BitPacket Manual

Release 1.0.0

Aleix Conchillo Flaqué

September 03, 2014

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Download

BitPacket is maintained in Savannah (and mirrored in github and gitorious). Savannah is the central point for development, maintenance and distribution of official GNU software (and other non-GNU software, like BitPacket).

You can download the latest BitPacket release from the project's website, or alternatively, you can also clone the source repository.

```
git clone git://git.sv.gnu.org/bitpacket.git
```

Or, if you are behind a firewall, you might use the HTTP version:

```
git clone http://git.savannah.gnu.org/r/bitpacket.git
```

1.2 Build and install

BitPacket is distributed as a Distribute (setuptools) module, so the usual commands for building and installing setuptools modules can be used. However, this means that you need setuptools installed in your system.

Once the BitPacket tarball is decompressed, you can build BitPacket as a non-root user:

```
python setup.py build
```

If the built is successful, you can then install it, as root, with the following command:

```
python setup.py install
```

1.3 Usage

Using BitPacket in your application is straightforward. You only need to add the following import in your Python scripts:

```
from BitPacket import *
```

1.4 History

The first version of BitPacket was released in 2007.

The validation guys from the project I was working on were building a test environment to validate a software which involved a lot of network packet management. They started by accessing packet fields with indexes. This was very error prone, hard to maintain, hard to read and hard to understand. So, I start digging through the web for something that could help us, but I only found the struct module. However, it does not solve the indexing problem neither it supports bit fields.

Then, I found the BitVector class which was able to work with bits given a byte array, and I built BitPacket in top of it. Initially, BitPacket consisted on three classes: BitField (for single bit fields), BitStructure (a BitField itself, to build packets as a sequence BitFields) and BitVariableStructure (something like a meta BitStructure).

At the end of 2009, a refactoring of the test environment was necessary, and I knew BitPacket was very slow and hard to extend. Between 2007 and 2009, I discovered a great Python library for building and parsing packets, construct. construct is great and performs its jobs very well. It is a very complete and powerful library for working with packets in a declarative way. The problem was that we had a lot of code that need to be reused written with BitPacket, so construct was not an option.

Finally, I decided I needed to refactor BitPacket, while learning more in the path, and create a small library, much simpler than struct and much more powerful and fast than the old BitPacket. This is how BitPacket 1.0.0 was born.

CHAPTER

TWO

CONCEPTS

2.1 Packets and fields

packets.

CHAPTER

THREE

THE BASE FIELD

Base abstract class for all BitPacket fields.

API reference: `Field`

The `Field` class is the abstract root class for all other BitPacket classes. Initially, a field only has a name and no value. `Field` subclasses must provide field details, such as the size of the field, the implementation of how the field value will look like, that is, how the field should be built, and other field related details.

3.1 Naming fields

The most simple field accessor is its name. A field name is built upon creation but can be changed at run-time (special care should be taken, though). It is recommended to follow python variable naming when assigning a name to a field. This is because with the `Container` subclass (and its subclasses) fields can be accessed directly as class members.

Note: changing the field name at run-time is not recommended unless you know what you are doing.

3.2 Building and parsing fields

The main purpose of BitPacket is to provide an easy way to represent packets (or, say it another way, data structures). In BitPacket, packets can be built from and to an string of bytes, arrays and streams.

A field subclass, then, needs to provide the following methods:

```
def value():_
```

This method returns the actual value of the field, whatever that is, a number, a string, etc.:

```
def set_value(value):_
```

This method sets a new value to the current field. The value might be a number, a string, etc. depending on the field's type:

```
def size():_
```

This method must return the field's size. Note that some fields are bit-oriented, so the method might return values for different units (basically, bits and bytes):

```
def str_value():_
```

This method must return the text string representation for the given field:

```
def str_hex_value():_
```

This method must return the hexadecimal string representation for the given field. That is, how the field looks like in memory. For example, for a float value, the hexadecimal representation could bbe the bytes forming the IEEE-754 representation:

```
def str_eng_value():_
```

This method must return the text string representation of the result obtained after applying the field's calibration curve. Therefore, it is necessary to call the calibration curve of the field first and then return the result (after applying any extra desired formatting):

```
def _encode(stream):_
```

This method will write the field's value into the given stream (byte or bit oriented):

```
def _decode(stream):_
```

This method will convert the given stream (byte or bit oriented) into the internal field representation.

3.3 Calibration curves

Sometimes a field might need to be expressed in another way, or a calculation might be necessary to determine the final value of the field. Consider, for example, a numeric field that represents a temperature with a 16-bit precision and covers a range from 0 to 50 celsius degrees. The calibration curve comes in handy by letting the user to specify this conversion function:

```
def set_calibration_curve(self, curve):_
```

This method lets the user to provide a function to compute the calibration curve. The function must be unary taking the field's value as it's argument and computing (using a the desired conversion function) a result.

For the temperature example mentioned above, the calibration function could be something like:

```
def temp_conv(x):
    return (x / 65535.0) * 50.0
```

CHAPTER

FOUR

SINGLE FIELDS

4.1 Bit fields

A field to represent single bit fields.

API reference: `BitField`

A packet might be formed by multiple fields that can be single bit fields, numeric fields, etc. Sometimes, byte-aligned fields are also formed by bit fields internally. The purpose of `BitField` is to provide these single bit fields.

For example, the first byte of the IP header is formed by two nibbles:

version	hlen
4 bits	4 bits

The first nibble, *version*, can be constructed by the following piece of code:

```
>>> bf = BitField("version", 4, 15)
>>> print bf
(version = 0x0F)
```

That is, a 4 bits field with a default, optional, value 15.

4.1.1 Booleans

A bit field for representing a boolean type.

API reference: `Boolean`

A `Boolean` is a `BitField` with a single bit. So only True or False can be set.

```
>>> b = Boolean("Option", True)
>>> print b
(Option = True)
```

There a couple of helper function to enable or disable the field value:

```
>>> b.disable()
>>> print b
(Option = False)

>>> b.enable()
>>> print b
(Option = True)
```

4.1.2 Flags

A bit field for representing a flag.

API reference: `Flag`

A `Flag` is a `BitField` with a single bit. As in the `Boolean` type, it can have only two values Active and Inactive. Basically, it is just a helper class to print Active or Inactive instead of True or False. The same behavior could be achieved by using the `Boolean` type.

```
>>> f = Flag("Flag", Flag.Active)
>>> print f
(Flag = Active)
```

There a couple of helper function to activate or deactivate the flag:

```
>>> f.deactivate()
>>> print f
(Flag = Inactive)

>>> f.activate()
>>> print f
(Flag = Active)
```

4.1.3 Masks

A field to represent bit masks.

API reference: `Mask`

A bit mask is a field where each bit has a different meaning. Multiples bits can be set or unset at once, but each one will represent a single thing. Bit masks are commonly used to define a set of options.

For example, we can define a 2-bit mask of whether our packet includes audio, video or both:

audio/video
2 bits

This can be simply constructed by the following code:

```
>>> m = Mask("audio/video", 0, AUDIO = 0x01, VIDEO = 0x02)
>>> m.mask(m.AUDIO | m.VIDEO)
>>> print m
(audio/video = 0x03)
```

We can also unmask a single option:

```
>>> m.unmask(m.AUDIO)
>>> print m
(audio/video = 0x02)
```

Or unmask all of them with *set_value*:

```
>>> m.set_value(0)
>>> print m
(audio/video = 0x00)
```

4.2 Numeric fields

An abstract class for numeric values (integer or real).

API reference: `Value`

This is the base class for numeric fields. Internally, it uses Python's struct module to define the numeric value size and the byte order (little-endian or big-endian).

The following code creates an 32-bit unsigned integer with a little-endian byte ordering.

```
>>> v32 = Value("value", "<I", 67436735)
>>> print v32
(value = 67436735)
```

Fortunately, BitPacket already defines most of numeric values that are commonly used.

4.2.1 Integer fields

This module provides classes to define signed and unsigned integers bit fields, from 8-bit to 64-bit.

Signed and unsigned

Multiple signed and unsigned integer classes are available. It is, for example, very easy to create a new 16-bit signed integer bit field:

```
>>> value = Int16("int16", -1345)
>>> print value
(int16 = -1345)
```

or a 16-bit unsigned one:

```
>>> value = UInt16("uint16", 0x8000)
>>> print value
(uint16 = 32768)
```

Helper classes

Default helper classes use network byte order (big-endian):

Size	Unsigned	Signed
8	`UInt8`	`Int8`
16	`UInt16`	`Int16`
32	`UInt32`	`Int32`
64	`UInt64`	`Int64`

Little-endian helper classes:

Size	Unsigned	Signed
8	`UInt8LE`	`Int8LE`
16	`UInt16LE`	`Int16LE`
32	`UInt32LE`	`Int32LE`
64	`UInt64LE`	`Int64LE`

Big-endian helper classes:

Size	Unsigned	Signed
8	`UInt8BE`	`Int8BE`
16	`UInt16BE`	`Int16BE`
32	`UInt32BE`	`Int32BE`
64	`UInt64BE`	`Int64BE`

4.2.2 Real fields

This module provides classes to define float (32-bit) and double (64-bit) fields.

Floats and doubles

A float value can be easily created with the `Float` class:

```
>>> value = Float("f", 1.967834)
>>> print value
(f = 1.96783399582)
```

Some times, it is also useful to see the hexadecimal value that forms this float number.

```
>>> print value.str_hex_value()
0x3FFBE1FC
```

The same might be applied for doubles:

```
>>> value = Double("f", 0.0087552)
>>> print value
(f = 0.0087552)
```

Helper classes

Default helper classes use network byte order (big-endian):

Size	Class
32	`Float`
64	`Double`

Endianness helper classes:

Size	Little-Endian	Big-Endian
32	`FloatLE`	`FloatBE`
64	`DoubleLE`	`DoubleBE`

4.3 String and text fields

4.3.1 String of characters

Fields to store a string of characters.

> **API reference**: `String`

A `String` field lets you store a string of characters of any size. The length of the string needs to be specified at creation time.

The simplest case is a string with a fixed length. In the next example we create a string field with sixteen characters:

```
>>> data = String("data", 16)
>>> data.set_value("this is a string")
>>> print data
(data = 0x7468697320697320612073747269 6E67)
```

As usual, we can easily get back the original string:

```
>>> "".join(data.value())
'this is a string'
```

Note that, above, "print data" returns a human-readable string with hexadecimal values and "data.value()" returns the actual string.

For convenience, there is a `Text` field that inherits from `String` and always returns the actual string.

```
>>> text = Text("text", 16)
>>> text.set_value("this is a string")
>>> print text
(text = this is a string)
```

So, `Text` is supposedly to be used with only text while `String` is to be used with any character.

4.3.2 Unpacking strings

Instead of a fixed length, we can specify a length function that will tell us what length the string should have. In the following structure we create a `Structure` with a numeric "length" field and a string of unknown size.

length	string
1 byte	length

BitPacket already provides a helper the `Data` field which contains a length field and a string.

```
>>> packet = Structure("string")
>>> l = UInt8("length")
>>> s = String("data", lambda root: root["length"])
>>> packet.append(l)
>>> packet.append(s)
```

If we print the initial contents of the structure we can see that the length is 0 and that we still have an empty string.

```
>>> print packet
(string =
  (length = 0)
  (data = ))
```

We can try to assign a value to our string directly and see what happens:

```
>>> try:
...   s.set_value("this is a string")
... except ValueError as err:
...   print "Error: %s" % err
Error: Data length must be 0 (16 given)
```

A *ValueError* exception is raised indicating that string length should be 0. This is because the "length" field has not been assigned a value yet.

Finally, we can provide to the structure all the necessary information, for example, in an array:

```
>>> data = array.array("B", [0x10, 0x74, 0x68, 0x69, 0x73, 0x20,
...                          0x69, 0x73, 0x20, 0x61, 0x20, 0x73,
...                          0x74, 0x72, 0x69, 0x6E, 0x67])
>>> packet.set_array(data)
>>> print packet
(string =
  (length = 16)
  (data = 0x74686973206973206120737472696E67))
```

4.4 Meta fields

4.4.1 MetaField field

API reference: `MetaField`

CHAPTER

FIVE

CONTAINER FIELDS

Abstract root class for field containers.

API reference: `Container`

Packets can be seen as field containers. That is, a packet is formed by a sequence of fields. The `Container` class provides this vision. A `Container` is also a `Field` itself. Therefore, a `Container` might also accomodate other Containers.

Consider the first three bytes of the IP header:

version	hlen	tos	length
4 bits	4 bits	1 byte	2 bytes

We can see the IP header as a `Container` with a sub-`Container` holding two bit fields (*version* and *hlen*) and two additional fields (*tos* and *length*).

The `Container` class is just an abstract class that allows adding fields. That is, it provides the base methods to build Containers.

5.1 Bit fields containers

A container implementation for bit fields.

API reference: `BitStructure`

The `BitStructure` class must be used, in conjunction with `BitField`, to create byte-aligned fields formed, internally, by bit fields.

It is really important to understand that BitPacket is byte oriented, therefore, a `BitStructure` must be byte-aligned.

Consider the first byte of an IP header packet:

version	hlen
4 bits	4 bits

This packet can be constructed as:

```
>>> ip = BitStructure("IP")
```

The line above creates an empty structure named *IP*. Now, we need to add fields to it. As BitStructure is a `Container` subclass the `Container.append()` function can be used:

```
>>> ip.append(BitField("version", 4, 0x0E))
>>> ip.append(BitField("hlen", 4, 0x0C))
>>> print ip
(IP =
```

```
  (version = 0x0E)
  (hlen = 0x0C))
```

Note that the size of a `BitStructure` is returned in bytes. Remember that the purpose of a `BitStructure` is to create a byte-aligned value that is built internally with bits:

```
>>> ip.size()
1
```

5.1.1 Accessing fields

BitStructure fields can be obtained as in a dictionary, and as in any `Container` subclass. Following the last example:

```
>>> ip["version"]
14
>>> ip["hlen"]
12
```

5.1.2 Packing bit structures

As with any BitPacket field, packing a BitStructure is really simple. Considering the IP header exampe above we can easily create an array of bytes with the contents of the structure:

```
>>> ip_data = array.array("B")
>>> ip.array(ip_data)
>>> print ip_data
array('B', [236])
```

Or also create a string of bytes from it:

```
>>> ip.bytes()
'\xec'
```

5.1.3 Unpacking bit structures

To be able to unpack an integer value or a string of bytes into a BitStructure, we only need to create the desired structure and assign data to it.

```
>>> bs = BitStructure("mypacket")
>>> bs.append(BitField("id", 8))
>>> bs.append(BitField("address", 32))
>>> print bs
(mypacket =
  (id = 0x00)
  (address = 0x00000000))
```

So, now we can unpack the following array of bytes:

```
>>> data = array.array("B", [0x38, 0x87, 0x34, 0x21, 0x40])
```

into our previously defined structure:

```
>>> bs.set_array(data)
>>> print bs
(mypacket =
```

```
  (id = 0x38)
  (address = 0x87342140))
```

Also, new data can also be unpacked (old data will be lost):

```
>>> data = array.array("B", [0x45, 0x67, 0x24, 0x98, 0xFB])
>>> bs.set_array(data)
>>> print bs
(mypacket =
  (id = 0x45)
  (address = 0x672498FB))
```

5.2 Structures

A container implementation for fields of different types.

API reference: `Structure`

The `Structure` class provides a byte-aligned `Container` implementation. This means that all the fields added to a `Structure` should be byte-aligned. This does not mean that a `BitField` can not be added, but if added, it should be added within a `BitStructure`. This is because bit and byte processing is done differently, and that's why `BitStructure` was created.

Consider the first three bytes of the IP header:

version	hlen	tos	length
4 bits	4 bits	1 byte	2 bytes

For simplicity, we can create only a `Structure` with the last two fields, *tos* and *length*.

```
>>> ip = Structure("IP")
```

The line above creates an empty packet named 'IP'. Now, we can add the two fields to it with an initial value:

```
>>> ip.append(UInt8("tos", 3))
>>> ip.append(UInt16("length", 146))
>>> print ip
(IP =
  (tos = 3)
  (length = 146))
```

5.2.1 Accessing fields

`Structure` fields, as in any other `Container`, can be obtained like in a dictionary, that is, by its name. Following the last example:

```
>>> ip["tos"]
3
>>> ip["length"]
146
```

5.2.2 Packing structures

As with any other BitPacket field, packing a `Structure` is really simple. Considering the IP header exampe above, we can easily create an array of bytes with the contents of the structure:

```
>>> ip_data = array.array("B")
>>> ip.array(ip_data)
>>> print ip_data
array('B', [3, 0, 146])
```

Or also create a string of bytes from it:

```
>>> ip.bytes()
'\x03\x00\x92'
```

5.2.3 Unpacking structures

To be able to unpack an integer value or a string of bytes into a Structure, we only need to create the desired packet and assign data to it.

```
>>> bs = Structure("mypacket")
>>> bs.append(UInt8("id"))
>>> bs.append(UInt32("address"))
>>> print bs
(mypacket =
  (id = 0)
  (address = 0))
```

So, now we can unpack the following array of bytes:

```
>>> data = array.array("B", [0x38, 0x87, 0x34, 0x21, 0x40])
```

into our previously defined structure:

```
>>> bs.set_bytes(data.tostring())
>>> print bs
(mypacket =
  (id = 56)
  (address = 2268340544))
```

5.2.4 Structures as classes

An interesting use of structures is to subclass them to create our own reusable ones. As an example, we could create the structure defined in the previous section as a new class:

```
>>> class MyStructure(Structure):
...     def __init__(self, id = 0, address = 0):
...         Structure.__init__(self, "mystructure")
...         self.append(UInt8("id", id))
...         self.append(UInt32("address", address))
...
...     def id(self):
...         return self["id"]
...
...     def address(self):
...         return self["address"]
...
>>> ms = MyStructure(0x33, 0x50607080)
>>> print ms
(mystructure =
  (id = 51)
  (address = 1348497536))
```

We can now use the accessors of our class to print its content:

```
>>> print "0x%X" % ms.id()
0x33
>>> print "0x%X" % ms.address()
0x50607080
```

5.2.5 Structure based fields

Arrays

An structure for fields of the same type.

API reference: `Array`

Sometimes we need to create packets that have a number of repeated fields in it. Normally, these kind of packets have a counter field indicating the number of repeated fields after it.

An `Array` is a subclass of `Structure`. Initially, it contains a length field, which is the one that will indicate how many fields the array holds. The type of the length field is specified in the *Array* constructor.

count	id	address
1 byte	1 byte	4 bytes
	count times	

In order to create an `Array` for the depicted packet above, we can define the base type of the fields (all of the same type) that this array will contain. We will create a *MyStructure* class that contains two fields, *id* and *address*.

```
>>> class MyStructure(Structure):
...     def __init__(self, name = "mystructure", id = 0, address = 0):
...         Structure.__init__(self, name)
...         self.append(UInt8("id", id))
...         self.append(UInt32("address", address))
```

Now, we can define an `Array` that contains the default counter field of our desired name and size and a single argument function that tells how to create the array fields.

```
>>> packet = Array("mypacket", UInt8("counter"),
...                lambda root: MyStructure())
```

As a second argument to the *Array* constructor, we specify the field that specifies how many elements the array contains. As the third argument, we have provided how to create the array elements. The anonymous function takes an argument which is the top-level root `Container` that the Array belongs to.

So, let's try to unpack some data and see what happens:

```
>>> data = array.array("B", [0x01, 0x54, 0x10, 0x20, 0x30, 0x40])
>>> packet.set_array(data)
>>> print packet
(mypacket =
  (counter = 1)
  (0 =
    (id = 84)
    (address = 270544960)))
```

At this point the array contains one field of type *MyStructure* as it has unpacked the given array, seen that the *counter* field had value 1 and therefore read one *MyStructure* field. It is worth noting that the fields added to an `Array` are automatically named in a zero-based scheme. That is, to access the first *address* field value we could do:

```
>>> packet["0.address"]
270544960
```

We can also easily add some data to the array. Consider again our packet:

```
>>> packet = Array("mypacket", UInt8("counter"),
...                lambda root: MyStructure())
```

Adding a *MyStructure* field is as easy as adding a field to any other *Structure*:

```
>>> packet.append(MyStructure("foo", 54, 98812383))
>>> print packet
(mypacket =
  (counter = 1)
  (0 =
    (id = 54)
    (address = 98812383)))
```

It is interesting to see that if we add something else that is not a *MyStructure*, a *TypeError* exception will be raised notifying about the problem:

```
>>> try:
...   packet.append(UInt8("wrong", 12))
... except TypeError as err:
...   print "Error: %s" % err
Error: Invalid field type for array 'mypacket' (expected <class 'MyStructure'>, got <class 'BitPacket
```

Accessing fields

`Array` fields, as in any other `Container` can be obtained like in a dictionary, that is, by its name. Following the last example:

```
>>> packet["0.id"]
54
```

Note that the *id* field could be another array instead of a numeric field, thus we could access further by using the dot field separator (.).

Complex arrays

We can also build more complex packets, such as the one below, where we have one `Array` inside another.

count1	id	count2	address
1 byte	1 byte	1 byte	4 bytes
			count2 times
	count1 times		

We will first create a structure for the list of addresses. It will contain the *count2* counter and an `Array` whose number of elements is provided by *count2* and that will be filled with 32-bit unsigned integers.

```
>>> class AddressList(Structure):
...     def __init__(self):
...         Structure.__init__(self, "addresslist")
...         self.append(UInt8("id"))
...         self.append(Array("address",
...                           UInt8("count2"),
...                           lambda root: UInt32("value")))
```

Now, we can build our packet as an structure with the *count1* counter and an `Array` whose number of elements is provided by *count1* and that will be filled by address lists (that, remember, already has another `Array`).

```
>>> s = Array("mypacket",
...           UInt8("count1"),
...           lambda root: AddressList())
```

So, let's try to set some data to this packet. As we have seen before with the simplest case, data should be propagated and `Array` meta properties will be used to build the desired fields.

```
>>> s.set_array(array.array("B", [0x02, # count1
...                               0x01, # id (1)
...                               0x01, # count2 (1)
...                               0x01, 0x02, 0x03, 0x04,
...                               0x02, # id (2)
...                               0x02, # count2 (2)
...                               0x05, 0x06, 0x07, 0x08,
...                               0x09, 0x0A, 0x0B, 0x0C]))
>>> print s
(mypacket =
  (count1 = 2)
  (0 =
    (id = 1)
    (address =
      (count2 = 1)
      (0 = 16909060)))
  (1 =
    (id = 2)
    (address =
      (count2 = 2)
      (0 = 84281096)
      (1 = 151653132))))
```

It works! As we see, our packet consists of a *mystructure* that contains two *AddressList* fields. The first one with a single address and the second with two.

Data

An structure that holds a `String` and its length.

API reference: `Data`

A `Data` field lets you store a string of characters (divided by words) and keeps its length in another field. By default, the size of a word is 1 byte. Basically, `Data` is a `Structure` with two fields in this order: length and data (internally named *Data*). The length is a numeric field and specifies how many words the *Data* field contains.

In the next example we create a `Data` field with six characters and a length field of 1 byte (thus, a maximum of 255 characters can be hold):

```
>>> data = Data("data", UInt8("Length"));
>>> data.set_value("abcdef")
>>> print data
(data =
  (Length = 6)
  (Data = 0x616263646566))
```

We can easily get back the six characters by creating the string again:

```
>>> "".join(data["Data"])
'abcdef'
```

Note that, above, "print data" returns a human-readable string with hexadecimal values and "data.value()" returns the actual string.

Word sizes

The length field tells us how many words the *Data* field contains. Above, we just saw an example with the default word size of 1. But a 12 character string and a word size of 4, gives us 3 words.

```
>>> data = Data("data", UInt8("Length"), 4);
>>> data.set_value("abcdefghigkl")
>>> print data
(data =
  (Length = 3)
  (Data = 0x616263646566676869676B6C))
```

Note that data length needs to be a multiple of the word size.

It is also possible to obtain the word size from the value of another field.

WSize	Length	Data
1 byte	1 byte	Length * WSize

Thus, instead of passing a number to the word size parameter, we pass it a single-argument function. The single-argument, as in all other BitPacket fields, is the top-level root `Container` field where the *Data* field belongs to.

```
>>> packet = Structure("packet")
>>> packet.append(UInt8("WSize"))
>>> data = Data("data", UInt8("Length"), lambda root: root["WSize"]);
>>> packet.append(data)

>>> buffer = array.array("B", [2, 3, 40, 55, 22, 45, 34, 89])
>>> packet.set_array(buffer)
>>> print packet
(packet =
  (WSize = 2)
  (data =
    (Length = 3)
    (Data = 0x2837162D2259)))
```

CHAPTER

SIX

WRITERS

API reference: `Writer` **API reference**: `WriterTextBasic` **API reference**: `WriterTextTable` **API reference**: `WriterTextXML` **API reference**: `WriterGtkTreeView` **API reference**: `WriterGtkTreeModel`

CHAPTER

SEVEN

API REFERENCE

7.1 Field

class **Field**(*name*)

Abstract root class for all other BitPacket classes. Initially, a field only has a name and no value. `Field` subclasses must provide field details, such as the size of the field, the implementation of how the field value will look like, that is, how the field should be built, and other field related details.

Initialize the field with the given *name*. And identity (returning the field's value) calibration curve is set by default.

array(*array*)

Returns the given *array* appended with the field byte representation to it.

bytes()

Returns a string of bytes representing this field.

calibration_curve()

Returns the calibration curve function.

eng_value()

Returns the engineering value of this field. The engineering value is the result of applying a calibration curve to the value of this field. Some fields might represent temperatures, angles, etc. that need to be converted from its digital form to its analog form. This function will return the value after the conversion is done.

fields()

Returns a list of the children of this field. An empty list is returned if the field does not have any children.

hex_value()

Returns the hexadecimal integer representation of this field. That is, the bytes forming this field in its integer representation.

name()

Returns the name of the field.

parent()

Returns the parent of this field, or None if the field is not part of any other field.

root()

Returns the root of this field. The root is the top level container that this field belongs to, if any. If the field is not part of any other field the root is the field itself.

set_array(*array*)

Sets the given *array* bytes to the field. This function does the same as calling *set_bytes* with the bytes of the array.

set_bytes(*bytes*)
Sets a string of bytes to the field.

set_calibration_curve(*curve*)
Sets the calibration curve to be applied to this field in order to obtain a desired conversion. Some fields might represent temperatures, angles, etc. that need to be converted from its digital form to its analog form. The calibration curve provides the functionality to perform this conversion.

set_stream(*stream*)
Sets this field with the contents of the given stream. Note that only the bytes necessary for this field will be obtained from the stream. This means that the stream cursor will only advance as many bytes as the size of this field.

set_value(*value*)
Sets a new *value* to the field.

size()
Returns the size of the field.

str_eng_value()
Returns a human-readable representation of the engineering value. This function will first calculate the engineering value (by applying the calibration curve) and will return the string representation of it.

str_hex_value()
Returns a human-readable representation of the hexadecimal value of this field. Note that the type of the field can be a float, integer, etc. This is the real representation (in memory) of the value.

str_value()
Returns a human-readable representation of the value of this field. Note that the type of the field can be a float, integer, etc. So, the representation might be different for each type.

stream(*stream*)
Fill the given byte stream with the contents of this field.

value()
Returns the value of the field.

7.1.1 BitField

class BitField(*name*, *size*, *value=0*)
Bases: `BitPacket.Field.Field`

This class represents bit fields to be used by `BitStructure` in order to build byte-aligned fields. Remember that BitPacket only works with byte-aligned fields, so it is not possible to create mixed (bit and byte) fields, that's why `BitField` can only be used inside a `BitStructure`.

Initialize the field with the given *name* and *size* (in bits). By default the field's value will be initialized to 0 or to *value* if specified.

set_value(*value*)
Sets a new unsigned integer *value* to the field.

size()
Returns the size of the field in bits.

str_eng_value()
Returns a human-readable representation of the engineering value. This function will first calculate the engineering value (by applying the calibration curve) and will return the string representation of it. In case of bit fields the representation is an hexadecimal value.

str_hex_value()
Returns a human-readable representation of the hexadecimal value of this field. This will return the same as *str_value*.

str_value()
Returns a human-readable representation of the value of this field. In case of bit fields the representation is an hexadecimal value.

value()
Returns the value of this field. As single bit fields do not have a concrete type (signed integers, float...) this will return the unsigned integer representation of this field.

Boolean

class Boolean(*name*, *value=False*)
Bases: BitPacket.BitField.BitField

This class represents a boolean field. It is a convenient class to represent True and False values.

Initialize the field with the given *name*. The default value is False.

disable()
Set the value to False.

enable()
Set the value to True.

str_eng_value()
Returns a text string with True or False.

str_value()
Returns a text string with True or False.

Flag

class Flag(*name*, *value=0*)
Bases: BitPacket.BitField.BitField

This class represents a boolean field. It is a convenient class to represent True and False values.

Initialize the field with the given *name*. The default value is Inactive.

Active = 1

Inactive = 0

activate()
Activate the flag.

deactivate()
Deactivate the flag.

str_eng_value()
Returns a text string with Active or Inactive.

str_value()
Returns a text string with Active or Inactive.

Mask

class Mask (*name*, *value*, ***kwargs*)
Bases: BitPacket.BitField.BitField

This class represents a bit mask field. Each bit has a unique meaning. Bit masks can be set, unset and tested.

Initialize the field with the given *name* and *value*. The list of all masks and their values are be given in *kwargs* with an arbitrary number of arguments (FLAG_1 = 0x01, FLAG_2 = 0x02 ...).

is_set (*mask*)
Tests whether the given bit *mask* is set. That is, all the given bits must be set.

mask (*mask*)
Mask the given bits in *mask*. The current bits will be kept and the new ones will be additionally masked.

str_eng_value ()
Returns a text string with all the bit mask names that are set. Bit masks names are separated by |.

str_value ()
Returns a human-readable representation of the hexadecimal value of this field.

unmask (*mask*)
Unmask the given bits in *mask*. The current bits will be kept and the new ones will be additionally unmasked.

7.1.2 Container

class Container (*name*)
Bases: BitPacket.Field.Field

This is an abstrat class to create containers. A `Container` is just a field that might contain a sequence of fields (that can be containers as well), thus forming a bigger field.

Initialize the `Container` with the given *name*. By default, it does not contain any fields.

append (*field*)
Appends a new *field* into the `Container`. A *NameError* exception will be raised if a field with the same name is already in the container.

field (*name*)
Returns the field identified by *name*. *name* accepts a dot (.) separator to indicate sub-fields of a container (multiple separators are allowed). If the field does not exist a *KeyError* exception is raised.

fields ()
Returns the (ordered) list of fields of this `Container`.

keys ()
Returns the list of fields' names recursively (i.e. if fields are also containers). In case one or more of the fields are containers, its fields will be suffixed with a dot separator. As an example, "a.b.c" is the key for a field *c* inside a *b* container which is also inside a root *a* container.

reset ()
Remove all the fields from this `Container`.

size ()
Returns the size of the field in bytes. That is, the sum of all byte sizes of the fields in this `Container`.

BitStructure

class **BitStructure**(*name*)
Bases: BitPacket.Container.Container

This class represents an structure formed by bit fields. The resulting structure must be byte-aligned and is to be used with other BitPacket types.

Initialize the bit structure field with the given *name*. By default an structure field does not contain any fields.

size()
Returns the size of the field in bytes. This function will add all the bit field sizes in order to calculate the byte size of the container.

Structure

class **Structure**(*name*)
Bases: BitPacket.Container.Container

This class provides a byte-aligned Container implementation. All the fields added to it should be byte-aligned.

Initialize the structure with the given *name*. By default, it does not contain any fields.

Array

class **Array**(*name*, *lengthfield*, *fieldtype*)
Bases: BitPacket.Structure.Structure

An Array is an structure for fields of the same type. It contains a length field to count the number of elements that the array holds. After the length field, all the rest of fields (of the same type) are stored.

Initialize the array with the given *name*, a *lengthfield* for the counter field and *fieldtype* for a single argument function that will return a new array member. The single argument is a reference to the top-level root Container field where the array belongs to.

append(*field*)
Appends a new *field* to the array. The given *field* must be of the same type specified when creating the array, otherwise a *TypeError* exception is raised.

It is important to note that the given *field* name will be changed by its index in the array.

Data

class **Data**(*name*, *lengthfield*, *wordsize=1*)
Bases: BitPacket.Structure.Structure

This class lets you store strings of characters (divided by words) and also provides a field to hold its length. It is a Structure with two fields: length and data (internally created with name *Data*). The length is a numeric field and specifies how many words the *Data* field contains. The *Data* field is internally a String.

Initialize the field with the given *name* and a *lengthfield*. The *lengthfield* must be a numeric field instance. *wordsize* specifies how many bytes a word contains and it can be a numeric value or a unary function that knows where to get the word size, it has a default value of 1. So, the total length in bytes of a Data field is the length field multiplied by the word size. If *wordsize* is a function, it takes the top-level root Container field as a single argument. This way, it is possible to provide a word size that depends on the value of another field.

set_value(*value*)
Sets a new string to the *Data* field. The given string length must be a mutliple of the word size and must fit in the length field (i.e. 300 characters are too long if the length field is `UInt8`, as only 255 characters fit), otherwise a *ValueError* exception will be raised.

value()
Returns the value of the *Data* field as a string.

7.1.3 MetaField

class MetaField(*name*, *fieldfunc*)
Bases: `BitPacket.Field.Field`

type()

7.1.4 String

class String(*name*, *length*)
Bases: `BitPacket.Field.Field`

A `String` field lets you store a string of characters of any size. Usually, to unpack a string we need to extract the length of the string from another field, in which case we need to specify a function that will know where to get the length from. However, it is also possible to specify a fixed length string.

Initialize the string field with a *name* and a *length*. *length* can be a fixed number or a single arument function that returns the length of the string. The single argument is a reference to the top-level root `Container` field where the string belongs to. A possible function could be:

```
lambda root: root["Length"]
```

where we get the length of the string from a *Length* field.

set_value(*data*)
Sets a new string of characters to the field.

size()
Returns the size in bytes of the string.

str_eng_value()
This is equivalent of calling *str_value ()*.

str_hex_value()
This is equivalent of calling *str_value ()*.

str_value()
Returns a text string with the hexadecimal value of each character of the string. A prefix of 0x is added. So, for "hello", "0x68656C6C6F" would be returned.

value()
Returns the string of characters.

class Text(*name*, *length*)
Bases: `BitPacket.String.String`

`Text` is basically a `String` but conceived to be used with only text strings. It does not perform any check on the data. It simply returns the internal string (which should be text) instead of generating a text string with the hexadecimal values of the string.

str_eng_value()
This is equivalent of calling *str_value ()*.

str_hex_value()
This is equivalent of calling *str_value ()*.

str_value()
Returns the text string.

7.1.5 Value

class Value(*name*, *format*, *value*)
Bases: BitPacket.Field.Field

This is the base class for numeric fields. Internally, it uses Python's struct module to define the numeric value size and the byte order (little-endian or big-endian).

Initialize the field with the given *name* and *value*. The *format* is a string conforming the Python's struct module format strings.

hex_value()
Returns the hexadecimal integer representation of this field. That is, the bytes forming this field in its integer representation. This will vary depending on the field's endiannes and size, so UInt16LE will return a different hexadecimal value than UInt16BE for the same number.

set_value(*value*)
Sets the new numeric *value* to this field. The value must fit in this field, otherwise an exception is raised.

size()
Returns the size in bytes of this field.

str_eng_value()
Returns a human-readable representation of the engineering value. This function will first calculate the engineering value (by applying the calibration curve) and will return the string representation of it.

str_hex_value()
Returns a human-readable representation of the hexadecimal representation of this field. This internally uses *hex_value()*.

str_value()
Returns a human-readable representation of the numeric value of this field.

value()
Returns the numeric value of this field.

Integer

Int8
alias of Int8BE

UInt8
alias of UInt8BE

Int16
alias of Int16BE

UInt16
alias of UInt16BE

Int32
alias of Int32BE

UInt32
alias of UInt32BE

Int64
alias of Int64BE

UInt64
alias of UInt64BE

class Int8LE(*name*, *value=0*)
Bases: BitPacket.Value.Value

class UInt8LE(*name*, *value=0*)
Bases: BitPacket.Value.Value

class Int8BE(*name*, *value=0*)
Bases: BitPacket.Value.Value

class UInt8BE(*name*, *value=0*)
Bases: BitPacket.Value.Value

class Int16LE(*name*, *value=0*)
Bases: BitPacket.Value.Value

class UInt16LE(*name*, *value=0*)
Bases: BitPacket.Value.Value

class Int16BE(*name*, *value=0*)
Bases: BitPacket.Value.Value

class UInt16BE(*name*, *value=0*)
Bases: BitPacket.Value.Value

class Int32LE(*name*, *value=0*)
Bases: BitPacket.Value.Value

class UInt32LE(*name*, *value=0*)
Bases: BitPacket.Value.Value

class Int32BE(*name*, *value=0*)
Bases: BitPacket.Value.Value

class UInt32BE(*name*, *value=0*)
Bases: BitPacket.Value.Value

class Int64LE(*name*, *value=0*)
Bases: BitPacket.Value.Value

class UInt64LE(*name*, *value=0*)
Bases: BitPacket.Value.Value

class Int64BE(*name*, *value=0*)
Bases: BitPacket.Value.Value

class UInt64BE(*name*, *value=0*)
Bases: BitPacket.Value.Value

Real

Float
alias of FloatBE

Double
alias of DoubleBE

class FloatLE(*name*, *value=0.0*)
Bases: BitPacket.Value.Value

class FloatBE(*name*, *value=0.0*)
Bases: BitPacket.Value.Value

class DoubleLE(*name*, *value=0.0*)
Bases: BitPacket.Value.Value

class DoubleBE(*name*, *value=0.0*)
Bases: BitPacket.Value.Value

7.2 Writer

class Writer(*config=<BitPacket.writers.WriterConfig.WriterConfig object at 0x7f3689c2c510>*)
This the abstract class for all bit fields sub-classes. All bit fields must inherit from this class and implement the non-implemented methods in it.

config()

end_block(*field*, *userdata=None*)

level()

start_block(*field*, *userdata=None*)

write(*field*, *userdata=None*)
Returns the name of the field.

7.2.1 WriterStream

class WriterTextStream(*stream*, *config=<BitPacket.writers.WriterTextStreamConfig.WriterTextStreamConfig object at 0x7f3689c2cb90>*)
Bases: BitPacket.writers.Writer.Writer

indent()

indentation()

stream()

WriterTextBasic

class WriterTextBasic(*stream*, *config=<BitPacket.writers.WriterTextStreamConfig.WriterTextStreamConfig object at 0x7f3689c2cb90>*)
Bases: BitPacket.writers.WriterTextStream.WriterTextStream

end_block(*field*, *userdata=None*)

start_block(*field*, *userdata=None*)

write(*field*, *userdata=None*)

WriterTextTable

class WriterTextTable(*stream*, *config=<BitPacket.writers.WriterTextTableConfig.WriterTextTableConfig object at 0x7f3689c34110>*)

Bases: BitPacket.writers.WriterTextStream.WriterTextStream

write(*field*, *userdata=None*)

WriterTextXML

class WriterTextXML(*stream*, *config=<BitPacket.writers.WriterTextStreamConfig.WriterTextStreamConfig object at 0x7f3689c2cb90>*)

Bases: BitPacket.writers.WriterTextStream.WriterTextStream

end_block(*field*, *userdata=None*)

start_block(*field*, *userdata=None*)

write(*field*, *userdata=None*)

7.2.2 WriterGtk

WriterGtkTreeView

class WriterGtkTreeView

Bases: BitPacket.writers.Writer.Writer

view()

write(*field*, *userdata=None*)

WriterGtkTreeModel

class WriterGtkTreeModel(*field*)

Bases: gtk.GenericTreeModel

CLASS_COLUMN = 1

COLUMN_NAMES = ['Name', 'Class', 'Size', 'Raw', 'Hexadecimal', 'Engineering']

COLUMN_TYPES = (<GType gchararray (64)>, <GType gchararray (64)>, <GType gint (24)>, <GType gchararray (64)>, <

ENG_VALUE_COLUMN = 5

HEX_VALUE_COLUMN = 4

NAME_COLUMN = 0

NUM_COLUMNS = 6

RAW_VALUE_COLUMN = 3

SIZE_COLUMN = 2

on_get_column_type(*n*)

on_get_flags()

on_get_iter(*path*)

on_get_n_columns()

on_get_path(*rowref*)

on_get_value(*rowref*, *column*)

on_iter_children(*parent*)

on_iter_has_child(*rowref*)

on_iter_n_children(*rowref*)

on_iter_next(*rowref*)

on_iter_nth_child(*parent*, *n*)

on_iter_parent(*child*)

7.3 WriterConfig

class WriterConfig(***kwargs*)

Bases: `object`

set_config(***kwargs*)

7.3.1 WriterStreamConfig

class WriterTextStreamConfig(***kwargs*)

Bases: `BitPacket.writers.WriterConfig.WriterConfig`

WriterTextTableConfig

class WriterTextTableConfig(***kwargs*)

Bases: `BitPacket.writers.WriterTextStreamConfig.WriterTextStreamConfig`

CHAPTER

EIGHT

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

b

A

B

C

D

E

F

H

I

V

W