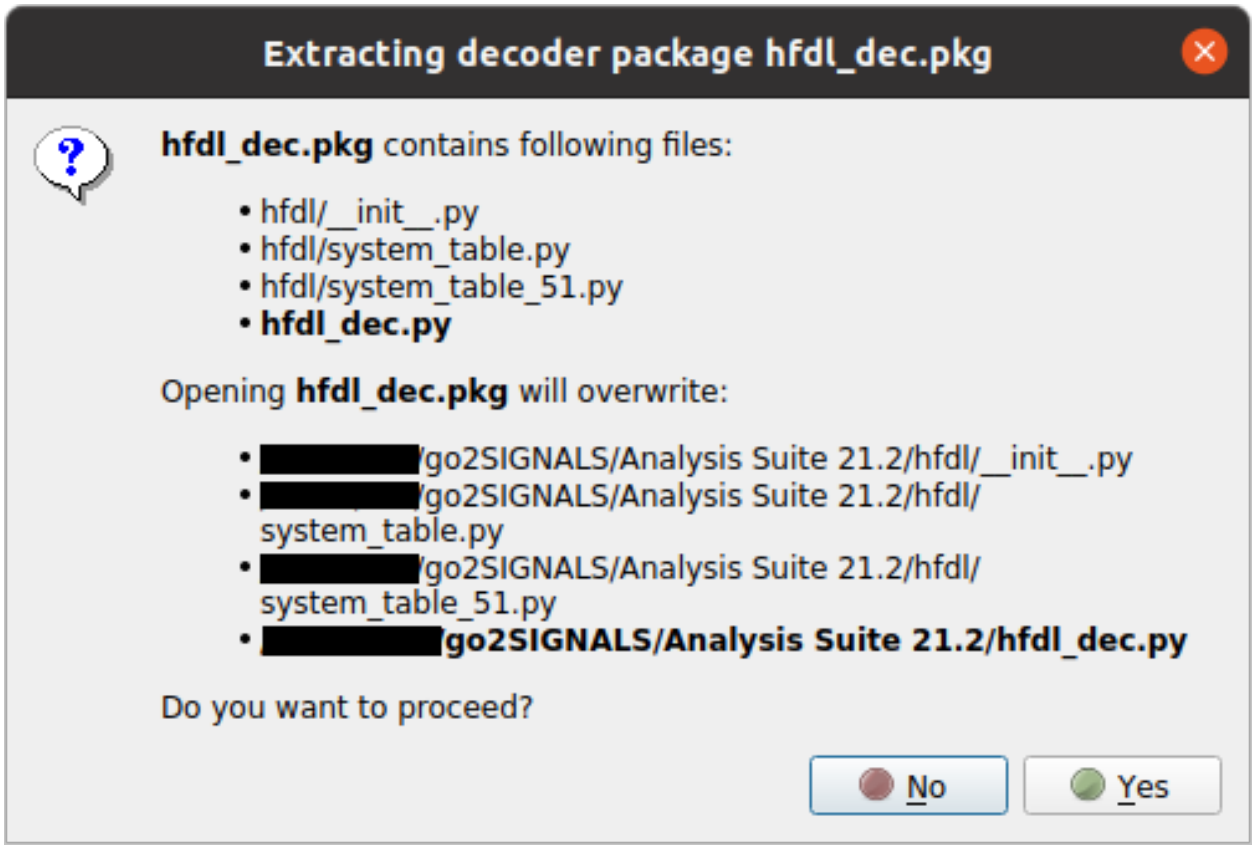


Figure 2: Dialog shown upon opening an existing decoder

> **Note**
>
> Once done, use the `File` → `Export decoder package` menu entry to update the decoder package.

Instead of using the `Edit descripton` button in *Modem Lab*, you can and use the `File` → `Open...` menu entry in the IDE to open a decoder (`_dec.py` file) or a decoder package (`_pkg.py` file) from any directory. Alternatively press `Ctrl+O` or use the `Open file` toolbar button.

1.3.3. Running a Decoder

By running a decoder you can see its output and verify whether the behaviour meets expectations. Run a decoder by using the `Run` → `Run` menu entry, pressing `F5` or using the `Run` button in the toolbar. You can also use the `Run last file` action (`F6`) displayed next to the `Run` action. This will execute the previous run/debug action regardless which file is currently open. This is especially useful while a module used in a decoder is being edited.

The very first time a decoder is started a configuration dialog appears (see Fig. 3). The dialog can be opened at any time using the `Run` → `Configuration per file...` menu entry, pressing `Ctrl+F6` or using the `Run settings` button in the toolbar (next to the `Run` button). Settings configured in the dialog are saved individually for each decoder and recalled the next time a decoder is executed.

There are two different execution modes: `Decoder only` and `Signal processing and decoder`. They are explained in *Execution Mode Decoder only* and *Execution Mode Signal processing and decoder* respectively. For details about "decoder parameters", see *Decoder Parameters*.

Confirm the settings and run the decoder by pressing the Run button in the dialog. The decoder's output appears in `Decoder Output` in the upper right area of Spyder's window. See *Examining and Saving Decoder's Output* for details.

The IDE can also be used to execute "conventional" Python code by selecting `Run generic Python script` instead of `Run Decoder Development` at the top of the configuration dialog (see Fig. 3).

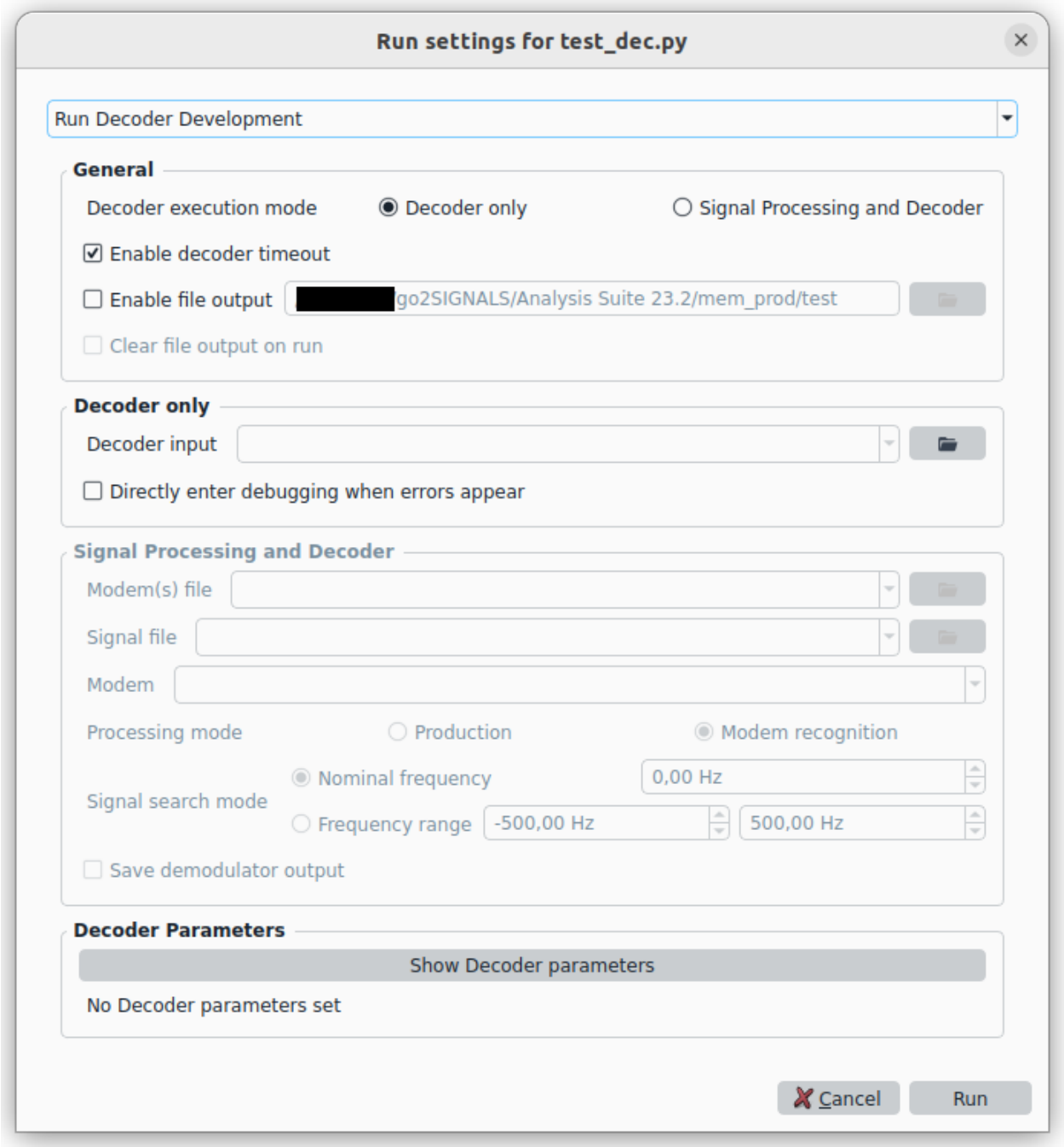


Figure 3: Configuration dialog for decoder execution

1.3.3.1. Execution Mode *Decoder only*

`Decoder only` is the execution mode selected by default. The mode is best suited for the development and testing of new decoders and for generic decoder development and testing. Only the decoder's code is executed, i.e. the execution is not affected by the APC's signal processing and runtime behaviour.

The input for the decoder is a so called `.rec` file which contains the bit stream generated by a demodulator. Alongside the raw bit stream other information such as burst information, timestamps and soft symbols are saved. Such a file can be generated using the `Record demodulator output` option in *Modem Lab* (see chapter 3.6.5 in the *Modem Lab* manual). Alternatively, use the *Execution Mode Signal processing and decoder* and set the option `Save demodulator output`.

You must select an input `.rec` file for the decoder. Do so by providing a file path in the `Decoder input` input line or invoke the file browser using the button right to the input line.

The `Decoder only` execution mode has the option to automatically start the debugger when an error occurs. This allows the inspection of variables (see *Examining variable values*) at the point the error occurred. Check the option `Directly enter debugging when errors appear` in the configuration dialog (see Fig. 3) to make use of this feature.

1.3.3.2. Execution Mode *Signal processing and decoder*

The mode `Signal processing and decoder` is mainly intended to test and debug the behaviour of decoders as they are executed inside the APC. Especially the behaviour and interaction with other modems (decoders) in *Modem Recognition* (see chapter 3.2.3 in the *Modem Lab* manual) is of interest.

The input file is a signal (`.wav`) file recorded by a receiver or a sound card, or generated synthetically (e.g. by SOMO). It is subject to the usual signal processing in the APC where eventually the bit stream generated by a demodulator is passed to a decoder.

The following settings must be defined to run a decoder in the `Signal processing and decoder` mode; they resemble settings used in *Modem Lab*:

- `Modem(s)`: Select the modem (`.ver`) or modem list (`.cmf`) file to be used for the execution of the decoder. Provide a file path in the input line or select a file by invoking the file browser using the button right to the input line. `.ver`/`.cmf` files can be created in *Modem Lab*, see chapters 3.7 and 3.8 in the manual. Make sure that the decoder is in fact used inside the `.ver`/`.cmf` file.
- `Signal file`: A `.wav` containing the signal to be processed. Provide a file path in the input line or select a file by invoking the file browser using the button right to the input line.
- `Modem`: A modem list file (`.cmf` file) may contain multiple modems. The modem to be executed can be selected here. Note that only modems with a matching name are displayed, i.e. at least one modem must use the decoder currently being edited.
- `Processing mode`: Either `Production` or `Modem recognition`; see chapter 3.2.3 in the *Modem Lab* manual for details.
- `Signal search mode`: Define the *absolute* frequency location of the signal of interest in the input file: See chapter 3.14.2.2 in the *Modem Lab* manual for details on the difference between `Nominal frequency` and `Frequency range`.

  > **Note**
  >
  > You must provide the *absolute* frequency location *inside the file*.

- `Save demodulator output`: Set this setting to save the demodulator output to a `.rec` file. These files can be used in the *Execution Mode Decoder only*.

  This option requires the general option `Enable file output` to be enabled.

1.3.3.3. Decoder Parameters

Decoders may receive and make use of externally set parameters. They are set when the decoder is started and can be used to configure the behaviour of the decoder. Inside the decoder, the parameters are accessible via `apc.parameters`.

Decoder parameters are set and modified in dialog windows described below. The parameters are saved individually per decoder, and per modem file and modem in case of *Execution Mode Signal processing and decoder*.

Setting and modifying decoder parameters in *Execution Mode Decoder only* is always possible by using the `Show Decoder parameters` button in the configuration dialog (Fig. 3). This brings up a dialog as shown in Fig. 4. Use the buttons in the toolbar to add and remove entries (multiple entries can be selected for deletion). Double click on a column to modify the corresponding value.

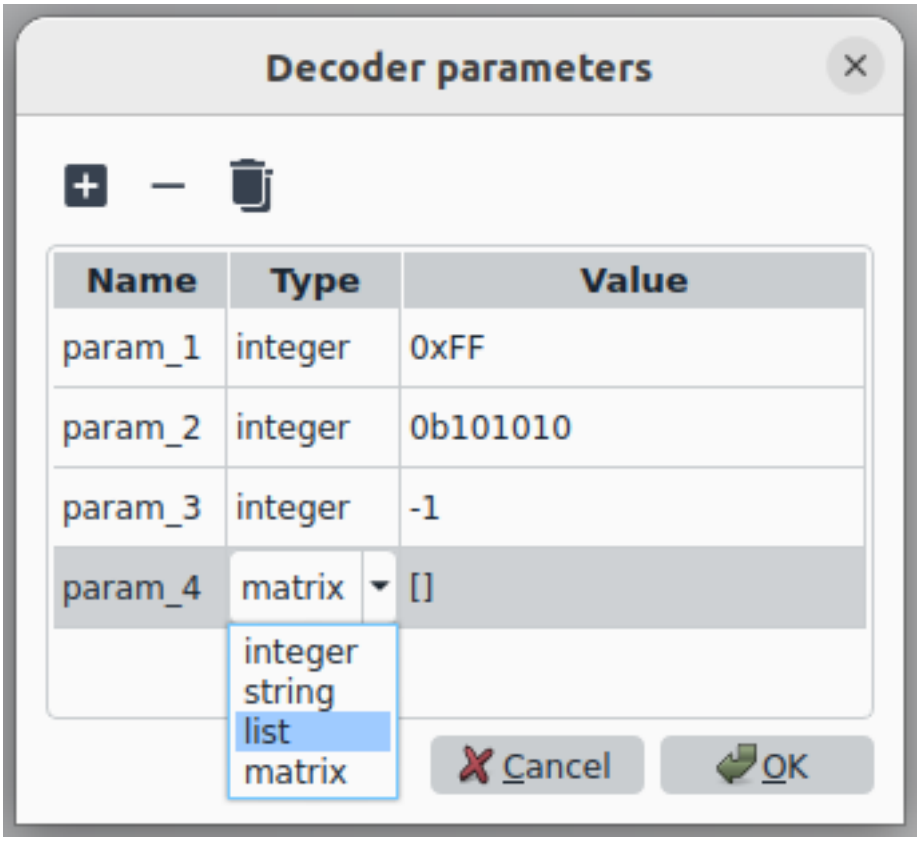


Figure 4: Dialog for decoder parameters in *Execution Mode Decoder only*

Setting and modifying decoder parameters in *Execution Mode Signal processing and decoder* is only possible if decoder parameters have been defined in the modem file. See chapter 3.14.4.4 of the *Modem Lab* manual for details. If decoder parameters are available, a dialog (Fig. 5) to modify them is brought by using the `Show Decoder parameters` button. Hover over GUI elements in the dialog to show tooltips about the corresponding decoder parameter. Modified values – those differing from default values defined in the modem file – are marked bold. `Restore Defaults` restores *all* values to defaults from the modem file.

1.3.4. Examining and Saving Decoder's Output

The decoder's output will appear in `Decoder Output` in the upper right corner of Spyder's window (see Fig. 1 and Fig. 6). By default the output of all output channels declared by the decoder is shown. There is also an additional tab containing the raw XML output. This can be used to examine the output of `apc.output.xml`.

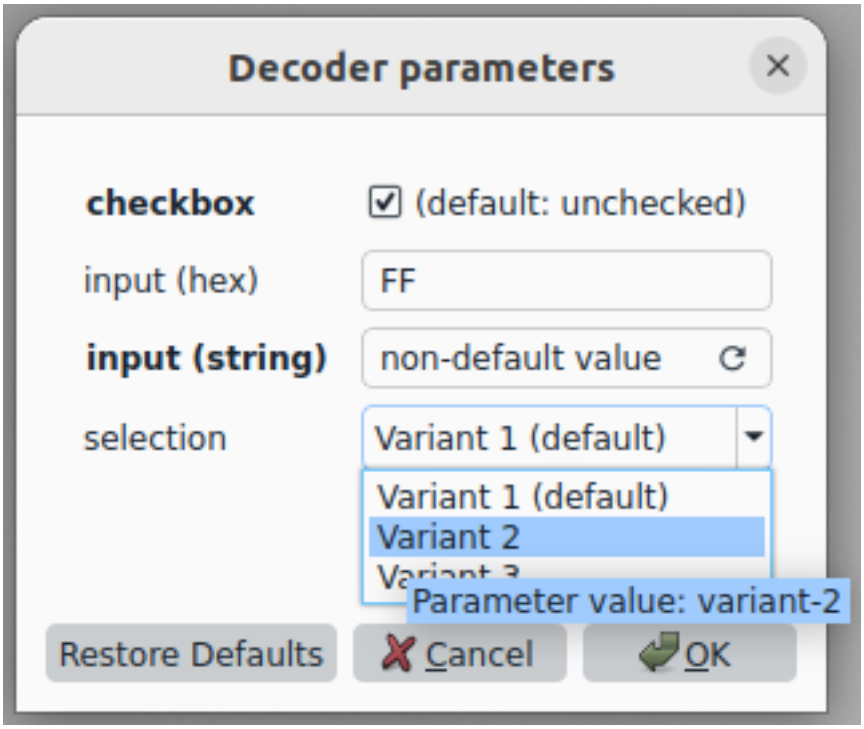


Figure 5: Dialog for decoder parameters in *Execution Mode Signal processing and decoder*

Please note that output on the `notify` channel is handled in a special way: It is displayed on all output tabs in gray color (this also applies to other PROCITEC products), unless output of the channel is disabled via *Output Settings*.

To find some text in the output, use the shortcut for `Find text` (the same one as in the editor; by default `Ctrl+F`).

Just below the decoder's output two additional lines with information are displayed. The first line (text `(no status output)` in Fig. 6) displays the decoder's output to the `status` channel. This special output channel can be used to output status messages. For example, in a point-to-point communication, it might be of interest to display the addresses of the current communication partners. These status messages are also displayed in a special manner in other PROCITEC products.

Below the status output of the decoder, the following information about its execution state is displayed:

- The run status. For decoders in execution mode *Decoder only* the run status is always `Running...`. With the execution mode *Signal processing and decoder* the run status may be any of `Search ongoing...`, `Search finished...`, `Production ongoing...` or `Production finished...`. These values reflect the current status of the APC's signal and decoder processing.
- The current position of the decoder's input bitstream (*`apc.data.position`*). This is only available when debugging a decoder. Note: The value is not updated while the decoder is running (i.e. after using `Continue`). It is only updated when executing the decoder stepwise and when a breakpoint is hit.
- The progress of the decoder execution. This is approximate only, especially so in the `Signal processing and decoder` mode. Moreover, a rewind of the input stream or the demodulator is not reflected.

The run status and the position of the input bitstream is also displayed in the statusbar of Spyder's window (see Fig. 9).

The output channels to be shown can be selected by opening the `Output Settings` dialog (see Fig. 7). Use the `Show selected channels` option and select the desired output channels. The list of channels is updated every time the decoder is executed. Execute the decoder once if the list is empty.

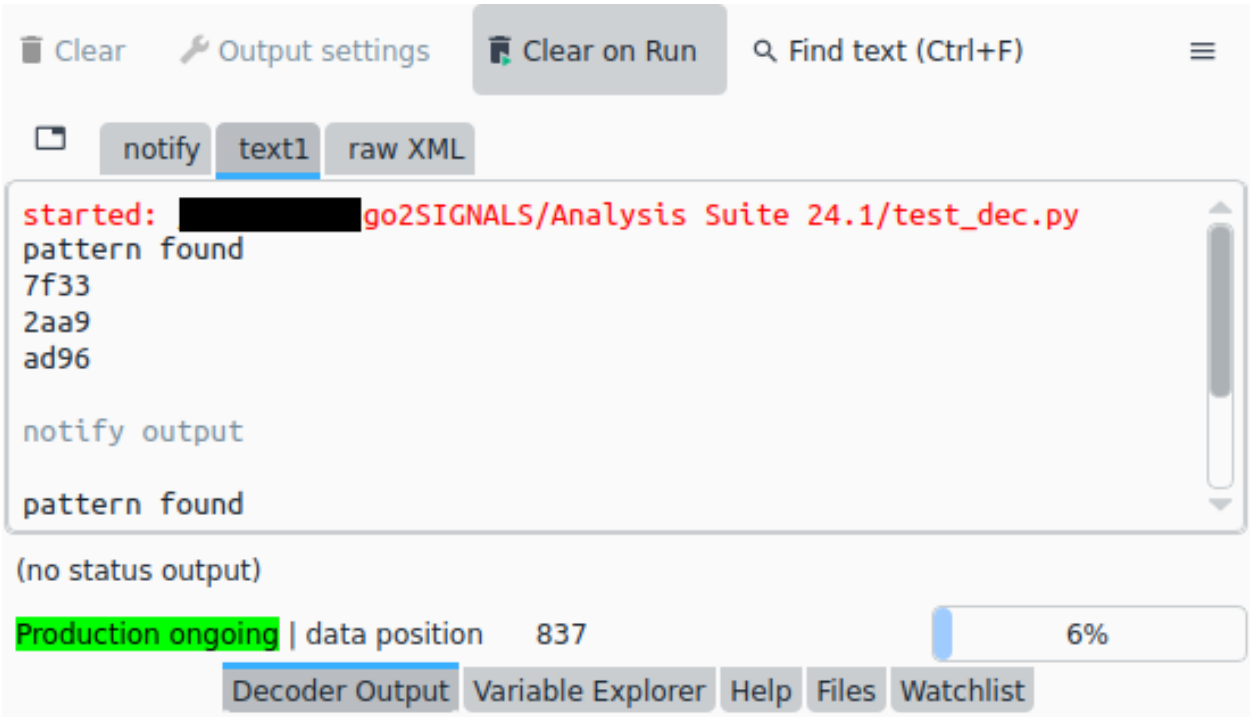


Figure 6: Decoder output widget in the upper right corner of Spyder's window. Note the special handling of output on the `notify` channel: It is displayed on all output tabs in gray color.

The display of the raw XML output is controlled by the `Show raw XML result` setting.

The decoder's output can be saved to files. Each output channel is saved into an individual file (UTF-8 encoded). To enable this feature, use the `Save decoder output` setting and select the output directory.

All settings made in the `Output Settings` dialog are saved individually for each decoder. They are recalled when Spyder is restarted.

1.3.5. Debugging a Decoder

Debugging is the process of finding and fixing errors in program code. This is facilitated by being able to halt the program's execution at specific points in the code. These points are called breakpoints. There are two ways to set or clear them:

- Use the `Debug` → `Set/Clear breakpoint` menu entry or press F12 to set/clear a breakpoint at the current line in the file.
- Place the mouse to the right of a line number (shown in the left part of the code editor, see Fig. 8). Use left mouse button to set/clear a breakpoint at that line.

Debugging can be started by using the `Debug` → `Debug` menu entry, pressing `Ctrl`+F5 or using the `Debug` button in the toolbar. You can also use the `Run last file` action (F6) displayed next to the `Run` action. This will execute the previous run/debug action regardless which file is currently open. This is especially useful while a module used in a decoder is being edited.

The decoder will be started and will run until a breakpoint is hit. Note that the dialog for decoder execution configuration appears if the decoder has not been run or debugged before (see *Running a Decoder*).

Once a breakpoint is hit, see the `Debug` menu for available options to control further execution; they are explained below. You can also use the corresponding keyboard shortcuts or buttons in the toolbar.

- `Step` (`Ctrl`+F10) executes the current line and goes to the next.

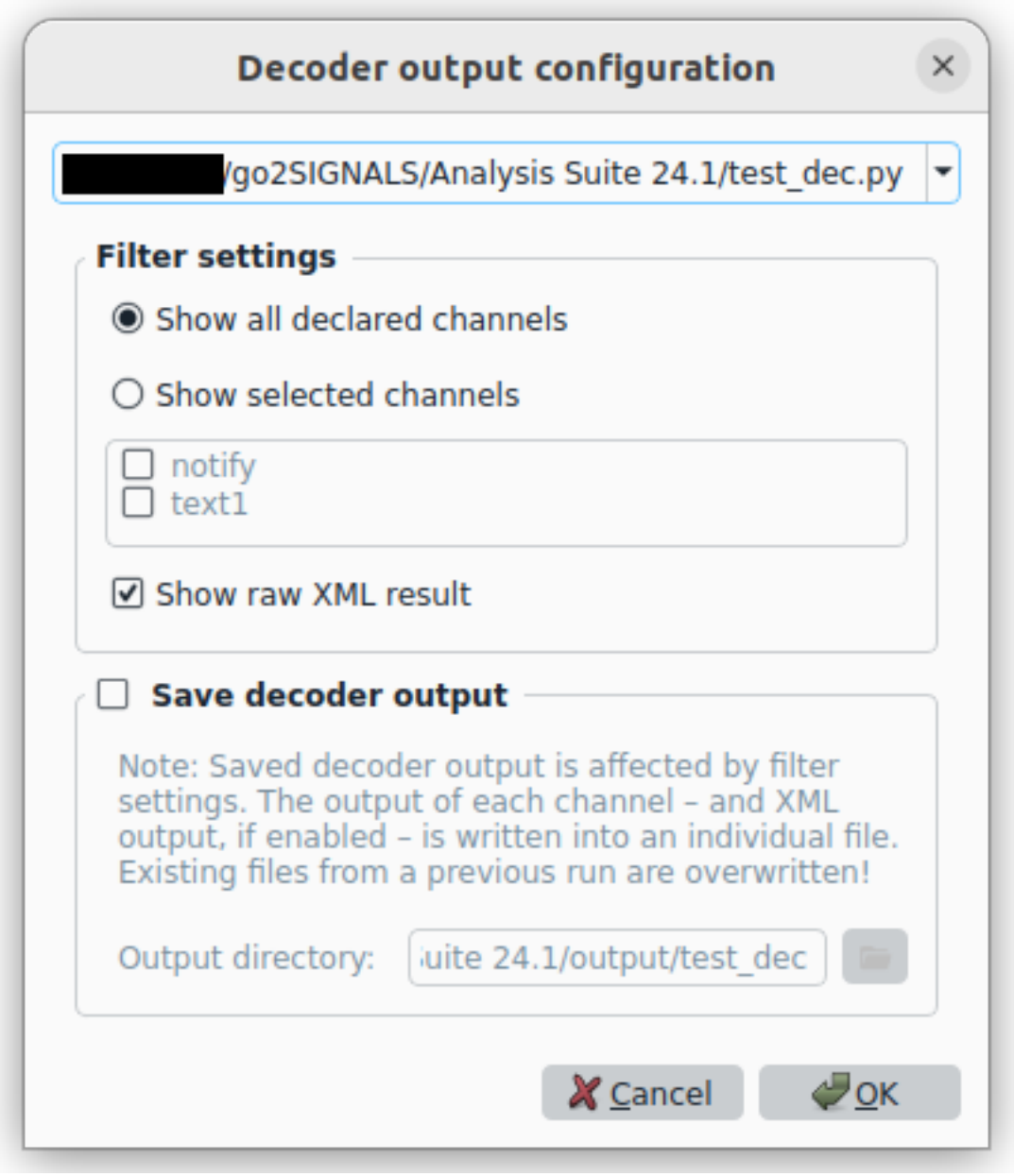


Figure 7: Configuration dialog for decoder output

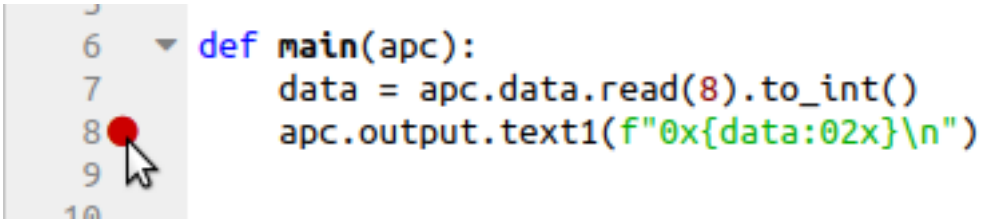


Figure 8: Setting or clearing a breakpoint using the mouse; use left mouse button

- `Step into` (`Ctrl`+`F11`): If the current line contains a function call (possibly nested ones), "jump" into that function. This only works if the function's definition (Python code) is available. Execution stops at the point of the function's definition. If the function is defined in an other file, that file will be opened.
- `Step return` (`Ctrl`+`Shift`+`F10`) execute the current function up to the point where it returns (the return statement is not executed).
- `Continue` (`Ctrl`+`F12`) executes code until the next breakpoint is hit.
- `Stop` (`Ctrl`+`Shift`+`F12`) stops debugging.

You can use the `IPython console` in the lower right corner of Spyder's window to execute Python expressions when code execution is halted (see Fig. 9). This also includes the manipulation of variables. The command line can also be used to control the debugger; see Debugger Commands for details. Debugger commands must be prefixed with an exclamation mark (`!`).

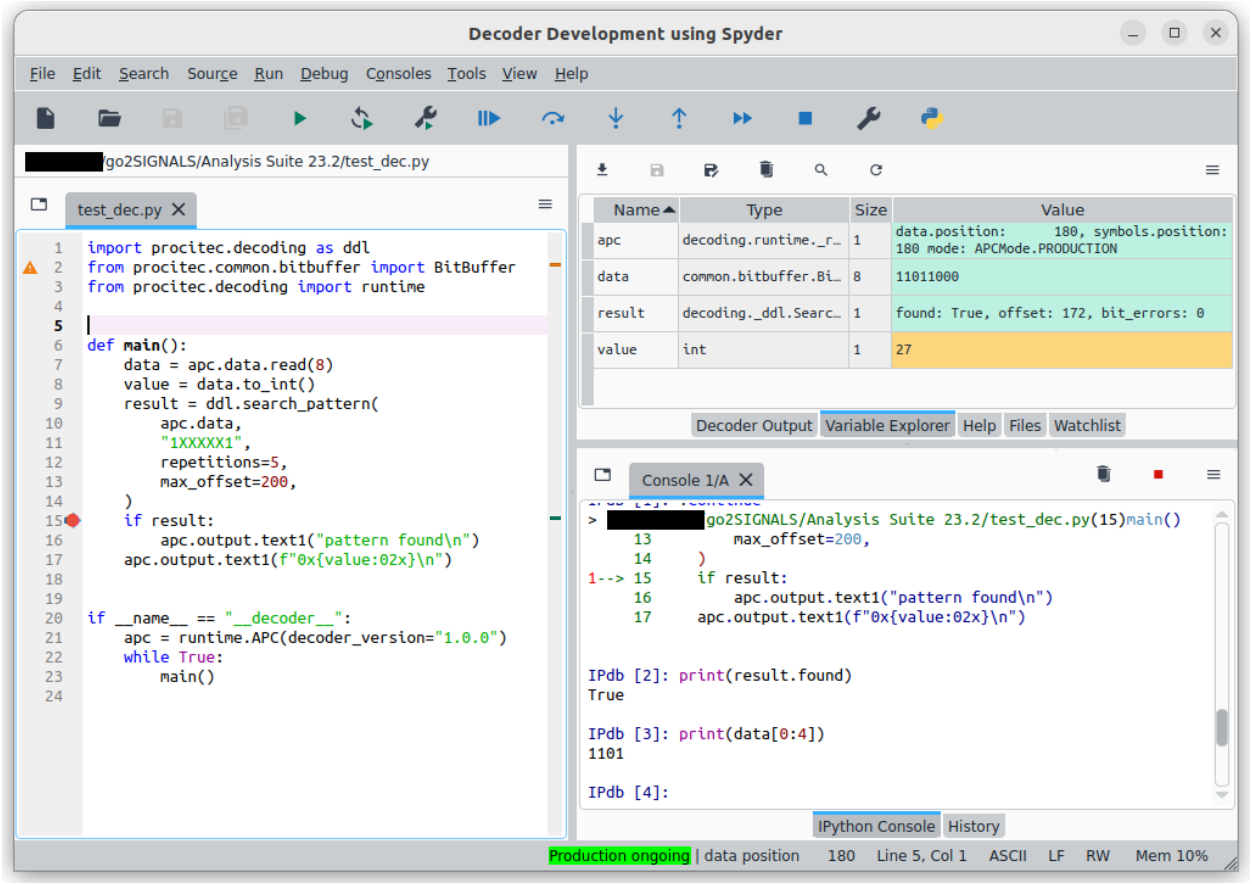


Figure 9: A running debug session. `Production ongoing` and `data position` in the statusbar is described in *Examining and Saving Decoder's Output*.

1.3.5.1. Examining variable values

There a multiple ways to examine variables when code execution is halted:

The `Variable explorer` in the upper right corner of Spyder's window shows *all* local variables (see Fig. 9). If you want to see only a specific set of variables, you can either use the `IPython console` or the `Watchlist` described hereafter.

The `IPython console` console in the lower right corner of Spyder's window can be used to print variable values. For Python's built-in data types the `print()` can be left out, i.e. it is sufficient to enter the variable's name only to view its value. This may apply to other types as well. If the displayed value is not "human readable", enclose the expression in `print()` or `str()`.

The `Watchlist` in the upper right corner of Spyder's window can be used to automatically execute Python expressions, similarly to the console (see Fig. 10). The "stringified" (see `str`) result of the statement is displayed in the `Value` column whenever code execution is halted. Errors during the execution of expressions are denoted by `<exception name>` in the `Value` column; the full error message is displayed in the tooltip of the `Value` column.

Watchlist entries can be added, removed and modified by using one the following ways:

- Use the toolbar buttons above the table to add and remove expressions.
- Use the right-click context menu inside the table to add and remove expressions.
- Use the `Delete` key to removed selected expressions.
- You can select multiple entries for removal.
- Use a double click below the last row in the table to add an expression.
- Use a double click onto an existing expression to modify it.
- Drag & Drop text into the table. Each line of the dropped text will be added as an expression to the list.
- Rearrange entries by using Drag & Drop. This is only possible if a single entry is selected.

The value of an expression can by copied using one of the folling ways:

- Select the value (click in the second column in the desired row) and use `Ctrl+C` shortcut.
- Use the `Copy value` action in the right-click context menu.

Note that the console and the watchlist can be used to execute any Python expression, including expressions with side effects. These side effects will affect subsequent code execution.

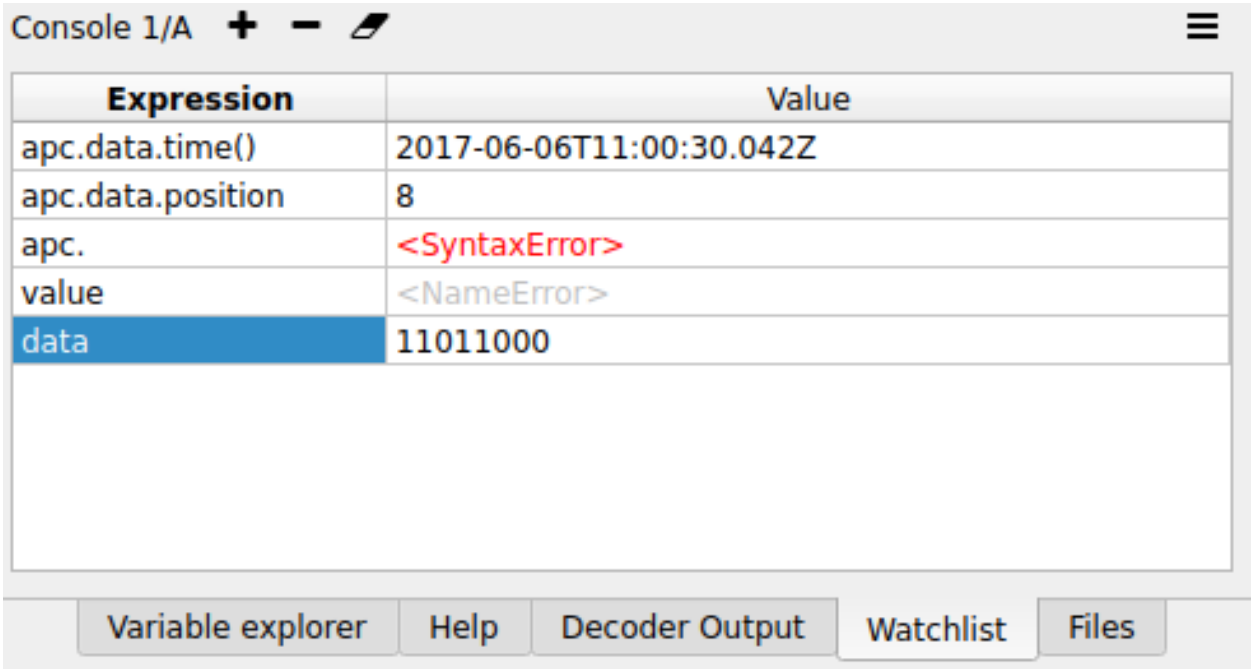


Figure 10: Watchlist plugin. Syntax errors are displayed in red. A `NameError` either indicates that a variable is not defined yet or that it is misspelled.

1.3.5.2. Graphical Display of a BitBuffer

The content of *BitBuffer* can be displayed in a graphical manner (see Fig. 11 and Fig. 12). To do so, perform a double click on a BitBuffer variable in the Variable explorer or use the Plot entry of the right click context menu. The display has the following features:

- Zoom by rotating the mouse wheel.
- The first N bits in the BitBuffer can be ignored (display offset). Modify the first input field at the top of the display.
- The number of columns can be modified in the second input field.
- The display style can be selected in the middle drop-down list. The first symbol denotes the symbol used for a 0 bit, the second symbol the symbol for a 1 bit.
- If available, display of burst marks can be enabled. Bits which are a burst start or end are highlighted with a green color. Hover over such a bit to display a tooltip with more information about the burst mark. (See Fig. 12.)
- If available, display of quality of bits can be enabled. The quality of bits is displayed in grayscale. A darker color indicates a better quality. (See Fig. 12.)
- A row-wise selection can be performed by clicking onto a bit and dragging the mouse. Press and hold Shift before the click and drag action in order to make a block-wise selection. Hovering over a selection displays a tooltip with information about the selection (see Fig. 11). A right click context menu allows copying of the selected bits into the clipboard.

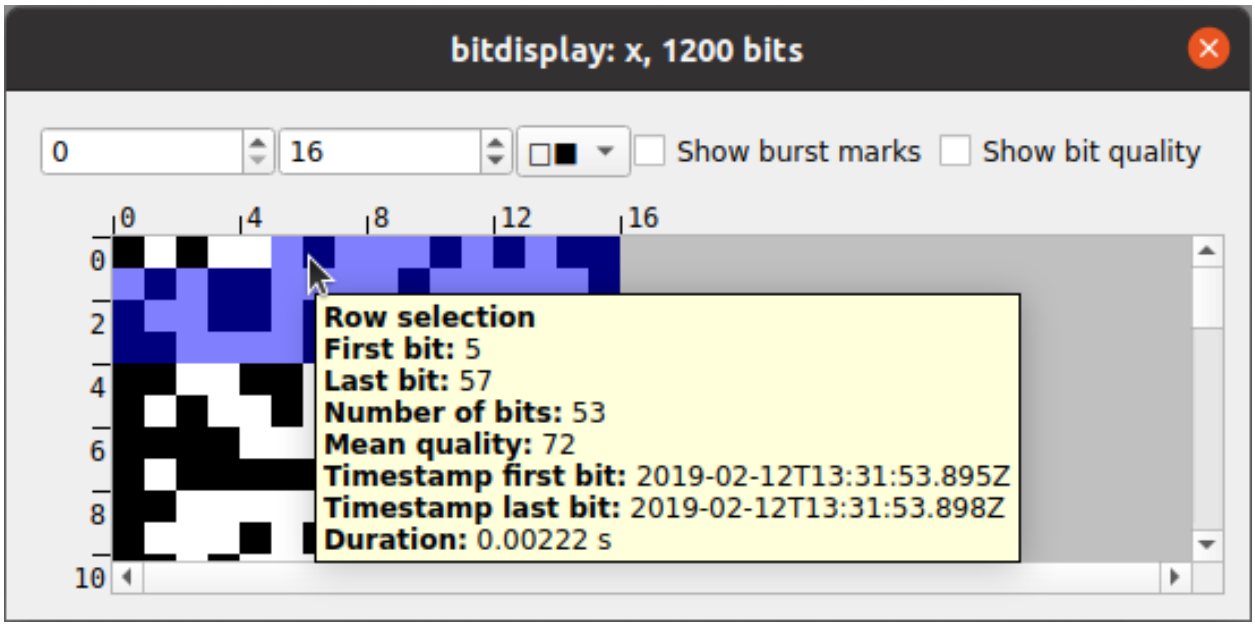


Figure 11: Graphical display of a BitBuffer

1.3.6. Profiling a Decoder

Profiling is the process of measuring a program's performance. The collected information aids optimization of programs by revealing bottlenecks (long-running portions of code). These bottlenecks may be possibly optimized to run faster.

The decoder development environment includes a profiler which measures how often individual lines of code are executed and the amount of time the execution took. To run the line profiler, use the Run → Run line profiler menu entry.

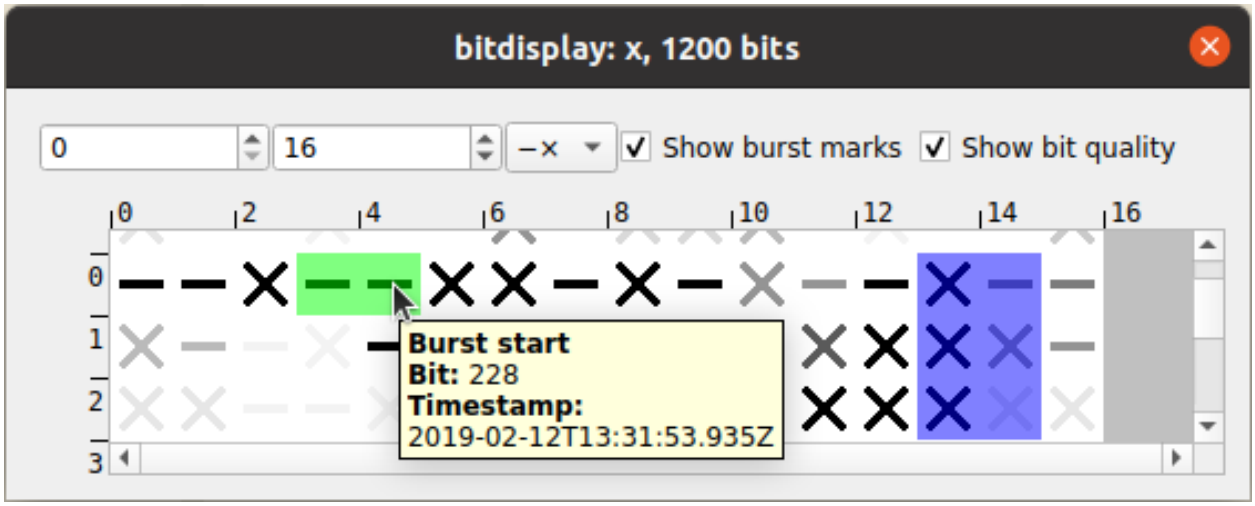


Figure 12: Graphical display of a BitBuffer

There are two requirements:

- The decoder must be configured for execution in `Decoder only` mode (see *Running a Decoder*).
- Each function to be profiled must be marked with a decorator specific to the line profiler. To do so, add `@profile` just before the function definition:

```
@profile
def func():
    pass
```

Results are presented in a tabular manner for each function (Fig. 13). They can by sorted by the available columns:

- `Line #`: Line number of code
- `Hits`: Number of times the line has been executed
- `Time (ms)`: The total amount of time execution of the line took, in millisecond
- `Per hit (ms)`: Average execution time of the line (`Time (ms)` divided by `Hits`), in millisecond
- `% Time`: `Time (ms)` relative to the total execution time of the function. The total execution time of the function is displayed in the header of each function entry. Entries with a high `% Time` have a background with a strong color, whereas entries with a low `% Time` use a pale color.

1.3.7. Known Spyder Issues

1.3.7.1. OpenGL Error

Starting Spyder may fail due to an OpenGL error as shown in Fig. 14 or the opened window stays black. The error mainly occurs on Windows systems and is related to graphic card drivers. To resolve the error perform the following steps:

- Open the file *<user directory>*`/AppData/Roaming/procitec/spyder/config/spyder.ini`. Note: Disable hiding of system directories in the options of Windows Explorer.
- Search for the line starting with `opengl =`
- Change the value after the equal sign from `automatic` to one of the following values (whichever works):

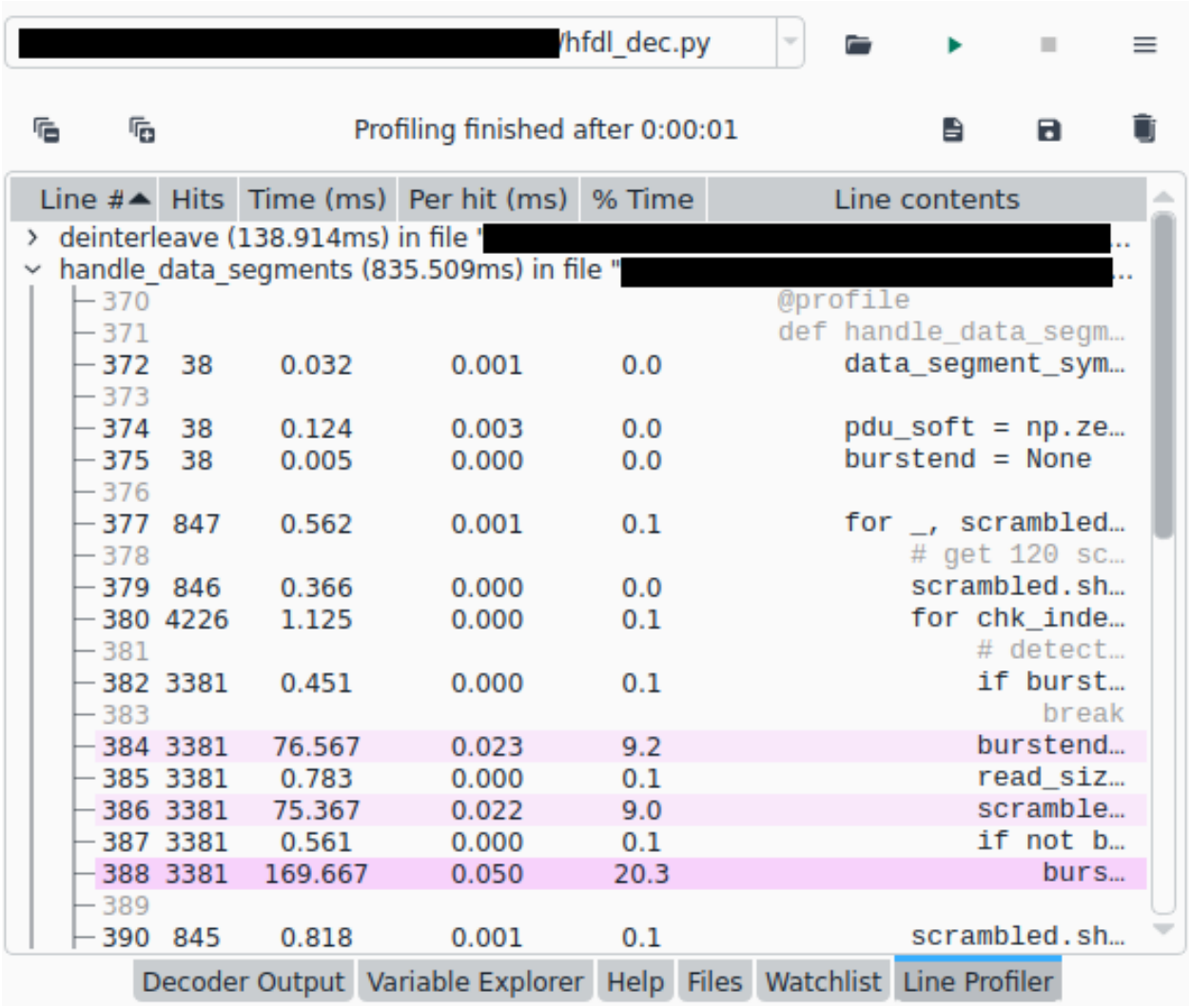


Figure 13: Example output of the line profiler

- `gles`
- `desktop`
- `software`

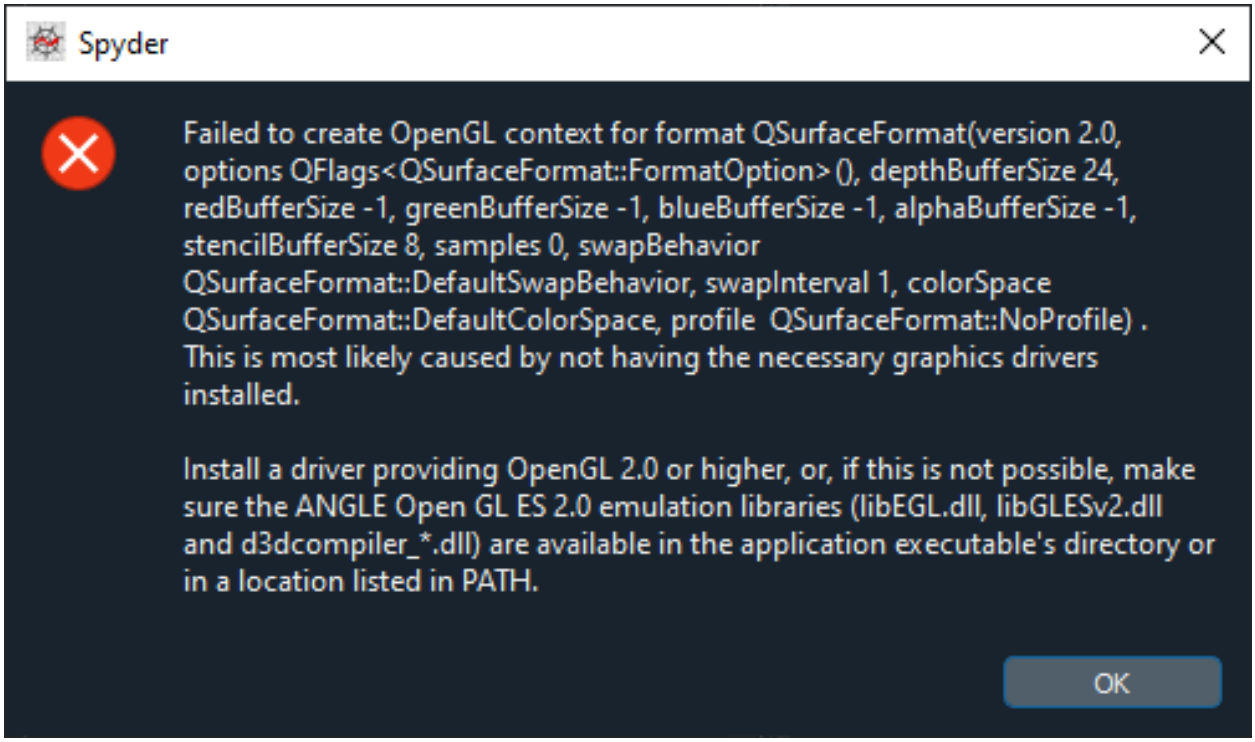


Figure 14: OpenGL error

1.4. Using Alternative IDEs

Beside using Spyder for decoder development (see *Decoder Development Using Spyder*), it is also possible to use other IDEs. The most convenient way to do so is to perform the following steps:

- Configure the IDE to use the Python interpreter provided by the *Analysis Suite* installation. The path depends on the used platform:
  - Windows: `C:\Program Files\procitec\analysis-suite\python`
  - Linux: `/opt/procitec/analysis-suite/python`.
- Add the following lines just below the `if __name__ == "__decoder__":` block in the decoder's main file:

```
if __name__ == "__main__":
    from procitec.decoding import runtime
    runtime.main(__file__, PATH_TO_REC_FILE)
```

  See *procitec.decoding.runtime.main()* for available options which control decoder execution and decoder output.
- Configure the IDE to run the decoder's main file as a script.

Alternatively, perform the following steps:

- Configure the IDE to use the Python interpreter provided by the *Analysis Suite* installation. The path depends on the used platform:
  - Windows: `C:\Program Files\procitec\analysis-suite\python`
  - Linux: `/opt/procitec/analysis-suite/python`.
- Configure the IDE to run the module `procitec.decoding.runtime`
- Configure the IDE to pass the following arguments to the module:
  - Path to a `.rec` file to be used as data input
  - Path to the decoder's main file
- Note: The runtime command line interface supports the same options as *`procitec.decoding.runtime.main()`*. Pass `--help` to see a description of the command line interface.

Further notes:

- The IDE may provide an option called "Emulate terminal". Enabling this option may provide better experience. It may be even necessary to enable this option depending on the used platform and IDE.
- When using a debugger, the `enable_timeout` argument of *`procitec.decoding.runtime.main()`* must be `False` (which is the default).

1.5. Customizing Python environments

Like any executable written in Python, pyDDL decoders inside applications like *Modem Lab*, go2MONITOR or Spyder utilize Python modules and/or packages for decoding execution. Depending on the application, different modules and packages are present in the respective Python environments. It's possible to add modules or packages to these environments to individually extend the decoders functionalities.

However, the path in which the custom modules/packages need to be installed into are application dependent.

1.5.1. Adding new packages

To install additional Python packages it is recommended to use the command line tool *pip*, the Package Installer for Python. One can install additional Python packages from the Python Package Index site if a working internet connection exists with

```
python -m pip install insert_package_name_here
```

The command should be carried out in a console window with **administrative privileges** as we need write access to the installation folder (for an alternative see *Non-priviliged package installation in Spyder*). Here it is important to call the python executable which is in the respective installation folder.

Analysis Suite
Windows: *C:\Program Files\procitec\analysis-suite\python*
Linux: */opt/procitec/analysis-suite/python*

go2MONITOR
Windows: *C:\Program Files\procitec\<go2APP>\python\bin\python*
Linux: */opt/procitec/go2monitor/python/bin/python*

It is necessary to repeat the above steps for every go2signals application for which the additional Python packages are needed.

Some Python packages consist of pure Python files and some contain, in addition, C/C++ libs. If the package maintainer does not provide a pre-compiled version of these libs for your specific platform then *pip* will download source code and compile it locally. A C/C++ toolchain is then a pre-requisite. Otherwise, the installation will fail.

To avoid downloading of source code packages it is possible to give *pip* an additional parameter *–only-binary*

```
python -m pip install --only-binary :all: insert_package_name_here
```

If there is no internet connection available *pip* allows to install a local distribution file. First download the distribution file from a computer with an internet connection.

```
python -m pip download --only-binary :all: insert_package_name_here -d insert_download_
↪folder_here
```

This will download the package *and* all of its dependencies in compressed wheel format (type .whl) and put them in the specified downlaod folder, e.g. *C:\tmp\downloads*.

Then copy this folder to the destination computer which has no internet connection and install the package there.

```
python -m pip install C:\tmp\downloads\insert_package_name_here.whl --find-links C:\tmp\
↪downloads --no-index
```

1.5.2. Non-priviliged package installation in Spyder

During the development of a decoder in Spyder it might be necessary to install the packages first in a user folder where no administrative privileges are needed. The installation should be carried out directly from the IPython console within the Spyder IDE.

```
import pip
pip.main(['install', 'insert_package_name_here', '--user'])
```

This will install the *insert_package_name_here* package and its dependencies in a hidden user folder.

Windows: *%APPDATA%Python\Python311\site-packages*
Linux: *~/.local/lib/python3.11/site-packages*

It is necessary to add this path to the PYTHONPATH in Spyder. Therefore, call *Tools->PYTHONPATH manager* from its menu and add the respective path.

To install into a package into a specific folder, e.g. *C:\users\username\my_decoder*

```
import pip
pip.main(['install', 'insert_package_name_here', '--target', 'C:\\users\\username\\my_
↪decoder'])
```

If the lcoal computer has no internet access and the packages were downloaded beforehand and put e.g. in the folder *C:\tmp\downloads*

```
import pip
pip.main(['install', 'c:\\tmp\\downloads\\insert_package_name_here.whl','--find-links',
↪'C:\\tmp\\downloads','--no-index','--user'])
```

Note: It is recommended to install the additional package directly into the same folder as the decoder. Then the new module will be automatically found when imported and then be put into the decoder package when the decoder is exported (if it is a pure Python package with no C/C++ libs). Otherwise this path must be added to PYTHONPATH, too.

To get more help for *pip* type

```
python -m pip help install
```

or consult the pip homepage.

1.5.3. Decoder module/package inspection in Spyder

As stated above, depending on whether a decoder is executed within Spyder or *Modem Lab*/go2MONITOR, the environment – and thus – Python modules/packages which are accessible from within the decoder differ. As a rule of thumb one can say that the Python environment within Spyder is more wide-ranging than *Modem Lab*/go2MONITOR's APC environment. This is due to the fact that Spyder is meant for development. A broader variety of modules (e.g. for plotting, analyzing, etc...) is therefore in the developer's favour, whereas for mere decoding, these modules are uncalled-for.

If a developer packs a decoder file (.py file) and its modules/packages within Spyder into a decoder package (.pkg file) containing modules/packages not present in *Modem Lab*/go2MONITOR's APC Python environment, a warning will pop up listing the modules/packages in question as seen below:

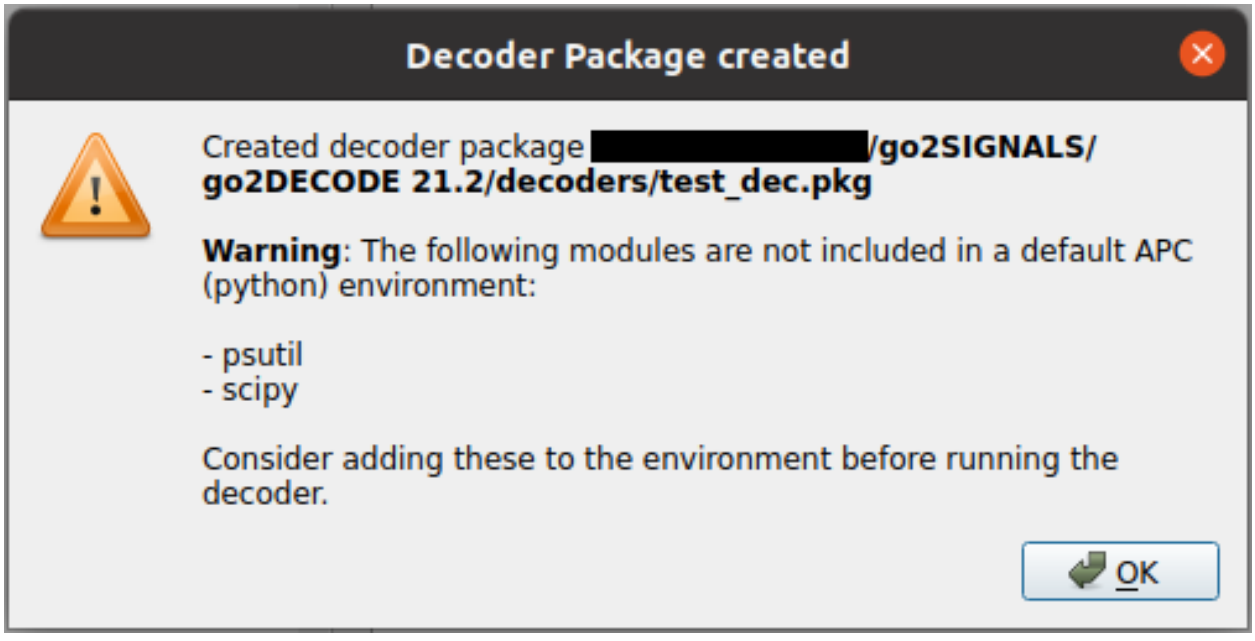


Figure 15: Decoder module/package warning in Spyder

For the decoder to run in *Modem Lab*/go2MONITOR, the developer will have two options:

- Remove the eligible modules/packages from the decoder

or

- Add the eligible modules/packages to the respective Python environment as outlined above.

Note

Module/Package inspection also takes place in CLI version of the packager. If a warning as seen above applies, the corresponding warning output will appear in the command line.

1.6. Using DLLs/Shared Libraries

- *Creating Shared Libraries*
- *Loading External Libraries*
- *Calling Function from Loaded Libraries*
  - *Passing Scalar Values by Value and by Pointer*
  - *Passing Arrays*
  - *Passing an array of pointers (multidimensional arrays)*
  - *Using Structures*
  - *Retaining Independent States and Using Classes*
- *Using Libraries Created for Non-Python DDL*

Python's `ctypes` module provides a way to call functions in dynamic-link libraries (DLLs) (*Microsoft Windows*) and shared libraries (*Unix*-like systems). This way you can incorporate existing third-party or your own libraries for special commands or additional functions such as interfacing functions or decryption.

This section provides a guide for the creation of external libraries and the usage of the `ctypes` module. For a full reference of the module see `ctypes`. All code examples presented hereafter can be found the folder `examples/decoder` inside the installation folder of the application (`testlib.cpp`, `userlib.h` and `dlltest_dec.py`).

1.6.1. Creating Shared Libraries

You can use a development system of your choice to create a shared library in C. In fact, you can use any programming language as long as there is a way to generate a shared library with a C application binary interface (ABI).

When using a C++ compiler you must prefix function declarations with `extern "C"`. C++ features affecting function declarations – templates, default arguments and overloading – can not be used. However, you can call functions using these features inside a function prefixed with `extern "C"`. This means that you can not use C++ features in the interface of a function, but you can use C++ features inside the function. Example:

```
#include <algorithm>

extern "C" int cpp_max(int a, int b)
{
    return std::max(a, b);
}
```

When using the *Microsoft Visual C++ (MSVC)* compiler you must prefix function which are to be accessed by `ctypes` by `__declspec(dllexport)`. In order to ease platform-independent development the following preprocessor macro can be used.

```
#ifdef _MSC_VER
    #define LIB_EXPORT __declspec(dllexport)
#else
    #define LIB_EXPORT
#endif
```

Use LIB_EXPORT as part of function declarations:

```
#include <algorithm>

extern "C" LIB_EXPORT int cpp_max(int a, int b)
{
    return std::max(a, b);
}
```

1.6.2. Loading External Libraries

A shared library can be loaded by instantiating the CDLL class. As argument a full path to the library, including system-specific portions, must be provided. We recommend the usage of the helper function *find_library()* to write portable (system-independent) decoders which do not depend on an absolute path; see *find_library()* for details.

```
>>> import ctypes
>>> from procitec.common import find_library
>>> lib = ctypes.CDLL(find_library("dlltest"))
```

1.6.3. Calling Function from Loaded Libraries

Functions from shared libraries are accessed as attributes of the CDLL instance, e.g.:

```
>>> result = lib.function_name(1, None) # pass an int and NULL pointer
```

Some Python data types can be passed directly as parameters in these functions calls. By default functions are assumed to return the C type `int`. This means that often no further steps are required to call functions after loading a library. However, we recommend to provide the required argument types and the return type. This is done by settings the `argtypes` respectively the `restype` attribute of the function:

```
>>> # function expects an int and void* and returns a long
>>> lib.function_name.argtypes = [ctypes.c_int, ctypes.c_void_p]
>>> lib.function_name.restype = ctypes.c_long
```

Doing so protects against wrong argument types and enables conversions of arguments to correct types if possible.

The following sections provide examples for calling of functions using the `ctypes` module.

1.6.3.1. Passing Scalar Values by Value and by Pointer

Assume the following C function

```
extern "C" LIB_EXPORT float simple(int a, float *b)
{
    *b = *b - 1;
    return a + *b;
}
```

where so called "fundamental data types" (`int`, `short`, `float` etc.) are passed by value and by pointer. Passing values by pointer (also known as "by reference") allows the manipulation of the corresponding memory location. This works as expected when interfacing the function through the `ctypes` module.

The initialization of the function in Python is as follows

```
>>> # function expects an int and float* and returns a float
>>> lib.simple.argtypes = [ctypes.c_int, ctypes.POINTER(ctypes.c_float)]
>>> lib.simple.restype = ctypes.c_float
```

Usage of `ctypes.POINTER()` indicates that the given type is passed by pointer. See fundamental data types for a list of supported data types. Make sure that the `ctypes` and C data types match. This is especially important for the types `short`, `int`, `long` and `long long` whose size is implementation defined in C and C++.

The function can be invoked using the following code

```
>>> a = 10
>>> b = ctypes.c_float(1)
>>> result = lib.simple(a, ctypes.byref(b)) # or lib.simple(10, b)
>>> assert b.value == 0.0
>>> assert result == a
```

Python's `int` data type can be passed directly assuming that the argument type in C is one of the possible integer types in C. Passing values by pointer requires the creation of a `ctypes` type and usage of `ctypes.byref()`. A function requiring a pointer argument – as set in `argtypes` – also accepts an object of the pointed-to type. The required `byref()` conversion is applied automatically (see type conversions in ctypes for details).

The modification of the pointed-to value in the C code works just as expected. To pass a `NULL` pointer to a C function use `None` in Python (directly, calling `ctypes.byref()` is not required).

1.6.3.2. Passing Arrays

Assume the following C function

```
extern "C" LIB_EXPORT int sum(int *arr, size_t len)
{
    int tmp = 0;
    for ( size_t i = 0; i < len; ++i )
    {
      tmp += arr[i];
      arr[i] = 0;
    }
    return tmp;
}
```

The initialization of the function in Python is as follows

```
>>> lib.sum.argtypes = [ctypes.POINTER(ctypes.c_int), ctypes.c_size_t]
>>> lib.sum.restype = ctypes.c_int
```

The function can be invoked using the following code

```
>>> array = (ctypes.c_int * 30)()
>>> for i in range(len(array)):
>>>     array[i] = i
>>> result = lib.sum(array, len(array))
>>> assert sum(range(len(array))) == result
>>> assert all([val == 0 for val in array])
```

Array types in `ctypes` are created by multiplying a data type with a positive integer (which is the length of the array). You must call the constructor of the resulting class to create an instance of the array type. These instances can be passed to C functions. As in C you can pass an array to a function accepting a pointer (see type conversions in ctypes for details).

Elements of `ctypes` arrays can be read written using the standard subscript access. Bound checks are performed just like with an ordinary Python `list` or `tuple`.

If the C function *always* expects an array of a specific size, the corresponding type in `argtypes` should be `ctypes.<type> * length`. This prevents passing of an array with wrong size.

1.6.3.3. Passing an array of pointers (multidimensional arrays)

Assume the following C function which depends on code from the previous section

```
extern "C" LIB_EXPORT int sum_arrays(int **arrays, size_t *array_size, size_t array_
↪count)
{
  int tmp = 0;
  for ( size_t i = 0; i < array_count; ++i )
      tmp += sum(arrays[i], array_size[i]);
  return tmp;
}
```

The initialization of the function in Python is as follows

```
>>> lib.sum_arrays.argtypes = [
>>>     ctypes.POINTER(ctypes.POINTER(ctypes.c_int)),
>>>     ctypes.POINTER(ctypes.c_size_t),
>>>     ctypes.c_size_t,
>>> ]
>>> lib.sum_arrays.restype = ctypes.c_int
```

In order to pass 2 arrays to the C function, first define an array which will hold pointers pointing to the first element of the arrays:

```
>>> arrays = (ctypes.POINTER(ctypes.c_int) * 2)()
```

Initialize differently sized arrays:

```
>>> array0 = (ctypes.c_int * 30)()
>>> for i in range(len(array0)):
...     array0[i] = i
...
>>> array1 = (ctypes.c_int * 10)()
>>> for i in range(len(array1)):
...     array1[i] = i
...
```

Set array pointers:

```
>>> arrays[0] = array0
>>> arrays[1] = array1
```

Initialize the array which holds the lengths of the arrays passed to the function:

```
>>> arrays_len = (ctypes.c_size_t * len(arrays))()
>>> arrays_len[0] = len(array0)
>>> arrays_len[1] = len(array1)
```

Finally invoke the function by:

```
>>> result = lib.sum_arrays(arrays, arrays_len, 2)
>>> assert result == sum(range(10)) + sum(range(30))
>>> assert all([val == 0 for val in array0])
>>> assert all([val == 0 for val in array1])
```

1.6.3.4. Using Structures

Assume the following `struct` declaration

```
typedef struct
{
    double x;
    double y;
} vector_t;
```

The corresponding structure definition for `ctypes` is as follows

```
>>> class vector_t(ctypes.Structure):
>>>     _fields_ = [("x", ctypes.c_double),
...                 ("y", ctypes.c_double)]
...
```

`ctypes` structures are defined by deriving from `ctypes.Structure` and defining the `_fields_` attribute. Each element in the list of `_fields_` must be a 2-tuple containing the field name and the field type. The field type can be any `ctypes` type.

Instances of structure types in Python are created by calling the constructor of the defined type. Passed arguments are used to initialize the members of the structure in the same order as they appear in `_fields_`. You can also pass keyword arguments (in any order) which will set the corresponding member. (These rules correspond to the initialization rules for structures in C.) Fields of the structure can be read and modified using attribute access.

```
>>> vec = vector_t(0, 1.0)
>>> print(vec.x, vec.y)
0.0 1.0
>>> vec = vector_t(y = 2.0, x = 1.0)
>>> vec.x = 3.0
>>> print(vec.x, vec.y)
3.0 2.0
```

As an example for the usage of structures in `ctypes` the following code is assumed

```
#include <cmath> // C++ compiler is assumed

extern "C" LIB_EXPORT double vector_length(vector_t vec)
{
    return std::hypot(vec.x, vec.y); // requires a C++ compiler
}

extern "C" LIB_EXPORT void reset_vector(vector_t* vec)
{
    vec->x = 0.;
    vec->y = 0.;
}
```

The initialization of the functions in Python is as follows

```
>>> lib.vector_length.argtypes = [vector_t]
>>> lib.vector_length.restype = ctypes.c_double
>>>
>>> lib.reset_vector.argtypes = [ctypes.POINTER(vector_t)]
>>> lib.reset_vector.restype = None
```

A return type of `None` indicates that the C function's return type is `void`. The functions can be invoked as shown in *Passing Scalar Values by Value and by Pointer*:

```
>>> vec = vector_t(1.0, 1.0)
>>> result = lib.vector_length(vec) # pass by value
>>> assert math.isclose(result, math.hypot(1.0, 1.0))
>>>
>>> lib.reset_vector(vec) # pass by pointer; byref() is applied automatically
>>> assert vec.x == 0.0 and vec.y == 0.0
```

A function requiring a pointer argument – as set in `argtypes` – also accepts an object of the pointed-to type. The required `byref()` conversion is applied automatically (see type conversions in ctypes for details).

1.6.3.5. Retaining Independent States and Using Classes

When working with a shared library it may be necessary to retain a state (a data structure) which is passed to different functions. Moreover, there may be a need to maintain multiple independent states, e.g. in order to decrypt multiple independent data streams. The state variable is most likely a C `struct` or a C++ `class` which must be initialized in some way.

A possible solution look as follows:

Create an initialization function which initializes the required data structure. If the data structure is a C++ `class`, the function should either return a `void` pointer to an instance of the class or the pointer should be saved in a `void **` passed as an argument. If the data structure is a C `struct`, you can use a pointer to the `struct` type and the techniques presented in *Using Structures*.

An example using a `void` pointer:

```
extern "C" LIB_EXPORT void init(void** instance)
{
    *instance = new MyClass;
}

/* or a function returning a pointer
extern "C" LIB_EXPORT void* init()
```

(continues on next page)

(continued from previous page)

```
{
    return new Data;
}
*/
```

The initialization of the function in Python is as follows

```
>>> lib.init.argtypes = [ctypes.POINTER(ctypes.c_void_p)]
>>> lib.init.restype = None
>>> # or lib.init.restype = ctypes.c_void_p if the initialization function return a␣
↪pointer
```

The pointer created by the C function must be saved in a variable in Python

```
>>> instance_pointer = ctypes.c_void_p()
>>> lib.init(ctypes.byref(instance_pointer))
>>> # or instance_pointer = lib.init()
```

The instance pointer can now be passed to every function requiring access to the data structure. If the data structure has been created using dynamic memory allocation (`malloc()` or `new`), you must also provide a function to free the corresponding memory.

The technique presented above can be used to make use of classes even though only C can be used in the interface of functions: For each member function you would like to access, create a function with the required arguments. As an additional argument a `void` pointer pointing to an instance of the class must be passed.

1.6.4. Using Libraries Created for Non-Python DDL

Non-Python *Decoder Description Language (DDL)* requires a specific function interface to make use of shared libraries (see DDL Manual chapter 5.23):

```
#define MAX_PARA 30
#define ERR_TXT_LEN 256

extern "C" LIB_EXPORT int myFunc(int* piValue[MAX_PARA],
                                 const int aiLen[MAX_PARA],
                                 const VARTYPE_T eType[MAX_PARA],
                                 char acErrText[ERR_TXT_LEN])
{
    // ...
}
```

The initialization of the function in Python is as follows

```
>>> lib.myFunc.argtypes = [
...     ctypes.POINTER(ctypes.c_int) * 30, # array of pointers to int
...     ctypes.c_int * 30,                 # array of int (lengths of arrays above)
...     ctypes.c_int * 30,                 # VARTYPE_T enum
...     ctypes.c_char * 256,               # array of char
... ]
>>> lib.myFunc.restype = ctypes.c_int
```

For a complete example for the usage see the function `test_myFunc`, defined in `dlltest_dec.py` in the folder `examples/decoder` inside the installation folder of the application.

1.7. Tools for testing, executing and packaging of decoders

For decoder development Spyder can be used to implement, execute and test a decoder with offline input data. It is also possible to bundle all needed modules of a decoder into a single file called decoder package. For information about using Spyder see *Decoder Development Using Spyder*.

Besides using Spyder, there are other ways to run a decoder or bundle them together into a decoder package. This can be used for automatic testing and deployment of decoders.

In all cases the Python executable supplied by the **Spyder Decoder Development Environment** is required. The executable can be found in the following places within the installation folder

- Windows: *C:\Program Files\procitec\analysis-suite\python*
- Linux: */opt/procitec/analysis-suite/python*

1.7.1. Executing a decoder

There are two ways to execute a decoder

- Using a Python script
- Using the command line

A decoder typically produces two outputs. One output is textual and in XML format. The other is file-based and writes files to a specified directory. For example, received binary files or decoded voice are stored there.

See also *procitec.decoding.runtime.ProductionMemory*

1.7.1.1. Using a Python script

The function *procitec.decoding.runtime.run()* can be used to execute a decoder. In the following example the decoder package baudot11_dec.pkg is run with the input data from the symbol recording file baudot11.rec. The output XML is written to standard output.

```
from procitec.decoding.runtime import run
run( "/tmp/baudot11_dec.pkg", "/tmp/baudot11.rec" )
```

To capture the output into a variable and enable file output to production memory (memprod) additional parameters and some glue objects have to be provided. It is also possible to specify parameters accessible in the decoder. Especially for long running decoders the result handler is handy, because it allows access to progress and intermediate results.

Changed in version 24.2.0: Production memory no longer needs to be explicitly created.

```
from procitec.decoding.runtime import run

class ResultHandler:
    def __init__(self):
        self.result_buffer = ""

    def write(self, new_results: str) -> None:
        self.result_buffer += new_results

    def set_progress(self, progress: float) -> None:
        print(f"{progress:2.1%}")

    def store(self, path: str) -> None:
```

(continues on next page)

(continued from previous page)

```
        open(path, "w").write(self.result_buffer)


result_handler = ResultHandler()
decoder_parameters = {"var1": "string xy", "var2": 123}

run(
    "/tmp/baudot11_dec.pkg",
    "/tmp/baudot11.rec",
    run_status=result_handler,
    mem_prod_path="/tmp/memprod/",
    parameters=decoder_parameters,
)

result_handler.store("/tmp/decoder_result.xml")
```

Note

Besides a decoder package (*_dec.pkg) also a plain decoder (*_dec.py) file can be used.

1.7.1.2. Using the command line

The package *procitec.decoding.runtime* itself can be executed directly on the command line Using the same input as in the example above:

```
python -m procitec.decoding.runtime --mem_prod_dir "/tmp/memprod/" -p var1="string xy" -
↪p var2=123 "/tmp/baudot11.rec" "/tmp/baudot11_dec.pkg" >> "/tmp/decoder_result.xml"
```

There is also a help page available:

```
python -m procitec.decoding.runtime -h
```

1.7.2. Bundle a decoder into a decoder package

There are two ways to bundle a decoder into a decoder package

- Using a python script
- Using the command line

See also *Creating and Using a New Decoder* (using Spyder).

1.7.2.1. Using a Python script

The function *procitec.packaging.builder.packager.create_decoder_package()* can be used to create a decoder package. In the following example the decoder package baudot11_dec.pkg is created from the decoder baudot11_dec.py

```
from procitec.packaging.builder.packager import create_decoder_package
package_path, non_env_module_names = create_decoder_package("/tmp/baudot11_dec.py",␣
↪output_path="/tmp/", comment="comment1")
```

1.7.2.2. Using the command line

The package `procitec.packaging.builder` itself can be executed directly on the command line Using the same input as in the example above:

```
python -m procitec.packaging.builder --output "/tmp" "/tmp/baudot11_dec.py"
```

There is also a help page available:

```
python -m procitec.packaging.builder -h
```

2. Reference

This chapter contains detailed description of the build-in functionalities for decoder implementation. These are split in three different parts: The decoder runtime describes the interface for the data input and output as well as the communication of other parts of the processing system, like the demodulator or search state handling. The next part, BitBuffer describes the main data type for handling bit-level data and functions for low-level data manipulation. Finally, the Decoding Library contains higher-level functions for common decoding tasks.

2.1. Decoder Runtime

This is the documentation for the Python module *procitec.decoding.runtime*.

This module contains types and functions to access input data, produce results or query or set runtime states like the search state or demodulator parameters.

- *Top-Level API*
- *Input*
- *Output*
- *State and Parameters*
- *File Output*
- *Audio Output*
- *Standalone runtime*

2.1.1. Top-Level API

The main interface for interaction with the decoder runtime is an instance of *APCGateway*. It is obtained by by calling *APC()* and usually named apc:

```
>>> from procitec.decoding import runtime
>>> apc = runtime.APC()
>>> apc.data.read(8).to_str()
00000000
>>> apc.output.text1("I have read 8 bits")
```

The above code shows how to obtain the interface object and exemplary usage. The functionality is often grouped into sub-objects, e.g. *apc.output* which contains all of the text output related functionality. *apc.output* is an instance of *Output* which is *documented below*.

procitec.decoding.runtime.**APC**(** (Keyword-only parameters separator (PEP 3102)), decoder_version: str = '1.0.0', namespaces: Sequence[str] = (), **kwargs: str*) → *APCGateway*

Get the instance of the decoder runtime API and set some common decoder properties

Parameters

- **decoder_version** (*str*) – set the decoder version in *DecoderProperties*
- **namespaces** (*list*) – list of namespaces this decoder will use in its output
- ****kwargs** – additional parameters to forward to *apc.decoder_properties()*

Returns
instance of the decoder runtime API for this decoder

Return type
APCGateway

class procitec.decoding.runtime.**APCGateway**

An object which acts as a gateway between a decoder and the APC

burst_end_time(*when*)

See *ignore_burst_detector()* for details.

Note: *flush* is called internally to ensure that a corresponding pair of burst start and end time is sent to output. So if *burst_end_time()* is called when an *EndOfDataError* occurs any output afterwards will be discarded.

Parameters
when (ProTS *or int*) – A timestamp or an absolute bit position in the input stream where the end of a burst occurs.

burst_start_time(*when*)

See *ignore_burst_detector()* for details.

Parameters
when (ProTS *or int*) – A timestamp or an absolute bit position in the input stream where the start of a burst occurs.

extend_search(*extension*, ***, *reference=None*)

Extend the maximum time of search to the specified amount of bits relative to the current read position or optionally the specified reference position.

The search phase is normally limited to around 30 seconds, i.e. if the decoder does not send a positive or negative identification message (i.e. *SearchStateHandler.ident()*, *SearchStateHandler.no_sync()*, etc.) within 30 seconds during the search phase, the search will be aborted. This command can be used to extend the maximum time of search beyond 30 seconds.

Parameters

- **extension** (*int*) – Minimum time of search phase in bits, starting at current position or given *reference*.
- **reference** (*int, optional*) – Input buffer index designating an absolute reference position for the extension parameter *extension*

Notes

This parameter influences the search behavior of _all_ modems currently in the search list and can lead to longer search times even if one modem is identified (search best mode in HF). Use with care.

`ignore_burst_detector`(*ignore_start=True*, *ignore_end=True*)

If a burst demodulator is used, the demodulator's measurement of burst start and end times can be overwritten by the decoder. Either the burst start, the burst end or both times can be ignored and replaced by new values using *burst_start_time()* respectively *burst_end_time()*. This may be necessary to provide correct and precise information about the timing of bursts if it cannot be produced correctly by the demodulator or if the demodulator's results are not reliable. This also may be necessary if a continuous demodulator is used for burst signals.

The burst timing information will be reported in the XML stream of the production results. For more details refer to the document PROCITEC-ICD-APC_Results-E.pdf.

Parameters

- **ignore_start** (*bool, optional*) – Ignore start times of bursts determined by the demodulator if True. The start time of bursts must be provided using *burst_start_time()*.
- **ignore_end** (*bool, optional*) – Ignore end times of bursts determined by the demodulator if True. The end time of bursts must be provided using *burst_end_time()*.

`init_codec`(*name*, *raw_file_postifx*, *index*)

Initialize an audio codec as a decoder's output

Created file type depends on availability of audio codec:

- A `.codec` file which contains raw (encoded) audio frames if requested audio codec is **not** supported
- A `.wav` file which contains decoded audio frames if requested audio codec is supported

Parameters

- **name** (*str*) – Name of audio codec

  Available codecs are: AMBE_DVSI_HR (half rate, AMBE+2), AMBE_DVSI_FR (full rate, IMBE),``G721``, TETRA
- **raw_name_postfix** (*str*) – Part of the filename of generated `.codec` file. For example, this can be used to add the codec's name to the filename.
- **index** (*int*) – Must be greater than 0. Part of the filename of generated `.codec` and `.wav` files. For example, this can be used to differentiate between different slots in a time-division multiple access (TDMA) system.

Return type
AudioCodec

`input_channel_mode`(*interleaved=True*, *selected=[]*)

Input channel mode configuration

Parameters

- **interleaved** (*bool, optional*) – If True (default), all demodulated channels are interleaved into a single channel in *SymbolStream* and *BitStream*. Otherwise, the demodulated channels are provided by separate channels. N symbols from M demodulated channels are provided as follows, where `s[i,k]` is symbol k from demodulated channel i:

  - With interleaving (default): Symbol-/InputStream channel 0: `[s[0,0], s[1, 0], ..., s[M-1,0], s[0,1], s[1,1], ..., s[M-1,1], ..., s[0,N-1], s[1,N-1], ..., s[M-1,N-1]]`
  - Without interleaving:
    * Symbol-/InputStream channel 0: `[s[0,0], s[0,1], ..., s[0,N-1]]`
    * Symbol-/InputStream channel 1: `[s[1,0], s[1,1], ..., s[1,N-1]]`
    * ...
    * Symbol-/InputStream channel M-1: `[s[M-1,0], s[M-1,1], ..., s[M-1, N-1]]`

- `selected` (*`list(int), optional`*) – If the list is empty (default), all channels are selected, otherwise only the listed channels are provided in the Symbol- and InputStream. The values in the list must be monotonically increasing and they must be in the range `[0; M-1]`, where `M` is the number of available demodulated channels.

  For example with `M = 8` and `selected = [0, 4, 7]` only the demodulated channels 0, 4 and 7 are provided. Note that they are accessed by indices 0, 1 and 2 in the Symbol- and InputStream.

Notes

Input channel mode must be set **before** reading any data. It cannot be changed afterwards as it directly affects the internal bit storage. This is especially important when only some channels were initially selected and one want to change this later. In this case it is recommended to use the `read`/`peek` methods with the parameter `channel_index`.

`input_channel_sync`(*channel=0*)

In case of enabled interleaving, this function consumes all symbols up to the first symbol, belonging to the specified channel. If the next to be read symbol already belongs to the specified channel or interleaving is not enabled, the function has no effect.

`preprocessing`(**args*)

Set the preprocessing steps to be executed before bits are placed in *`data`*.

Preprocessing steps are created with the functions in *`procitec.decoding.preprocessing`*. To support conditional steps (e.g. via a conditional expression) arguments with value `None` are ignored.

This function can only be called during the initialization of decoder. That is, before any data input is read.

When *`apc.demodulator.modify`* is executed, then input buffers including preprocessing are cleared. In this case *`apc.preprocessing()`* with specification of preprocessing chain must be issued again.

`set_production_hold_time`(*timeout*)

One criterion to stop a modem production is loss of signal. Normally this will be assumed if more than 5 seconds have elapsed without a reasonable signal. The decoder can hold production by commands sync, ident or accept if the demodulator works in a continuous mode. In burst mode however this is not possible, as the decoder has no data input and output during the time of burst gaps. This results normally in a production termination whenever a burst gap longer than 5 seconds occurs.

In some cases this might be an unwanted behavior. By this command the 5 seconds time limit can be extended or shortened.

Parameters
timeout (*int*) – Timeout value in milliseconds.

`set_timeout`(*timeout*)

A timeout error occur if too much time elapses within the decoder without returning control to the APC instance.

The default time limit for this period is 5 seconds. Timeouts can be caused by programming errors, like endless loops. However in rare cases this can happen also for correct programs. The reason could be routines consuming much calculation time and/or processing of very long data packages. For such cases the timeout limit can be extended to any other value above 0.25 seconds. However this must be handled with care. Too long timeouts can also cause other real time problems or hang ups of the complete program.

Parameters
timeout (*float*) – Timeout value in seconds.

`property current_mode`

Current mode of the APC

The mode is one of the following types:

- *APCMode.SEARCH*
- *APCMode.PRODUCTION*

Examples

A string representation of the APC mode can be retrieved by accessing the attribute `name` of the returned value, i.e.

```
>>> apc.current_mode.name
'PRODUCTION'
```

Conditions on the current mode can be implemented, using *APCMode*, i.e.

```
>>> from procitec.decoding import runtime
>>> if apc.current_mode == runtime.APCMode.PRODUCTION:
>>>     ...
>>> else:
>>>     ...
```

Type
APCMode

`property data`

Input bit stream, i.e. the bit stream after symbol decision

Type
procitec.common.bitbuffer.BitStream

`property decoder_properties`

add meta data to the decoder output

Type
DecoderProperties

`property decoder_version`

access decoder version as set in the constructor

Type
str

`property demodulator`

Interface to the demodulator

Type
DemodControl

`property interleaved_channels`

Number of interleaved channels

Type
int

`property modem`

Provides functions to report the state of the decoder

Type
SearchStateHandler

`property output`

Provides functions to generate textual decoder output

Type
Output

`property parameters`

Decoder parameters for this modem

The object is a read-only mapping (`dict`) supporting all methods of `collections.abc.Mapping` except for `values()`, `__eq__()` and `__ne__()`.

Type
dict-like

`property production_memory`

Provides functions to generate files as a decoder output

Type
ProductionMemory

`property receiver_frequency`

Current absolute receiving frequency of demodulator. Returned value is `None` if receiver frequency is not available (e.g. replay from recorded bit file). Note that this property should only be queried after a data exchange has happened. It is recommended to call `apc.data.peek(1)`

Type
Optional[float]

`property symbols`

Input symbol stream generated by the demodulator

Type
SymbolStream

`property version`

APC version

Type
str

2.1.2. Input

The main bit-level input class is *procitec.common.bitbuffer.BitStream* which is documented as part of the *procitec.common.bitbuffer*. The class *SymbolStream* here contains symbol-level input data and acts as source for *apc.data*. However, the `position` in these streams is always synchronized.

`class procitec.decoding.runtime.SymbolStream`

An object which provides access to a symbol stream generated by a demodulator

`peek`(*out*, *channel_index=0*)

Peek into the symbol stream

Peek into the stream (read from the stream) without consuming any data, i.e. the position in the stream is not modified.

This is the underlying function for *read()*.

Parameters

- `out` (*buffer*) – A buffer object with a dimension of 2. The length of the first dimension determines the number of symbols to read. The length of the second dimension must be 2. The meaning of these two elements depends on the sample format; see *sample_format*. The elements in the buffer must of format `uint16_t`.
- `channel_index` (*int, optional*) – The channel to read from. By default (0) the first channel is read.

Returns

Number of symbols written into the output buffer. An *EndOfDataError* is thrown if the stream is closed (see *closed*) and if additionally one of the following applies:

- The internal buffer does *not* contain the requested amount of symbols (`size` parameter greater than 0). (If the buffer does contain the requested amount of symbols they will be returned even if the stream is closed).
- The internal buffer is empty (when `size` parameter is zero).

Return type
int

`read`(*size=None*, *channel=0*)

Read symbols from the demodulator

This is equivalent to a *peek()* combined with a `consume()`.

Parameters

- `size` (*int, optional*) – Number of symbols to read. If *None* (default), all currently available symbols are read.
- `channel` (*int, optional*) – The channel to read from. By default (0) the first channel is read.

Returns

Symbols from demodulator. An *EndOfDataError* is thrown if the stream is closed (see *closed*) and if additionally one of the following applies:

- The internal buffer does *not* contain the requested amount of symbols (`size` parameter greater than 0). (If the buffer does contain the requested amount of symbols they will be returned even if the stream is closed).

- The internal buffer is empty (when `size` parameter is zero).

Each element in the list is a list with two elements. The meaning of these two elements depends on the sample format; see *sample_format*.

Return type
list(list(int))

`rewind()`

Rewind the stream by the given number of symbols, if possible. Throws `procitec.common.bitbuffer.RewindError` if not enough symbols are buffered.

`set_position`(*position: int*) → int

Rewind or consume symbols to get to the given absolute position, if possible. Returns the new position or throws `procitec.common.bitbuffer.RewindError` if not enough symbols are buffered.

Parameters
position (*int*) – The absolute position to jump to.

Returns
The new position

Return type
int

`symbol_rate`(*position: int | None = None*) → float | None

Get symbol rate of symbol at given absolute position, if available.

Added in version 24.1.0.

Parameters
position (*int, optional*) – Position of the symbol for which the symbol rate is to be determined.

Returns
Symbol rate in baud (1/s) of the symbol at the requested position.

Return type
float | None

`time`(*position: int*) → *procitec.common.ProTS*

Get timestamp of symbol at given absolute position

Parameters
position (*int*) – Position of the symbol for which the timestamp is to be determined.

Returns
Timestamp of the symbol at the requested position.

Return type
procitec.common.ProTS

`property bits_per_symbol`

Bits per symbol

Type
int

`property channels`

Number of available channels

Type
int

property closed

True if the symbol stream is closed, False otherwise

Type
bool

property position

Returns the total number of consumed symbols

Type
int

property sample_format

Sample format returned by *read()*

The sample format is one of the following types:

- SymbolStream.FSK
- SymbolStream.I_Q
- SymbolStream.MAG_PHASE

A string representation of the sample format can be retrieved by accessing the attribute name of the returned value, i.e.

```
>>> format = apc.symbols.sample_format
>>> format.name
'MAG_PHASE'
```

Type
SampleFormat

2.1.3. Output

The text output functionality is provided by *apc.output* which allows decoders to report text-based decoding result streams. In addition, structured output of decoding result can be done with *Output.xml()*.

Multiple of output streams may be used in parallel, each is an instance of *OutputChannel*. These can be obtained via item or attribute access on *apc.output*:

```
>>> apc.output.text1("foo")
>>> apc.output["text1"]("foo")
>>> std_out = apc.output.text1
>>> std_out("foo")
```

class procitec.decoding.runtime.Output

Object which holds and maintains channels for decoder output

Output channels for a decoder can be added using *declare_channel()*. Output performed to a specific channel will be enclosed in a XML element whose name is the channel's name. Every defined channel is listed inside the XML element decoderProperties.

The channels text1, notify and status are always predefined.

The following output channels are used for a specific purpose and displayed in the graphical user interfaces in a certain way.

- text1 / textX: The decoding result in generic textual format. Multiple text channels can be used.
- notify: Output of notifications directed to the user. These notifications should not contain any decoded content.

- `status`: The status channel includes important parameters, results and live status of the active decoder. Examples of the decoder status are the currently active transmission type (idle, data, voice, ...), recognized decoding modes (interleaving, scrambling, code rate, ...) or network and country codes. Writing to the status channel always overwrites the last status. If the same status string is written again, no output is made and the content does not occur in the result xml.

Examples

Declare a channel called `channel_foo`:

```
>>> apc.output.declare_channel("channel_foo", "Output channel foo")
```

The channel can be accessed using its name as an attribute or as an index in the *Output* object:

```
>>> apc.output.channel_foo.write("bar")
>>> apc.output["channel_foo"].write("bar")
```

A channel object (*OutputChannel*) can also be saved in a variable and used instead:

```
>>> out_foo = apc.output.channel_foo
>>> out_foo.write("bar")
```

Channel objects also support a function call interface (see `object.__call__()`) which is equivalent to using *OutputChannel.write()*:

```
>>> out_foo("bar") # equivalent to out_foo.write("bar")
```

`__getattr__`(*name: str*) → *OutputChannel*

Access a specific output channel

Parameters
name (*str*) – Name of the channel to be accessed

Returns
Output channel object

Return type
OutputChannel

`__getitem__`(*name*)

see *__getattr__()*

`declare_channel`(*name: str*, *description: str*, ***, *description_de: str | None = None*)

Declare an output channel for the decoder

Declaring an already existing channel (determined by `name`) is an error. The name must not be `Error` (reserved for error reporting).

Parameters

- **name** (*str*) – Name of the output channel. Only the characters A-Z, a-z, 0-9 and :_.- are allowed, however, the name must not start with 0-9 or .-. The channel name must not be `Error` (reserved for error reporting).

- **description** (*str*) – Description of the channel which will appear in decoderProperties.
- **description_de** (*str, optional*) – German description of the channel which will appear in decoderProperties.

flush()

Enforce immediate flush of output. Usually this is done automatically and no manual intervention is necessary. It is recommended to use this command only in special cases.

new_info_unit(*time:* ProTS) → None

Determines that from the given time position the decoder output can be split into a new emission.

Parameters

time (ProTS) – Timestamp of the new info unit.

xml(*element_or_name*, *content=None*, ***kwargs*)

Append XML data to the output of a decoder

Parameters

- **element_or_name** (*str or xml.etree.ElementTree.Element*) – If an instance of Element is passed it is serialized to XML. Other arguments must not be passed.

  If a string is passed a XML element with the name element_or_name is constructed. The content of the element is determined by content and **kwargs.
- **content** (*object, optional*) – May only be used if element_or_name is a string. The string representation of object (see str) defines the content of the generated XML element. If content is None (default) the element's content will be empty.
- ****kwargs** – May only be used if element_or_name is a string. The keyword arguments define the attributes of the generated XML element. The string representation of an argument's value (see str) is used as an attribute's value. If no keyword arguments are passed no attributes will be added to the generated XML element.

Examples

In the simple mode the name of the XML element, its content and attributes are passed:

```
>>> apc.output.xml("frequency", 12000, unit="Hz", mode="full")
```

This will generate the following XML element in the decoders output:

```
<frequency unit="Hz" mode="full">12000</frequency>
```

Attributes and content are optional:

```
>>> apc.output.xml("mode_switch", counter=10)
>>> apc.output.xml("resync")
```

```
<mode_switch counter="10" />
<resync />
```

More complex (nested) XML elements can be produced using Python's `xml.etree.ElementTree` module and passing the resulting `xml.etree.ElementTree.Element` object:

```
>>> from xml.etree.ElementTree import Element, SubElement
>>> frame = Element("frame", count="10")
>>> mode = SubElement(frame, "mode")
>>> mode.text = "full"
>>> apc.output.xml(frame)
```

```
<frame count="10"><mode>full</mode></frame>
```

See `xml.etree.ElementTree` for more details.

class procitec.decoding.runtime.OutputChannel

An object representing an output channel of a decoder; see *Output* for details and usage.

__call__(*object*)

see *write()*

write(*data: str*) → None

Write output to the channel.

In the case of the `status` channel, repeated writing of the same string data is ignored as the decoder status has not changed.

Parameters

data (*str*) – String to be appended to the channel output

2.1.4. State and Parameters

class procitec.decoding.runtime.DecoderProperties

Decoder properties represent meta data on the current decoder. They are emitted with the output and can not be changed afterwards.

Some properties like 'name' are build-in and can not be changed. Others, like 'version' and 'namespaces' have special setter-functions.

```
>>> apc.decoder_properties.version("1.0.0")
>>> apc.decoder_properties.namespaces("foo", "bar")
```

Custom properties can be set by using the call operator

```
>>> apc.decoder_properties(flag1="foobar", flag2="other")
```

__call__(***kwargs*)

Add the passed key-value-pairs the decoder properties (emitted with the first output)

add(*key: str, value: str*)

Add a single decoder property

This function can be use circumvent the limitations for keyword args using the call-operator.

namespaces(**args*)

Adds the given namespaces to the list of XML namespaces emitted by this decoder

class procitec.decoding.runtime.SearchStateHandler

`accept`(*when=None*)

Mark modem as accepted

Parameters

when (ProTS *or int, optional*) – Specify a timestamp of the start of the signal. If a position (int) is given it is converted to a timestamp.

If no parameter is given then the beginning of the current input block is taken. Another call with an explicit new timestamp/position will overwrite an existing value.

`ident`(*when=None*, *weight=50*)

Mark modem as identified

Parameters

- **when** (ProTS *or int, optional*) – Specify a timestamp of the start of the signal. If a position (int) is given it is converted to a timestamp.

  If no parameter is given then the beginning of the current input block is taken. Another call with an explicit new timestamp/position will overwrite an existing value.

- **weight** (*int, optional*) – If it is expected that two different decoders may send their identification message Ident() for the same input signal simultaneously because further differentiation is not possible, this message can be weighted. The default value for the identification weight is 50. It can be set to greater or lower values by simple assignments. Greater values will cause the automatic production channel to prefer this decoder. By this, a certain sort of decoder precedence can be arranged.

`no_sync`()

Mark modem as NO_SYNC

`stop`()

Mark modem as STOPPED

`sync`()

Mark modem as SYNC

class procitec.decoding.runtime.**APCMode**

Members:

SEARCH

PRODUCTION

PRODUCTION = <APCMode.PRODUCTION: 0>

SEARCH = <APCMode.SEARCH: 1>

property name

property value

class procitec.decoding.runtime.**DemodulatorParameters**

An object which provides access to demodulator parameters

Possible parameters are:

- `'primary_modulation'` (*PrimaryModulationType*)
- `'demodulator_type'` (*DemodulatorType*)

- `'symbol_rate'` (`float` in Hz)
- `'symbol_rate_tolerance'` (`float` in Hz)
- `'modulation_order'` (`int`)
- `'shift'` (`float` in Hz)
- `'shift_tolerance'` (`float` in Hz)
- `'synchronization'` (`str`)
- `'burst_mode'` (`bool`)
- `'minimum_burst_pause'` (`float` in seconds)
- `'minimum_burst_length'` (`float` in seconds)
- `'maximum_burst_length'` (`float` in seconds)
- `'PSK_version'` (`str` `'A'` or `'B'`)
- `'PSK_equalizer'` (`bool`)
- `'equalizer'` (*`EqualizerMode`*)
- `'equalizer_filter_taps'` (`int`)
- `'frequency_tracking'` (`bool`)
- `'tone_duration'` (`float` in seconds)
- `'tone_duration_tolerance'` (`float` in seconds)
- `'number_of_tones'` (`int`)
- `'tone_distance'` (`float` in Hz)
- `'number_of_channels'` (`int`)
- `'channel_distance'` (`float` in Hz)
- `'channels_equidistant'` (`bool`)
- `'morse_cpm_range'` (`int`)
- `'fixed_cpm_range'` (`float` in CPM)
- `'cpm_range_tolerance'` (`float` in CPM)
- `'symbol_table'` (`list` (`int`))
- `'channel_frequencies'` (`list` (`float`) in Hz)
- `'nominal_frequency_offset'` (`float` in Hz)

Example

The available parameters depend on the used demodulator. If a specific parameter is available for the current demodulator, can be checked by

```
>>> if 'symbol_rate' in apc.demodulator.parameters:
>>>     value = apc.demodulator.parameters['symbol_rate']
```

class DemodulatorType

Enumeration of supported demodulator types:

Members:

Morse

ASK2_OOK

FSK_disc

FSK_2_matched

MSK

DPSK

Multitone_FSK

Multichannel_FSK_2

Multichannel_DPSK

PSK_absolute

Multichannel_PSK_absolute

F1A

OFDM

OQPSK

QAMN

PSK_data_aided

FSK_auto_shift

ASK2PSK2

ASK2PSK2 = <DemodulatorType.ASK2PSK2: 43>

ASK2_OOK = <DemodulatorType.ASK2_OOK: 2>

DPSK = <DemodulatorType.DPSK: 6>

F1A = <DemodulatorType.F1A: 21>

FSK_2_matched = <DemodulatorType.FSK_2_matched: 4>

FSK_auto_shift = <DemodulatorType.FSK_auto_shift: 36>

FSK_disc = <DemodulatorType.FSK_disc: 3>

MSK = <DemodulatorType.MSK: 5>

Morse = <DemodulatorType.Morse: 1>

Multichannel_DPSK = <DemodulatorType.Multichannel_DPSK: 9>

Multichannel_FSK_2 = <DemodulatorType.Multichannel_FSK_2: 8>

Multichannel_PSK_absolute = <DemodulatorType.Multichannel_PSK_absolute: 20>

Multitone_FSK = <DemodulatorType.Multitone_FSK: 7>

OFDM = <DemodulatorType.OFDM: 22>

OQPSK = <DemodulatorType.OQPSK: 23>

PSK_absolute = <DemodulatorType.PSK_absolute: 15>

PSK_data_aided = <DemodulatorType.PSK_data_aided: 27>

QAMN = <DemodulatorType.QAMN: 24>

property name

property value

class EqualizerMode

Enumeration of supported equalizer modes:

Members:

OFF : No equalisation, only regular control loop

LMS : LMS based equalisation

LINSOLVER : Equalisation based on a system of linear equations

LINSOLVER = <EqualizerMode.LINSOLVER: 2>

LMS = <EqualizerMode.LMS: 1>

OFF = <EqualizerMode.OFF: 0>

property name

property value

class PrimaryModulationType

Enumeration of supported primary modulation types:

Members:

AM : Amplitude Modulation

FM : Frequency Modulation

LSB : Lower Side Band

USB : Upper Side Band

AM = <PrimaryModulationType.AM: 0>

FM = <PrimaryModulationType.FM: 1>

LSB = <PrimaryModulationType.LSB: 2>

USB = <PrimaryModulationType.USB: 3>

property name

property value

`__bool__()`

True if parameters for current demodulator are available

`__getitem__(key: str)`

Returns value for given key or raises exception if parameter is not available.

`get(key: str, default_value: object)`

Returns value for given key or default_value if parameter is not available.

property burst_mode

Burst_mode enabled or None if parameter not available

Type
bool or None

property demodulator_type

Demodulator type or None if parameter not available

Type
DemodulatorType or None

property equalizer

Equalizer working mode: {*OFF*, *LMS*, *LINSOLVER*}

Type
EqualizerMode

property equalizer_filter_taps

number of equalizer filter taps (when in *LINSOLVER* mode), odd number from set {3, 5, . . . , 31}

Type
int

property frequency_tracking

enable/disable frequency tracking for equaliser when in *LINSOLVER* mode

Type
bool

property modulation_order

Modulation order or None if parameter not available

Type
int or None

property primary_modulation

Primary modulation type {*USB*, *LSB*, *AM*, *FM*}

Type
PrimaryModulationType

property symbol_rate

Symbol rate or None if parameter not available

Type
float or None

class procitec.decoding.runtime.DemodControl

`modify()`

Initiates and returns a context manager to modify the demodulator or to perform a rewind of the demodulator.

The returned context object provides a `dict`-like interface to request a modification of the demodulator. The parameters which can be changed are described in *DemodulatorParameters*. Unmodified parameters inherit values which are in place just before the context is entered, e.g. changing the demodulator type only will not affect other parameters.

The parametrisation may fail with invalid key/value/type or runtime exception thrown. Make sure the parametrised fields are available in demodulator type context and parameter interdependencies are met. Always protect the parametrisation with try/except clause.

The returned context object also provides the *rewind()* method to request a rewind of the demodulator to the timepoint specified as a *ProTS*.

You can either request a demodulator change only, rewind only or both at the same time. Upon exit from the context the demodulator is rewound and changed. The request – especially a rewind request – may fail; an exception is thrown in this case.

The input data buffer, including configured *preprocessing chain*, is cleared after demodulator modification. Therefore the preprocessing chain should be restored if needed directly after execution of demodulator modification.

> **Warning**
>
> This is a very advanced feature. You must not ignore the exception which is thrown when the requested change fails.
>
> Rewinding into the past is possible in a limited manner only – expect not more than about 30 seconds (reference timepoint is the current position in the input stream). Rewinding into the "future" is only possible if the requested timepoint is already known in the input stream, i.e. the farthest possible timepoint for rewind is `apc.data.time(last_known_pos)` where `last_known_pos = apc.data.position + apc.data.available - 1`.

Examples

Change the demodulator type to absolute PSK with a symbol rate of 2400 and perform a rewind:

```
with apc.demodulator.modify() as mod:
    mod['demodulator_type'] = runtime.DemodulatorParameters.DemodulatorType.
↪PSK_absolute
    mod['symbol_rate'] = 2400
    mod.rewind(time) # time must be of type ProTS
```

Change demodulator parameters to a predefined set of parameters:

```
# predefined parameter sets
param0 = {'demodulator_type': runtime.DemodulatorParameters.DemodulatorType.
↪PSK_absolute,
    'symbol_rate': 2400,
    'PSK_version': 'B',
    'modulation_order': 4}
param1 = {'demodulator_type': runtime.DemodulatorParameters.DemodulatorType.
↪Multichannel_DPSK,
    'symbol_rate': 300,
    'channel_distance': 900,
```

(continues on next page)

(continued from previous page)

```
    'PSK_version': 'A',
    'modulation_order': 8,
    'number_of_channels': 3}
new_params = # ... logic to determine new parameter set; can also be placed␣
↪inside the `with' clause
with apc.demodulator.modify() as mod:
    for name, value in new_params.items():
        mod[name] = value
```

Restore preprocessing chain after demodulator modification:

```
from procitec.decoding import preprocessing as pre

with apc.demodulator.modify() as mod:
    mod['demodulator_type'] = runtime.DemodulatorParameters.DemodulatorType.
↪DPSK
    mod['symbol_rate'] = 2400
    mod.rewind(time) # time must be of type ProTS

# set/restore preprocessing chain directly after demodulator modification
# apc.preprocessing() may be called only if input buffers are still empty
apc.preprocessing(pre.reverse_symbol_bits(), pre.descramble(0xC1))
```

set_burst_preamble(*value:* BitBuffer | *Sequence[complex | None]*, *mask:* BitBuffer, *sync_offset: int = 0*, *msb_first: bool = True*) → None

Sets a preamble to search for a burst.

This command will affect all PSK demodulators operating in "Burst Mode". In this mode demodulation will only take place during the time of a burst. Normally a burst is detected by signal energy changes only. However, in some cases a burst start can also be detected by a known symbol pattern at a defined position within the burst, e.g. a synchronization preamble, training sequence etc. This method can result in more reliable detections in the presence of non-continuous noise.

The command can be repeated with different parameters to indicate different possible preambles.

To ensure correct demodulator operation, it is mandatory to use this command at the beginning of the decoder execution.

Note that the burst end detection will still be based on signal energy changes as before.

Parameters

- **value** (*BitBuffer or Sequence[complex | None]*) –
  - Preamble consisting of the pattern and arbitrary placeholders for unknown bits or symbols.
  - If the type `BitBuffer` is used then parameter `mask` can be used to specify unknown bits in the pattern. The bits in the BitBuffer are used to form an IQ-symbol pattern with the symbol table of the demodulator. The length of the pattern (size of BitBuffer) should be a multiple of bits per symbol.
  - To indicate IQ symbols directly, a list of complex IQ values can be used. At positions with unknown values the list must contain `None`, `0+0j` or `NAN`.
- **mask** (*BitBuffer*) –

  - Data mask of the length of value to distinguish between variable (payload) data and fixed bit patterns. A binary value 1 designates a share of fixed bit pattern and the value 0 a share of variable (payload) data.
  - Only allowed if the type of parameter `value` is a BitBuffer.
  - If omitted, then assumed to be all-ones.
- `sync_offset` (*int*) –
  - Distance of burst start to the first symbol of specified pattern.
  - If the type of parameter `value` is `BitBuffer`, `sync_offset` should be given in bits - a multiple of bits per symbol is recommended - else in symbols.
  - Maximum value: `length of burst – length of value`
  - The default is 0.
- `msb_first` (*bool*) – Specifies the order in which the bits within an individual symbol are to be interpreted. Has only an effect if the type of parameter `value` is `BitBuffer`. The default is `True`.

Examples

Given a "pseudo"-burst with a preamble of some unknown(U) and 6 known(K) symbols at the beginning:

UUUU KKKK UU KK UUUUUUUUUU...

Known pattern KKKK UU KK is one of the following possibilities:

```
>>> preamble = [0+1j, 1+0j, 0+1j, -1+0j, None, None, 0-1j, 1+0j]
>>> preamble = [0+1j, 1+0j, 0+1j, -1+0j, 0+0j, 0+0j, 0-1j, 1+0j]
>>> preamble = np.array([0+1j, 1+0j, 0+1j, -1+0j, 0+0j, 0+0j, 0-1j, 1+0j])
```

The command to set the burst-preamble is

```
>>> apc.demodulator.set_burst_preamble(preamble, sync_offset=4)
```

With the use of the demodulator constellation and a Bitpattern the function call differs:

```
>>> constellation_qpsk = {0: 1+0j, 2: 0+1j, 1: -1+0j, 3: 0-1j}
```

Known pattern for the preamble defined above KKKK UU KK is

```
>>> preamble = BitBuffer.from_int(0b0011_0000_1001_0001)
>>> pre_mask = BitBuffer.from_int(0b1111_0000_1111_1111)
```

Then the command to set the burst-preamble is

```
>>> apc.demodulator.set_burst_preamble(preamble, pre_mask, sync_offset=4*2)
```

It is important to observe the bit order of the symbol table and that the unit of `sync_offset` differs when using the IQ-type or bit pattern function call.

Should the mask be all-ones, it may be omitted.

```
>>> preamble = BitBuffer.from_int(0b1001_0001)
>>> pre_mask = BitBuffer.from_int(0b1111_1111)  # it may be omitted.
```

Then the command to set the burst-preamble is as before

```
>>> apc.demodulator.set_burst_preamble(preamble, pre_mask, sync_offset=4*2)
```

but in this case equivalent to

```
>>> apc.demodulator.set_burst_preamble(preamble, sync_offset=4*2)
```

The parameter `msb_first` is used to specify the order in which the bits correspond to a symbol. The following two commands have the equivalent effect, in case of 2 bits per symbol.

```
>>> apc.demodulator.set_burst_preamble(BitBuffer.from_int(0b1001_0001_1110))
>>> apc.demodulator.set_burst_preamble(BitBuffer.from_int(0b0110_0010_1101),␣
↪msb_first=False)
```

When working with PSK version B, the bit sequence is to be specified exactly as received in the decoder or bit view (LSB left/first), please use `msb_first=False` if unsure. If an I/Q symbol sequence is used, it should be provided the way the symbols appear in the waveform. It means that in a comparison (or a conversion) between a bit sequence and an equivalent I/Q sequence the symbol rotation introduced by version B must be considered.

Note

The preamble in the above examples is kept short for clarity. In practical use cases the bit/symbol sequences of known values should span over significantly more than 10 symbols. Think in terms of the probability of detection in gaussian noise a random sequence of symbols.

`set_filter`(*filter_type: str*, *roll_off: float*) → None

Sets the filter form of the demodulators receive filter.

This command is used exclusively in connection with the demodulator PSK Data Aided. For this demodulator, the symbol filter can be matched to the special modem similar to a Matched Filter. The respective methods and parameters mostly result from published modem descriptions. The use of this command is optional. Without this instruction, a universal and wider filter will be used, which does require a better signal-to-noise ratio but also tolerates larger fluctuations in center frequency and symbol rate.

To ensure correct demodulator operation, it is mandatory to use this command at the beginning of the decoder execution.

Parameters

- **filter_type** (*str*) – Form of filter. Possible values are `"RC"`/`"rc"` and `"RRC"`/`"rrc"` for Raised-Cosine and Root-Raised-Cosine.
- **roll_off** (*float*) – Filter roll-off in range 0.0-1.0

Examples

```
>>> apc.demodulator.set_filter('rrc', 0.35)
```

`set_soft_symbols(format)`

The demodulator will be advised to deliver soft symbols (or samples) in addition to the conventional bit stream.

Soft symbols are always delivered as pairs of unsigned 16-bit integers. Each pair is one symbol. However, the meaning of the pair elements differs depending on the specified symbol format.

For magnitude/phase symbols (`MAG_PHASE`):

- The first element of a sample corresponds to the magnitude and uses the full 16 bit range. A value of 2**15 (the middle of the value range) matches a magnitude of 1.
- The second element of a sample corresponds to the phase. The phase range `[0, 2*pi]` is mapped to full 16 bit.
- Only the PSK demodulators can provide these values.

For fsk symbols (`FSK`):

- The first element of a sample corresponds to the magnitude and uses the full 16 bit range.
- The second element is always 0.
- Only the demodulator `'FSK_discriminator'` can provide these values.

Parameters

`format` (`SampleFormat`) – Format of the requested soft symbols. Has to be `MAG_PHASE` or `FSK`.

Examples

Informing the demodulator to deliver magnitude/phase pairs. This should be done once at decoder start.

```
>>> from procitec.decoding import MAG_PHASE
>>> apc.demodulator.set_soft_symbols(MAG_PHASE)
```

Read and print the soft-symbols

```
>>> soft_symbols = apc.symbols.read(size=4)
>>> soft_symbols
[[31632, 160], [35824, 16327], [31152, 65376], [31648, 32416]]
```

Convert these values into normalized float values (which is normally not necessary), with usage of numpy.

```
>>> np.array(soft_symbols) / [ 2**15, 2**16 /(2*np.pi) ]
[[ 0.97  0.02]
 [ 1.09  1.57]
 [ 0.95  6.27]
 [ 0.97  3.11]]
```

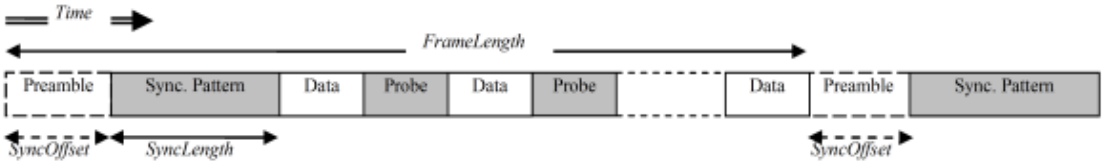
You can see that these values belong to a noisy QPSK constellation.

`set_training_pattern`(*frame*, *frame_mask*, *sync_offset*, *sync_length*, *min_length*, *msb_first*)

Specifies a training pattern used for equalization of distorted signals.

This command is used exclusively for the demodulator PSK Data Aided. This demodulator has been designed for adaptive equalization based on specific training patterns. As a result of such equalization, multipath fading effects are corrected. The correct phase reference of absolute PSK signals is, as a result, matching the training pattern.

The areas shaded in grey comprise symbol sequences, partially used as training patterns for adaptive filtering and partially for time synchronization of the entire sequence. Additionally, an "unusable" time segment, e.g. a pre-carrier, may be located in front of the synchronization pattern start. To support the synchronization, you may specify this time range as well. This may be required, e.g. if the synchronization pattern is very short and there is a risk of false "random hits". Parameters described below specify the entire frame. Typically, the training pattern is interleaved with the payload data as follows



To ensure correct demodulator operation, it is mandatory to use this command at the beginning of the decoder execution.

Parameters

- `frame` (`BitBuffer` *or Sequence(complex or None)*) –
  - Data frame consisting of the pattern and arbitrary placeholders for unknown bits or symbols respectively payload.
  - If the type BitBuffer is used the use of argument frame_mask is mandatory, else forbidden. The bits in the BitBuffer are used to form an IQ-symbol pattern with the symbol table of the demodulator. The length of the pattern (size of BitBuffer) should be a multiple of bits per symbol.
  - To indicate IQ symbols directly, a list of complex IQ values can be used. At positions with unknown values, the list must contain `None`, `0+0j` or `NAN`.
  - Sequences used for synchronisation should be long and unique enough to enable robust detection.
  - Contiguous training sequences (probes) should be at least 6 symbols long, otherwise they cannot be optimally used for equaliser training.
- `frame_mask` (`BitBuffer`) –
  - Data mask of the length of value to distinguish between variable (payload) data and fixed bit patterns. A binary value 1 designates a share of fixed bit pattern and the value 0 a share of variable (payload) data. Please note that the area of the synchronization pattern must be fully masked with 1 without interruptions.
  - Only allowed if the type of argument value is a BitBuffer.
- `sync_offset` (*int*) –
  - If the defined frame comprises a prekey, i.e. a start unsuitable for synchronizing, make sure to enter the minimum length of this preamble here. However, this is only required if there is a risk of confusion with the synchronization symbols within the prekey (e.g. with short and/or regular synchronization patterns). Otherwise enter the value 0. The frame defined using `frame` (and

frame_mask) must contain a preamble of this length under all circumstances but the values entered there are irrelevant.

- If the type of frame is BitBuffer then sync_offset should be given in bits - a multiple of bits per symbol is recommended - else in symbols.
- Valid range: 0 - min_length

- **sync_length** (*int*) –
  - Length of synchronization pattern. This pattern is necessary to determine the exact reference position of the frame defined. Therefore, this pattern should be clearly identifiable due to random variety and length.
  - If the type of frame is BitBuffer then sync_length should be given in bits - a multiple of bits per symbol is recommended - otherwise in symbols.
  - Valid range: 1 - min_length
- **min_length** (*int*) –
  - Some modems use different frame lengths. Shorter frames then have the same start followed by the same structure, but the payload data section at the end is shorter. If so, enter the minimum frame length, otherwise the length of frame will be used.
  - If the type of frame is BitBuffer then min_length should be given in bits - a multiple of bits per symbol is recommended - otherwise in symbols.
  - Valid range: 1 - length of frame
- **msb_first** (*bool*) – Specifies the order in which the bits within an individual symbol are to be interpreted. Has only an effect if the type of parameter frame is BitBuffer. The default is True.

Examples

Given a "pseudo"-pattern for a PSK burst modem with unknown(U), known(K), sync symbols(S) and a (useless) prekey (P)

PPPPP SSSS UUU KK UUU KK UUU

The different patterns could be defined as follows

```
>>> sync = [1+0j, 0+1j, -1+0j, 0-1j]
>>> known = [0+1j, 0+1j]
>>> unknown = [None] * 3
>>> preamble = [None] * 5
```

The combined pattern

```
>>> pattern = preamble + sync + unknown + known + unknown + known + unknown
>>> set_training_pattern(pattern, sync_offset=5, sync_length=4, min_length=22)
```

The same can be achieved using a bitpattern and the symbol table of the demodulator. Given the symbol table

```
>>> constellation_qpsk = {0b00: 1+0j, 0b01: 0+1j, 0b10: -1+0j, 0b11: 0-1j}
```

The full pattern:

```
>>> pattern = BitBuffer.from_int(0b000000_0101_000000_0101_000000_11100100_
↪000000000000, 44)
>>> pt_mask = BitBuffer.from_int(0b000000_1111_000000_1111_000000_11111111_
↪000000000000, 44)
>>> set_training_pattern(pattern, pt_mask, sync_offset=5*2, sync_length=4*2,␣
↪min_length=22*2)
```

It is important to observe the bit order of the symbol table and that the unit of `sync_offset`, `sync_length` and `min_length` differs when using the IQ-type or bit pattern function call.

Note

In version 25.1 and earlier, the SYNC sequence (S) must be directly followed by a payload (U) sequence.

`property parameters`

Current demodulator type and parameters

Type
DemodulatorParameters

`class procitec.decoding.runtime.DemodControlModify`

A context manager returned by *`modify()`*. It allows access modification of demodulator parameters and potential rewind operation.

`rewind(`*time*`)`

Rewind to the given time position in an input signal, so that potentially modified demodulator can process the data again. Usually used together with reparametrization of demodulator, see also *`apc.demodulator.modify`* for more detailed description and examples.

Warning

Make sure there is no infinite loop created. In doubt limit the number of rewinds with a counter.

This is a very advanced feature. You must not ignore the exception which is thrown when the requested rewind fails.

Rewinding into the past is possible in a limited manner only – expect not more than about 30 seconds (reference timepoint is the current position in the input stream). Rewinding into the "future" is only possible if the requested timepoint is already known in the input stream, i.e. the farthest possible timepoint for rewind is `apc.data.time(last_known_pos)` where `last_known_pos = apc.data.position + apc.data.available - 1`.

Parameters
`time` (ProTS) – target timestamp for rewind

Example

```
# demodulator reparametrization and rewind are allowed only in production
if apc.current_mode == runtime.APCMode.PRODUCTION:
    # use try/except to catch potentially failed rewind()
    try:
        with apc.demodulator.modify() as mod:  # get context manager
```

(continues on next page)

(continued from previous page)

```
            # reparametrize demodulator:
            # mod['symbol_rate'] = 2400
            # rewind data stream to a position given by a timestamp
            mod.rewind(time) # time must be of type ProTS
    except RuntimeError:
        pass  # rewind failed

    # input buffers and preprocessing are empty here
```

2.1.5. File Output

class procitec.decoding.runtime.**ProductionMemory**

An object providing access to file output facilities in a decoder.

It is recommended to use the following API for any file output. This way it is ensured that all created files are stored in the so-called *production memory folder* (e.g. C:\users\username\go2SIGNALS\go2DECODE\mem_ The name of a file stored there has a fixed scheme and contains among others a timestamp which is taken from the current read position within the input stream.

Files generated with the builtin Python functions or generated by external libraries should be copied with the *ProductionMemory.copy()* function.

Note that only files created in the *production memory folder* will be reported and made available to the monitoring system.

copy(*filename: str | Path*) → str | None

Copies a file to production memory. This can be used for files generated by external libraries or files generated with Pythons built-in packages, e.g. tempfile.

The file is renamed to match the default production memory filename template. The file extension is truncated to the first 10 characters. If the file has no extension, ".bin" is used. This is due to internal filename constraints.

If the production memory is not available no copy happens.

The file is reported automatically. There should be no additional call of *ProductionMemory.report_file()*.

Added in version 24.1.0.

Parameters

filename (*str*) – Absolute path of file to be copied. It must contain an absolute path with forward slashes or backslashes.

Returns

The path to the copied file in production memory if copy was successful.

Return type

str or None

Raises

- **FileNotFoundError** – If the file to copy was not found.
- **IsADirectoryError** – If the filename does not contain a file, but a valid directory.
- **RuntimeError** – If the file could not be read.

Examples

```
>>> from tempfile import NamedTemporaryFile
>>> file = NamedTemporaryFile(suffix=".data", delete=False)
>>> file.write(b"1234")
>>> file.close()
>>> print(file.name)
/tmp/tmpxhfc6ecv.data
>>> memprod_filename = apc.production_memory.copy(file.name)
>>> print(memprod_filename)
.../mem_prod/decoder_xy/20170606/20170606-110030-008__0000000000_D01.data
```

open(*extension: str, mode: str = 'w', encoding: str | None = None, msb_first: bool | None = None*)

Open a file in the production memory

Opening of a file fails if too many files are opened. A `RuntimeError` with an appropriate error message is thrown in that cases.

If production memory is not available the opened file is stub and *write()* discards all data.

Note that a physical file is only created when the file is actually written to with *write()*

Parameters

- **extension** (*str*) – File extension to be used for the file. Only the characters a-z, A-Z, 0-9, . and _ are allowed. The length of the extension must be in the range (including) 1–10.

  Changed in version 24.1.0: Relaxed restrictions on length of extension

- **mode** (*str, optional*) – Mode in which the file is opened. By default (`"w"`) the file is opened for writing in text mode. In text mode (the default, or when `"t"` is included in `mode`), only strings (`str` objects) may be passed to *write()*. The strings are written in the character encoding defined by `encoding` to the file.

  The file can be opened in binary mode by including `"b"` in `mode`. In binary mode only `bytes` and `bytearray` objects or *BitBuffer* may be passed to *write()*. The bytes are written as is to the file. In case of BitBuffer bit ordering can be set by `msb_first`.

  Newline characters (`"\n"` and `"\r"`) are never modified in any way (unlike with files created using Python's *open()*).

- **encoding** (*str, optional*) – Encoding to be used in text mode; may only be provided for a file in text mode. By default (`None`) UTF-8 is used (unlike with files created using Python's *open()*, where a platform dependent encoding is used). For a list of supported encodings see `codecs`.

- **msb_first** (*bool, optional*) – Specifies whether the bytes should be written as MSB (most significant bit) or LSB (least significant bit) first. This parameter is only taken into account when `data` is of type BitBuffer in *write()*. Default is false (LSB).

  If the binary output should is byte-oriented, e.g. an additional application layer then MSB is recommended. However, if the output is bit-oriented, e.g. original signal without channel decoding (detector only) or there is still some framing (start/stop bits) left then LSB is advised.

  Changed in version 22.2.0: Added msb_first

Return type

ProductionMemoryBinFile or *ProductionMemoryTextFile*

Raises
`RuntimeError` – If too many files are opened at once.

`report_file(`*filename: str*`)` → None

Inform the system about a new result file created independently of pyDDL API. This e.g. can be used to report files created by an external shared lib (DLL) which is loaded by the decoder. To ensure that the external DLL also writes to the production memory folder use *name()* to get the path and then give this path to the external DLL.

The filename must contain an absolute path with forward slashes or backslashes.

Parameters
`filename` (*str*) – Absolute path of new created file

`property enabled`

Check if the production memory is enabled

`class procitec.decoding.runtime.ProductionMemoryBinFile`

An object to access a production memory file

This is the base type returned by *ProductionMemory.open()*. All methods and attributes described herein after are supported by files in text and binary mode.

The object provides a context manager, i.e. it can be used in a `with` statement in Python:

```
>>> with apc.production_memory.open("ext") as fd:
>>>     foo = "0x{:x}".format(42)
>>>     fd.write(foo)
```

The file is closed automatically once all code in the with statement body has been executed, even if an exception is thrown therein.

`close()`

Close the production memory file

`write(`*data*`)`

Write data to the production memory file

Parameters
`data` (*str or bytes or bytearray or* `BitBuffer`) – Data to write to the file. For files in text mode `data` must be a `str` object which is converted using the encoding specified at *ProductionMemory.open()*. For files in binary mode `data` must be a `bytes` or `bytearray` object which is written as is into the file. For files in binary mode `data` can also be of type *BitBuffer*. The parameter `msb_first` specified in *ProductionMemory.open()* determines then the bit ordering when converting BitBuffer to bytes. It is not possible to use another type for `data` after the first writing has occurred.

Changed in version 22.2.0: The argument can now be a BitBuffer

`property closed`

`True` if the file is closed, `False` otherwise

`property name`

Filename currently being written to, if any.

This is set to `None` until the underlying storage-handler opens a file - usually with the first write-operation.

Returns
filename (including path) in-use of the last write-operation

Return type
str or None

class procitec.decoding.runtime.**ProductionMemoryTextFile**

All methods and attributes of *ProductionMemoryBinFile* are supported

property **encoding**

Character encoding used for writing data to the file

Type
str

2.1.6. Audio Output

class procitec.decoding.runtime.**AudioCodec**

An object for an audio codec

close()

Close audio codec and associated files

decode(*data*, *timestamp*, ***kwargs*)

Decode an encoded frame and write data to output files

Decoded audio frames are written to `.wav` files. If the requested audio codec is not supported then raw (encoded) data is written to `.codec` files.

Parameters

- **data** (*bytes or* BitBuffer) – Raw (encoded) audio frame. This is written to the `.codec` file if requested audio codec is not supported.

  Changed in version 21.2.0: The argument can now be a BitBuffer
- **timestamp** (ProTS) – Timestamp of the audio frame
- ****kwargs** – Decode parameters

set_parameters()

Set audio decoder parameters

May only be called before the very first call to *decode()* (right after the audio codec object is returned by *APCGateway.init_codec()*) or between *start_new_file()* and *decode()*.

Parameters
****kwargs** – Decoder parameters

start_new_file()

Start new output files

Close current output files (`.codec` and `.wav`) and start new ones. This can be used to start new files after a period of silence.

property **closed**

True if the codec is closed, False otherwise

2.1.7. Standalone runtime

procitec.decoding.runtime.**run**(*source: Callable[[*APCGateway*], None] | Path | str | tuple[Path | str, str], datafile: Path | str, *, parameters: dict[str, int | float | str | Sequence[str] | Sequence[Sequence[str]]] | None = None, name: str | None = None, run_status: Any = None, mem_prod_path: Path | str | None = None, enable_timeout: bool = False, **kwargs: str*) → None

Execute a decoder offline using a datafile as input.

Changed in version 24.2.0: The parameter `mem_prod_handler` has been replaced with `mem_prod_path`: Production memory no longer needs to be explicitly created.

Parameters

- **source** (*callable or path-like or tuple[path-like, str]*) – The decoder to run. May be one of the following:
  - A callable that takes an *apc* object as argument
  - A path to a file whose content is read and used as the decoder's code
  - A tuple containing a path and the decoder's code as a string (the file pointed to by the path is not read).
- **datafile** (*path-like*) – A .rec-file to use as input data
- **parameters** (*dict, optional*) – Decoder parameters, which can be queried in the decoder. see *procitec.decoding.runtime.APCGateway.parameters*.
- **name** (*str, optional*) – Decoder name to be used used in the output's XML header. If None (default) and if `source` is a path to a decoder, the filename of the decoder is used (`_dec` suffix and file extension is removed).
- **run_status** (*object, optional*) – If None (default), decoder's output is written to `stdout`. Otherwise `run_status` must be an object providing the following methods:
  - The callable `write` accepting a `str` argument: Used to report decoder's text output.
  - The callable `set_progress` accepting a `float` argument: Used to report decoder's approximate progress (0-1).
- **mem_prod_path** (*path-like, optional*) – Used to specify where non-text decoder result (files) are stored. If not given, there is no file output.

  Added in version 24.2.0: Replaces removed `mem_prod_handler` parameter
- **enable_timeout** (*bool*) – A timeout error occur if too much time elapses within the decoder without returning control to the APC instance.

  The time limit can be set within the decoder itself - see *procitec.decoding.runtime.APCGateway.set_timeout()*. The default is false.
- ****kwargs** – Keyword arguments are passed to *APC()*

Examples

see *Executing a decoder*

procitec.decoding.runtime.**main**(*source: Callable[[APCGateway], None] | Path | str | tuple[Path | str, str], datafile: Path | str, *, parameters: dict[str, int | float | str | Sequence[str] | Sequence[Sequence[str]]] | None = None, name: str | None = None, mem_prod_path: Path | str | None = None, enable_timeout: bool = False, output_progress: bool = True, output_mode: str = 'ALL', output_save: None | Path = None, **kwargs: str*) → int

A wrapper for *run()* providing formatted, human-readable output

Parameters

- **source** (*callable or path-like or tuple[path-like, str]*) – See *run()*
- **datafile** (*path-like*) – See *run()*
- **parameters** (*dict, optional*) – See *run()*
- **name** (*str, optional*) – See *run()*
- **mem_prod_path** (*path-like, optional*) – See *run()*
- **enable_timeout** (*bool, optional*) – See *run()*
- **output_progress** (*bool, optional*) – If `True` (default), display approximate progress of decoder execution and most current output of status channel.
- **output_mode** (*str, optional*) – Select output mode. The mode can be one of:
  - `"ALL"` (default): Display all output channels, formatted and grouped by decoder result sections.
  - `"RAW"`: Display raw XML output as produced by the runtime
  - Any other value is assumed to be the name of a channel to be displayed.
- **output_save** (*pathlib.Path, optional*) – Save decoder output in the provided directory. The output of every channel is saved into an individual file. An additional file is created for raw XML output as produced by the runtime. All files are UTF-8 encoded.

  Warning: Existing files are overwritten.
- ****kwargs** – See *run()*

Returns

status_code – An integer indicating successful execution or an error. Can be directly passed to `sys.exit()`.

Return type

int

Notes

Added in version 24.2.0.

2.2. BitBuffer

This module contains two central types for handling bit-level data, BitBuffer and BitStream. The former represents data of known size along with meta-data like a timestamp, the demodulation quality or burst information. BitBuffer object supports indexing individual bits as well as bitwise and bitshift operators.

A BitStream represents an infinite stream of bits, which can be accessed by reading chunks into a BitBuffer. Additional methods allow querying meta-data or manipulating the current position within the stream.

The functions in this module are used to query or manipulate BitBuffer/BitStream objects. The main categories are shorthands for recurring operations and the free-form of the supported operators on BitBuffer object, allowing customized behavior.

- *BitBuffer*
- *BitStream*
- *Helper Functions*
- *Shift Operations*

2.2.1. BitBuffer

This class is used to access individual bits stored in an internal or external buffer. In the context of decoder development BitBuffer objects are mostly created as a result of reading data provided by the demodulator via the input buffer *apc.data* which is an instance of *BitStream*.

Aside from creating BitBuffer objects with an empty buffer, there are a number functions to convert values provided as `str` or `int` and vice versa.

In the following example a BitBuffer with *size=16* is created from an integral value:

```
>>> bb = BitBuffer.from_int(0xf531)
>>> bb.size
16
>>> bb.to_int()
62769
>>> hex(bb.to_int())
'0xf531'
```

Note, that while the conversion back to `int` yields the original value, a conversion to a string reveals the internal bit and byte ordering:

```
>>> bb.to_str()
'1000110010101111'
>>> bb[0], bb[1], bb[2], bb[3], bb[4], bb[5], bb[6]
(True, False, False, False, True, True, False)
>>> bytes(bb).hex()
'31f5'
```

The least significant bit (LSB) is first – it is assigned the index 0 and therefore shown first. This can also be seen when individual bits are accessed by index. The positional notation LSB `first` is also known as LSB `left` or MSB `right` (MSB: most significand bit). Integer representation always uses positional notation MSB `first` (also known as MSB `left`). Furthermore, if converted to bytes, the content of the internal buffer is returned – here formatted as hex value. The byte order is little-endian, which means that the byte holding bits 0 to 7 is first. That is why the bytes are `31f5` in comparison to the original notation in the integer literal.

Aside from the data itself BitBuffer objects can store meta-information which includes

- Timestamps: Using *BitBuffer.time()* a timestamp for any position within the buffer is returned.
- Quality: The demodulation quality for each bit, if available. See *BitStream.read()*.

BitBuffer supports shift operators (<< and >>) and bitwise operators (&, |, ^, ~). These require operands of equal size, if used with another BitBuffer. In addition they may be used with `int` value directly:

```
>>> # decimal value of 35 is in binary representation 0b100011
>>> bb = BitBuffer.from_int(0b100011,7)
>>> # left shift is equal to multiplication by 2
>>> (bb << 1).to_int()
70
>>> # Note, shift operation is carried out in integer represention (MSB first)
>>> # This means shift direction in BitBuffer (LSB first) is the other way round.
>>> bb.to_str()
'1100010'
>>> (bb << 1).to_str()
'0110001'
>>> # right shift is equal to division by 2
>>> (bb >> 1).to_int()
17
```

```
>>> # example for bitwise and operator ``&``
>>> bb = BitBuffer.from_int(0b1101)
>>> str(bb & BitBuffer.from_int(0b1100))
'0011'
>>> str(bb & 0b1100)
'0011'
```

All of the operators functionality is also available as functions allowing additional features.

`class procitec.common.bitbuffer.BitBuffer(`*reserved_or_buffer*, *size: int | None = None*, *offset: int = 0*`)`

Creates a new *BitBuffer* by either allocating memory or referencing the data passed as first argument using the buffer protocol.

Parameters

- **reserved_or_buffer** (*int or* BitBuffer *or bytes or bytearray*) – If an int is passed it is number of bits for which memory is allocated. This will be the *capacity* and is always rounded up to an integer multiple of 8. The memory is initialized with all 0

  Else, the memory of the passed buffer object shall be reused. If this is an immutable, the content of the returned *BitBuffer* can not be modified.

- **size** (*int, optional*) – Defines *size* of *BitBuffer*

  If `reserved` is given:
  Must not exceed `reserved - offset`. If None (default) *size* is set to `reserved - offset`.

  If `buffer` is given:
  If None (default) *size* is set to the size in bits of `buffer` minus `offset`. If this exceeds the size of `buffer`, new memory is allocated and `buffer` is not used at all.

- **offset** (*int, optional*) – Defines *offset*, must be in range 0 to 7, denoting the index of the first used bit in the memory. See also `size` parameter.

`static from_bytes(value: bytes, byteorder: str = 'little') → BitBuffer`

Creates a new *BitBuffer* and initialize its content with the `bytes value`

Added in version 22.2.0.

Parameters

- **value** (*bytes*) – Content of the new *BitBuffer*.
- **byteorder** (*str*) – Endianness of the bytes: `little` or `big`

Return type
BitBuffer

Examples

```
>>> BitBuffer.from_bytes( b"\x12\x3F").to_str()
'0100100011111100'
>>> BitBuffer.from_bytes( b"\x12\x3F", byteorder="big").to_str()
'1111110001001000'
```

`static from_hex(value: str, byteorder: str = 'little') → BitBuffer`

Creates a new *BitBuffer* and initialize its content with symbols 0 to F in the `str value`. This behaves similar to `from_bytes` method. Whitespace between bytes is ignored.

Added in version 25.2.0.

Parameters

- **value** (*str*) – Content of the new *BitBuffer*. String with symbols [0-F].
- **byteorder** (*str*) – Endianness of the bytes: `little` or `big`

Return type
BitBuffer

Raises

- **ValueError** – `str value` contains values which are not in [0-9], [a-f] or [A-F] or length is not even (ignoring whitespace between bytes)
- **ValueError** – *byteorder* neither `little` nor `big`

Examples

```
>>> BitBuffer.from_hex("123f").to_str()
'0100100011111100'
>>> BitBuffer.from_hex("123f", byteorder="big").to_str()
'1111110001001000'
```

`static from_int(value: int, size: int | None = None) → BitBuffer`

Creates a new *BitBuffer* and initializes its content with the `int` `value`

Parameters

- **value** (*int*) – Content (LSB first) of the new *BitBuffer*.
- **size** (*int, optional*) – Specifies the *size* of the returned BitBuffer. By default (None), the *size* is equal to the required number of bits to represent `value` (ignoring the sign bit).

Return type
BitBuffer

Examples

```
>>> BitBuffer.from_int(0x123, 16).to_str()
'1100010010000000'
```

`static from_iter(value: Sequence[int], size: int) → BitBuffer`

Creates a new *BitBuffer* and initializes its content from all `int` objects in `value`

Added in version 23.2.0.

Parameters

- **value** (*Sequence[int]*) – Content (LSB first per element) of the new *BitBuffer*.
- **size** (*int*) – Specifies the number of bits for each value given. The resulting *size* of the returned BitBuffer is therefore the number of values multiplied with this parameter.

Return type
BitBuffer

Examples

```
>>> BitBuffer.from_iter([1, 2, 7], 4).to_str()
'100001001110'
```

Numpy arrays are also supported.

```
>>> import numpy as np
>>> iter_ = np.arange(8)
>>> BitBuffer.from_iter(iter_, 3).to_str()
'000100010110001101011111'
```

`static from_str(value: str) → BitBuffer`

Create a new *BitBuffer* and initializes its content with the 1s and 0s in the `str` `value`. The order is left-to-right.

Parameters
value (*str*) – Bits encoded as "0" and "1" characters from left to right. Characters contained in " _\t" are ignored and can be used as separators.

Raises
ValueError – If characters are found that are not contained in "01 _\t"

Return type
BitBuffer

Example

```
>>> BitBuffer.from_str("1100_0000")
BitBuffer(8, 8, 0)
```

`__and__()`
Bitwise AND operation of two *BitBuffers*, returning the result in a new *BitBuffer*. All *ShadowBuffers* and Marks are copied to the returned *BitBuffer*. Bitwise AND operation of *BitBuffer* and int, returning the result in a new *BitBuffer*. All *ShadowBuffers* and Marks are copied to the returned *BitBuffer*.

`__invert__()`
Inverts the whole *BitBuffer* and returns result in a new *BitBuffer*. All *ShadowBuffers* and Marks are copied to the returned *BitBuffer*.

`__or__()`
Bitwise OR operation of two *BitBuffers*, returning the result in a new *BitBuffer*. All *ShadowBuffers* and Marks are copied to the returned *BitBuffer*. Bitwise OR operation of *BitBuffer* and int, returning the result in a new *BitBuffer*. All *ShadowBuffers* and Marks are copied to the returned *BitBuffer*.

`__xor__()`
Bitwise XOR operation of two *BitBuffers*, returning the result in a new *BitBuffer*. All *ShadowBuffers* and Marks are copied to the returned *BitBuffer*. Bitwise XOR operation of *BitBuffer* and int, returning the result in a new *BitBuffer*. All *ShadowBuffers* and Marks are copied to the returned *BitBuffer*.

`clear()` → None
Fills the buffer with zeros and resets 'size' and *offset* to *0*

`copy()` → *BitBuffer*
Copies the bitbuffer and the underlying memory.

Returns
The copied buffer.

Return type
BitBuffer

`extend(`*buffer:* BitBuffer, *include_shadows: bool = True*`)` → None
Extends the *BitBuffer* by the given `buffer`

This increases the *size* by the size of `buffer`. No reallocation of memory is performed, therefore the *capacity* of the *BitBuffer* must not be exceeded. When the *BitBuffer* has no time information, it uses the one from the given `buffer`.

Parameters

- **buffer** (BitBuffer) – Bits to append.
- **include_shadows** (*bool, optional*) – Also extend with the shadow buffers if true.

Return type
None

Raises
ValueError – If the remaining capacity is not large enough to hold the additional bits.

Examples

```
>>> bb = BitBuffer(16, 0)
>>> bb.size
0
>>> bb.extend(BitBuffer.from_int(3, size=2))
>>> bb.extend(BitBuffer.from_int(0, size=3))
>>> bb.size, bb.to_str()
(5, '11000')
```

`fill`()

fill with py::buffer

`pack`(*value: int, fmt: str, position: int = 0*) → int

Pack `value` to `BitBuffer` with given format.

It is possible to `pack` an integer value to the `BitBuffer` instead of using the `from_int` method.

The pack method has at least two parameters. The first parameter is the value, the second the format string. Optionally one can define a position in `BitBuffer` for packing.

Parameters

- **`value`** (*int*) – Integer value to be packed in `BitBuffer`. If the value is not representable with the specified number of bits the upper bits of value are discarded.
- **`fmt`** (*str*) – The format string of pack contains three parts:
  - byte order: > or < (the arrow indicates little-endian(<) or big-endian(>) byte ordering in BitBuffer)
  - bit order: > or < (the arrow indicates the first bit form the left(<) or from the right in BitBuffer)
  - **size**

    unsigned: digit(s) (> 0), B (8 bit), H (16 bit), I (32 bit), Q (64 bit)

    signed: digit(s) (< 0), b (8 bit), h (16 bit), i (32 bit), q (64 bit)
- **`position`** (*int, optional*) – The index of bit position to get packed bits. The default is 0.

Returns

Index in bitbuffer one after the last bit with packed bits written.

Return type

int

Raises

`ValueError` – If BitBuffer has not enough bits after position needed due to specified format `fmt` or if format `fmt` is incorrect.

Notes

Special case if the size is not a multiple of 8. Then the remaining bits (parts of byte) are ignored.

Examples

```
>>> bb = BitBuffer(16)
>>> bb.pack(0x0123, ">>16")  # or ">>H"
>>> bb.to_str()
'0000000100100011'
```

The following four examples demonstrates the working with different byte and bit ordering.

```
>>> bb = BitBuffer(16)
>>> bb.pack(0x0123, ">>13")
>>> bb.to_str()
'0000100100011000'
```

```
>>> bb = BitBuffer(16)
>>> bb.pack(0x0123, "<>13")
>>> bb.to_str()
'0010001100001000'
```

```
>>> bb = BitBuffer(16)
>>> bb.pack(0x0123, "<<13")
>>> bb.to_str()
'1100010010000000'
```

```
>>> bb = BitBuffer(16)
>>> bb.pack(0x0123, "><13")
>>> bb.to_str()
'1000011000100000'
```

An example with a negative value.

```
>>> bb = BitBuffer(16)
>>> bb.pack(-0x0123, ">>-13")
>>> bb.to_str()
'0111101101110100'
```

`set_time_info`(*timestamp*, *rate*, *repetition=1*, *offset=0*) → None

Set the values of *time_info*

Parameters

- **timestamp** (ProTS) – The timestamp of the first bit in the buffer
- **rate** (*float*) – Rate in bits per second.
- **repetition** (*int, optional*) – number of consecutive bits with identical timestsamp
- **offset** (*int, optional*) – see *TimeInfo.offset*

Return type

None

`split`(*split_size: int | Sequence[int], *, reverse: bool = False*) → list[*BitBuffer*]

Splits the BitBuffer into several BitBuffers and returns them as a list. The returned BitBuffers are views to a copy of the BitBuffer.

Added in version 22.1.0.

Changed in version 25.2.0: Returns an empty list instead of throwing an exception when BitBuffer is empty.

Parameters

- `split_size` (*int or Sequence[int]*) – If int, the *BitBuffer* is split into multiple *BitBuffers* of size *split_size*. The size of the *BitBuffer* has to be a multiple of *split_size*`. If a list is given, the *BitBuffer* is split into multiple BitBuffers of arbitrary lengths given in the list. The sum of the list must be less than or equal to the size of the *BitBuffer* and no element in the list is allowed to be 0. An exception is thrown in case of improper *split_size*.
- `reverse` (*bool*) – If true, each individual returned BitBuffer is bit-reversed. Default is False.

Return type
list[*BitBuffer*]

Raises

- `ValueError` – If *split_size* is of type int: *split_size* is 0 or size of the buffer is not a multiple of *split_size*. If *split_size* is of type Sequence[int]: The sum of all individual sizes is greater than the size of the buffer or one of the sizes is 0.
- `TypeError` – If *split_size* is of type Sequence[int]: One of the sizes is negative.

Examples

```
>>> for b in BitBuffer.from_str("100101111").split(3):
>>>     print(b)
100
101
111
```

```
>>> for b in BitBuffer.from_str("100101111").split([2, 3, 2], reverse=True):
>>>     print(b)
01
010
11
```

```
>>> b0, b1, b2 = BitBuffer.from_str("100_101_111").split(3)
```

`split_to_array`(*split_size: int | Sequence[int], *, reverse: bool = False, signed: bool = False, dtype: numpy.dtype | None = None*) → numpy.ndarray

Splits the BitBuffer into several ranges, casts them to integer and returns a numpy.ndarray of these integers.

Added in version 23.2.0.

Changed in version 25.2.0: Returns an empty array instead of throwing an exception when BitBuffer is empty.

Parameters

- **split_size** (*int | Sequence[int]*) – If int, the *BitBuffer* is split into multiple ranges of size `split_size`. The size of the BitBuffer has to be a multiple of `split_size`. If a list is given, the *BitBuffer* is split into multiple ranges of arbitrary length given in the list. The sum of the list must be less than or equal to the size of the *BitBuffer* and no element in the list is allowed to be 0. An exception is thrown in case of improper `split_size` or in case of an empty BitBuffer.
- **reverse** (*bool*) – If true, each individual range is bit-reversed before casting to int. Default is False.
- **signed** (*bool*) – If true, the cast is performed signed (two's complement), else unsigned. The sign is interpreted before assignment to the array. Signed cast together with an unsigned array type results into an integer with binary representation of the two's complement value. Default is False.
- **dtype** (*numpy.dtype, optional*) – Data type of desired numpy array. All signed and unsigned types up to 64 bit are allowed (`numpy.(u)int8 to numpy.(u)int64`) If dtype is not given, numpy.uint32 is used.

Return type

numpy.ndarray

Raises

- **ValueError** – If *split_size* is of type int: *split_size* is 0 or size of the buffer is not a multiple *split_size*. If *split_size* is of type Sequence[int]: The sum of all individual sizes is greater than the size of the buffer or one of the sizes is 0.
- **TypeError** – If *split_size* is of type Sequence[int]: One of the sizes is negative.

Examples

```
>>> BitBuffer.from_str('100101111').split_to_array(3)
array([1, 5, 7], dtype=uint32)
```

```
>>> import numpy as np
>>> BitBuffer.from_str('100101111').split_to_array(3, dtype=np.int8)
array([1, 5, 7], dtype=int8)
```

Adds 1 to each 3 bit element in BitBuffer.

```
>>> buffer = BitBuffer.from_str('100_101_111')
>>> array = buffer.split_to_array(3)
>>> array = (array + 1) % 8
>>> buffer = BitBuffer.from_int(array, 3)
'010011000'
```

Notes

This auxiliary function is helpful to interpret binary data as numbers and to calculate with them. Especially for large or very long arrays, it is useful because calculations with Numpy are usually much faster than an own implementation in Python.

split_to_int(*split_size: int | Sequence[int]*, *, *reverse: bool = False*, *signed: bool = False*) → list[int]

Splits the BitBuffer into several ranges, casts them to integer and returns a list of these integers.

Added in version 22.1.0.

Changed in version 25.2.0: Returns an empty list instead of throwing an exception when BitBuffer is empty.

Parameters

- **split_size** (*int | Sequence[int]*) – If int, the *BitBuffer* is split into multiple ranges of size `split_size`. The size of the BitBuffer has to be a multiple of `split_size`. If a list is given, the *BitBuffer* is split into multiple ranges of arbitrary length given in the list. The sum of the list must be less than or equal to the size of the *BitBuffer* and no element in the list is allowed to be 0. An exception is thrown in case of improper `split_size`.
- **reverse** (*bool*) – If true, each individual range is bit-reversed before casting to int. Default is False.
- **signed** (*bool*) – If true, the cast is performed signed (two's complement), else unsigned. Default is False.

Return type
list[int]

Raises

- **ValueError** – If *split_size* is of type int: *split_size* is 0 or size of the buffer is not a multiple *split_size*. If *split_size* is of type Sequence[int]: The sum of all individual sizes is greater than the size of the buffer or one of the sizes is 0.
- **TypeError** – If *split_size* is of type Sequence[int]: One of the sizes is negative.

Examples

```
>>> BitBuffer.from_str('100101111').split_to_int(3)
[1, 5, 7]
```

```
>>> BitBuffer.from_str('100101111').split_to_int([2, 3, 2], reverse=True)
[2, 2, 3]
```

```
>>> BitBuffer.from_str('110101111').split_to_int([3, 3, 2], signed=True)
[3, -3, -1]
```

```
>>> packet = BitBuffer.from_str('1101 0111 1')
>>> parameter0, parameter1, flag0 = packet.split_to_int([4, 4, 1])
>>> print(parameter0, parameter1, flag0)
11 14 1
```

Notes

This utility function is helpful when unpacking packets or headers with the same or different field sizes.

`time`(*position: int = 0*) → *ProTS*

Get timestamp of bit at the specified `position`. Note, that consecutive bits may have the same timestamp, e.g. if they were transmitted within a single multi-bit symbol.

Parameters
`position` (*int*) – The index of bit to get the timestamp

Return type
ProTS

`to_bytes`(*length: int | None = None, byteorder: str = 'little'*) → bytes

Converts the *BitBuffer* into `bytes`. This gives the same results as converting to an integer and then to bytes.

Added in version 22.2.0.

Parameters

- **`length`** (*int, optional*) – Specifies the size of the resulting byte object. An error is raised if this size is not sufficient to store the BitBuffer. If not specified, the minimum number of bytes is used.
- **`byteorder`** (*str*) – Endianness of the bytes: `little` or `big`

Return type
bytes

Examples

```
>>> BitBuffer.from_int(0x123f).to_bytes()
b'?\x12'
>>> BitBuffer.from_int(0x123f).to_bytes(byteorder="big")
b'\x12?'
>>> BitBuffer.from_int(0x123f).to_bytes(10)
b'?\x12\x00\x00\x00\x00\x00\x00\x00\x00'
```

`to_hex`(*num_bytes: int | None = None, byteorder: str = 'little', sep: str = '', bytes_per_sep: int = 1*) → str

Converts the *BitBuffer* into `str` in hexadecimal format. It behaves similar to `to_bytes` method.

Added in version 25.2.0.

Parameters

- **`num_bytes`** (*int, optional*) – Specifies the number of bytes of the resulting hexadecimal string object. If not specified, the minimum number of bytes is used.
- **`byteorder`** (*str*) – Endianness of the bytes: `little` or `big`.
- **`sep`** (*str*) – Symbol which is used as separator between bytes.
- **`bytes_per_sep`** (*int*) – Number of bytes which are separated.

Return type
str

Raises

- `OverflowError` – If *num_bytes* is smaller than the size of the *BitBuffer*.
- `ValueError` – If *byteorder* neither `little` nor `big`.

Examples

```
>>> BitBuffer.from_int(0x123f).to_hex()
'3f12'
>>> BitBuffer.from_int(0x123f).to_hex(byteorder="big")
'123f'
>>> BitBuffer.from_int(0x123f).to_hex(5)
'3f12000000'
>>> BitBuffer.from_int(0x123f).to_hex(5, sep="-")
'3f-12-00-00-00'
```

`to_int`(*signed: bool = False*) → int

Converts the *BitBuffer* into an *int*. Bits are interpreted LSB-first. Default behaviour, or if `signed` parameter is set to `false` data is considered *unsigned*. If data is to be interpreted *signed*, see the following optional parameter. This function returns 0 when called on an empty *BitBuffer* (*size* == 0).

Parameters

`signed` (*bool, optional*) – If true, interpret bitbuffer data as signed and return respective integer value as such. If false, interpret bitbuffer data as unsigned.

Added in version 21.2.0.

Return type

int

Example

```
>>> bb = BitBuffer.from_str("1101")
>>> bb.to_int()
11
>>> bb.to_int(signed=True)
-5
```

Notes

It is possible to use the builtin `int` for this.

```
>>> bb = BitBuffer.from_str("1101")
>>> int(bb)
11
```

`to_str`(*truncate: int = 0*) → str

Get string representation of bits. The bit at index 0 is first.

Parameters

`truncate` (*int, optional*) – If greater than 0, the string representation is truncated so that it contains only the first `truncate` bits.

Return type
str

unpack(*fmt: str*, *position: int = 0*) → int

Unpack value from `BitBuffer` with given format.

It is possible to `unpack` an integer value from the BitBuffer instead of using the `to_int` method.

Parameters

- **fmt** (*str*) – The format string of unpack contains three parts:
  - byte order: > or < (the arrow indicates little-endian(<) or big-endian(>) byte ordering in BitBuffer)
  - bit order: > or < (the arrow indicates to the first bit form the left(<) or from the right in BitBuffer)
  - **size**
    signed: digit (value 1 to 64), B (8 bit), H (16 bit), I (32 bit), Q (64 bit)

    unsigned: digit (value -64 to -1), b (8 bit), h (16 bit), i (32 bit), q (64 bit)

  Additional formats for unpack for interpretion as signed values:
- **position** (*int*) – The index of bit position to get unpacked bits.

Returns
Unpacked value

Return type
int

Examples

```
>>> bb = bitbuffer.BitBuffer(40)
>>> bb.pack( 0x69, "<<B", position=0)
8
>>> hex( bb.unpack( "<<B", position=0) )
'0x69'
```

Alternatively one can get the content as signed value.

```
>>> bb = bitbuffer.BitBuffer(40)
>>> bb.pack( 0x96, ">>B", position=0)
8
>>> bb.unpack( ">>B", position=0)
150
>>> bb.unpack( ">>b", position=0)
-106
```

Instead of a letter in the format string one can use a digit, which defines the number of bits for packing and unpacking.

```
>>> bb = bitbuffer.BitBuffer(40)
>>> bb.pack( 0x96, ">>5", position=0)
5
>>> hex( bb.unpack( ">>5", position=0) )
'0x16'
```

`property capacity`
: Max number of bits this *BitBuffer* can hold

  Type
  int

`property data`
: Access raw memory of *BitBuffer*

  Type
  memoryview

`property offset`
: Number of bits to ignore from the underlying byte memory

  Type
  int

`property quality`
: quality information, if available

`property size`
: Number of bits in this *BitBuffer*

  Type
  int

`property soft_bits`
: soft bits, if available

`property time_info`
: Returns the TimeInfo

  Type
  TimeInfo

The following class holds the information necessary to compute timestamps for individual bits. While the terminology is avoided here, these fields are inspired by bits originating from a transmission of symbols describing multiple bits. That is why consecutive bit positions may report the same timestamp.

`class procitec.common.bitbuffer.TimeInfo`

`property bitrate`
: Rate in bits per second.

  Type
  float

`property offset`
: Offset in range 0 to $repetition$ - 1. Indicates if the first bit in the *BitBuffer* does not correspond to the first bit of a group with identical timestamps (see also $repetition$).

  Type
  int

`property repetition`
: Repetition factor, if $repetition$ consecutive bits have an identical timestamp.

  Type
  int

`property timestamp`
: Timestamp corresponding to the first bit in the *BitBuffer*.

  Type
  ProTS

2.2.2. BitStream

This class handles a stream of bit-level data. Main usage within decoders is as interface to access the demodulator output, *apc.data*. This uses an internal buffer which is filled by the runtime with chunks of demodulated bits. Reading data from the stream may trigger the runtime to request more data behind the scenes.

The current position within the stream is always absolute, starting with the first incoming bit. Reading from the stream moves the position forward, allowing the runtime to load more data from the demodulator. In addition, it is possible to access data without changing the position, which may be required for the detection of the transmission format (trial and error).

`class procitec.common.bitbuffer.BitStream(`*buffer*`)`

A class which implements a stream like behaviour on a BitBuffer

Each *read()* call remembers the new position in the bit stream and continues accordingly.

`__iter__()`

Return an iterator for each individual bit in the bit stream

`__str__()`

Return str(self).

`bit_rate(`*position: int | None = None*`)` → float | None

Get bit rate at given absolute position if available.

Added in version 24.1.0.

Parameters
`position` (*int, optional*) – If None (default) the bit rate at the current position is returned; otherwise the bit rate at the requested absolute position

Returns
bit rate (1/s) if available, None otherwise.

Return type
float | None

`consume(`*num_bits*`)`

Consume (skip) bits in the stream

Parameters
`num_bits` (*int*) – Number of bits to consume (to skip)

`iter_read(`*size, channel_index=0*`)`

Return an iterator for the bit stream reading a specific number of bits from a channel

Parameters

- `size` (*int*) – Number of bits to read in each iteration
- `channel_index` (*int, optional*) – Index of the channel to read from

`peek(`*size=None, *, channel_index=0, include_quality=False*`)`

Peek into the bit stream

Peek into the stream (read from the stream) without consuming any data, i.e. the position in the stream is not modified.

This is the underlying function for *read()*.

Parameters

- **size** (*int, optional*) – Number of bits to peek. If *None* (default), *available* bits are peeked.
- **channel_index** (*int, optional*) – Index of the channel to peek into
- **include_quality** (*bool, optional*) – Add quality information for each bit

Raises
EndOfStreamError – If the requested number of bits can not be returned. This means the stream has been closed and data remaining in internal buffer is not sufficient.

Returns
the requested bits

Return type
BitBuffer

read(*size=None*, ***, *channel_index=0*, *include_quality=False*)

Read bits from bit stream. This is equivalent to a *peek()* combined with a *consume()*.

Parameters

- **size** (*int, optional*) – Number of bits to read. If *None* (default), *available* bits are read.
- **channel_index** (*int, optional*) – Index of the channel to read from. Note that bits on all other channels are consumed.
- **include_quality** (*bool, optional*) – Add quality information for each bit

Raises
EndOfStreamError – If the requested number of bits can not be returned. This means the stream has been closed and data remaining in internal buffer is not sufficient.

Returns
the requested bits

Return type
BitBuffer

rewind(*num_bits: int*) → None

Rewind the bit stream by a given amount of bits

Parameters
num_bits (*int*) – Number of bits to go back in the bit stream

Raises
RewindError – If the requested *num_bits* exceeds the buffered data length

set_position(*position: int*) → int

Set position in the bit stream to an absolute value

This is equivalent to calling *rewind()* if the new position is smaller than the current *position* respectively to calling *consume()* if the new position is larger than the current.

Note that if new position is beyond current end of buffer only available bits will be consumed and thus not trigger the runtime to request new data.

No action is performed if current and new position are equal.

Parameters
position (*int*) – Absolute position in the bit stream to jump to

Returns
The new position in the bitstream. This position can differ from the requested value if the requested position is beyond the currently available bits in the buffer.

Return type
int

Raises

- *RewindError* – If the requested position can not be reached because it is to far in the past
- *EndOfStreamError* – If the requested position can not be reached because it is past the end of the input data stream

`time`(*position: int | None = None*) → *ProTS*

Get timestamp at given absolute position

Parameters
position (*int, optional*) – If None (default) the timestamp at the current position is returned; otherwise the timestamp at the requested absolute position

Returns
Timestamp

Return type
ProTS

`property available`

Number of bits that can be read without blocking

Type
int

`property channels`

Number of channels

Type
int

`property closed`

True if the stream is closed

Type
bool

`property inverted`

True if read bit are inverted

Type
bool

`property position`

Position in the stream, i.e. the index of the next bit to be read

Type
int

`exception procitec.common.bitbuffer.RewindError`

Raised if *BitStream.rewind()* or *BitStream.set_position()* fails

`exception procitec.common.bitbuffer.EndOfStreamError`

Raised from *BitStream.read()* and *BitStream.peek()* if more data is requested than can ever be made available.

exception procitec.common.bitbuffer.BitStreamError

Common base class of *EndOfStreamError* and *RewindError*.

2.2.3. Helper Functions

procitec.common.bitbuffer.**mean_quality**(*data*, *size*, *channel_index=0*, *past=False*)

Calculate mean quality of either a BitBuffer or a BitStream. In case a BitStream is passed to this function, channel_index, *size* ` and past must be passed as well. Parameter channel_index will be ignored in case of APCGateway input stream if input_channel_mode is configured as interleaved (= default). In case a BitBuffer is provided, channel_index, size and past *must not* be passed. Slicing is supported for BitBuffer.

Added in version 21.1.0.

Parameters

- **data** (BitStream *or* BitBuffer) – Input data type of which mean quality shall be calculated.
- **size** (*int*) – Size of bitstream data in bits, beginning at current read pointer position, of which the mean quality shall be calculated. Must *not* be passed, if data is of type BitBuffer.
- **channel_index** (*int, optional*) – Channel index of BitStream of which mean quality shall be calculated (default = 0). Must *not* be passed if data is of type BitBuffer or using APCGateway input stream with configuration interleaved.
- **past** (*bool, optional*) – If True the bits before the read_pointer are used. If there are not enough bits the calculation length is truncated. Returns 0.0 if there are no bits available in the past. Must *not* be passed if data is of type BitBuffer.

Raises

RewindError – If data is of type BitStream, past is True and the requested size exceeds the buffered data length.

Returns

- Mean quality of all passed BitBuffer values, in case data is a BitBuffer.
- Mean quality of BitStream values, as defined by size, channel_index and past.

procitec.common.bitbuffer.**convert_symbols**(*data*, ***, *conv_dict*, *symbol_size_in*, *symbol_size_out=None*, *replacement=0*, *msb_first=False*, *include_meta_info=False*) → *BitBuffer*

Convert symbols of a BitBuffer as described in python dictionary conv_dict. Slicing is supported for BitBuffer.

Note: The size of data must be an integer multiple of symbol_size_in.

Added in version 21.2.0.

Parameters

- **data** (BitBuffer) – Input data of which symbols shall be converted.

- **conv_dict** (*dict(int, int)*) – Python dictionary specifying input and output symbol mapping. Where dictionary keys represent input symbols, and values represent the respective output symbol. The dictionary *does not* have to cover all possible symbols but can contain any arbitrary number of key/value pairs <= $2^{symbol\_size\_in}$. Symbols within `data` which are not found in `conv_dict` will therefore be converted as given by `replacement`.
- **symbol_size_in** (*int*) – Describes the number of bits in `data` that shall be interpreted as one symbol.
- **symbol_size_out** (*int, optional*) – Describes the number of bits that each output symbol will cover in the resulting output `BitBuffer`. Make sure that this value fits the required number of bits needed for `conv_dict` output symbols. If no value is given, `symbol_size_out` is assumed to be equal to `symbol_size_in`.
- **replacement** (*int, optional*) – This value will be put into the output `BitBuffer` in place for an input symbol in `data` which could not be found in `conv_dict` listed input symbols (default = 0).
- **msb_first** (*bool, optional*) – This flag specifies the bit-endianness of which each input symbol of `data` shall be interpreted. If this parameter is not specified (default = False), input symbols are interpreted as LSBit first.
- **include_meta_info** (*bool, optional*) – If `False` then any metadata (e.g. quality, time, burst info) is discarded. An error is raised if `symbol_size_in` and `symbol_size_out` are not identical but metadata should be preserved.

  Added in version 24.2.0.

Return type
BitBuffer with converted symbols as given by `data`.

Examples

Swap symbol '3' with '2':

```
>>> from pocitec.common.bitbuffer import BitBuffer, convert_symbols
>>> input = BitBuffer.from_str('00100111') # 0,1,2,3
>>> # convert '11' to '01' and '01' to '11' (3 => 2 and 2 => 3)
>>> output = convert_symbols(input, conv_dict={0:0, 1:1, 2:3, 3:2}, symbol_size_
↪in=2)
>>> print(f"in:  {input}\nout: {output}\nxor: {input ^ output}")
in:  00100111
out: 00101101
xor: 00001010
```

`pocitec.common.bitbuffer.concat(*args)` → *BitBuffer*

Concatenate multiple BitBuffers. Meta information is removed.

Parameters
***args** (`BitBuffer`) – Variable length argument list.

Return type
BitBuffer

procitec.common.bitbuffer.**insert**(*buffer*, *position*, *new_data*)

Inserts a BitBuffer into another BitBuffer at given position. Existing bits are shifted to the right.

Parameters

- **buffer** (BitBuffer) – existing data
- **position** (*int*) – position where to insert the new data
- **new_data** (BitBuffer) – new data to be inserted

Examples

```
>>> from procitec.common.bitbuffer import BitBuffer, insert
>>> buffer = BitBuffer.from_str('00100111')
>>> insert( buffer, 2, BitBuffer.from_str("11") )
>>> print(buffer)
00111001
```

procitec.common.bitbuffer.**reverse_symbol_order**(*buffer*, *bits_per_symbol*) → *BitBuffer*

Reverse the order of arbitrary sized symbols in the BitBuffer and all its shadow buffers. BitBuffer size has to be integer multiple of number of bits per symbol. Meta information is removed.

Parameters

- **buffer** (BitBuffer) – existing data
- **bits_per_symbol** (*int*) – position where to insert the new data

Return type
BitBuffer

Examples

Reverse the ordering of 3bit symbols

```
>>> from procitec.common.bitbuffer import BitBuffer, reverse_symbol_order
>>> buffer = BitBuffer.from_str('001 001 111 000')
>>> print(reverse_symbol_order(buffer,3))
000111001001
```

procitec.common.bitbuffer.**mirror_symbols**(*buffer*, *bits_per_symbol*, *include_meta_info=False*)

Reverses the order of bits within arbitrary sized symbols in a BitBuffer and all its shadow buffers. BitBuffer size has to be an integer multiple of number of bits per symbol.

Added in version 21.1.0.

Parameters

- **buffer** (BitBuffer) – Input data. Size has to be an integer multiple of number of bits per symbol.
- **bits_per_symbol** (*int*) – Size of one symbol in bits.

- **include_meta_info** (*bool, optional*) – If False then any metadata (e.g. quality, time, burst info) is discarded.

  Added in version 24.2.0: (note: older versions always kept quality but discarded other metainfo)

Return type
BitBuffer

Examples

reverse bit ordering in bytes

```
>>> from procitec.common.bitbuffer import BitBuffer, mirror_symbols
>>> buffer = BitBuffer.from_str('00100111 10100111')
>>> print(mirror_symbols(buffer,8))
1110010011100101
```

procitec.common.bitbuffer.**invert**(*buffer*, *ranges*)

Invert all bits inside given ranges of bitbuffer.

To invert all bits in a bitbuffer please use '~' operator, it returns a new bitbuffer with all bits inverted, but does not modify the origin.

Parameters

- **buffer** (BitBuffer) – Input data
- **ranges** (*list*) – Defines ranges of bits, which should be inverted.

Examples

Invert given bit ranges in-place.

```
>>> from procitec.common.bitbuffer import BitBuffer, invert
>>> buffer = BitBuffer.from_str('00100111')
>>> invert( buffer, [[1,2],[5,6]])
>>> print(buffer)
01000001
```

Invert complete bitbuffer in-place.

```
>>> from procitec.common.bitbuffer import BitBuffer, invert
>>> buffer = BitBuffer.from_str('00100111')
>>> invert( buffer, [(0,buffer.size)])
>>> print(buffer)
11011000
```

Return a new inverted bitbuffer by using '~' operator.

```
>>> from procitec.common.bitbuffer import BitBuffer
>>> buffer = BitBuffer.from_str('00100111')
>>> print(~buffer)
11011000
>>> print(buffer)
00100111
```

2.2.4. Shift Operations

These mirror the shift operators defined for BitBuffer objects with additional parameters to perform e.g. cyclic shifting.

procitec.common.bitbuffer.**rshift**(*buffer*, *shift*, ***, *circular=False*, *include_meta_info=True*) → *BitBuffer*

Shifts specified number of bits to the right, returning result in a new BitBuffer Optionally as circular operation and excluding metainformation.

Parameters

- **buffer** (BitBuffer) – Input data
- **shift** (*int*) – number of bits to be shifted
- **circular** (*bool, optional*) – If True then the shift operation is circular, i.e. bits dropping out on the right will be inserted at the left.
- **include_meta_info** (*bool, optional*) – If False then any metadata (e.g. quality, time, burst info) is discarded.

Return type
BitBuffer

procitec.common.bitbuffer.**rshift_inplace**(*buffer*, *shift*, ***, *circular=False*, *include_meta_info=True*)

Shifts specified number of bits to the right in the given BitBuffer. Optionally as circular operation and excluding metainformation.

Parameters

- **buffer** (BitBuffer) – Input data
- **shift** (*int*) – number of bits to be shifted
- **circular** (*bool, optional*) – If True then the shift operation is circular, i.e. bits dropping out on the right will be inserted at the left.
- **include_meta_info** (*bool, optional*) – If False then any metadata (e.g. quality, time, burst info) is discarded.

procitec.common.bitbuffer.**lshift**(*buffer*, *shift*, ***, *circular=False*, *include_meta_info=True*) → *BitBuffer*

Shifts specified number of bits to the left, returning result in a new BitBuffer Optionally as circular operation and excluding metainformation.

Parameters

- **buffer** (BitBuffer) – Input data
- **shift** (*int*) – number of bits to be shifted
- **circular** (*bool, optional*) – If True then the shift operation is circular, i.e. bits dropping out on the left will be inserted at the right.
- **include_meta_info** (*bool, optional*) – If False then any metadata (e.g. quality, time, burst info) is discarded.

Return type
BitBuffer

`procitec.common.bitbuffer.lshift_inplace(buffer, shift, *, circular=False, include_meta_info=True)`

Shifts specified number of bits to the left in the given `BitBuffer`. Optionally as circular operation and excluding metainformation.

Parameters

- **buffer** (`BitBuffer`) – Input data
- **shift** (*int*) – number of bits to be shifted
- **circular** (*bool, optional*) – If `True` then the shift operation is circular, i.e. bits dropping out on the left will be inserted at the right.
- **include_meta_info** (*bool, optional*) – If `False` then any metadata (e.g. quality, time, burst info) is discarded.

2.3. Decoding Library

This is the documentation for the Python module *procitec.decoding*.

This module contains functions and classes to perform common decoding tasks. These can be categorized into *Synchronisation and Search*, *Error Correction and Detection* and other *Utilities*, that ease reoccurring tasks.

Most of the functionality aims to be generic and works as building-blocks to decode bit-level transmissions of various types of complexity. Some of these need to keep state between invocations or require non-trivial computation to be initialized. In these cases a class, rather than a plain function, is provided - even if it has only a single method. This gives control over of the scope where state is kept and avoids the overhead of initialization when used many times.

All of the objects types from the *bitbuffer* module are used as primary bit-level data input and output format. Algorithms working with soft metrics use multi-dimensional arrays provided by the Python package numpy for symbol-level data. Conversion bit-level soft metrics, however, results in *BitBuffer* objects with the soft-bit values stored as meta-data in *BitBuffer.soft_bits*. This way consuming functions, like *ViterbiDecoder* can easily switch behavior.

- *Synchronisation and Search*
- *Error Correction and Detection*
- *Burst Operations*
- *Pre-Processing*
  - *Bit-Level Pre-Processing*
  - *Symbol-Level Pre-Processing*
  - *Utilities*
- *Alphabets*
- *Utilities*
- *File Output Helpers*
- *Bit Formatting*
- *Encoding*
- *Miscellaneous*

2.3.1. Synchronisation and Search

The functions here are used in many decoders to acquire the position of known or repeated patterns within the incoming data stream. Therefore they read directly from the input stream, reading and consuming data as needed.

Both the number of bits to be searched, the `max_offset` and the maximum number of bit errors can be specified to control the behavior of search runs. It is advised to choose a sensible `max_offset` and loop over multiple search calls. This allows reporting the decoders state between executions of search methods, see *procitec.decoding.runtime.SearchStateHandler*.

To search for different patterns in parallel, the search functions can be instructed not to consume data from the input stream. Subsequent searches run off the same position. The logic for looping over patterns and choosing an acceptable result needs to be implemented by the user.

procitec.decoding.**search_pattern**(*data*, *pattern*, *mask*, ***, *repetitions=1*, *max_errors=0*, *max_offset=None*, *auto_invert=False*, *consume_searched=True*)

Search for a certain bit pattern in the input stream

Parameters

- **data** (BitStream) – Input stream where the pattern is to be searched
- **pattern** (*str or* BitBuffer) – A string or a bit buffer describing the bit pattern to search.

  If a string is passed, relevant bit position, i.e. bits which must be set or cleared, are denoted with 1 respectively 0. Bits which may take any value ("don't care") are denoted with x or X. Any other value in the string is ignored.
- **mask** (BitBuffer) – A 1 in the mask denotes that the corresponding bit in `pattern` must be present; a 0 denotes that the corresponding bit may take any value ("don't care").

  Must not be passed if `pattern` is a string.

  Changed in version 22.1.0.

  If `pattern` is a BitBuffer, `mask` can be omitted. In this case `mask` is evaluated as if all bits of `mask` are 1, or in other words: Each bit in `pattern` counts.
- **repetitions** (*int, optional*) – search for `pattern` repeated `repetitions` times (default 1 repetition)
- **max_errors** (*int, optional*) – Tolerate up to `max_errors` bit errors (default 0), i.e. maximum allowed Hamming distance between the repeated `pattern` and the pattern found in `data`.
- **max_offset** (*int, optional*) – Maximum offset, until which the search is performed, i.e this is the furthermost offset, where a pattern can be found. If None (default), the search range is only limited by `data` and if no pattern is found the search won't be aborted until `data` ends (see *EndOfDataError*).
- **auto_invert** (*bool, optional*) – If True, a search with inverted `data` (see *BitStream.inverted*) is executed as well. The first match within the limits is considered as result, i.e. the smallest offset for which the number of errors is less or equal `max_errors`. If at this offset, both, the number of errors for the non-inverted and inverted search are less or equal `max_errors`, the non-inverted match is considered as result. If a inverted search is successful, the stream's `inverted` property is toggled.

  `False` by default.

- **consume_searched** (*bool, optional*) – If True (default), the searched bits in data are consumed.

  If the pattern is found:
  SearchResult.offset bits are consumed, i.e. *data.position* is at the first bit of the found pattern.

  If the search is aborted:
  data is consumed so that *data.position* equals the first offset, where the search would have been continued. This means if the search has been aborted due to max_offset, exactly max_offset + 1 bits are consumed.

Returns
The result of the search: *first* match considering also max_errors

Return type
SearchResult

Notes

The hard decision *search_pattern()* command returns *first* match, the soft decision *search_pattern_soft()* returns *best* match. Therefore the search results may be drastically different when replacing hard decision processing with soft decision.

If the pattern to be found is very long and contains too many "don't care" bits in relation to the known bits and repetitions is small then it might be better to use the function *bit_correlation_and_maxima()*.

procitec.decoding.**search_alphabet**(*data*, *alphabet*, ***, *repetitions*, *max_bit_errors*, *max_offset=None*, *auto_invert=False*, *sign_pattern=None*, *interleaving=None*, *consume_searched=True*)

Search for an arbitrary sequence of codewords defined in an *Alphabet*.

Parameters

- **data** (BitStream) – bit sequence to be searched on
- **alphabet** (Alphabet) – Defines valid codewords to consider in search. Only an *Alphabet* with a fixed-length code is supported.
- **repetitions** (*int*) – length of the codeword sequence to search
- **max_bit_errors** (*int*) – Total allowed number of bit errors in the found sequence compared to a valid one, i.e. the maximum allowed Hamming distance between the found sequence and a valid one. The number of permitted bit errors per codeword is limited to $\left\lfloor \frac{\text{max\_bit\_errors} + \text{codeword\_repetitions} - 1}{\text{codeword\_repetitions}} \right\rfloor$
- **max_offset** (*int, optional*) – Maximum offset, until which the search is performed, i.e this is the furthermost offset, where a pattern can be found. If None (default), the search range is only limited by data and if no pattern is found the search won't be aborted until data ends (see *EndOfDataError*).
- **auto_invert** (*bool, optional*) – If True, a search with inverted data (see *BitStream.inverted*) is executed as well. The first match within the limits is considered as result, i.e. the smallest offset for which the number of errors is less or equal max_bit_errors. If at this offset, both, the number of errors for the non-inverted and inverted search are less or equal max_bit_errors, the match with less errors is considered as result. If both have equal number of errors, the non-inverted match is considered as result. If an inverted search is successful, the stream's *inverted* property is toggled.

  False by default.

- `sign_pattern` (*int, optional*) – Enables search with varying inversion of codewords if *not* None (the default is None): For every bit set in `sign_pattern` the corresponding codeword in `data` is inverted before comparison with valid codewords in `alphabet`.

  This option is only available for `repetitions <= 64`.

  This option is *not* available if the parameter `interleaving` is used.

- `interleaving` (*tuple(int, int), optional*) – Search assuming bit-interleaved `data`. See *extract_interleaved()*.

  This option is not available if the parameter `sign_pattern` is used.

- `consume_searched` (*bool, optional*) – If True (default), the searched bits in `data` are consumed.

  If the pattern is found:
  SearchResult.offset bits are consumed, i.e. *data.position* is at the first bit of the found pattern.

  If the search is aborted:
  `data` is consumed so that *data.position* equals the first offset, where the search would have been continued. This means if the search has been aborted due to `max_offset`, exactly `max_offset + 1` bits are consumed.

Returns
The result of the search

Return type
SearchResult

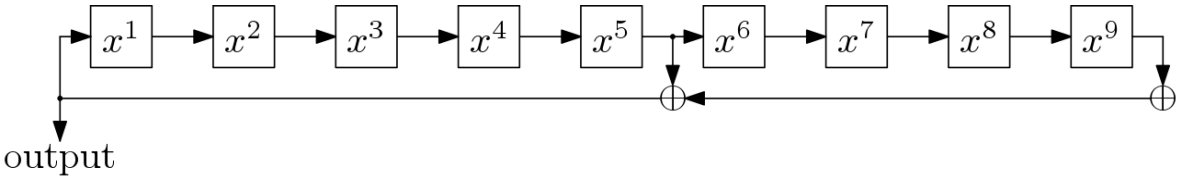
`procitec.decoding.search_lfsr_sequence`(*data:* BitStream, *poly:* BitBuffer | *int*, *window_length: int*, ***, *max_errors: int = 0*, *max_offset: int | None = None*, *auto_invert: bool = False*, *consume_searched: bool = True*)

Search for a sequence generated by an linear feedback shift register (LFSR).

Linear feedback shift registers are defined by polynomials of order n with binary coefficients c_i:

$$p(x) = 1 + \sum_{i=1}^{n} c_i x^i$$

An example for the LFSR with the polynomial $1 + x^5 + x^9$ is given below.



operating principle

- For each bit position `window_length` bits are considered from the input data. A check is performed on these bits to determine whether they match with a LFSR sequence with a given maximum number of errors. If so, the search is terminated with a positive result. If not, the next bit position is considered.
- When checking a single bit position, the polynomial is applied `window_length - polynomial_length + 1` times to the window of `window_length` bits. The error count is then compared to the maximum error count.

- A single bit error in the input data can lead to multiple errors in the search, depending on `poly`.

Added in version 24.1.0.

Parameters

- **data** (`BitStream`) – bit sequence to be searched on
- **poly** (`BitBuffer | int`) – Polynomial of the LFSR as a bit mask where the least significant bit corresponds to the coefficient of exponent 0. The coefficient for exponent 0 must be present. The position of the highest bit set (zero based counting) corresponds to n. The search for a polynomial with a length of up to 256 bit is supported.
- **window_length** (*int*) – Length of the search window to search for the LFSR sequence. Has to be >= the length of the polynomial (to perform a least one check). Maximum is 65536.
- **max_errors** (*int*) – Total allowed number of errors in the found sequence. A single bit error in the input data normally leads to multiple errors in the search. Default is 0.
- **max_offset** (*int, optional*) – Maximum offset, until which the search is performed, i.e this is the furthermost offset, where a LFSR sequence can be found. If `None` (default), the search range is only limited by `data` and if no LFSR sequence is found the search won't be aborted until `data` ends (see *EndOfDataError*).
- **auto_invert** (*bool, optional*) – If `True`, a search with inverted `data` (see *BitStream.inverted*) is executed as well. The first match within the limits is considered as result, i.e. the smallest offset for which the number of errors is less or equal `max_errors`. If at this offset, both, the number of errors for the non-inverted and inverted search are less or equal `max_errors`, the match with less errors is considered as result. If both have equal number of errors, the non-inverted match is considered as result. If an inverted search is successful, the stream's *inverted* property is toggled.

  `False` by default.
- **consume_searched** (*bool, optional*) – If `True` (default), the searched bits in `data` are consumed.

  If the LFSR sequence is found:
  SearchResult.offset bits are consumed, i.e. *data.position* is at the first bit of the found sequence.

  If the search is aborted:
  `data` is consumed so that *data.position* equals the first offset, where the search would have been continued. This means if the search has been aborted due to `max_offset`, exactly `max_offset + 1` bits are consumed.

Returns
The result of the search

Return type
SearchResult

Raises
ValueError – If the polynomial is inappropriate or `window_length` is too short.

Notes

If a LFSR sequence is embedded within some other data, the search may not found a bit position with the exact bit offset but one in front of it with some errors, especially in case of `max_errors > 0`. This is the correct behaviour of the algorithm.

Example

Generate a sequence of length 100 using the polynomial $x^9 + x^5 + 1$. This sequence is placed in a random bitbuffer of length 200 at offset 42. A search for the sequence is performed.

```
import random

from procitec.common.bitbuffer import BitBuffer, BitStream
from procitec.decoding import generate_lfsr_sequence, search_lfsr_sequence

l = 200
n = 100
poly = 0x221

random.seed(1)
random_bits = BitBuffer.from_int(random.randint(0, 2**l - 1), l)
random_bits[42:142] = generate_lfsr_sequence(poly, initial_state=123, length=n)
search_result = search_lfsr_sequence(BitStream(random_bits), poly, 70, max_
↪errors=0)
print(search_result)
```

```
>>> found: True, offset: 42, bit_errors: 0
```

`class procitec.decoding.SearchResult`

Result returned by *search_pattern()*, *search_alphabet()* and *search_lfsr_sequence()*

`__bool__()`

same value as *found*

`property errors`

If `found` is `True`, number of bit errors in the found sequence, i.e. the Hamming distance between the searched pattern and the found pattern in the given data; otherwise undefined.

Type
int

`property found`

`True` if the search has been successful, `False` otherwise.

Type
bool

`property offset`

If `found` is `True`, offset at which the pattern has been found, otherwise undefined.

Type
int

procitec.decoding.**search_pattern_soft**(*data*, *pattern*, *, *sample_format=MAG_PHASE*, *bits_per_symbol=None*, *method='phase'*, *range*, *max_symbol_errors*, *consume_searched=True*)

Searches for a symbol pattern in the input buffer by comparing corresponding nominal and input soft symbols. The position found is not the first matching pattern, but the best based on an error metric.

Parameters

- **data** (*SymbolStream or array_like*) – Input stream or buffer where the pattern is to be searched. A buffer has to satisfy the Python Buffer Protocol (`https://docs.python.org/3/c-api/buffer.html`) and should be a NumPy array. It is possible to cast symbols read from a SymbolStream to a NumPy array. See examples below.
- **pattern** (*list(tuple(float, float)) or array_like*) – The symbol pattern to search for. A buffer (NumPy-Array) should be preferred because it is faster, especially for long patterns. See examples below.
- **sample_format** (*SampleFormat, optional*) – The format of symbols in data and pattern. Has to be MAG_PHASE or FSK. If the type of data is *SymbolStream*, this option is ignored and the sample format defined within the symbol stream is used.

  For MAG_PHASE, the method has to be `'phase'` or `'euclid'`.

  For FSK, the method has to be `'fsk'`.
- **bits_per_symbol** (*int, optional*) – Number of bits per symbol. This value is used to determine the presence of a symbol error. Must be provided if `data` is not of type *SymbolStream*.

  Changed in version 25.2.0.

  bits_per_symbol is now optional if `data` is of type SymbolStream.
- **method** (*str, optional*) – Method for symbol error and error metric calculation. One of the following:
  - `'phase'`: based on phase difference, magnitude ignored
  - `'euclid'`: based on euclidean distance
  - `'fsk'`: based on absolute symbol distance
- **range** (*int*) – Length of search range in number of symbols. The search range must cover the maximum search offset plus the length of the pattern. This means, the furthermost offset, where a pattern can be found is exactly at `range` minus length of pattern.
- **max_symbol_errors** (*int*) – Maximum number of symbol faults allowed for a successful search.
- **consume_searched** (*bool, optional*) – If `True` (default), the searched symbols in `data` are consumed, except if `data` is not a *SymbolStream*, in this case this argument is ignored.

  If the pattern is found:
  SearchPatternSoftResult.offset symbols are consumed, i.e. *data.position* is at the first symbol of the found pattern.

  If the search is aborted:
  `data` is consumed so that *data.position* equals the first offset, where the search would have been continued. This means if the search has been aborted due to the `range`, exactly `range - len(pattern) + 1` bits are consumed, where `len(pattern)` equals the length of `pattern`.

Returns
The result of the search: *best* match within the `range` considering `max_symbol_errors`

Return type
SearchPatternSoftResult

Examples

We want so search for a 4-symbol QPSK preamble which could be defined with a list or with a NumPy array. The symbols are given in `MAG_PHASE` format.

```
>>> from procitec.decoding import search_pattern_soft
>>> import numpy as np
>>> pattern_list = [(1, 0), (1, 0), (1, np.pi), (1, np.pi/2)]
>>> pattern_numpy_float = np.array(pattern_list, dtype=np.float32)
>>> pattern_numpy_uint16 = np.array([(2**15, 0), (2**15, 0), (2**15, 2**15),␣
↪(2**15, 2**14)], dtype=np.uint16)
```

These pattern variables are equivalent in *search_pattern_soft()*.

```
>>> for p in pattern_list, pattern_numpy_float, pattern_numpy_uint16:
>>>     print(search_pattern_soft(apc.symbols, p, method="euclid", bits_per_
↪symbol=2, range=200, max_symbol_errors=0, consume_searched=False))
SearchPatternSoftResult(True, 42, 0, 0.0)
SearchPatternSoftResult(True, 42, 0, 0.0)
SearchPatternSoftResult(True, 42, 0, 0.0)
```

search_pattern_soft() can also be used to search for a pattern in a variable.

```
>>> search_me = np.array([(1, 0), (1, np.pi), (1, 0), (1, 0), (1, np.pi), (1, np.
↪pi/2), (1, np.pi), (1, np.pi)], dtype=np.float32)
>>> search_pattern_soft(search_me, pattern_list, method="euclid", bits_per_
↪symbol=2, range=200, max_symbol_errors=0)
SearchPatternSoftResult(True, 2, 0, 0.0)
```

Notes

The result from *apc.symbols.read* cannot be directly used in *search_pattern_soft()*. It has to be converted to a NumPy array first.

```
>>> search_me = np.array(apc.symbols.read(20), dtype=np.uint16)
>>> search_pattern_soft(search_me, pattern_list, method="euclid", bits_per_
↪symbol=2, range=200, max_symbol_errors=0)
SearchPatternSoftResult(True, 8, 0, 0.0)
```

The hard decision *search_pattern()* command returns *first* match, the soft decision *search_pattern_soft()* returns *best* match. Therefore the search results may be drastically different when replacing hard decision processing with soft decision.

`class procitec.decoding.SearchPatternSoftResult`

Result returned by *search_pattern_soft()*; fields from *SearchResult* are available as well. Offset and errors in symbols, not bits.

`__bool__()`
same value as *found*

`__str__()`
Return str(self).

`property error_metric`
Depends on search method. Undefined if `found=False`

- `'phase'`: averaged phase difference in range [0, 2*pi]
- `'euclid'`: averaged euclidian symbol distance
- `'fsk'`: averaged absolute symbol distance

Type
float

`property errors`
If `found` is `True`, number of symbol errors in the found sequence, otherwise undefined.

Type
int

`property found`
`True` if the search has been successful, `False` otherwise.

Type
bool

`property offset`
If `found` is `True`, offset at which the pattern has been found, otherwise undefined.

Type
int

`procitec.decoding.bit_correlation_and_maxima(`*data*, *pattern*, ***, *normalize=False*, *count_maxima=0*`)`

Computes binary correlation of two BitBuffers and searches for the requested number of maxima in the correlation values

The correlation is computed according to

$$c[k] = \sum_{i=0}^{N} d[i+k] \oplus p[i],$$

where d and p are `data` and `pattern` respectively and N is the size of `pattern`.

Added in version 21.1.0.

Parameters

- **data** (`BitBuffer`)
- **pattern** (`BitBuffer`) – Size must not exceed the size of `data`.
- **normalize** (*bool*) – If `False` (default), the correlation values correspond to the number of identical bits for the specific shift. Consequently, the values are in the range $[0, N]$, with N being the size of `pattern`.

  If `True`, the returned correlation values are $c'[k] = c[k]/(N/2) - 1$, i.e. the returned values are in the range $[-1.0, +1.0]$.

- `count_maxima` (*int*) – If greater than 0, this defines the number of maxima to be searched on the correlation result.

Returns

- **correlation** (*numpy.ndarray*) – Correlation values, where the index corresponds to the shift between `pattern` and `data`.
- **maxima** (*list(tuple(int, float))*) – List of maxima, where each maxima is a tuple of position and value of the maxima. The maxima are sorted according to the values in descending order.

Example

```
>>> data = BitBuffer.from_int(0b0110001111000111, 16)
>>> pattern = BitBuffer.from_int(0b1111, 4)
>>> corr, maxima = bit_correlation_and_maxima(data, pattern, count_maxima=2)
>>> corr
array([3.       , 2.       , 1.       , 1.0000001, 2.       , 3.       ,
       4.       , 3.       , 2.       , 1.       , 1.       , 1.9999999,
       2.       ], dtype=float32)
>>> maxima
[(6, 4.0), (0, 3.0)]
```

2.3.2. Error Correction and Detection

The functionality in this section may be used to decode various forward error correction codes or verify the correctness of a transmission.

`class procitec.decoding.BlockDecoder(`*P*, *d*`)`

Decoder for binary linear block codes described by a generator matrix.

The code must be systematic, i.e. its generator matrix G must have the form $G = [I|P]$ where I is a $k \times k$ identity matrix and P is a $k \times (n-k)$ matrix defining the equations for the parity bits. n is the length of a codeword (a block) in bits and k is the number of information bits.

Parameters

- `P` (*list(list(int))*) – Matrix P as defined above. Each sub-list is a row of the matrix and the elements must be either 0 or 1.
- `d` (*int*) – minimum distance of the block code

Example

Initialize a decoder for a $C(7,4,3)$ Hamming code with

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

```
>>> import procitec.decoding as ddl
>>> P = [[1, 1, 0], [1, 0, 1], [0, 1, 1], [1, 1, 1]]
>>> dec = ddl.BlockDecoder(P, 3)
```

__call__(*data*)

see *decode()*

decode(*data*)

Perform error correction of a codeword

Parameters

data (BitBuffer) – The length of data must be at least the size a codeword (i.e. $\geq n$). The order of bits must be $[i_0 i_1 \ldots i_{k-1} p_0 p_1 \ldots p_{n-k-1}]$ (in order of ascending indices in data). This is the ordering one gets when performing encoding as iG where i is a row vector of information bits.

Returns

- **data** (*BitBuffer*) – Corrected codeword. The input codeword is returned unchanged in case of a decoding failure.
- **errors** (*int*) – Number of corrected bits. -1 if decoding is not possible (too many errors).

MAX_BLOCK_LEN = 32

MAX_MIN_DISTANCE = 9

MAX_PARITY_LEN = 20

MIN_BLOCK_LEN = 3

MIN_MIN_DISTANCE = 3

property d

minimum distance of the code as provided at initialization

property k

number of information bits in a codeword; determined from matrix P at initialization

property n

block length of the code in bits; determined from matrix P at initialization time

procitec.decoding.crc(*poly*, *data*, ***, *initial_state=0*, *flush_zeros=True*)

Universal cyclic redundancy check (CRC)

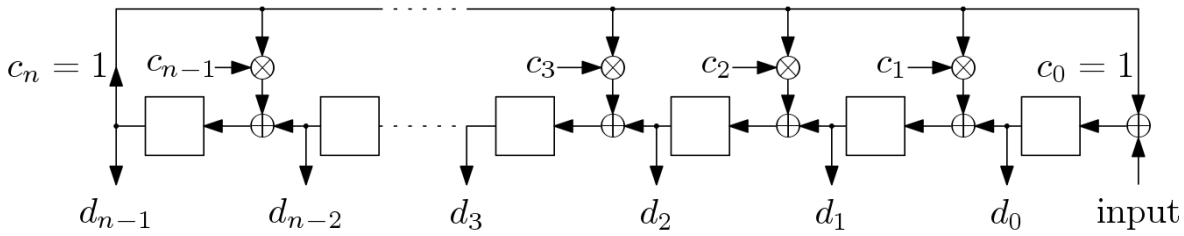
A cyclic redundancy check is defined by a generator polynomial of order n with binary coefficients c_i:

$$g(x) = 1 + \sum_{i=1}^{n} c_i x^i$$

The CRC (digest) of a data block viewed as a polynomial $i(x)$ is usually determined by

$$i(x) \bmod g(x)$$

using $\mathrm{GF}(2)$ arithmetic. This operation may be implemented using the shift register circuit shown below. Note that some references may reverse the order of the labels or the shift direction. The CRC is the content of the shift register after shifting-in the last bit.



Some standards define the CRC as

$$x^n \cdot i(x) \bmod g(x)$$

which is equivalent to appending n zero bits to the data block; use the option `flush_zeros` in this case.

Parameters

- **poly** (*int*) – Generator polynomial of the CRC as a bit mask where the least significant bit corresponds to the coefficient of exponent 0 (always 1). The next significant bit is coefficient c_1 and so on. The position of the highest bit set (zero based counting) corresponds to n.
- **data** (`BitBuffer`) – Update the shift register (the CRC) using bits in `data`.
- **initial_state** (*int*) – Initial state of the shift register used for the CRC calculation.
- **flush_zeros** (*bool*) – If `True`, flush the shift register with zeros before returning the result.

`procitec.decoding.correct_majority`(*data*, *repeat_count*)

Correct bits of a repeated frame using majority decision rule

If the number of repetitions (`repeat_count`) is even and the count of zeros and ones of a bit in the frame is equal, bit 1 is used as the corrected value.

Parameters

- **data** (`BitBuffer`) – The repeated frame. Length must be a multiple of `repeat_count`.
- **repeat_count** (*int*) – count of frame repetitions (>=3)

Returns

- **data** (*BitBuffer*) – corrected frame of length `len(data) / repeat_count`
- **bit_flipped** (*int*) – overall number of bits flipped due to correction process, in all repeated frames

Examples

- frames of length 8 and `repeat_count = 3`

00101011 first data frame
00011011 second data frame
01011001 third data frame

00011011 output, 4 bits flipped (in 2nd, 3rd, 4th and 7th column)

Each frame is placed into a single row for better explanation. The correction according to the majority rule is applied across the columns, i.e. using corresponding bits of the frame.

- frames of length 3 and `repeat_count = 5`

```
001
001
011
101
110
---
001, 5 bits flipped (2 bits in 1st and 2nd column, 1 bit in 3rd)
```

- frames of length 4 and `repeat_count = 4`

```
0011
0101
1101
1101
----
1101, 4 flipped (2 bits in 1st columns, 1 bit in 2nd and 3rd)
```

If the number of zeros and ones is equal – as in the first column – bit 1 is returned.

`class procitec.decoding.ViterbiDecoder(*args, **kwargs)`

A Viterbi decoder for non-recursive convolutional codes with rate $1/n$

1. **ViterbiDecoder** (*mode, polys, *, puncture_pattern=[], use_soft_bits=False, initial_state=0, final_state=0, metric_type="trellis"*)
2. **ViterbiDecoder** (*mode, trellis_table, rate, *, puncture_pattern=[], use_soft_bits=False, initial_state=0, final_state=0*)

As implied by the name, a convolutional code may be viewed as a filter with arithmetic in $\mathrm{GF}(2)$. Due this correspondence a convolutional code with rate $1/n$ is usually described by n transfer functions (generator polynomials), each in the form

$$g_i(D) = 1 + \sum_{k=1}^{K-1} c_k D^k$$

where K is the constraint length of the code and $c_k \in \{0, 1\}$. An example for the implementation of a convolutional encoder is given below.

Parameters

- **mode** (*{"terminated", "truncated", "tailbiting", "streaming"}*) – The mode of operation of the decoder which is determined by the encoding process:
  - `"terminated"`: The encoder is flushed with a known sequence of $K-1$ bits (usually zeros). This way the final state of the encoder is known to the decoder. Also known as *flushed* encoding.
  - `"truncated"`: The final state of the encoder is determined by the last bits of the information block, i.e. the final state is not known to the decoder.

– `"tailbiting"`: The encoder is preloaded with the last bits of the information block, i.e. the initial and final state of the encoder are equal.

– `"streaming"`: The encoder is never flushed; an infinite stream of information bits is assumed. It is possible to use a defined or undefined `initial_state` for the first decoding block. Manual flushing with termination to defined state is possible.

Changed in version 23.1.0.

The default `initial_state` in mode `"streaming"` changed from undefined to 0. For former behaviour `initial_state` has to be set to None.

The encoding modes `"terminated"`, `"truncated"` and `"tailbiting"` can be regarded as block codes, i.e a fixed amount of data is encoded. Therefore all bits from one encoded block are required for decoding and the decoder does not retain an internal state. With `"streaming"` the decoder is fed with blocks of the infinite encoded bit stream and keeps an internal state for consecutive processing. Flushing the decoder is possible either with a known terminating end state or an unknown end state. See *`flush()`* for flushing in mode `"streaming"`.

- **`polys`** (*`list(int)`*) – A list of polynomials defining the convolutional code. Each element must be a bit mask where the least significant bit corresponds to the coefficient of exponent 0. The position of the highest bit set (zero based counting, in any of the polynomials) determines the shift register length used in the encoding process ($K-1$).

  Note: the required representation of the generator polynomials is reversed compared to commonly used octal representation. The common representation as found in most books and references puts the coefficient of the highest exponent into the least significant bit; the coefficient for exponent zero is consequently at the position of the highest bit set. See examples below.

  Not allowed in combination with parameters `trellis_table` and `rate`.

- **`trellis_table`** (*`list(tuple(tuple(int, int), tuple(int, int), int))`*) – A list which describes the trellis. See *`trellis_table`* for a more detailed parameter description. This allows to configure a trellis which cannot be abstracted from the polynomials easily.

  The length of the list has to be a power of 2. Not allowed in combination with parameter `polys`.

  Added in version 21.1.0.

- **`rate`** (*`int`*) – Specifies the rate of the code (mother code if punctured). This setting equals to the number of polynomials, so this setting is only allowed in combination with parameter `trellis_table`, since the rate cannot be extracted directly from the table.

  Added in version 21.1.0.

- **`puncture_pattern`** (*`list(int) or str, optional`*) – A list or string describing the puncturing performed in the encoding process. A 1 indicates that the corresponding is *not* punctured, i.e. transmitted; a 0 indicates that the corresponding bit is punctured, i.e. *not* transmitted.

  The puncture pattern is traversed periodically. This means:

  – A puncture matrix can be passed by writing the elements of the matrix columnwise into the list.

  – A list whose length is equal to the length of an information block is traversed only once.

By default (empty list) no puncturing is assumed. In mode `"streaming"` it is possible to change the puncturing between blocks.

- **use_soft_bits** (*bool, optional*) – If `True`, make use of soft-bits when decoding. By default (`False`) hard decision bits are used.
- **initial_state** (*int | None, optional*) – Initial state of the encoder. This is the initial state of the shift register interpreted as a binary number (default 0). Only used if `mode` is `"terminated"`, `"truncated"` or `"streaming"`; ignored otherwise, unless the mode is changed using *switch_mode()*. If None is passed, all states are treated as equally probable. In mode `"tailbiting"` initial and final states are calculated internally from the given coded data.
- **final_state** (*int, optional*) – Final state of the encoder. This is the content of the shift register interpreted as a binary number, after flushing has been performed. Only relevant and used if `mode` is `"terminated"` or the decoder is flushed in mode `"streaming"` using *flush()*; ignored otherwise, unless the mode is changed using *switch_mode()*.
- **metric_type** (*{"trellis", "reencode"}, optional*) – Defines how the returned error metric in calls of *decode()* or *flush()* is calculated.
  - `"trellis"`: The metric is calculated based on the error metric of the final state in trellis.

    In case of hard-bit processing this value matches the number of detected bit-errors in the coded data.

    In case of soft-bit processing an approximation comparable with the hard-bit value is done. This approximation is poor or meaningless when noisy soft-bits are used. Even when the corresponding hard-bits are error free, the approximation of the bit-errors could be bad. It may be better to use the metric_type `"reencode"` in this case.

    An approximation is also done for mode `tailbiting` (soft and hard-bit processing), since the input or part of it is put in the trellis decoding at least two times, which leads to an higher error metric of the final state.
  - `"reencode"`: The decoded bits are encoded again and compared with the input bits. The error metric is the hamming distance.

    If soft-bit processing is used, the input soft-bits are converted to hard bits before comparing. This error metric is more meaningful when working with soft-bits, since soft information is discarded for metric calculation and it is easier to determine an error limit in order to check input bits for correct convolutional coding.

  The default is `"trellis"`.

  Not allowed together with parameter `trellis_table`.

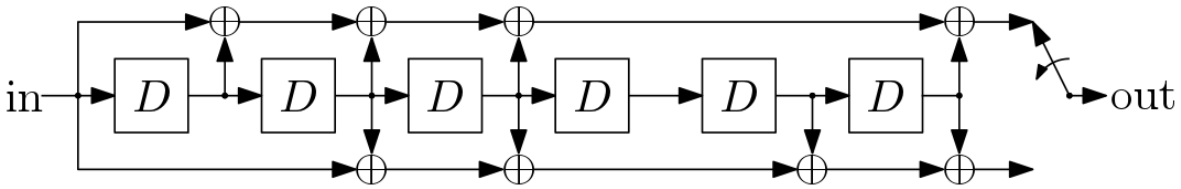
  Added in version 25.1.0.

Examples

A popular convolutional code with rate $1/2$ and constraint length $K = 7$ is

$$g_1(D) = 1 + D + D^2 + D^3 + D^6$$
$$g_2(D) = 1 + D^2 + D^3 + D^5 + D^6.$$

The corresponding encoder implementation is depicted below. The state of the encoder is the content of the shift register interpreted as a binary number.



The common octal representation of the polynomials is 171 respectively 133. The octal representation is retrieved by interpreting the coefficients of the polynomial with ascending exponents as a binary number, here 1111001 respectively 1011011. Note this implies that the coefficient for exponent 0 is the most significant set bit and the coefficient for $K-1$ is at the least significant bit (LSB)

The initialization of the Viterbi Decoder requires a reversed representation, i.e the coefficient for exponent 0 is at the LSB and the coefficient for $K-1$ is at bit $K-1$ (zero based counting). Consequently the required representation for the polynomials given above is binary 1001111/octal 117/ hexadecimal 4F respectively 1101101/155/6D.

For terminated encoding, also called flushed encoding, the decoder is initialized as shown below. By default the initial and final state of the encoder as assumed to be 0.

```
>>> import procitec.decoding
>>> vit_dec = procitec.decoding.ViterbiDecoder("terminated", [0b1001111, 0x6D])
```

`decode`(*data:* BitBuffer | *Sequence[Literal[0, 1]]*) → tuple[*BitBuffer*, int]

Decode one block of data

Unless mode is `"streaming"`, all calls to *decode()* are independent and assume that the whole coded block is passed. With `"streaming"` mode an infinite stream of coded blocks of different sizes may be passed. However, no output will be made as long less than approximately *decoding_delay* * *approximate_rate* bits have been provided.

For flushing the decoder in mode `"streaming"` see *flush()*.

Parameters

data (*BitBuffer or list(int)*) – Encoded and punctured data to be decoded. In case of a BitBuffer and soft-decision active, the soft decision values are expected to be placed within shadow-buffer “soft”.

In case of an integer list each list entry represents one bit. For soft-decision the integer values have to be within range [-127, 127] and for hard decision only the values [0, 1] are allowed. All other values lead to undefined behavior.

For a soft bit the most confident 1 is soft value -127, the most confident 0 is soft value 127.

Note: With a soft value of 0 it is possible to puncture the code manually.

Returns

- **data** (*BitBuffer*) – decoded bits; empty if mode is `"streaming"` and not enough data has been provided
- **metric** (*int*) – Error metric of the coded input data. The value depends on the selected `metric_type`. See description of `metric_type` in constructor of *ViterbiDecoder*.

`flush`(*terminate: bool = False*) → tuple[*BitBuffer*, int]

Only useful in mode `"streaming"`. For the other modes the flush is always included in the call of *decode()*.

Due to the decoding delay some bits remain in the decoders trellis even after the last call of *decode()*. This method flushes these remaining bits either with a known final state (most times 0) or in a `"truncated"` fashion.

The decoder is also reset (see *reset()*).

Added in version 23.1.0.

Parameters

`terminate` (*bool*) – If `False`, the remaining bits in the trellis are flushed up to the trellis state with best metric (as in mode `"truncated"`). Otherwise the set final state (see *set_final_state()*) is used for flushing (as in mode `"terminated"`).

The default is `False`.

Returns

- **data** (*BitBuffer*) – decoded bits; empty if mode is not `"streaming"` or not enough data has been provided.
- **metric** (*int*) – Error metric of the coded input data. The value depends on the selected `metric_type`. See description of `metric_type` in *ViterbiDecoder*.

`reset`() → None

Reset internal state of the decoder

Has only an effect if `mode` is `"streaming"` (all other modes are stateless).

`set_final_state`(*final_state: int*) → None

Set final state of the encoder

Parameters

`final_state` (*int*) – see *ViterbiDecoder* for details

`set_initial_state`(*initial_state: int | None*) → None

Set initial state of the encoder

Parameters

`initial_state` (*int or None*) – see *ViterbiDecoder* for details

`set_puncture_pattern`(*puncture_pattern: Sequence[Literal[0, 1] | str]*) → None

Change the puncture pattern

The internal puncturing position is reset to 0. In mode `"streaming"` it is possible to change the puncturing between blocks.

Parameters

`puncture_pattern` (*Sequence[int] or str*) – see *ViterbiDecoder* for details

`switch_mode`(*mode: Literal['terminated', 'truncated', 'tailbiting', 'streaming']*) → None

Switch the decoding mode

The decoder is also reset (see *reset()*).

Parameters

`mode` (*{"terminated", "truncated", "tailbiting", "streaming"}*) – The new mode to use in the following calls to *decode()*; see *ViterbiDecoder*.

property approximate_rate

Approximate rate of the configured code including puncturing. For non-punctured codes the rate is $1/n$.

Type
float

property decoding_delay

Minimal internal decoding delay. Of importance for "streaming" mode only.

Type
int

property last_final_state

Last final state of the decoder

Final state of the decoder (of the trellis) reached in the last call to *decode()* or *flush()*. Only valid if *decode()* has been called at least once. The value depends on mode:

- "terminated" / "streaming" after call of *flush()* with terminate=True: The end state either defined at initialization time of the decoder or set by *set_final_state()*.
- otherwise: The state with the best metric.

Type
int

property memory_usage

Approximate memory usage for internal buffers in bytes

Type
int

property mother_rate

The value of n. This is the inverse rate of the configured mother code (i.e. puncturing is ignored).

Type
int

property puncture_pattern

Current puncture pattern

Type
list(int)

property trellis_table

Trellis table of the configured code

Each element of the returned list of length 2**reg_len is a tuple describing a state in the trellis of the code. The members of the tuple are:

- (s0, s1): previous states. The transition to state s0 was caused by information bit 0, transition to state s1 by information bit 1.
- (c0, c1): coded bits output by the encoder by a transition to the current state, packed in an integer. c0 is the output if the transition to the current state was caused by information bit 0; c1 is the output if the transition was caused by information bit 1.
- info: information bit corresponding to the current state, i.e. information bit which caused a transition to the current state.

Type
list[tuple[tuple[int, int], tuple[int, int], int]]

class procitec.decoding.**ReedSolomonDecoder**(*n*, *k*, *prim_poly*, ***, *first_root=1*, *shorten=0*, *puncturing_pos=[]*)

Decoder for Reed-Solomon (RS) codes defined over $\mathrm{GF}(2^m)$.

Decoder for Reed-Solomon codes defined over an extension Galois field of order 2^m ($\mathrm{GF}(2^m)$). Shortened and punctured codes are supported. Erasure positions, i.e. positions of symbol which are known to be erroneous, may be provided to aid the decoder. Systematic encoding is assumed.

The order m of the primitive polynomial `prim_poly` for the construction of $\mathrm{GF}(2^m)$ defines

- the length n of the un-shortened and un-punctured "mother" code (in symbols)
- and the length of a symbol in bits.

The generator polynomial for a RS code is usually defined by

$$g(x) = \prod_{i=b}^{n-k+b-1} (x - \alpha^i)$$

where b is the position of the first root (usually 1), k is the number of information symbols and α is a primitive element of the field. The number of redundancy symbols $n-k$ defines the correction capability of the code: The number of correctable errors is limited by $2t + e \leq n - k$ where t is the number of errors and e is the number of punctures and erasures.

Parameters

- **n** (*int*) – Length of the un-shortened and un-punctured mother code. Must be 2^m-1 where m is the order of the primitive polynomial `prim_poly`.
- **k** (*int*) – Number of information symbols of the un-shortened mother code
- **prim_poly** (*int*) – Primitive polynomial for the construction of the Galois field. This must be a bit mask where the least significant bit corresponds to the to coefficient of exponent 0. The position of the highest bit set (zero based counting) determines m.
- **first_root** (*int, optional*) – Position of the first root in the generator polynomial of the code $g(x)$ (b in the equation above). Must be greater than 0 (default 1).
- **shorten** (*int, optional*) – Number of symbols by which the mother code is shortened (default 0)
- **puncturing_pos** (*list(int), optional*) – Position of punctured symbols. By default (empty list) a code without puncturing is assumed.

  Only redundancy symbols may be punctured, i.e. the positions must be in the range $[0; n-k[$. Number of punctured symbols must be smaller than $n-k$.

Examples

A decoder for a RS code with 9 information symbols in $\mathrm{GF}(2^4)$ using the primitive polynomial x^4+x+1. The code is not shortened or punctured:

```
>>> from procitec.decoding import ReedSolomonDecoder as RS
>>> rs_dec = RS(15, 9, 0x13)
```

A decoder for a code with the following specification:

- mother code with $n = 63$ and $k = 45$

- shortened by 10 symbols
- first 4 symbols of the redundancy are punctured
- $x^6 + x + 1$ as primitive polynomial

The resulting codewords have a length of $n' = n - \text{shorten} - 4 = 59$ symbols and contain $k' = k - \text{shorten} = 35$ information symbols.

```
>>> from procitec.decoding import ReedSolomonDecoder as RS
>>> rs_dec = RS(63, 45, 0x43, shorten = 10, puncturing_pos = [0, 1, 2, 3])
```

__call__(*data*, *, *shorten=None*, *puncturing_pos=None*, *erasure_pos=[]*)

see *decode()*

decode(*data*, *, *shorten=None*, *puncturing_pos=None*, *erasure_pos=[]*)

Perform error correction

Parameters

- **data** (BitBuffer) – The codeword to be corrected. The length of data must be at least the size of a codeword in bits, i.e. $\geq m \cdot (n - \text{shorten} - \text{number of punctures})$. The codeword polynomial must be given in order of descending exponents, i.e. the lower $k' = k - \text{shorten}$ symbols are information symbols (systematic encoding). The bits of symbols must be given in order of descending exponents as well.
- **shorten** (*int, optional*) – If *None* (default), the shortening amount provided at initialization time is used. Otherwise, the provided value is used.
- **puncturing_pos** (*list(int), optional*) – If *None* (default), the puncturing positions provided at initialization time are used. Otherwise, the provided list of puncturing positions is used.
- **erasure_pos** (*list(int), optional*) – If the list is empty (default) no erasures are considered when decoding. Otherwise the list defines the positions of erasures (symbols which are known to be erroneous).

  The following rules apply:

  – Erasure positions must be provided in terms of the un-shortened and un-punctured codeword.

  – Duplicate erasure and/or puncturing positions must not be provided.

  – Erasure positions must be in range $[0; n - \text{shorten})$.

Returns

- **data** (*BitBuffer*) – Corrected codeword. The input codeword is returned unchanged in case of a decoding failure.
- **errors** (*int*) – Number of corrected symbols. -1 if decoding is not possible (too many errors).

class procitec.decoding.**BCHDecoder**(*n*, *t*, *prim_poly*, *, *shorten=0*)

Decoder for binary and primitive Bose-Chaudhuri-Hocquenghem (BCH) codes defined over $\mathrm{GF}(2^m)$.

Decoder for binary and primitive Bose-Chaudhuri-Hocquenghem codes defined over an extension Galois field of order 2^m ($\mathrm{GF}(2^m)$). Shortened codes are supported. Erasure positions, i.e. positions of bits which are likely erroneous, may be provided to aid the decoder. Systematic encoding is assumed.

The order m of the primitive polynomial for the construction of $\mathrm{GF}(2^m)$ defines the length n of the un-shortened "mother" code: $2^m - 1`(inbits).(Note : non - primitiveBCHcodeshavealength : math : ` < 2^m - 1)$.

The generator polynomial $g(x)$ for a binary, primitive and t-error correcting BCH code is defined as the lowest-degree polynomial with coefficients in $GF(2)$ which has

$$\alpha, \alpha^2, \ldots \alpha^{2t}$$

as its roots, i.e. $g(\alpha^i) = 0$ for $1 \leq i \leq 2t$. t is the error correction capability of the code and α is a primitive element of the field. The number of correctable errors is limited by $2t' + e \leq t$ where t' is the number of errors and e is the number of erasures. There is no closed-form formula to determine the number of information bits k in an BCH-Code. However, for many BCH-Codes $k = n - mt$ applies.

Parameters

- **n** (*int*) – Length of the un-shortened mother code. Must be $2^m - 1$ where m is the order of the primitive polynomial `prim_poly`
- **t** (*int*) – Error correction capability of the code
- **prim_poly** (*int*) – Primitive polynomial for the construction of the Galois field. This must be a bit mask where the least significant bit corresponds to the to coefficient of exponent 0. The position of the highest bit set (zero based counting) determines m.
- **shorten** (*int, optional*) – Number of bits by which the mother code is shortened (default 0).

Examples

__call__(*data*, *, *shorten=None*, *erasure_pos=[]*)

see *decode()*

decode(*data*, *, *shorten=None*, *erasure_pos=[]*)

Perform error correction

Parameters

- **data** (BitBuffer) – The codeword to be corrected. The length of `data` must be at least the size a codeword in bits, i.e. $\geq n - \mathrm{shorten}$. The codeword polynomial must be given in order of descending exponents, i.e. the lower $k' = k - \mathrm{shorten}$ bits are information bits (systematic encoding).
- **shorten** (*int, optional*) – If *None* (default), the shortening amount provided at initialization time is used. Otherwise, the provided value is used.
- **erasure_pos** (*list(list(int)), optional*) – If the list is empty (default) no erasures are considered when decoding. Otherwise the list defines the positions of erasures (symbols which are known to be erroneous).

  The following rules apply:

  – Erasure positions must be provided in terms of the un-shortened codeword.

- Duplicate erasure positions must not be provided.
- Erasure positions must be in range $[0; n - \text{shorten})$.

Returns

- **data** (*BitBuffer*) – Corrected codeword (payload+redundancy). The input codeword is returned unchanged in case of a decoding failure.
- **errors** (*int*) – Number of corrected bits. -1 if decoding is not possible (too many errors).

`class procitec.decoding.LDPCCode(**kwargs)`

A specification for systematic LDPC-Codes. This object is needed to specify the code for *LDPCDecoder* and *LDPCEncoder*.

1. **LDPCCode** (*, parity_check_matrix: Sequence[Sequence[int]])
2. **LDPCCode** (*, n: int, k:int, parity_check_matrix_indices: Sequence[Sequence[int]])
3. **LDPCCode** (*, blocksize: int, quasicyclic_prototype_matrix: Sequence[Sequence[int]])
4. **LDPCCode** (*, alist: str)
5. **LDPCCode** (* code_id: *LDPCCodeID*)

All parameters are keyword only.

Added in version 25.2.0.

Parameters

- `parity_check_matrix` (*Sequence[Sequence[int]]*) – The full 2-D parity check matrix of shape `(n-k) x n`. All elements have to be either 0 or 1.
- `n` (*int*) – Number of codebits.
- `k` (*int*) – Number of information bits. Has to be `< n`
- `parity_check_matrix_indices` (*Sequence[Sequence[int]]*) – Column indices per row with a 1 in parity check matrix.
- `blocksize` (*int*) – Size of circulant shifted eye-matrices. Used together with parameter quasicyclic_prototype_matrix.
- `quasicyclic_prototype_matrix` (*Sequence[Sequence[int]]*) – Prototype matrix of quasicyclic ldpc code. It is used to create a parity check matrix consisting only of circulant shifted eye matrices of shape `blocksize * blocksize`. The values of the prototype matrix elements describe the circulation right shift. A value of -1 indicates a zero submatrix.
- `alist` (*str*) – Special string format to describe a ldpc parity check matrix.
- `code_id` (LDPCCodeID) – Some internal predefined codes can be selected with this parameter.

Raises

`ValueError` – The arguments does not specify a valid systematic ldpc code.

Examples

Construction with full parity check matrix

```
>>> from procitec.decoding import LDPCCode
>>> code = LDPCCode(
        parity_check_matrix=[
            [1, 1, 0, 1, 0, 0],
            [0, 1, 1, 0, 1, 0],
            [1, 0, 0, 0, 1, 1],
            [0, 0, 1, 1, 0, 1],
        ]
    )
```

Construction with full parity check indices, same code as above.

```
>>> code = LDPCCode(
        n=6,
        k=2,
        parity_check_matrix_indices=[
            [0, 1, 3],
            [1, 2, 4],
            [0, 4, 5],
            [2, 3, 5],
        ],
    )
```

Construction of quasicyclic code (IEEE 802.11-2020 Annex F with n = 648 and rate = 5/6)

```
>>> code = LDPCCode(
        blocksize=27,
        quasicyclic_prototype_matrix=[
            [17, 13, 8, 21, 9, 3, 18, 12, 10, 0, 4, 15, 19, 2, 5, 10, 26, 19, 13,␣
↪13, 1, 0, -1, -1],
            [3, 12, 11, 14, 11, 25, 5, 18, 0, 9, 2, 26, 26, 10, 24, 7, 14, 20, 4,␣
↪2, -1, 0, 0, -1],
            [22, 16, 4, 3, 10, 21, 12, 5, 21, 14, 19, 5, -1, 8, 5, 18, 11, 5, 5,␣
↪15, 0, -1, 0, 0],
            [7, 7, 14, 14, 4, 16, 16, 24, 24, 10, 1, 7, 15, 6, 10, 26, 8, 18, 21,␣
↪14, 1, -1, -1, 0],
        ],
    )
```

The prototype matrix is an efficient way to define larger parity check matrices. The following example shows a (for LDPC inappropriate) prototype matrix with corresponding parity check matrix. The upper left 3x3 sub-matrix is an eye matrix circular right shifted by one.

```
>>> code = LDPCCode(
        blocksize=3,
        quasicyclic_prototype_matrix=[
            [1, 0, -1, -1],
            [2, -1, -1, 2],
            [-1, -1, 1, 0],
        ],
    )
>>> print(code.make_parity_check_matrix())
[
```

(continues on next page)

(continued from previous page)

```
  [0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
  [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
  [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
  [1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
  [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
  [0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0],
  [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1],
]
```

Construction of a simple code (7, 3) with an alist.

```
>>> alist = """7 3
              3 4
              1 1 1 2 2 2 3
              4 4 4
              1 0 0
              2 0 0
              3 0 0
              1 2 0
              1 3 0
              2 3 0
              1 2 3
              1 4 5 7
              2 4 6 7
              3 5 6 7
              """
>>> code = LDPCCode(alist=alist)
>>> print(code.make_parity_check_matrix())
[
  [1, 0, 0, 1, 1, 0, 1],
  [0, 1, 0, 1, 0, 1, 1],
  [0, 0, 1, 0, 1, 1, 1],
]
```

Construction of a predefined code.

```
>>> from procitec.decoding import LDPCCode, LDPCCodeID
>>> code = LDPCCode(code_id=LDPCCodeID.IEEE_802_11_648_5_6)
```

See *LDPCDecoder* and *LDPCEncoder* for using the the code object for decoding and encoding.

make_parity_check_matrix() → list[list[int]]

Reconstructs the full parity check matrix of shape (n-k) x n.

Returns

parity_check_matrix – The parity check matrix.

Return type

list[list[int]]

property k

Number of info bits

Type
int

property n

Number of code bits

Type
int

property rate

The code rate n/k

Type
float

class procitec.decoding.LDPCCodeID

An identifier for some internal predefined codes to be used in constructor of *LDPCCode*.

Members:

IEEE_802_11_648_1_2 : IEEE 802.11-2020 Annex F with n = 648 and rate = 1/2

IEEE_802_11_648_2_3 : IEEE 802.11-2020 Annex F with n = 648 and rate = 2/3

IEEE_802_11_648_3_4 : IEEE 802.11-2020 Annex F with n = 648 and rate = 3/4

IEEE_802_11_648_5_6 : IEEE 802.11-2020 Annex F with n = 648 and rate = 5/6

IEEE_802_11_1296_1_2 : IEEE 802.11-2020 Annex F with n = 1296 and rate = 1/2

IEEE_802_11_1296_2_3 : IEEE 802.11-2020 Annex F with n = 1296 and rate = 2/3

IEEE_802_11_1296_3_4 : IEEE 802.11-2020 Annex F with n = 1296 and rate = 3/4

IEEE_802_11_1296_5_6 : IEEE 802.11-2020 Annex F with n = 1296 and rate = 5/6

IEEE_802_11_1944_1_2 : IEEE 802.11-2020 Annex F with n = 1944 and rate = 1/2

IEEE_802_11_1944_2_3 : IEEE 802.11-2020 Annex F with n = 1944 and rate = 2/3

IEEE_802_11_1944_3_4 : IEEE 802.11-2020 Annex F with n = 1944 and rate = 3/4

IEEE_802_11_1944_5_6 : IEEE 802.11-2020 Annex F with n = 1944 and rate = 5/6

__str__()

Return str(self).

IEEE_802_11_1296_1_2 = <LDPCCodeID.IEEE_802_11_1296_1_2: 5>

IEEE_802_11_1296_2_3 = <LDPCCodeID.IEEE_802_11_1296_2_3: 6>

IEEE_802_11_1296_3_4 = <LDPCCodeID.IEEE_802_11_1296_3_4: 7>

IEEE_802_11_1296_5_6 = <LDPCCodeID.IEEE_802_11_1296_5_6: 8>

IEEE_802_11_1944_1_2 = <LDPCCodeID.IEEE_802_11_1944_1_2: 9>

IEEE_802_11_1944_2_3 = <LDPCCodeID.IEEE_802_11_1944_2_3: 10>

IEEE_802_11_1944_3_4 = <LDPCCodeID.IEEE_802_11_1944_3_4: 11>

IEEE_802_11_1944_5_6 = <LDPCCodeID.IEEE_802_11_1944_5_6: 12>

IEEE_802_11_648_1_2 = <LDPCCodeID.IEEE_802_11_648_1_2: 1>

IEEE_802_11_648_2_3 = <LDPCCodeID.IEEE_802_11_648_2_3: 2>

IEEE_802_11_648_3_4 = <LDPCCodeID.IEEE_802_11_648_3_4: 3>

IEEE_802_11_648_5_6 = <LDPCCodeID.IEEE_802_11_648_5_6: 4>

property name

property value

class procitec.decoding.LDPCDecodeResult

Result object of the LDPCDecoder.

__bool__()

Returns
True if all parity checks are fulfilled.

Return type
bool

__getitem__(*key: int*) → *BitBuffer* | int

Parameters
key (*int*) – 0 : Returns attribute data

1 : Returns attribute errors

Raises
IndexError – Parameter key is not 0 or 1.

Return type
Attribute data or errors.

property bad_parities

number of unsatisfied parity checks

Type
int

property corrections

number of changed bits compared to the input

Type
int

property data

The corrected codeword after decoding including corrected parity bits.

Type
BitBuffer

property errors

Number of corrected bits. -1 if decoding is not possible (not all parity checks are fulfilled)

Type
int

property iterations

number of executed iterations

Type
int

class procitec.decoding.**LDPCDecoder**(*code:* LDPCCode, *max_iterations: int*, *llr_magnitude: float | None = None*)

A decoder for arbitrary systematic low-density parity-check (LDPC) codes using iterative belief propagation decoding.

The decoder works generically for any codes and does not use customized algorithms for specific LDPC code types. Therefore, the performance and processing speed may be impaired for very long codes.

Added in version 25.2.0.

Parameters

- **code** (LDPCCode) – The ldpc code to be decoded.
- **max_iterations** (*int*) – If not all parity checks are fulfilled after a fixed count of iterations the decoding of the codeword is aborted.
- **llr_magnitude** (*float | None = None*) – The used algorithm works with llr (log-likelihood-ratio) values. For decoding hard bits (using the *decode()* method with a BitBuffer parameter) these bits have to casted to llr values. The given magnitude is used to transform a 0-Bit to `-llr_magnitude` and a 1-Bit to `+llr_magnitude`.

  If None is passed, the value is automatically calculated internally. The default is None

Examples

Correcting an erroneous codeword.

```
>>> from procitec.decoding import LDPCCode, LDPCDecoder
>>> from procitec.common.bitbuffer import BitBuffer
>>> code = LDPCCode(
        parity_check_matrix=[
            [1, 1, 0, 1, 0, 0],
            [0, 1, 1, 0, 1, 0],
            [1, 0, 0, 0, 1, 1],
            [0, 0, 1, 1, 0, 1],
        ]
    )
>>> decoder = LDPCDecoder(code, max_iterations=8)
>>> erroneous_codeword = BitBuffer.from_str("101011")
>>> result = decoder.decode(erroneous_codeword)
>>> print(result.data)
001011
>>> error_pattern = erroneous_codeword ^ result.data
>>> print(result.errors, error_pattern)
1 100000
```

It is possible to unpack the result object into the corrected codeword and number of errors.

```
>>> corrected, errors = result
```

`decode`(*data: BitBuffer | Sequence[float], **, llr_magnitude: float | None = None*) → *LDPCDecodeResult*

Tries to correct a codeword until all parity checks are fulfilled or the maximum number of iterations are reached.

Parameters

- **data** (*BitBuffer | Sequence[float]*) – The codeword to be decoded. If a value of type `BitBuffer` is passed, the bits are casted to llr values with magnitude `llr_magnitude`. Values of type `Sequence[float]` are considered to be llr values.
- **llr_magnitude** (*float | None = None*) – The value overrides the parameter given in the constructor, but is not saved for subsequent decode calls.

  Not allowed if parameter data is of type `Sequence[float]`.

Returns
The result object includes the corrected codeword, number of corrected errors and other information of the decoding.

Return type
LDPCDecodeResult

Raises
`ValueError` – Wrong length of the codeword.

`property llr_magnitude`

The used llr magnitude to cast hard bits to llr values.

Type
float

2.3.3. Burst Operations

`procitec.decoding.burstoperations.search_burst`(*data, *, max_offset=None, max_length=None, start_offset=0, consume_searched=True*)

Search for the next full burst and return its position and length

Parameters

- **data** (`BitBuffer` *or* `BitStream`) – Input where the search is performed
- **max_offset** (*int, optional*) – If `None` (default), the search range is not limited. However, the search will eventually stop depending on the type of the input `data`:
  - *BitBuffer*: The search range is limited by the buffer's size.
  - *BitStream*: The search is not aborted until a burst is found or until the stream ends (see *EndOfDataError*).

  A value other than `None` defines the maximum allowed offset for a burst start in the input (the returned offset for a found burst will not be greater than this value). This means that no more than `max_offset + 1` bits are searched; this amount of bits is consumed if no burst start is found.
- **max_length** (*int, optional*) – Limit the length of the burst to find to `max_length`, i.e. bursts longer than `max_length` are skipped. If `None` (default) a burst of any length is searched.
- **start_offset** (*int, optional*) – May only be used if `data` is a *BitBuffer*: Start search at given offset (default 0).

- **consume_searched** (*bool, optional*) – May only be used if data is a *BitStream*: If True (default), bits up to the found start of the burst are consumed. In case of an aborted search, all searched bits are consumed.

Returns

If no burst is found None is returned. Otherwise a tuple containing the offset (relative to start_offset) and the length of the found burst is returned.

> **Note**
>
> The offset points to the first bit of a burst. If consume_searched is False, consuming or reading offset bits from data (if it is a *BitStream*) will position the stream at that bit: a subsequent consume (read) will consume (return) the first bit of a burst. If consume_searched is True, the stream will be positioned at the first bit of a burst by *search_burst()*.
>
> If data is a *BitBuffer*, data[start_offset+offset] is the first bit of a burst.

Return type
tuple(int, int) or None

procitec.decoding.burstoperations.**search_burst_end**(*data*, ***, *max_offset=None*, *start_offset=0*, *consume_searched=True*)

Search for the next end of a burst and return its offset

See *search_burst()* for a description of the parameters.

Returns

Offset (relative to start_offset) of the found end of a burst or None if no end of a burst is found.

> **Note**
>
> The offset points to the last bit of a burst. If consume_searched is False, consuming or reading offset bits from data (if it is a *BitStream*) will *not* consume (return) that bit. Consume (read) offset+1 bits in order to include the last bit of a burst. The same logic also applies if consume_searched is set to True: Consume (read) an additional bit in order to skip (get) the last bit of a burst.
>
> If data is a *BitBuffer*, data[start_offset+offset] is the last bit of a burst.

Return type
int or None

procitec.decoding.burstoperations.**search_burst_start**(*data*, ***, *max_offset=None*, *start_offset=0*, *consume_searched=True*)

Search for the next start of a burst and return its offset

See *search_burst()* for a description of the parameters.

Returns

Offset (relative to start_offset) of the found start of a burst or None if no start of a burst is found.

> **Note**
>
> The offset points to the first bit of a burst. If `consume_searched` is `False`, consuming or reading `offset` bits from `data` (if it is a *BitStream*) will position the stream at that bit: a subsequent consume (read) will consume (return) the first bit of a burst. If `consume_searched` is `True`, the stream will be positioned at the first bit of a burst by *search_burst_start()*.
>
> If `data` is a *BitBuffer*, `data[start_offset+offset]` is the first bit of a burst.

Return type
int or None

`procitec.decoding.burstoperations.stream_until_burst_end`(*data:* BitStream, *, *max_offset=None*, *max_length=None*) → *BitStream* | None

Search for the next burst start and return a stream which ends at a following burst end or after `max_length` bits

First, all bits until a burst start are consumed. If the stream does not contain a burst start, the behavior depends on the value of `max_offset`:

- `max_offset` is `None`: The search for a burst start is not aborted until a burst start is found or until the stream ends (see *EndOfDataError*).
- `max_offset` is not `None`: If no burst start is found, `max_offset + 1` bits are consumed and `None` is returned.

As soon as a burst start is found, a stream is returned, otherwise `None`. The stream ends if one of the following occurs, regardless of the value of `max_length`:

- The underlying *BitStream* (argument `data`) ends.
- A burst end is found.

If `max_length` is not `None`, the stream additionally ends if `max_length` bits have been read. Note: This may occur before a burst end is found. This in turn means that an incomplete burst might be read.

Read operations from the returned stream will ignore any burst starts.

Parameters

- **data** (BitStream) – Input where the search is performed
- **max_offset** (*int, optional*) – Maximum offset at which a burst must start in the input. See description above for details.
- **max_length** (*int, optional*) – If not `None`, the value defines the maximum length of the returned stream. See description above for details.

Returns
See description above

Return type
BitStream or None

2.3.4. Pre-Processing

Pre-processing modifies incoming symbols before they are placed in the input buffer (*apc.data*).

This package contains a number of functions to create pre-processing steps to be passed as arguments to *apc.preprocessing()*.

Example

```
from procitec.decoding import preprocessing as pre
apc.preprocessing(
    pre.reverse_symbol_bits(),
    pre.nrz_decode(),
)
```

The order and number of steps is arbitrary. Each call to *apc.preprocessing()* overwrites the current set of pre-processing steps.

2.3.4.1. Bit-Level Pre-Processing

procitec.decoding.preprocessing.**nrz_decode**(*change_bit=1*)

Perform a None-Return-To-Zero (NRZ-I) decoding. By default (change_bit=1) changes between successive input bits result in a One-Bit.

Parameters

change_bit (*int, optional*) – Bit value to represent a change input bits.

- Non-return-to-zero Mark (NRZ-M): One-Bit on toggle
- Non-return-to-zero Space (NRZ-S): Zero-Bit on toggle

Returns

a preprocessing step

Return type

PreProcessor

procitec.decoding.preprocessing.**nrz_encode**(*change_bit=1*)

Perform a None-Return-To-Zero inverted (NRZ-I) encoding. By default (change_bit=1) incoming One-Bits cause an output value toggle and Zero-bits repeat the previous value.

Parameters

change_bit (*int, optional*) – Bit value that causes a level shift in the output.

- Non-return-to-zero Mark (NRZ-M): One-Bit will toggle output
- Non-return-to-zero Space (NRZ-S): Zero-Bit will toggle output

Returns

a preprocessing step

Return type

PreProcessor

procitec.decoding.preprocessing.**descramble**(*polynomial*)

Directs the entire input bit-stream through a descrambler by means of the shift-register equation:

$$\mathrm{OUT} = \mathrm{IN}(p_0 + p_1 z^{-1} + p_2 z^{-2} + \cdots + p_n z^{-n})$$

where n < 255.

In case the bit stream consists of multiple channels, then the descrambling is performed as if it were a single bit stream processing all channels from a single time slot before continuing with the next time slot.

Parameters
polynomial (*int*) – Describes to polynomial to be used. The LSB p_0 (int value 1) represents to current input bit. p_1 the previous one

Returns
a preprocessing step

Return type
PreProcessor

Example

The polynomial $1 + z^{-6} + z^{-7}$ means each bit is XOR combined with the values of those 6 and 7 before. The integer representation here is 0b11000001 or 0xC1.

procitec.decoding.preprocessing.**invert**()

Invert all incoming bits

Returns
a preprocessing step

Return type
PreProcessor

2.3.4.2. Symbol-Level Pre-Processing

procitec.decoding.preprocessing.**reverse_symbol_bits**()

Reverse bit order with in each symbol

procitec.decoding.preprocessing.**convert_symbols**(*table*)

Converts the input symbols with the provided table. Symbols are mapped using the table index as input.The bit rate remains unchanged.

Parameters
table (*list(int)*) – The table must have one entry for each incoming symbol (2**bits_per_symbol)

Returns
a preprocessing step

Return type
PreProcessor

Example

If used as shown below:

```
from procitec.decoding import preprocessing as pre
apc.preprocessing(
    pre.convert_symbols([1,0,2,3])
)
```

will convert the following sequence of symbols

3, 2, 1, 0, 0, 1, 2, 3

to

3, 2, 0, 1, 1, 0, 2, 3

procitec.decoding.preprocessing.**differential_encode**()

Converts the input symbol stream to a differentially-coded symbol stream. The main purpose of this command is to transform absolute-PSK- demodulated data streams to differentially-coded data streams. The bit rate remains unchanged.

Returns
a preprocessing step

Return type
PreProcessor

2.3.4.3. Utilities

class procitec.decoding.preprocessing.**PreProcessor**

Base class for pre-processing steps to be used in *apc.preprocessing()*.

This type can not be instantiated directly. It serves as a base class for all pre-processing steps (see above).

reset()

Set *enabled* to True and reset any internal state

Added in version 21.2.0.

property enabled

If False, skip this pre-processing step, i.e. go to the next one (if there is one)

Added in version 21.2.0.

Type
bool

2.3.5. Alphabets

Alphabets represents a mapping of bits (codewords) to characters and strings. The main benefit over raw dict objects is the ability to do fuzzy look-ups, allowing best-effort decoding of erroneous codewords. In addition, alphabets with multiple symbol layers and the state handling thereof is build-in.

class procitec.decoding.**Alphabet**(*table*, ***, *codeword_length=None*, *replacement_character='uFFFD'*, *msb_first=False*)

Create an Alphabet which represents a mapping of bits (codewords) to characters and strings

Both fixed- and variable-length codes are supported. A replacement character can be defined which is returned whenever a codeword is not defined by the alphabet. Support for codes which switch between different tables (level shifts) is available as well.

Parameters

- **table** (*dict((int or tuple(int, int)), list(str))*) – The key in the dictionary is a codeword (a sequence of bits). For a fixed-length code the key must be an integer representing the codeword. For a variable-length code the key must be a tuple where the first value is the codeword and the second value its length in bits.

  The value corresponding to a key in the dictionary must be a list containing strings (characters) which are output depending on the current level shift. The first value in the list belongs to the first level of the alphabet, the second value to the second level and so on. Switching between levels is performed by using the special value `procitec.decoding.LEVEL[i]` where `i` is the number of the level to switch to. Up to ten levels are possible, with `LEVEL[0]` being the first one.

- **codeword_length** (*int, optional*) – Defines the length of a codeword in bits. This parameter must be present and greater than zero for fixed-length codes. The value is ignored for variable-length codes.

- **replacement_character** (*str, optional*) – This string (character, by default "uFFFD" [Unicode U+FFFD]) is returned by search functions whenever a codeword can not be found in the alphabet (see *decode()*, *decode_alphabet()*).

- **msb_first** (*bool, optional*) – By default (`False`) the keys of the dictionary `table`, i.e. the codewords, are used as is. If `True`, the codewords are reversed.

See also

decode_alphabet(), *search_alphabet()*

Examples

Assume the following fixed-length mapping of bits to characters, with two different levels:

codeword	character	
bits	first level	second level
0b00000	a	1
0b01011	b	2
0b10101	c	3
0b11110	level switch	level switch

The corresponding *Alphabet* is initialized the following way:

```
>>> from procitec.decoding import Alphabet
>>> from procitec.decoding import LEVEL as L
>>> a = Alphabet(
...     {
...         0b00000: ["a", "1"],
...         0b01011: ["b", "2"],
...         0b10101: ["c", "3"],
...         0b11110: [L[1], L[0]]
...     },
...     codeword_length=5)
...
>>>
```

Note that the second argument (`codeword_length`) must be present for fixed-length codes. *decode()* can now be used the retrieve the character corresponding to a codeword:

```
>>> a.decode(0)
('a', 0)
```

By default the first level is used and only an exactly matching codeword (a Hamming distance of zero) is returned as a result. The second returned value is the Hamming distance between the passed codeword and the codeword found in the alphabet. If a codeword is not found, *replacement_character* is returned and the second value of the tuple is None:

```
>>> a.decode(0b0001)
('
uFFFD', None)
```

You may decode using another level and permit bit errors:

```
>>> a.decode(0b11111, level=1, max_errors=1)
(0, 1)
```

An integer as the returned value indicates a level shift to that level.

Assume the following variable-length mapping of bits to characters, with one level of characters:

codeword	character	
bits	code length	characters
0b11	2	1
0b1100	4	E
0b11011	5	T
0b11010	5	A

Variable-length codes are defined and used in a similar way as fixed-length codes, but `tuple` instead of `int` is used:

```
>>> from procitec.decoding import Alphabet
>>> a = Alphabet(
...     {
...         (0b11, 2):          ["1"],
...         (0b1100, 4):        ["E"],
...         (0b11011, 5):       ["T"],
...         (0b11010, 5):       ["A"],
...     }, msb_first=False)
...
>>>
```

decode() can now be used to retrieve the character corresponding to a codeword. Please note that you musst pass a tuple containing the codeword and its length.

```
>>> a.decode((0b11010, 5))
('A', 0)
```

decode(*codeword*, *, *level=0*, *max_errors=0*)

Decode a codeword, i.e. retrieve the corresponding string

Parameters

- **codeword** (*int or tuple(int, int)*) – For fixed-length codes provide the codeword to be decoded. For variable-length codes provide a tuple containing the codeword to be decoded and its length in bits.
- **level** (*int, optional*) – Use the level `level` for decoding. By default the first level is used.
- **max_errors** (*int, optional*) – Allow up to `max_errors` bit errors (default 0) when decoding the codeword. This means that the Hamming distance between `codeword` and the codeword actually used for decoding will be not greater than `max_errors`. This option may only be used with fixed-length codes.

Returns

- **string** (*str or int*) – Either a *str* corresponding to `codeword` at level `level` or an *int* denoting the level to switch to, if `codeword` represents a level shift at level `level`. If `codeword` is not found in the alphabet with given maximum number of errors, *replacement_character* is returned.
- **bit_errors** (*int or None*) – The number of bit errors, i.e. the Hamming distance between `codeword` and the codeword actually used for decoding. If `codeword` is not found in the alphabet the value is `None`. Note, that in case of a variable-length code, this value is either 0 if `codeword` is found in the alphabet and `None` otherwise, since in maximum number of errors is always 0 for variable-length codes

property codeword_length

codeword length for fixed-length codes; maximum codeword length for variable-length codes

property levels

number of levels in the alphabet

property replacement_character

The returned string if a codeword can not be found the alphabet.

property variable_length_code

`True` if the code is a variable-length code, `False` for fixed-length codes

class procitec.decoding.AlphabetDecoder(*alphabet*, *, *initial_level=0*, *force_level=False*)

A decoder to map a stream of bits to characters as defined by an *Alphabet*

Unlike *decode_alphabet()* an *AlphabetDecoder* retains a state between calls to *decode()*. This allows to pass partial data; decoded codewords will be returned as soon as possible when new data is provided using *decode()*.

Parameters

- **alphabet** (Alphabet) – Alphabet to be used for decoding
- **initial_level** (*int, optional*) – The initial level to assume at the start of decoding. The level to be used for the next *decode()* can be set by modifying the *level* property.

- **force_level** (*bool, optional*) – Level shifting codewords encountered during the decoding process are respected by default. With `force_level = True` any level shifting codewords are ignored and an empty string is returned instead. All other codewords are decoded using the level *level*.

  This settings can by modified changing the *force_level* property.

See also

decode_alphabet()
Stateless decoding of an *Alphabet*

decode(*data*, *max_errors=0*)

Decode a stream of bits by using the underlying *Alphabet* to translate bit sequences to characters

The data does not need to be complete, i.e the last codeword in `data` may have missing bits. As soon as new data is provided the partial codeword of the previous call to *decode()* will be returned as well.

Parameters

- **data** (BitBuffer) – The data to decode. The last codeword does not need to be complete, i.e. the length of `data` does not need to be a multiple of the codeword length (in case of a fixed-length code).
- **max_errors** (*int, optional*) – Allow up to `max_errors` bit errors (default 0) in each codeword: If a bit sequence from the input can not be found in `alphabet` a codeword with a Hamming distance of up to `max_errors` is searched. The codeword with the smallest Hamming is taken. If there are several candidates the first match is returned, i.e. the output depends on the order of definition of codewords in `alphabet`.

  Alphabet.replacement_character is used as output if the codeword cannot be found even after considering `max_errors`.

  Only applicable for an `alphabet` with a fixed-length code.

Returns
The decoding result

Return type
DecodeAlphabetResult

reset()

Resets *level* to the initial value and clears the internal buffer of possible remaining bits

property force_level

Current value of the setting. A change takes effect at the next *decode()*. When `False`, level shifting codewords are respected. Otherwise any level shifting codewords are ignored and an empty string is returned instead. All other codewords are decoded using the level *level*.

Type
bool

property level

Current level of decoder. May be set to define which level to use at next *decode()*

Type
int

class procitec.decoding.DecodeAlphabetResult

Result returned by *decode_alphabet()* and *AlphabetDecoder.decode()*

Example

```
>>> from procitec.decoding import Alphabet, decode_alphabet
>>> from procitec.common.bitbuffer import BitBuffer
>>> alphabet = Alphabet({ 0x00:["A"] }, codeword_length = 4)
>>> data = BitBuffer.from_int(0b1111_0111_0011_0001_0000, 5*4)
>>> result = decode_alphabet(data, alphabet, max_errors = 2)
'AAA
uFFFD
>>> result.message # Only the first three codewords where accepted.
uFFFD'
>>> result.bit_errors # This first three codewords contain 0, 1
>>>                   # and 2 bit errors respectively.
>>>                   # Note, that no bit errors of not accepted
>>>                   # codewords contribute to this value.
3
>>> result.codeword_errors # Two codewords aren't accepted.
2
```

__bool__()

True if *codeword_errors* is zero, False otherwise

__str__()

Returns *message*

property bit_errors

Total number of bit errors accepted in the decoding process due to the given max_errors. If any codeword error occurred then codeword_length is returned. Always 0 in case of an *Alphabet* with a variable-length code.

Changed in version 23.1.0: In older releases bit_errors had the value 0 if a codeword error did occurr.

Type
int

property codeword_errors

Number of codeword errors, i.e. how many times a bit sequence from the input is not found in the used *Alphabet*. This is the number of replacement characters of the used *Alphabet* in the output *message*. Note that in the case of several possible codewords (due to allowed max_errors > 0) a value of 0 is returned.

Type
int

property message

Decoded output message

Type
str

procitec.decoding.**decode_alphabet**(*data*, ***, *alphabet*, *initial_level=0*, *force_level=False*, *max_errors=0*)

Decode a sequence of bits by using an *Alphabet* to translate bit sequences to characters

Parameters

- **data** (BitBuffer) – Sequence of bits to be decoded. With a fixed-length code in alphabet the length of data must be a multiple of the codeword length.
- **alphabet** (Alphabet) – Mapping which defines how specific bit sequences are translated to characters
- **initial_level** (*int, optional*) – Initial level inside alphabet which is used for decoding. Only applicable for an alphabet with more than one level. By default the first level is used.
- **force_level** (*bool, optional*) – Level shifting codewords encountered during the decoding process are respected by default. With force_level = True any level shifting codewords are ignored and an empty string is returned instead. All other codewords are decoded using the level initial_level.
- **max_errors** (*int, optional*) – Allow up to max_errors bit errors (default 0) in each codeword: If a bit sequence from the input can not be found in alphabet a codeword with a Hamming distance of up to max_errors is searched. The first match is returned, i.e. the output depends on the order of definition of codewords in alphabet.

  Alphabet.replacement_character is used as output if the codeword cannot be found even after considering max_errors.

  Only applicable for an alphabet with a fixed-length code.

Returns
The decoding result

Return type
DecodeAlphabetResult

See also

AlphabetDecoder
Supports passing of partial data by retaining an internal state

procitec.decoding.**LEVEL: list**

This is a list of predefined marker objects used in class *Alphabet* to denote level changes. For usage details and examples see *Alphabet*.

procitec.decoding.alphabets.**get_ita2_alphabet**(*alphabet_name: str*) → *Alphabet*

Returns an ITA2 alphabet. 32 codewords: 0...31 (codeword_length=5)

Added in version 21.2.0.

Parameters
alphabet_name (*str*) – Specify the alphabet. Supported are the following:

- "ITA2"
  - (Baudot-Murray code
  - 2 levels: letter shift, figure shift

- "ITA2_ARABIC"
  - Arabic characters (Baghdad 80, ATU-80)
  - 2 levels: letter shift, figure shift
- "ITA2_CYRILLIC"
  - Third Shift Cyrillic (MTK-2)
  - 3 levels: letter shift, figure shift
- "ITA2_HEBREW"
  - Hebrew characters
  - 2 levels: letter shift, figure shift
- "custom" alphabet
  - searches for file "custom.yaml" in user and install folder
  - YAML file with dict (codeword length must fit)
  - filename is arbitrary but must match name of alphabet

Returns
result

Return type
Alphabet

`procitec.decoding.alphabets.get_ita5_alphabet`(*alphabet_name: str*) → *Alphabet*

Returns an ITA5 alphabet. 128 codewords: 0x00...0x7f (codeword_length=7)

Added in version 21.2.0.

Parameters
alphabet_name (*str*) – Specify the alphabet. Supported are the following:

- "ITA5"
  - ITA-5 (ASCII, ISO-646-US)
- "ITA5_GRAPH"
  - graphical representation of control characters using unicode `https://en.wikipedia.org/wiki/IS0_2047`
- "custom" alphabet
  - searches for file "custom.dict" in user and install folder
  - text file with Python dictionary (see ita2lower.dict as example)
  - filename is arbitrary but must match name of alphabet

Returns
result

Return type
Alphabet

`procitec.decoding.alphabets.ASCII_8_BIT = Alphabet(...)`

8 bit ASCII

- 256 codewords: 0x00...0xff (codeword_length=8)

It is recommended to use instead `to_bytes(..).decode(..)`

see `https://docs.python.org/3/library/codecs.html#standard-encodings`

`procitec.decoding.alphabets.ASCII_8_reduced = Alphabet(...)`

8 bit ASCII reduced

- 95 codewords: 0x20...0x7e (codeword_length=8)
- no control characters and characters above 0x7e are omitted

`procitec.decoding.alphabets.CCIR476 = Alphabet(...)`

CCIR 476

CCIR 476 is a character encoding used in radio data protocols such as SITOR, AMTOR and Navtex. It is a recasting of the ITA2 character encoding, known as Baudot code, from a five-bit code to a seven-bit code. In each character, exactly four of the seven bits are mark bits, and the other three are space bits. This allows for the detection of single-bit errors.

Added in version 22.1.0.

`procitec.decoding.alphabets.ITA2 = Alphabet(...)`

ITA-2 (Baudot-Murray code)

- 32 codewords: 0...31 (codeword_length=5)
- 2 levels: letter shift, figure shift

If a different replacement character or bit ordering is desired then create an alphabet directly from the _ITA2_TAB dictionary

```
>>> from procitec.decoding import Alphabet
>>> from procitec.decoding.alphabets import _ITA2_TAB
>>> alphabet=Alphabet(_ITA2_TAB, codeword_length=5, msb_first=True,␣
↪replacement_character = '')
```

Added in version 21.2.0.

`procitec.decoding.alphabets.ITA2P_EVEN = Alphabet(...)`

ITA2P with even parity (ARQ-1A)

- 5 bit ITA2 alphabet extended to 7 bits with 3 additional characters
- leading '0' for all original ITA-2 characters and '1' for additional characters
- trailing parity bit to have overall even parity

Added in version 22.2.0.

`procitec.decoding.alphabets.ITA2P_ODD = Alphabet(...)`

ITA2P with odd parity (ARQ-1A)

- 5 bit ITA2 alphabet extended to 7 bits with 3 additional characters
- leading '0' for all original ITA-2 characters and '1' for additional characters
- trailing parity bit to have overall odd parity

Added in version 22.2.0.

`procitec.decoding.alphabets.ITA3 = Alphabet(...)`

ITA-3 (CCIR342-2)

- 32 codewords plus 3 codes for IDLE and RQ (codeword_length=7)
- 2 levels: letter shift, figure shift

Added in version 22.1.0.

`procitec.decoding.alphabets.ITA5 = Alphabet(...)`

ITA-5 (ASCII, ISO-646-US)

- 128 codewords: 0x00...0x7f (codeword_length=7)

Added in version 21.2.0.

2.3.6. Utilities

`procitec.decoding.extract_pattern(`*data*, *pattern*`)`

Extracts a set of single bit positions defined by a bit pattern and returns a new `BitBuffer` containing only the extracted bits

For each bit in `pattern`, if it is 1, the bit at the same position in `data` is copied into the returned `BitBuffer`. Thereby, the size of the returned `BitBuffer` will match the hamming weigth of `pattern`. The size of `pattern` has to be equal or less than the size of `data`.

Example

data:	[	b_0,	b_1,	b_2,	b_3,	b_4,	b_5,	b_6,	b_7]
pattern:	[	0,	0,	1,	0,	1,	1		]
returns:	[			b_2,		b_4,	b_5		]

Parameters

- **data** (`BitBuffer`) – `BitBuffer` from which bits are extracted
- **pattern** (`BitBuffer`) – length of a symbol in bits

Returns
`BitBuffer` containing only the extracted bits

Return type
BitBuffer

`procitec.decoding.extract_interleaved(`*data*, ***, *bit_dist*, *char_dist*, *char_size*, *char_ix*, *block_len*`)`

Extract and deinterleave bits of one symbol from interleaved bits.

Parameters

- **data** (`BitBuffer`) – interleaved bits
- **bit_dist** (*int*) – distance between two successive bits of the same symbol (measured in bits)
- **char_dist** (*int*) – distance between the first bit of two successive symbols (measured in bits)
- **char_size** (*int*) – length of a symbol in bits
- **char_ix** (*int*) – index of the symbol to be extracted (zero based counting)

- **block_len** (*int*) – Length of interleaving block. All indices used to access bits in data are taken modulo this value.

Returns
deinterleaved symbol of size char_size

Return type
BitBuffer

Example

Assume the following 16 bits contain 4 interleaved symbols of size 4. The first symbol starts at bit_position = 0, the second at bit_position = 3, then 6 and 9.

```
>>> interleaved = BitBuffer.from_str("1001000110011110")  # size = 16
>>>                       # symbol 0:  0   1   2   3       => 1011
>>>                       # symbol 1:    0   1   2   3     => 1110
>>>                       # symbol 2:    3   0   1   2     => 0010
>>>                       # symbol 3:  2   3   0   1       => 0100
```

To extract the third symbol (zero based counting, therefore char_ix=2) we use the following command:

```
>>> deinterleaved_symbol2 = ddl.extract_interleaved(interleaved, bit_dist=4, char_
↪dist=3, char_size=4, char_ix=2, block_len=interleaved.size)
>>> str(deinterleaved_symbol2)
'0010'
```

This function can also be used as a matrix deinterleaver. char_size would then correspond to the number of rows, bit_distance to the number of columns. And char_distance should be set to 1.

Notes

See also *RandomInterleaver* and *RegularInterleaver*.

procitec.decoding.**extract_interleaved_symbols**(*data*, ***, *symb_len*, *symb_dist*, *group_dist*, *group_size*, *group_ix*, *block_len*)

Deinterleave modulation symbols (sequences of bits) of an interleaved symbol stream

Parameters

- **data** (BitBuffer) – interleaved modulation symbols (as a stream of bits)
- **symb_len** (*int*) – length of a symbol in bits
- **symb_dist** (*int*) – distance between two successive symbols (measured in symbols)
- **group_dist** (*int*) – distance between two successive symbol groups (measured in symbols)
- **group_size** (*int*) – length of a symbol group in symbols
- **group_ix** (*int*) – index of the symbol group to be extracted (zero based counting)
- **block_len** (*int*) – Length of interleaving block in bits. All indices used to access bits in data are taken modulo this value.

Returns
deinterleaved symbol group

Return type
BitBuffer

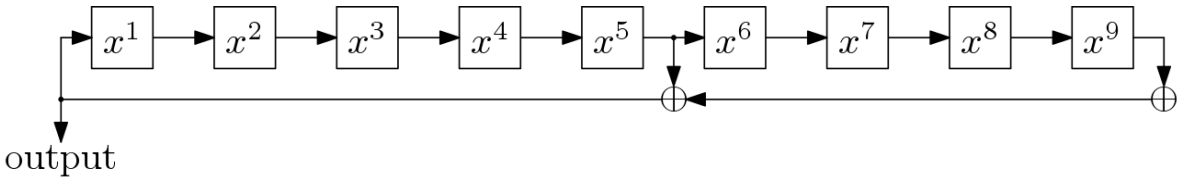
`procitec.decoding.generate_lfsr_sequence(poly, initial_state, length)`

Generate a pseudo-random sequence using a linear feedback shift register (LFSR)

Linear feedback shift registers are defined by polynomials of order n with binary coefficients c_i:

$$p(x) = 1 + \sum_{i=1}^{n} c_i x^i$$

An example for the operation of the LFSR with the polynomial $1 + x^5 + x^9$ is given below. This is also the mode of operation of this function.



Parameters

- **poly** (int) – Polynomial of the LFSR as a bit mask where the least significant bit corresponds to the coefficient of exponent 0. The coefficient for exponent 0 must be present. The position of the highest bit set (zero based counting) corresponds to n.
- **initial_state** (int) – Initial state of the shift register: The first output is computed using `initial_state` as the state of the shift register.

  Must be > 0 and $< 2^n$ The least significant bit defines the most recent bit in the shift register (x^1 in the example above), the most significant bit the oldest bit (x^9 in the example above).
- **length** (int) – determines the length of the generated sequence

Returns
generated bit sequence

Return type
BitBuffer

Example

Generate a sequence of length 32 using the polynomial $x^9 + x^5 + 1$ and all ones initial state

```
>>> from procitec.decoding import generate_lfsr_sequence
>>> seq = generate_lfsr_sequence(0x221, 0b111111111, 32)
>>> print(seq)
00000111101111100010111001100100
```

procitec.decoding.**de_stuff**(*data*, ***, *length*, *chain_limit*, *chain_polarity*)

Reverses a bit stuffing, i.e. bits will be removed which are inserted on transmission side to interrupt longer 1- or 0-sequences

Parameters

- **data** (BitBuffer) – Input data to be de-stuffed
- **length** (*int*) – Length of bit field to de-stuff in data starting with LSB
- **chain_limit** (*int*) – Length of a bit sequence, which was meant to be interrupted by bit stuffing
- **chain_polarity** (*int*) – 1 or 0 = Polarity of interrupted bit sequence = inverse polarity of stuff bit

Returns

- **result** (*BitBuffer*) – De-stuffed bit field
- **number_stuffs** (*int*) – Number of de-stuffed bits

Examples

```
>>> from procitec.decoding import de_stuff
>>> from procitec.common.bitbuffer import BitBuffer
>>> stuffed = BitBuffer.from_int(0b10100101111101101,17)
>>> de_stuffed, number = de_stuff(stuffed, length=17, chain_limit=5, chain_
↪polarity=1)
>>> str(stuffed)
'10110111110100101'
>>> str(de_stuffed)
'10110111111001010'
>>> number
1
```

procitec.decoding.**golay_parity_matrix**(*mat_spec: str*) → List[List[int]]

Returns the transposed P (P^) part of a Golay parity check matrix (H) according to selected input. P as in H = [P^ | T^]. For reasons of symmetry P = P^ applies.

Added in version 21.2.0.

Parameters

mat_spec (*str*) – Parity matrix specifier string. Supported are the following:

- "AE3"
- "C75"

The names are based on the polynoms used to construct them.

Returns

P-Part of Parity Check Matrix as given by input.

Return type

list(list(int))

procitec.decoding.**hamming_generator_matrix**(*m: int*) → List[List[int]]

Returns a generator matrix for systematic hamming code $(2^m-1, 2^m-m-1, 3)$ without the systematic part. The resulting matrix is suitable to use it directly with *BlockDecoder*.

Added in version 22.1.0.

Parameters
m (*int*) – Number of parity bits of hamming code. Has to be >= 3.

Returns
Generator matrix of hamming code without systematic code part.

Return type
list(list(int))

procitec.decoding.**hamming_distance**(*data1*, *data2*)

Determine the Hamming distance (number of different bits) of two buffers. An error is raised if the two buffers differs in size.

Parameters

- **data1** (BitBuffer) – first data buffer
- **data2** (BitBuffer) – second data buffer

Returns
Hamming distance of data1 and data2

Return type
int

procitec.decoding.**hamming_weight**(*data*)

Determine the Hamming weight (number of ones) of a buffer

Parameters
data (BitBuffer) – sequence of bits for which the Hamming weight is to be determined

Returns
Hamming weight of data

Return type
int

procitec.decoding.**time_skip**(*stream*, *desired_difference*)

Skips the specified amount of time in the given BitStream stream

Parameters

- **stream** (BitStream) – BitStream, on which time shall be skipped
- **desired_difference** (*float*) – Desired time difference in seconds

Returns
The actual skipped amount of time in seconds

Return type
float

procitec.decoding.**bit_metric_psk**(*symbols*, *table*)

Calculates soft bits for PSK constellations.

The soft value of a bit is based on the distance from the symbol of that bit to the next decision boundary. See also *bit_metric_distance()*.

Parameters

- **symbols** (*array_like*) – Soft symbol values in magnitude and phase format (MAG_PHASE)
- **table** (*list(tuple(int, float, float)) or list(tuple(int, int, int))*) – Symbol table. The tuple contains the bit pattern and the soft value as a combination of magnitude and phase, in that order. The length of the list has to be a power of 2 and the maximum number of symbols is 16.

Returns

Hard decided bits and corresponding soft bits in *BitBuffer.soft_bits*.

Return type

BitBuffer

Example

See *bit_metric_distance()*

procitec.decoding.**bit_metric_distance**(*symbols*, *table*, *format*)

Calculates soft bits for arbitrary symbol constellations.

For each symbol and bit within that symbol the two nearest reference symbols with a corresponding one and zero at the matching bit position are searched for in the given symbol table. The soft value is based on a linear interpolation between the distances of the received symbol to the 2 reference symbols.

The resolution of soft bits is 8 bit. The most reliable hard decision for a 0 is represented by +127, the most reliable decision for a 1 by -127; smaller absolute values indicate a less reliable decision.

See also *set_soft_symbols()*

Parameters

- **symbols** (*array_like*) – Soft symbol values. e.g. data from a SymbolStream
- **table** (*list(tuple(int, float, float)) or list(tuple(int, int, int))*) – Symbol table. The tuple contains the bit pattern and the soft value as a combination of 2 values depending on the symbol format. The length of the list has to be a power of 2 and the maximum number of symbols is 256. The float representation (`tuple(int, float, float)`) is a helper for simplified or more understandable input and internally translated to the `tuple(int, int, int)` format. See examples below.
- **format** (SampleFormat) – `symbols` and `table` are expected in this format. Only MAG_PHASE or FSK are supported.

Returns

Hard decided bits and corresponding soft bits in *BitBuffer.soft_bits*.

Return type

BitBuffer

Examples

Given are 4 received soft symbols read from a SymbolStream

```
>>> soft_symbols = apc.symbols.read(size=4)
>>> soft_symbols
[[31632, 160], [35824, 16327], [31152, 65376], [31648, 32416]]
```

Or in float represantation (16 bit are linear mapped to magnitude range [0, 2] and phase range [0, 2*pi].

```
>>> import numpy as np
>>> soft_symbols_flt = np.array(soft_symbols) / [ 2**15, 2**16 /(2*np.pi) ]
>>> soft_symbols_flt
[[ 0.97  0.02]
 [ 1.09  1.57]
 [ 0.95  6.27]
 [ 0.97  3.11]]
```

The corresponding QPSK symbol table could be

```
>>> qpsk_tab = [[0b00, 32768,     0],
                [0b01, 32768, 16384],
                [0b10, 32768, 32768],
                [0b11, 32768, 49152]]
>>> qpsk_tab_flt = [[0b00, 1.0,   0*np.pi],
                    [0b01, 1.0, 1/2*np.pi],
                    [0b10, 1.0,   1*np.pi],
                    [0b11, 1.0, 3/2*np.pi]]
```

The bits for the soft symbols and table can now be calculated using the `bit_metric_distance` command.

```
>>> from procitec.decoding import bit_metric_distance, MAG_PHASE
>>> bits = bit_metric_distance(np.array(soft_symbols, dtype=np.uint16), qpsk_tab,␣
↪format=MAG_PHASE)
>>> print(bits)
00100001
>>> soft_bits = np.array(bits.soft_bits, dtype=np.int8)
>>> print(soft_bits)
[ 120  120 -112  112  118  118  118 -118]
```

The following commands are here equivalent:

```
>>> bit_metric_distance(np.array(soft_symbols, dtype=np.uint16), qpsk_tab,␣
↪format=MAG_PHASE)
>>> bit_metric_distance(np.array(soft_symbols, dtype=np.uint16), qpsk_tab_flt,␣
↪format=MAG_PHASE)
>>> bit_metric_distance(np.array(soft_symbols_flt, dtype=np.float32), qpsk_tab,␣
↪format=MAG_PHASE)
>>> bit_metric_distance(np.array(soft_symbols_flt, dtype=np.float32), qpsk_tab_flt,
↪ format=MAG_PHASE)
```

class procitec.decoding.**RandomInterleaver**(*pattern*)

Creates an interleaver used for interleaving or deinterleaving bits or other array-like types.

Added in version 21.2.0.

Parameters

pattern (*list(int)*) – The interleaving pattern used to interleave the input. The largest value specifies the size of the interleaved output.

deinterleave(*data*, *with_soft=False*)

Deinterleaves the data accordingly to the set pattern. This reverses the process of the interleave function.

For the set pattern p the output is calculated as follows:

```
out[p[0]] = input[0]

out[p[1]] = input[1]

out[p[2]] = input[2]

...
```

Parameters

- **data** (*BitBuffer or sequence or 1D-array*) – Input data to be deinterleaved.
- **data_out** (*BitBuffer*) – Only allowed if data is also a BitBuffer. Instead of returning a new BitBuffer, the result of deinterleaving is written into this buffer if this parameter is given. Overlapping range of data and data_out is not allowed.
- **with_soft** (*bool*) – If the input is a BitBuffer, the soft-bit information is also deinterleaved if True. Not allowed if data is not a BitBuffer.

Example

```
>>> interleaver = ddl.RandomInterleaver([0, 4, 5, 2, 1, 3])
>>> deinterleaved = interleaver.deinterleave(BitBuffer.from_str("110101"))
>>> print(deinterleaved)
    101110
```

To clarify the interleaving process a simple list is deinterleaved as follows:

```
>>> interleaver = ddl.RandomInterleaver([2, 1, 3, 0])
>>> deinterleaved = interleaver.deinterleave(['a', 'b', 'c', 'd',])
>>> print(deinterleaved)
['d', 'b', 'a', 'c']
```

interleave(*data*, *data_out=None*, *with_soft=False*)

Interleaves the data accordingly to the set pattern. This function can also be used for deinterleaving.

For the set pattern p the output is calculated as follows:

```
out[0] = input[p[0]]

out[1] = input[p[1]]

out[2] = input[p[2]]

...
```

Parameters

- **data** (*BitBuffer or sequence or 1D-array*) – Input data to be interleaved.
- **data_out** (*BitBuffer*) – Only allowed if data is also a BitBuffer. Instead of returning a new BitBuffer, the result of interleaving is written into this buffer if this parameter is given. Overlapping range of data and data_out is not allowed.
- **with_soft** (*bool*) – If the input is a BitBuffer, the soft-bit information is also interleaved if True. Not allowed if data is not a BitBuffer. Default is False.

Return type

Interleaved data or data_out if data_out is given.

Example

```
>>> interleaver = ddl.RandomInterleaver([0, 4, 5, 2, 1, 3])
>>> interleaved = interleaver.interleave(BitBuffer.from_str("101110"))
>>> print(interleaved)
    110101
```

Several identical bit positions are allowed.

```
>>> interleaver = ddl.RandomInterleaver([0, 4, 4, 5, 5, 3, 4, 0])
>>> interleaved = interleaver.interleave(BitBuffer.from_str("101110"))
>>> print(interleaved)
    11100111
```

To clarify the interleaving process a simple list is interleaved as follows:

```
>>> interleaver = ddl.RandomInterleaver([2, 1, 3, 0])
>>> interleaved = interleaver.interleave(['a', 'b', 'c', 'd',])
>>> print(interleaved)
['c', 'b', 'd', 'a']
```

property deinterleaved_size

the set interleaving pattern

property interleaved_size

the number of interleaved bits

property pattern

the set interleaving pattern

class procitec.decoding.**RegularInterleaver**(*size*, *offset*, *start=0*, *count=None*)

Creates an interleaver used for interleaving or deinterleaving bits or other array-like types in a regular sequence.

Added in version 21.2.0.

Parameters

- **size** (*int*) – Block size of the interleaver.
- **offset** (*int*) – Offset between two following bits or elements.
- **start** (*int*) – Start index of first bit or element to interleave. The default is 0.

- **count** (*int*) – Number of elements to interleave. If None the given size is used for count. The default is None.

deinterleave(*data*, *with_soft=False*)

Deinterleaves the data This reverses the process of the interleave function.

For the set pattern p the output is calculated as follows:

```
out[(start + offset * 0 ) % size] = input[0]
out[(start + offset * 1 ) % size] = input[1]
out[(start + offset * 2 ) % size] = input[2]
...
out[(start + offset * (count - 1) ) % size] = input[count - 1]
```

Parameters

- **data** (*BitBuffer or sequence or 1D-array*) – Input data to be deinterleaved.
- **data_out** (*BitBuffer*) – Only allowed if data is also a BitBuffer. Instead of returning a new BitBuffer, the result of deinterleaving is written into this buffer if this parameter is given. Overlapping range of data and data_out is not allowed.
- **with_soft** (*bool*) – If the input is a BitBuffer, the soft-bit information is also deinterleaved if True. Not allowed if data is not a BitBuffer.

Example

To clarify the interleaving process a simple list is deinterleaved as follows:

```
>>> interleaver = ddl.RegularInterleaver(size=5, offset=2, start=1)
>>> deinterleaved = interleaver.deinterleave(['a', 'b', 'c', 'd', 'e'])
>>> print(deinterleaved)
['c', 'a', 'd', 'b', 'e']
```

interleave(*data*, *data_out=None*, *with_soft=False*)

Interleaves the data. This function can also be used for deinterleaving.

For the set pattern p the output is calculated as follows:

```
out[0] = input[(start + offset * 0 ) % size]
out[1] = input[(start + offset * 1 ) % size]
out[2] = input[(start + offset * 2 ) % size]
...
out[count - 1] = input[(start + offset * (count - 1) ) % size]
```

Parameters

- **data** (*BitBuffer or sequence or 1D-array*) – Input data to be interleaved.
- **data_out** (*BitBuffer*) – Only allowed if data is also a BitBuffer. Instead of returning a new BitBuffer, the result of interleaving is written into this buffer if this parameter is given. Overlapping range of data and data_out is not allowed.
- **with_soft** (*bool*) – If the input is a BitBuffer, the soft-bit information is also interleaved if True. Not allowed if data is not a BitBuffer. Default is False.

Return type
Interleaved data. data_out if data_out is given.

Example

```
>>> interleaver = ddl.RegularInterleaver(size=5, offset=2, start=1)
>>> interleaved = interleaver.interleave(BitBuffer.from_str("10100"))
>>> print(interleaved)
    00110
```

To clarify the interleaving process a simple list is interleaved as follows:

```
>>> interleaver = ddl.RegularInterleaver(size=5, offset=2, start=1)
>>> interleaved = interleaver.interleave(['a', 'b', 'c', 'd', 'e'])
>>> print(interleaved)
['b', 'd', 'a', 'c', 'e']
```

property count
the set count

property offset
the set offset

property size
the set size

property start
the set start

2.3.7. File Output Helpers

class procitec.decoding.pcapFile(*file*, *link_type*, *byteorder='little'*)
A class to generate and write a pcap file.

Parameters

- **file** (*path-like or file-like object*) – Path-like objects are passed to Python's open() function. File-like objects are assumed to be already opened for writing in binary mode.
- **link_type** (*int*) – a link-layer header type as defined by libpcap (see http://www.tcpdump.org/linktypes.html)
- **byteorder** (*{"big", "little"}, optional*) – Order of bytes (endianess) when writing the file's global header and the headers for each packet. The default value is the system's byte order.

Notes

Path-like objects are described in `https://docs.python.org/3/glossary.html#term-path-like-object`

File-like objects are described in `https://docs.python.org/3/library/io.html`. It is sufficient to provide an implementation of `write()`, `close()` and the attribute `closed`.

`close()` → None

Close the pcap file.

`write`(*packet_data: bytes | bytearray, timestamp_sec: int, timestamp_usec: int*) → None

Write a packet to the pcap file.

Parameters

- **packet_data** (*bytes or bytearray*) – Packet data to be written to the pcap file. `packet_data` is not modified in any way. It is up to the caller to ensure that the data has the correct byte order (endiannes).

  The length of a data packet is limited to *_SNAPLEN*.

- **timestamp_sec** (*int*) – Timestamp of the packet as number of seconds since 1970-01-01T00:00:00Z. This is also known as known as UNIX time, Epoch time or seconds since the Epoch. The timezone must be UTC (+00:00).

- **timestamp_usec** (*int*) – Timestamp of the packet in microseconds as a positive offset to `timestamp_sec`. This value must be smaller than 1000000.

`_SNAPLEN = 1073741824`

Maximum packet length in bytes

Type
int

`property byteorder: str`

Order of bytes (endianess) when writing headers inside the pcap file. Either *"little"* or `"big"`.

Type
str

`property closed: bool`

`True` if the pcap file is closed, `False` otherwise

Type
bool

2.3.8. Bit Formatting

`class procitec.decoding.PrettyBitFormatter`(*data_format: str = 'H', num_columns_per_line: int = 16, *, num_chars_per_column: int | None = None, separator: str = ' ', column_prefix: str = '', always_full_lines: bool = False, line_prefix: str | None = None, line_postfix: str | None = None, msb_first: bool = False, show_flush: bool = True*)

The PrettyBitFormatter is used to format a continuous stream of bits or a single block of bits into a pretty textual representation.

The input bits are divided into blocks of bits or columns. It is possible to specify the number of characters per column and the number of columns. A single-line output is also possible.

Added in version 22.2.0.

Parameters

- **data_format** (*str, optional*) – Specifies the format of the data. A hexadecimal or binary output is possible. For hexadecimal format (0-9, A-F) use "h" or "H". If the upper case letter is used, the hexadecimal output also uses upper case letters instead of lower case letters for A-F. For the binary format(0,1) use "b". The default is "H".
- **num_columns_per_line** (*int or None, optional*) – Specifies the number of columns per line. If None is passed, everything is written in one line and there is no line break. The default is 16.
- **num_chars_per_column** (*int or None, optional*) – Specifies the number of characters per column or block. If None is passed, 8 bits per column are used. The default is None.
- **separator** (*str, optional*) – Separator between the columns. The default is " ".
- **column_prefix** (*str, optional*) – Arbitrary prefix before each column. The default is "".
- **always_full_lines** (*bool, optional*) – If True is passed, new characters are only returned when formatting of the whole line is completed. Must not be True if num_columns_per_line is None. The default is False.
- **line_prefix** (*str or None, optional*) – Must be None, "line" or "address". If None is passed there is no prefix before each line. With "line" line numbers are printed. With "address" the byte addresses are output hexadecimal. The default is None.
- **line_postfix** (*str or None, optional*) – Must be None or an encoding string. If None is passed, there is no line postfix. Otherwise, after each line the bits with the given encoding are output interpreted as characters. Preferably "ascii" should be passed, but "utf8" is also possible. The encodings offered by Python are supported. However, only single lines are interpreted in the output. This can be problematic with multibyte encodings. Non-printable characters or incorrect characters are replaced with ".". The default is None.
- **msb_first** (*Bool, optional*) – Specifies whether the MSB (most significant bit) or the LSB (least significant bit) per column should be output first. The default is False.
- **show_flush** (*True, optional*) – Specifies whether zero characters are to be output during flush if the number of bits does not completely fill the characters. The default is True.

__call__(*data:* BitBuffer | *None* = *None*, *flush: bool* = *False*) → str

See format_data

append(*data:* BitBuffer) → None

Adds bits to the internal buffer for later output. These can also be added directly during formatting,

Parameters

data (BitBuffer) – Bits to appended.

Return type

None.

`format_data`(*data:* BitBuffer | *None* = *None*, *flush: bool* = *False*) → str

Formats the data into a nice readable format and returns it as a string. These strings should be output one after the other without line breaks. Line breaks are included in the returned strings unless single line formatting is used.

Parameters

- **data** (*BitBuffer, optional*) – Bits to append. If None is passed no bits are added to the output. This is useful if you want to flush the last line. The default is None.
- **flush** (*bool, optional*) – If True is passed, the last line or the trailing bits which could not be output due to padding are flushed. The default is False.

Returns
output – Pretty formatted string representation.

Return type
str

`reset`() → None

Resets the formatter to the initial state.

Return type
None.

2.3.9. Encoding

class procitec.decoding.**ConvolutionalEncoder**(**args*, ***kwargs*)

A encoder for non-recursive convolutional codes.

Added in version 25.1.0.

ConvolutionalEncoder (*mode, polys, *, puncture_pattern=[], initial_state=0, final_state=0*) **ConvolutionalEncoder** (*decoder*)

See *ViterbiDecoder* for general information about convolutional coding.

Parameters
mode ({"terminated", "truncated", "tailbiting", "streaming"}) –

The mode of operation of the decoder which is determined by the
encoding process:

- "terminated": The encoder is flushed with a known sequence of bits (usually zeros). Also known as *flushed* encoding.
- "truncated": The final state of the encoder is determined by the last bits of the information block.
- "tailbiting": The encoder is preloaded with the last bits of the information block, i.e. the initial and final state of the encoder are equal.
- "streaming": The encoder is never flushed; an infinite stream of information bits is assumed. Manual flushing with termination to defined state is possible.

The encoding modes "terminated", "truncated" and "tailbiting" can be regarded as block codes, i.e a fixed amount of data is encoded. With "streaming" the encoder is fed with blocks of the infinite encoded bit stream and keeps the internal register state for consecutive processing. Flushing the encoder is possible. See *flush()* for flushing in mode "streaming".

polys
: [list[int]] A list of polynomials defining the convolutional code. Each element must be a bit mask where the least significant bit corresponds to the coefficient of exponent 0. The position of the highest bit set (zero based counting, in any of the polynomials) determines the shift register length used in the encoding process.

  Note: the required representation of the generator polynomials is reversed compared to commonly used octal representation. The common representation as found in most books and references puts the coefficient of the highest exponent into the least significant bit; the coefficient for exponent zero is consequently at the position of the highest bit set. See examples below.

puncture_pattern
: [list[int] or str, optional] A list or string describing the puncturing performed in the encoding process. A 1 indicates that the corresponding is *not* punctured, i.e. transmitted; a 0 indicates that the corresponding bit is punctured, i.e. *not* transmitted.

  The puncture pattern is traversed periodically. This means:

  - A puncture matrix can be passed by writing the elements of the matrix columnwise into the list.
  - A list whose length is equal to the length of an information block is traversed only once.

  By default (empty list) no puncturing is assumed. In mode `"streaming"` it is possible to change the puncturing between blocks.

initial_state
: [int, optional] Initial state of the encoder. This is the initial state of the shift register interpreted as a binary number (default 0). Only used if `mode` is `"terminated"`, `"truncated"` or `"streaming"`; ignored otherwise. In mode `"tailbiting"` initial and final states are calculated internally from the given input data.

final_state
: [int, optional] Final state of the encoder. This is the content of the shift register interpreted as a binary number, after flushing has been performed. Only relevant and used if `mode` is `"terminated"` or the decoder is flushed in mode `"streaming"` using *`flush()`*; ignored otherwise.

decoder
: [ViterbiDecoder] It is possible to use a decoder object to initialize the encoder with the proper code settings. The decoder has to be initialized with polynomials or not with a trellis table.

  No further parameters are allowed if this parameter is used.

  If the decoder was initilized with an unknown initial state (`initial_state=None`), the encoder uses 0 as initial state.

Examples

see *`ViterbiDecoder`*

`encode`(*data:* BitBuffer | *Sequence[int]*) → *BitBuffer*

Encode one block of data.

Unless mode is `"streaming"`, all calls to *`encode()`* are independent and assume that the whole information block is passed. With `"streaming"` mode an infinite stream of information bits of different sizes may be passed.

For flushing the decoder in mode `"streaming"` see *`flush()`*.

Parameters
data (*BitBuffer or Sequence[int]*) – Plain information bits to be encoded.

In case of an integer sequence each item represents one bit. Only the values [0, 1] are allowed. All other values lead to undefined output.

Returns
data – The encoded bits.

Return type
BitBuffer

`flush()` → *BitBuffer*

Only useful in mode `"streaming"`. For the other modes the flush is always included in the call of *encode()*.

This method flushes the remaining bits in the state register with the final state (most times 0).

The encoder is also reset (see *reset()*).

Returns
data – The encoded bits. Empty buffer if mode is not `"streaming"`.

Return type
BitBuffer

`reset()` → None

Reset internal state of the encoder

Has only an effect if `mode` is `"streaming"` (all other modes are stateless).

`set_final_state(`*final_state: int*`)` → None

Sets the final state of the encoder register. See *ConvolutionalEncoder*

`set_initial_state(`*initial_state: int*`)` → None

Sets the initial state of the encoder register. See *ConvolutionalEncoder*

`property register_state`

the current internal state of the shift register

`class procitec.decoding.LDPCEncoder(`*code:* LDPCCode, *include_infobits: bool = True*`)`

An encoder for systematic low-density parity-check (LDPC) codes.

The encoder works generically for any ldpc codes. Therefore, the performance and processing speed may be impaired for very long codes.

Nevertheless, a speed-optimized algorithm is used for codes with special designed parity check matrices. For this, the inverse of the right-hand side of the matrix (the right `(n-k) x (n-k)` part of the `(n-k) x n` parity check matrix) must be spare.

Added in version 25.2.0.

Parameters

- **code** (LDPCCode) – The definition of the code used.
- **include_infobits** (*bool*) – If `True`, the result of the *encode()* method is the full `n` bit codeword and includes `k` infobits. Otherwise only the `n-k` parity-check bits are returned. The default is `True`.

`encode`(*data:* BitBuffer | *Sequence[int]*) → *BitBuffer*

Encode information bits.

Parameters

data (BitBuffer | *Sequence[int]*) – The info bits to be encoded. The length has to match the specified code. If the type Sequence[int] is used, all integer elements have to be either 0 or 1.

Returns

encoded – The encoded data bits.

Return type

BitBuffer

Raises

ValueError – The length of the input is not correct.

2.3.10. Miscellaneous

`class procitec.decoding.SampleFormat`

Members:

MAG_PHASE : Magnitude and Phase, commonly used for PSK modulations

I_Q : Real and imaginary components, useful for QAM modulations

FSK : Value discriminating between frequencies/tones for FSK modulations

FSK = <SampleFormat.FSK: 3>

I_Q = <SampleFormat.I_Q: 2>

MAG_PHASE = <SampleFormat.MAG_PHASE: 1>

property name

property value

`exception procitec.decoding.EndOfDataError`

This exception is used by the decoder runtime to signal the end of production or search. It is raised when requesting input data or other information from the runtime after the *BitStream*, *apc.data*, has been closed. The runtime expects the decoder to terminate and will not report uncaught instances of the exception as an error.

EndOfDataError must not be raised by the decoder directly. It may, however, be caught (and re-raised) to perform clean-up tasks at the end of production. Note that upon occurrence of the exception no more data can be read and most of the runtime functions are ineffective. Only the output-related parts of the *runtime object* should be accessed. This includes for example *apc.output* and *apc.production_memory*.

EndOfDataError is a subclass of *procitec.common.bitbuffer.EndOfStreamError*.

2.4. Miscellaneous

class procitec.common.ProTS(**args*)

This class represents an absolute point in time (timestamp) based on the UNIX-epoch and the fractional part thereof. It is mostly used as meta-data for signals and decoding events. Construction needs either the full and factorial part of the second or, for convince, the human readable components of the corresponding date and time elements, see below.

Instance of this class can be formatted to a string or converted to regular `datetime.datetime` objects.

1. ProTS(*seconds, fraction=0*)

   Creates a timestamp from `seconds` (`int`) and `fraction` (`float`) of a second elapsed since 1970-01-01T00:00:00:000Z.

2. ProTS(*year, month, day, hour, minute, second, millisecond=0*)

   Creates a timestamp specified by `year`, `month`, `day`, `hour`, `minute`, `second` and optionally `millisecond`; all parameters are of type `int`. If the date is not valid, the timestamp is set to 1970-01-01T00:00:00:000Z.

The operators - and + allow to decrease or increase a timestamp with the specified amount of *seconds* (float). The subtraction of two timestamps results in a time difference in *seconds* (float). And it is possible to compare two timestamps using the <, > and == operators.

__format__(*format*)

Returns the timestamp as a formatted string, defined by the given format specifier

Parameters

`format` (*str, optional*) – Format specifier string. The default format specifier "`yyyy-MM-ddThh:mm:ss.zzzZ`" conforms to ISO 8601.

The following format codes are supported:

Date	
yyyy	the year as four digit number
yy	the year as two digit number (00-99)
MM	the month as number with a leading zero (01-12)
M	the month as number without a leading zero (1-12)
dd	the day as number with a leading zero (01-31)
d	the day as number without a leading zero (1-31)
Time	
hh	the hour with a leading zero (00-23 or 01-12 if AM/PM display)
h	the hour without a leading zero (0-23 or 1-12 if AM/PM display)
mm	the minute with a leading zero (00-59)
m	the minute without a leading zero (0-59)
ss	the second with a leading zero (00-59)
s	the second without a leading zero (0-59)
zzz	the milliseconds with leading zeroes (000-999)
zzzzzz	the microseconds with leading zeroes (000-999999)

Note: Excluded time components are truncated, not rounded.

Examples

```
>>> from procitec.common import ProTS
>>> t = ProTS(224133255, 0.9265359)
>>> print(format(t, 'yyyy-MM-ddThh:mm:ss.zzzZ'))
1977-02-07T03:14:15.926Z
>>> print(format(t, 'yyyy/dd/MM - hh:mm:ss.zzzzzz'))
1977/07/02 - 03:14:15.926535
>>> print(f"{t:hh}h {t:mm}m {t:ss}s")
03h 14m 15s
```

`__str__()`
: Returns the timestamp as an ISO 8601 formatted string including date and time.

`add_seconds()`
: add seconds to the time stamp

`datetime()`
: Convert the timestamp to `datetime.datetime` with the timezone as UTC

`to_float()`
: return as seconds since 1970 including fraction

property fraction

Fractional part

Type
float

property seconds

Full seconds part

Type
int

procitec.common.find_library(*lib: PathLike | str*) → str

Find the path of a dynamic library

The search order is:

- If `lib` is an absolute path and points to an existing file, the path is returned as is.
- If `lib` is a filename only or a relative path with a filename, system-specific prefixes and extensions are added to the filename. This means that the passed filename must be the name of the library without any system-specific portions. The library will be searched relatively in
  - the folder `modems` contained in the user folder
  - and in the folder `extdecod` contained in the installation folder.

Parameters
`lib` (*os.PathLike*) – The following can be passed:
- an absolute path to a file
- a filename only without any system-specific suffixes and extensions for a library
- a relative path whose filename (everything after the last slash) is as above

Returns
Path of the found library including system-specific suffixes and extensions. `FileNotFoundError` is raised if the library can not be found.

Return type
os.PathLike

Examples

Assume that the user folder is `USERNAME\procitec\common` and that the installation folder is `C:\Program Files\procitec\common{XX.Y}`. (/opt/procitec/common/{XX.Y} in case of Linux)

```
>>> import procitec.common
>>> procitec.common.find_library("decod")
```

The code above will check if any of the following files exist and returns the corresponding path:

- `USERNAME\procitec\common\modems\decod.dll`
- `C:\Program Files\procitec\common{XX.Y}\apc-ext\extdecod\decod.dll`

Relative paths are searched as shown below

```
>>> procitec.common.find_library("./vendor/decod") # leading "./" may be left out
```

- `USERNAME\procitec\common\modems\vendor\decod.dll`

- C:\Program Files\procitec\common{XX.Y}\apc-ext\extdecod\vendor\decod.dll

```
>>> procitec.common.find_library("../vendor/a/decod")
```

- USERNAME\go2SIGNALS\common\modems\..\vendor\a\decod.dll
- C:\Program Files\procitec\common{XX.Y}\apc-ext\extdecod\..\vendor\a\decod.dll

procitec.packaging.builder.packager.**create_decoder_package**(*decoder_filepath*, *output_path: str = '.'*, *comment: str = ''*) → Tuple[Path, List[str]]

Creates a decoder package file (*_dec.pkg) from a given decoder. The decoder imports are analyzed and all needed modules are bundled into the package.

Parameters

- **decoder_filepath** (*str or path-like*) – The decoder to pack.
- **output_path** (*str*) – Where to save the decoder package.
- **comment** (*str*) – A comment saved in the decoder packages header.

Returns

- **decoder_package** (*Path*) – The path of the created package.
- **non_env_module_names** (*list[str]*) – Imported modules in decoder which are not available in the default APC python environment. Consider adding it to the environment before running the decoder.

Raises

SyntaxError – In case of a syntax error in decoder source code and dependencies.

3. Porting Existing DDL Decoders to pyDDL

3.1. Automatic Conversion

The IDE for the development of pyDDL decoders (see *Decoder Development Using Spyder*) provides a tool for automatic conversion of a DDL decoder to pyDDL. It can be invoked from the `File → Convert DDL Decoder` menu entry. You will be asked to select a DDL decoder to convert. The conversion may take some time.

The automatic conversion is able to convert most DDL constructs to equivalent pyDDL code:

- Conditional branches
- `for`/`while` loops, including `break` statements
- `switch case` blocks
- `User defined functions` (chapter 5.13.9 in the DDL manual)
- `Subroutines` (chapters 5.13.2 and 5.13.3 in the DDL manual)
- `Symbol Tables` (chapter 5.17 in the DDL manual)
- Many DDL functions

Please note that there are limitations, the notable ones being:

- `GoTo()` statements can not be converted as they are not available in Python.
- `Pre-Processing` (chapters 4.1 and 5.2 in DDL manual) is not converted.
- Manipulation of variables (assignments, arithmetic and logical operations) are converted. However, the converter may make wrong assumptions about the intended data type in the DDL code.

DDL code which can not be converted is put in comments in the converted Python code. Comments from the DDL decoder are retained.

The produced code is likely *not* runnable. This is especially true for decoders with a complex execution flow due to `GoTo()` statements. The translation process basically converts DDL statements to corresponding pyDDL statements as listed in *Quick Reference*. This may result in suboptimal code and simplifications may be made.

3.2. Quick Reference

This section provides a quick reference for the translation of DDL code to pyDDL. Each section hereafter corresponds to chapter in the DDL Manual. A table of DDL commands and the corresponding pyDDL command is given.

- *Basic Programming and Language Elements*
- *Pre-Processing Functions*
- *Search Functions*

- *Read Functions*
- *Read Pointer Functions*
- *Fractionize Frame*
- *Reformat and Process Data Packages*
- *Arrays*
- *Analyzing, Code Checking and FEC*
- *Universal Decoding of Block Codes*
- *Functions for Bit Manipulation*
- *Operators*
- *Branch Commands*
- *Output Functions*
- *System Functions*
- *System Variables*
- *Symbol Tables*
- *Commands to Control Demodulator Parameters*
- *Knowledge-Base Demodulator Settings*
- *Special Commands for Morse Post-Processing*
- *Measurement Functions*
- *External Program Control and Interfacing*
- *Dynamic Link Libraries*
- *CVSD Speech Decoding*
- *Special Commands for Data Base Support*
- *Soft Decision Decoding*

3.2.1. Basic Programming and Language Elements

DDL	pyDDL
Decoder Input Values	*apc.parameters*
Input Buffer	*apc.data*
Read Pointer	*apc.data.position*
Variables	Python variables
LSB-First Logic	(same)
Value Annotations	Python literals -
Bit Patterns	Python `int` literals or `str` with `01x` (see *search_pattern()*)
Bit-Reverse Annotations	• N/A for Python `int` • `buffer[::-1]` for *BitBuffer*
Labels	N/A (no more `GoTo()`)
Comments	Comments use #

3.2.2. Pre-Processing Functions

DDL	pyDDL
PPInvert	*preprocessing.invert()*
PPDescramble	*preprocessing.descramble()*
PPBitCodeNRZ	*preprocessing.nrz_decode()*
PPBitCodeInvNRZ	*preprocessing.nrz_encode()*
PPBitCodeManch	NOT IMPLEMENTED
PPBitCodeBIPH	NOT IMPLEMENTED
PPConvertSymbol	*preprocessing.convert_symbols()*
PPConvertDiff	*preprocessing.differential_encode()*
PPSymbolBitReversal	*preprocessing.reverse_symbol_bits()*
PPIsLastCall	catch *EndOfDataError*; see *Performing Tasks on Decoder Exit* for example
PPSeparateChannels	*apc.input_channel_mode()*
PPSelectInpPacket	NOT IMPLEMENTED
PPRecombineChannels	NOT IMPLEMENTED
PPExtractChannels	*apc.input_channel_mode()*

3.2.3. Search Functions

DDL	pyDDL
SearchPattern	`search_pattern()`
SearchPatternVar	`search_pattern()`
SearchSymbolTab	`search_alphabet()`
SearchInterlSymbolTab	`search_alphabet()`
SearchSymbolSignPattern	`search_alphabet()`
SearchFrameRepeat	NOT IMPLEMENTED
SearchBurst	`search_burst()`
SearchBurstPause	`search_burst_end()`, `search_burst_start()`
SearchParity	NOT IMPLEMENTED, but see cis_14 decoder for substitute
SearchPolynom	`search_lfsr_sequence()`
SearchVectPatternMatch	OBSOLETE

3.2.4. Read Functions

DDL	pyDDL
GetFrame	`apc.data.read()`
GetDeInterl	`apc.data.read()` + `extract_interleaved()`
GetSymbol	`apc.data.rewind(-apc.data.position % apc.symbols.bits_per_symbol)` and then `apc.data.read(num_symbols * apc.symbols.bits_per_symbol)`
GetBurstSymbol	use `search_burst_end()` without consuming and then `apc.data.read()` see twinplex decoder

3.2.5. Read Pointer Functions

DDL	pyDDL
GetPos	*apc.data.position*
SetPos	*apc.data.set_position()*
Rewind	*apc.data.rewind()*
SymbolSync	combine *position* with *bits_per_symbol* and *apc.data.consume()*
ChannelSync	*apc.input_channel_sync()*
GetTime	*apc.data.time()*
TimeSkip	*procitec.decoding.time_skip()*
SelectInpBuffer	*apc.data.read(... ,channel_index=...)*
ClearInputBuffer	NOT IMPLEMENTED
BufferStatus	*procitec.common.bitbuffer.BitStream.available*
SymbolSize	*apc.symbols.bits_per_symbol*

3.2.6. Fractionize Frame

DDL	pyDDL
Extract	slicing in *BitBuffer*
ExtractInterl	*extract_interleaved()*
ExtractInterlLong	*extract_interleaved()*
ExtractSymbInterlLong	*extract_interleaved_symbols()*
ExtractPattern	*extract_pattern()*

3.2.7. Reformat and Process Data Packages

DDL	pyDDL
Destuff	*de_stuff()*
Join	*bitbuffer.concat()*
Place	slice-assignment on *BitBuffer*
Insert	*bitbuffer.insert()*
ConvertSymbol	*convert_symbols()*
VectorRead	Python `list` etc.
VectorWrite	Python `list` etc.
FindBytes	convert data to `bytes` or `str` and use `find()` method
StringLen	convert data to `bytes` or `str` type and use `len`
ReverseByteOrder	• *bitbuffer.reverse_symbol_order()* • convert data to `bytes` and use `reversed()` or `...[::-1]`
MirrorBytes	• *bitbuffer.mirror_symbols()* • reversed slicing in *BitBuffer* • alphabet with reversed bitorder (MSB/LSB)

3.2.8. Arrays

DDL	pyDDL
ArrayDef	• Python `list`, `tuple`, `bytes`, `bytearray` • *BitBuffer*

3.2.9. Analyzing, Code Checking and FEC

DDL	pyDDL
TestPolynom	NOT IMPLEMENTED
CorrectMajority	*correct_majority()*
CorrectExtGolay	*BlockDecoder* in junction with *golay_parity_matrix()*, see *Using Golay Block Decoder*
CorrectFECA	NOT IMPLEMENTED
CheckCRC	*crc()*
CheckHamm	*BlockDecoder* in junction with *hamming_generator_matrix()*, see *Using Hamming Block Decoder*
CorrectHamm	*BlockDecoder* in junction with *hamming_generator_matrix()*, see *Using Hamming Block Decoder*
Weight	*hamming_weight()*
AddDescramble	*generate_lfsr_sequence()*
IsTabSymb	*procitec.decoding.Alphabet.decode()*
ViterbiHDD	*ViterbiDecoder*
CorrectViterbi	*ViterbiDecoder*
CorrectViterbiSoft	*ViterbiDecoder*
CorrectBCH	*BCHDecoder*
CorrectRS	*ReedSolomonDecoder*
CorrectRSext	*ReedSolomonDecoder*
BitCorrelation	*bit_correlation_and_maxima()*

3.2.10. Universal Decoding of Block Codes

DDL	pyDDL
InitBlockCode	*BlockDecoder*
CorrectBlockCode	*BlockDecoder*
CloseBlockCode	*BlockDecoder*

3.2.11. Functions for Bit Manipulation

DDL	pyDDL
RotLeft	*lshift(circular=True)*
RotRight	*rshift(circular=True)*

3.2.12. Operators

DDL	pyDDL
Operator Assignment	Python has no custom assignment operator, object are never copied on assignment
Operators Addition, Subtraction, Multiplication, Division, Modulo	Python `int`
Operators Smaller, Smaller or Equal, Greater, ...	• *BitBuffer* supports == • Python `int`
Operators Bitwise And, Bitwise Or, Exclusive Or	• *BitBuffer* supports bit operators &, | and ^ on `int` or another BitBuffer • Python `int`; note: "unlimited" precision
Operators Bitwise Negation	• *invert()* • Python `int`; note: careful with negative values
Operators Shift Left	• *lshift()* • `BitBuffer << int`, `BitBuffer <<= int` • Python `int`; note: "unlimited" precision
Operators Shift Right	• *rshift()* • `BitBuffer >> int`, `BitBuffer >>= int` • Python `int`; note: "unlimited" precision, careful with negative values
Operators Logical Negation	Python `not`
Operators Logical And, Logical Or	Python `and`, `or`

3.2.13. Branch Commands

DDL	pyDDL
Unconditional Branch	Not Available - no more `GoTo()`
Subroutine Branch	Python functions
Return from Subroutine	Python `return`
Conditional Branches	Python `if`
While Loops	Python `while`
For Loops	Python `for`
Switch Case Blocks	Python `if...elif...else`
Break Instruction	Python `break`
User-Defined Functions	Python `def function_name()`
Return	Python `return`

3.2.14. Output Functions

DDL	pyDDL
SetOutBuf	Python `str` or `bytearray`
OutTab	*`procitec.decoding.Alphabet.decode()`* + *`apc.output.NAME()`*
OutTabHuffman	*`procitec.decoding.Alphabet.decode()`* + *`apc.output.NAME()`*
OutText	*`apc.output.NAME()`*
OutVal	*`apc.output.NAME()`* + Python formatting
OutValForm	*`apc.output.NAME()`* + Python formatting
OutTimeStamp	*`apc.output.NAME()`* + Python formatting + *ProTS*
OutDateStamp	*`apc.output.NAME()`* + Python formatting + *ProTS*
OutDateTime	*`apc.output.NAME()`* + Python formatting + *ProTS*
ClearOutputBuffer	OBSOLETE
Output Character Sets	Python `bytes.decode`
BinFileInit	*`apc.production_memory.open()`* in `with` (context manager)
BinFileOut	*`mem_file.write()`*
BinFileXMLOut	OBSOLETE, use e.g. `ElementTree`
BinFileXMLOpen	OBSOLETE, see above
BinFileXMLClose	OBSOLETE, see above
BinFileClose	*`mem_file.close()`*
BinFileReport	*`apc.production_memory.report_file()`*

3.2.15. System Functions

DDL	pyDDL
Sync	`apc.modem.sync()`
Ident	`apc.modem.ident()`
IdentAt	`apc.modem.ident(...)`
Accept	`apc.modem.accept()`
AcceptAt	`apc.modem.accept(...)`
Fail	`apc.modem.no_sync()`
StopModem	`apc.modem.stop()`
Quality	`mean_quality()`
SearchMode	`apc.current_mode`
ExtendSearch	`apc.extend_search()`
SetTimeOut	`apc.set_timeout()`
FlushOut	`apc.output.flush()`
SetAutoFlush	OBSOLETE
SetProdHoldTime	`apc.set_production_hold_time()`
ErrExit	Python `raise` or `exit()`
Version	`apc.version`
BurstStartAt	`apc.burst_start_time()` (ProTS or bit position)
BurstEndAt	`apc.burst_end_time()` (see above)
IgnoreBurstDetector	`apc.ignore_burst_detector()`

3.2.16. System Variables

DDL	pyDDL
_auto_invert	moved to search functions
_invert	moved to search functions
_idweight	`apc.modem.ident(weight=...)`
_burststart	use `search_burst()` beforehand
_burstend	use `search_burst()` beforehand
_class	`apc.decoder_properties.add('class', "ARQ")`
_mode	`apc.decoder_properties.add('mode', "SitorB")`

3.2.17. Symbol Tables

DDL	pyDDL
Table Attributes and Table Switches	*procitec.decoding.Alphabet*
List of Table Values	*procitec.decoding.Alphabet*

See *Converting Alphabets* for examples how to port tables defined in DDL.

3.2.18. Commands to Control Demodulator Parameters

DDL	pyDDL
SetDemodType	*apc.demodulator.modify*
ReadDemodType	*apc.demodulator.parameters*
SetDemodPara	*apc.demodulator.modify*
SetDemodParaSet	*apc.demodulator.modify*
ReadDemodPara	*apc.demodulator.parameters*
ReadDemodParaSet	*apc.demodulator.parameters*
ReadDemodAck	OBSOLETE, check is performed automatically by *apc.demodulator.modify*
WaitDemodAck	OBSOLETE, check is performed automatically by *apc.demodulator.modify*
RewindDemod	*apc.demodulator.modify*
ReadDemodRewindAck	OBSOLETE, check is performed automatically by *apc.demodulator.modify*
WaitDemodRewindAck	OBSOLETE, check is performed automatically by *apc.demodulator.modify*
SaveDemod	OBSOLETE, see example in *apc.demodulator.modify*
RecallDemod	OBSOLETE, see example in *apc.demodulator.modify*
SetBurstPreamble	*apc.demodulator.set_burst_preamble*

Note:

3.2.19. Knowledge-Base Demodulator Settings

DDL	pyDDL
PSKFilter	*apc.demodulator.set_filter*
SetTrainingPattern	*apc.demodulator.set_training_pattern*

3.2.20. Special Commands for Morse Post-Processing

DDL	pyDDL
Check_AR	NOT IMPLEMENTED
Check_CallSeq	NOT IMPLEMENTED

3.2.21. Measurement Functions

DDL	pyDDL
StartMeas (Optional)	NOT IMPLEMENTED
WaitMeasResult (Optional)	NOT IMPLEMENTED
ReadMeasResult (Optional)	NOT IMPLEMENTED

3.2.22. External Program Control and Interfacing

DDL	pyDDL
ProcLoad	Python subprocess
ProcUnload	Python subprocess
ProcIsRunning	Python subprocess
ProcWrite	Python subprocess
ProcWriteBin	Python subprocess
ProcSelectChannel	Python subprocess
ProcRead	Python subprocess
ProcReadWord	Python subprocess
ProcReadLine	Python subprocess
ProcCheckInput	Python subprocess
ProcFlushInput	Python subprocess
ProcCheckError	Python subprocess
ProcBytesToWrite	Python subprocess
ProcBytesToRead	Python subprocess
ProcWaitForResult	Python subprocess
ProcCloseWriteChannel	Python subprocess

3.2.23. Dynamic Link Libraries

See *Using DLLs/Shared Libraries* for a guide.

DDL	pyDDL
LibLoad	Python ctypes and *find_library()*
LibFunction	Python ctypes
Creating a User Library	Python extension modules
Errors in the Library Code	Python Exceptions

3.2.24. CVSD Speech Decoding

DDL	pyDDL
CVSDInit	NOT IMPLEMENTED
CVSDOut	NOT IMPLEMENTED
CVSDClose	NOT IMPLEMENTED

3.2.25. Special Commands for Data Base Support

DDL	pyDDL
NextUnit	*apc.output.xml(f"ch{channel:03d}-infounit", "")*

3.2.26. Soft Decision Decoding

DDL	pyDDL
DefineSoftSymbols	*apc.demodulator.set_soft_symbols*
SearchSoftPattern	*search_pattern_soft()*
SearchSoftPatternVar	*search_pattern_soft()*
GetMagPhase	*apc.symbols.read*
GetFrequency	*apc.symbols.read*
BitMetricPSK	*bit_metric_psk()*
BitMetricLLR	*bit_metric_distance()*
BitsSoft2Hard	hard decided bits included in bit_metric_xxx() functions
SymbolsSoft2Hard	NOT IMPLEMENTED
CalcNoiseRef	NOT IMPLEMENTED

3.3. Migration Guide

This section contains a collection of migration guides, each focusing on a different part of the DDL functionality and how it may be ported to Python and pyDDL.

3.3.1. Search pattern

This section outlines differences in the `search_pattern()` command to the corresponding DDL command `SearchPattern` when using `auto_invert`. The DDL variant will search in the normal bit stream until the gap limit first and if no match is found will rerun the search on the inverted bit stream second. In case of pyDDL the search will be carried in parallel on the normal and the inverted bit stream. This may result in case of a short search pattern to different results.

3.3.2. Performing Tasks on Decoder Exit

This section describes how decoders using the DDL command `PPIsLastCall()` can be ported to pyDDL. Typical use-cases are closing opened files and reporting data collected during production.

pyDDL decoders are usually terminated by raising the exception *procitec.decoding.EndOfDataError* from the runtime. This exception may be caught to perform actions just before the decoder exists.

The code below shows an exemplary structure that calls a function, `last_call()`, before termination:

```
import procitec.decoding as ddl

some_obj = ...  # requires explicit action before exit


def main():
    ...  # uses some_obj


def last_call():
    # example
    apc.output.text1(some_obj.dump_results())


apc = ddl.runtime.APC()

try:
    while True:
        main()
except ddl.EndOfDataError:
    last_call()
```

Actions performed in `last_call()` usually depend on state or objects collected in `main()`. In the example above this is represented by a module global object, `some_obj`. Alternatively, `main()` and `last_call()` can be methods of a class and `some_obj` an attribute of the same class.

3.3.3. Using Golay Block Decoder

This section briefly outlines an example of implementing a Golay Block decoder.

```
from procitec.decoding import BlockDecoder, golay_parity_matrix

golay_dec = BlockDecoder(golay_parity_matrix('AE3'), 8)  # using Golay Block Decoder␣
↪with 'AE3' matrix.
```

3.3.4. Using Hamming Block Decoder

This section briefly outlines an example of implementing a Hamming Block decoder.

```
from procitec.decoding import BlockDecoder, hamming_generator_matrix

hamming3_dec = BlockDecoder(hamming_generator_matrix(3), 3)  # using Hamming Block␣
↪Decoder with Hamming code (7, 4, 3).
hamming4_dec = BlockDecoder(hamming_generator_matrix(4), 3)  # using Hamming Block␣
↪Decoder with Hamming code (15, 11, 3).
```

3.3.5. Converting Alphabets

Converting an existing alphabet from DDL to pyDDL is a straightforward substitution of code syntax. In DDL, all alphabets are specified at the end of the decoder source file with a syntax as shown in the example below.

```
ALPHA_DEF(123)

BitNo = 7
NoLevels = 2
ErrSymb = _

TABLE

06_h,    <ACK>, 3
15_h,    <NAK>, 5
30_h,    0,     9
31_h,    1,     8
32_h,    <L2>,  <L1>

ENDTABLE
END_ALPHA_DEF
```

In pyDDL the same alphabet is defined with a builtin Python `dict` and *procitec.decoding.Alphabet* as follows

```
import procitec.decoding as ddl

L = ddl.LEVEL
alphabet123 = ddl.Alphabet(
    {
        0x06: ['<ACK>', '3'],
        0x15: ['<NAK>', '5'],
        0x30: ['0', '9'],
        0x31: ['1', '8'],
        0x32: [L[1], L[0]]
    },
```

(continues on next page)

(continued from previous page)

```
    codeword_length=7,
    replacement_character='_')
```

Note

To decode a stream of bits in pyDDL *procitec.decoding.AlphabetDecoder* should be used.

Warning

DDL source code always uses Latin1 encoding, whereas pyDDL source code has to be UTF-8. This must be taken into account when converting characters outside the 7-bit ASCII range.

In DDL an alphabet is often used for mapping integer values to strings without the need of any error correction or level shifting. For that purpose an alphabet is not necessary and a builtin `dict` is sufficient, as seen in the following example.

```
value=2
OutTab(value, 2, 123, 1, 0,,,,)

...

ALPHA_DEF(123)

BitNo = 2
NoLevels = 1
ErrSymb = u  ; limitation: only one character allowed

TABLE

0, short
1, medium
2, long

ENDTABLE
END_ALPHA_DEF
```

In pyDDL this operation can be implemented in a more compact way.

```
value = 2
out = { 0: "short",
        1: "medium",
        2: "long", }.get(value, "unknown")
apc.output.text1(out)
```

The optional `MSB_First` option in DDL causes a bit reversal of the input code. This is also possible in pyDDL using the parameter `msb_first=True` as shown in the following two codeblocks.

```
ALPHA_DEF(12)

BitNo = 4
NoLevels = 1
ErrSymb = x
MSB_First

TABLE

011_b, 1
111_b, 2
001_b, 0

ENDTABLE
END_ALPHA_DEF
```

```
import procitec.decoding as ddl

alphabet12 = ddl.Alphabet(
    {
        0b0011: ['1'],
        0b0111: ['2'],
        0b1001: ['0'],
    },
    codeword_length=4,
    replacement_character='x',
    msb_first=True)
```

The following alphabet definition is equivalent to the one above. Note the reversed order of bits and the usage of `msb_first=False` (which can be omitted as it is assumed by default).

```
alphabet12 = ddl.Alphabet(
    {
        0b1100: ['1'],
        0b1110: ['2'],
        0b1001: ['0'],
    },
    codeword_length=4,
    replacement_character='x',
    msb_first=False)
```

When replacing `OutTab` with `Alphabet.decode` or `AlphabetDecoder` then it should be noted that `OutTab` only returns a character if exactly one codeword matches the given bit sequence. In case of pyDDL and several possible codeword matches the first match will be returned. See *procitec.decoding.AlphabetDecoder* for details.

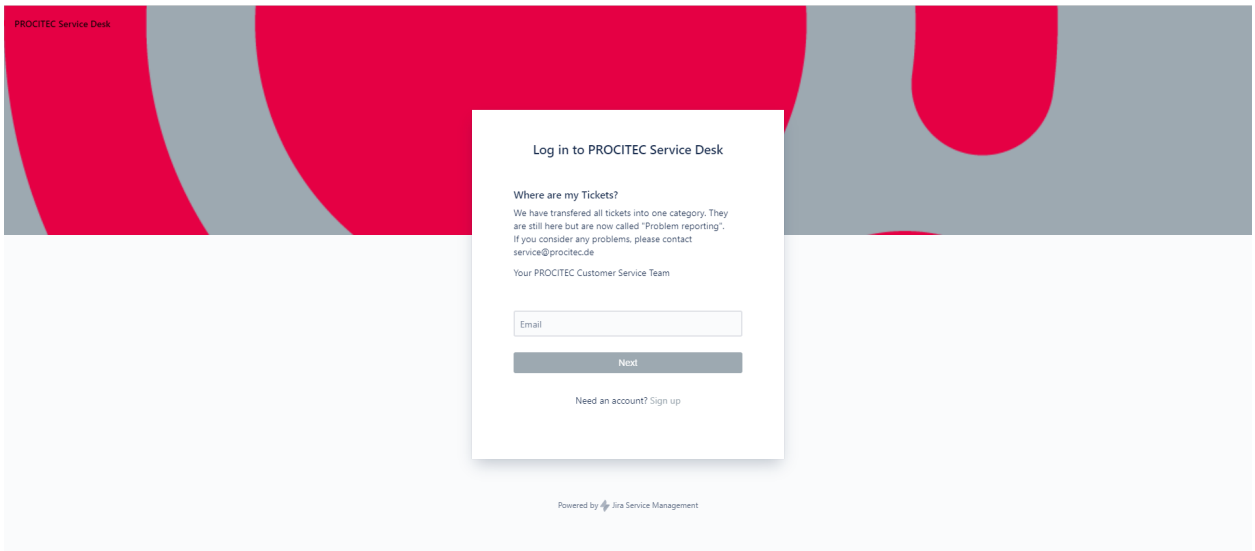
A. Support

Requests and suggestions?

All requests or suggestions regarding our go2signals product-range are very much appreciated; we would be delighted to hear from you.

Any questions? We are happy to assist you!

If you have any further questions, please do not hesitate to contact our Support Team for rapid assistance – just raise a service request at: **http://servicedesk.procitec.com**.



PROCITEC GmbH
Rastatter Straße 41
D-75179 Pforzheim
Phone: +49 7231 15561 0
Web: www.procitec.com
Email: service@procitec.com

List of Figures

1. Spyder main window . . . 8
2. Dialog shown upon opening an existing decoder . . . 9
3. Configuration dialog for decoder execution . . . 10
4. Dialog for decoder parameters in *Execution Mode Decoder only* . . . 12
5. Dialog for decoder parameters in *Execution Mode Signal processing and decoder* . . . 13
6. Decoder output widget in the upper right corner of Spyder's window. Note the special handling of output on the `notify` channel: It is displayed on all output tabs in gray color. . . . 14
7. Configuration dialog for decoder output . . . 15
8. Setting or clearing a breakpoint using the mouse; use left mouse button . . . 15
9. A running debug session. `Production ongoing` and `data position` in the statusbar is described in *Examining and Saving Decoder's Output*. . . . 16
10. Watchlist plugin. Syntax errors are displayed in red. A `NameError` either indicates that a variable is not defined yet or that it is misspelled. . . . 17
11. Graphical display of a BitBuffer . . . 18
12. Graphical display of a BitBuffer . . . 19
13. Example output of the line profiler . . . 20
14. OpenGL error . . . 21
15. Decoder module/package warning in Spyder . . . 24

List of Tables