

---

# **Bdec binary specifications**

*Release 0.7.2*

**Henry Ludemann**

July 10, 2011



# CONTENTS

<b>1</b>	<b>Binary file formats</b>	<b>1</b>
1.1	Why specifications are needed . . . . .	1
1.2	Manually created decoders . . . . .	1
1.3	Existing binary specifications . . . . .	1
<b>2</b>	<b>Tutorial</b>	<b>3</b>
2.1	Preparation . . . . .	3
2.2	Getting started . . . . .	3
2.3	Writing the specification . . . . .	4
2.4	Updating the specification . . . . .	4
2.5	Simplifying the specification using named references . . . . .	6
2.6	Refining the specification . . . . .	6
2.7	Using the specification . . . . .	8
2.8	Where to go from here . . . . .	8
<b>3</b>	<b>Bdec specifications</b>	<b>11</b>
3.1	Xml specification format . . . . .	11
3.2	Field entries . . . . .	12
3.3	Sequence entries . . . . .	14
3.4	Choice entries . . . . .	15
3.5	SequenceOf entries . . . . .	16
3.6	Reference entries . . . . .	17
3.7	Expressions . . . . .	18
3.8	Constraints . . . . .	21
<b>4</b>	<b>Tips</b>	<b>23</b>
4.1	Hiding entries . . . . .	23
4.2	Named references . . . . .	24
4.3	Optional entries . . . . .	24
4.4	Avoid duplication within choice options . . . . .	25
<b>5</b>	<b>Compiling to C</b>	<b>27</b>
<b>6</b>	<b>Python instance decoder</b>	<b>29</b>



# BINARY FILE FORMATS

Binary files contain data encoded in binary form; that is, a format which is typically easy for a computer to read and write, but contains no textual information that is directly accessible by a human reader.

Some binary formats include jpg pictures, mp3 audio files, or pdf documents.

The `bdec` library allows machine readable *specifications* to be written for existing binary formats, allowing high quality decoders to be automatically generated.

## 1.1 Why specifications are needed

Most binary file formats do not use a standardised format for specifying the data layout in the file; instead, the format specification is usually a human readable document describing what data is present in the file, and the layout in which that data is encoded.

If a file format is not intended for data exchange between programs, often a specification document for the format will not exist; there will instead be a single implementation capable of reading and writing to the file.

## 1.2 Manually created decoders

To create a decoder, a software developer will have to read the specification, and write software capable of loading that data into memory in a form that will be able to be processed. This is a labourous job, fraught with difficulties;

- The specification may be incomplete, inaccurate, or missing.
- Validating all the loaded data is very time consuming, and requires a lot of code. Failure to validate the data can result in the program crashing, potentially allowing malicious data files to execute code through buffer overflows.
- Parts of the specification that aren't commonly found in data files may be coded incorrectly, but not found during testing.
- Writing decoders often involves a large amount of repetition, which can frustrate the developer.

These problems make it difficult to read and write decoders.

## 1.3 Existing binary specifications

There are existing specifications for binary formats, such as `ASN.1` and `CSN.1`. These specifications have the problem that they cannot be retrofitted to existing binary formats.



# TUTORIAL

This tutorial will discuss writing a rudimentary specification for the `png` image file format.

## 2.1 Preparation

When decoding a format, first we need to find a specification for the file type we want to decode. Try searching for 'jpeg specification' or 'png file format') on google or [wotsit](#).

These documents can be very large and difficult to understand; they include information on not only the file format, but also how the data should be interpreted. For the purpose of decoding the file, we only care about how the data is represented on disk, so search through the document until you find the section about the on disk format. Finding the section on the header (the first part of the file) is an excellent place to start.

## 2.2 Getting started

To start, it is a good idea to attempt to decode a small and simple file in the target format. If you have access to an editor capable of creating files in the format, create (and save) an empty document. For this tutorial, I created a small 5x5 white png image.

We start by creating a new xml document, named `png.xml`:

```
<protocol>
  <sequence name="png">
  </sequence>
</protocol>
```

All specifications have the outer 'protocol' element, and the *sequence* is like a C struct (it contains other entries). Now we can try running 'bdecode', giving the specification and the file to decode.

```
bdecode -f white.png png.xml
```

Which gives the results:

```
<png>
</png>
Over 8 bytes undecoded!
hex (8 bytes): 89504e470d0a1a0a
656 bytes undecoded!
```

We have now started to decode the file, which is the first step in writing a working specification!

## 2.3 Writing the specification

We'll start by decoding the header; for example, the png specification has a section 'File structure', where we find out that all png files start with an 8 byte signature.

```
<protocol>
  <sequence name="png">
    <field name="signature" length="64" type="hex" value="0x89504e470d0a1a0a" />
  </sequence>
</protocol>
```

The *field* element is used to represent all data on disk; in this case we say it is an 8 byte data field with an expected value. Note that the length is 64, not 8! This is because all lengths in a specification are in bits, not bytes (so we need to multiply by eight). This decodes to:

```
<png>
  <signature />
</png>
Over 8 bytes undecoded!
hex (8 bytes): 0000000d49484452
592 bytes undecoded!
```

We have now started to decode the file! Files that aren't png will no longer decode, as they won't start with the required signature. Note that the value of the signature field isn't displayed in the output xml; this is because as the value of the field is implied in the specification; if it wasn't that data, decoding would have failed.

## 2.4 Updating the specification

Writing a new specification is an iterative process; ie: continually updating the specification while decoding a sample file. Ideally we'll do this with as few updates of the specification as possible before decoding again, as this will give us quick feedback necessary for not only error checking, but also inspiration to decode 'just a little more...'.

The next part of the png specification defines a 'Chunk' structure, which makes up the rest of the file. Each chunk has a length, a type, data, and a crc.

```
<protocol>
  <sequence name="png">
    <field name="signature" length="64" type="hex" value="0x89504e470d0a1a0a" />
    <sequenceof name="chunks">
      <reference name="unknown chunk" />
    </sequenceof>
  </sequence>

  <common>
    <sequence name="unknown chunk">
      <field name="data length:" length="32" type="integer" />
      <field name="type" length="32" type="hex" />
      <field name="data" length="${data length:} * 8" type="hex" />
      <field name="crc" length="32" type="hex" />
    </sequence>
  </common>
</protocol>
```

A few things to note:

- The *sequenceof* means the referenced 'unknown chunk' is repeated.

- We've put the 'unknown chunk' in the common section, and *reference* it from where it is to be used. It is a good idea to separate logical constructs.
- The 'data' entry is a variable length *field*.
- The name of the 'data length:' field has a trailing ':'. This acts as a hint to hide the output of 'data length:' field, so it will not be displayed.

Re-running the decode, we successfully decode four chunks before we run out of data with the error:

```
...
    </unknown-chunk>
    <unknown-chunk>
png.xml[11]: integer 'data length:' (big endian) - Asked for 32 bits, but only have 0 bits available
```

This is because the *sequenceof* entry doesn't know when to stop decoding; ie: there isn't a count, a length, or an end-entry. From reading the specification, we find that a png file is supposed to end with an 'IEND' chunk. Lets add it!

```
<protocol>
  <sequence name="png">
    <field name="signature" length="64" type="hex" value="0x89504e470d0a1a0a" />
    <sequenceof name="chunks">
      <choice name="chunk">
        <reference name="unknown chunk" />
        <sequence name="end">
          <reference name="end chunk" />
        <end-sequenceof />
      </sequence>
    </choice>
  </sequenceof>
</sequence>

<common>
  <sequence name="unknown chunk">
    <field name="data length:" length="32" type="integer" />
    <field name="type" length="32" type="hex" />
    <field name="data" length="{data length:} * 8" type="hex" />
    <field name="crc" length="32" type="hex" />
  </sequence>

  <sequence name="end chunk">
    <field name="data length:" length="32" type="integer" />
    <field name="type" length="32" type="text" value="IEND" />
    <field name="data" length="{data length:} * 8" type="hex" />
    <field name="crc" length="32" type="hex" />
  </sequence>
</common>
</protocol>
```

Things to note:

- We've added an 'end chunk' common entry
- We've added a *choice* entry, allowing a chunk to be either an 'unknown chunk' or an 'end chunk'.

Attempting to decode still has the out of data error! Wait; look at the choice; the 'unknown chunk' is before the 'end chunk'! The 'unknown chunk' will always be attempted first (and succeed), so the 'end chunk' is never attempted. We need to swap them around, like so:

```
<choice name="chunk">
  <sequence name="end">
    <reference name="end chunk" />
    <end-sequenceof />
  </sequence>
  <reference name="unknown chunk" />
</choice>
```

Running the decode again, the sample file to successfully decodes!

## 2.5 Simplifying the specification using named references

In the specification we have had to re-type the integer field several times. While this isn't too difficult, having more text can make it harder to read. We can use *references* to only specify these once:

```
<protocol>
  ...skipping...

  <common>
    <field name="dword" type="integer" length="32" />

    <sequence name="unknown chunk">
      <reference name="data length:" type="dword" />
      <field name="type" length="32" type="hex" />

    ...skipping...

    <sequence name="end chunk">
      <reference name="data length:" type="dword" />
      <field name="type" length="32" type="text" value="IEND" />
```

Even in this simple case, it has made the code easier to read. In more complicated situations, where complex encodings are used (eg: textual integers, big endian integers, ...) it can make your specification far easier to read and maintain.

## 2.6 Refining the specification

Of course, while we are successfully decoding the file, there are still many sections in the file that have been left undecoded. Lets flesh some of them out.

### 2.6.1 Header

The spec states that a png file must start with an IHDR chunk. This chunk includes information about image height, width, the encoding, etc.

```
<sequence name="png">
  <field name="signature" length="64" type="hex" value="0x89504e470d0a1a0a" />
  <reference name="begin chunk" />
  <sequenceof name="chunks">
    ...
  </sequenceof>
</sequence>

<common>
  <sequence name="begin chunk">
    <field name="data length:" length="32" type="integer" />
```

```

<field name="type" length="32" type="text" value="IHDR" />
<sequence name="header" length="${data length:} * 8">
  <field name="width" length="32" type="integer" />
  <field name="height" length="32" type="integer" />
  <field name="bit depth" length="8" type="integer" />
  <choice name="colour type">
    <field name="greyscale" length="8" value="0x0" />
    <field name="rgb" length="8" value="0x2" />
    <field name="palette" length="8" value="0x3" />
    <field name="greyscale and alpha" length="8" value="0x4" />
    <field name="rgba" length="8" value="0x6" />
    <field name="unknown" length="8" />
  </choice>
  <choice name="compression method">
    <field name="deflate" length="8" value="0x0" />
    <field name="unknown" length="8" />
  </choice>
  <choice name="filter method">
    <field name="adaptive" length="8" value="0x0" />
    <field name="unknown" length="8" />
  </choice>
  <choice name="interlace method">
    <field name="none" length="8" value="0x0" />
    <field name="adam 7" length="8" value="0x1" />
    <field name="unknown" length="8" />
  </choice>
</sequence>
<field name="crc" length="32" type="hex" />
</sequence>
...

```

Note that when decoding the header, we have used a choice of fields to represent an enumeration. Also note that we validate the data length of the packet by setting a length on the header sequence (we could also have set an expected value on the 'data length' field).

## 2.6.2 Image data

Of course, the rest of the information isn't very useful without the image data. In the case of png, the image data is compressed. As the bdec specification is only concerned with representing the on disk structure, decoding this data is beyond the scope of bdec (it is up to the code using bdec to decode this data). That said, we can identify the image data chunk.

```

...
<choice name="chunk">
  <reference name="image data" />
  <sequence name="end">
    <reference name="end chunk" />
  <end-sequenceof />
</sequence>
<reference name="unknown chunk" />
</choice>
...

<sequence name="image data">
  <field name="data length:" length="32" type="integer" />
  <field name="type" length="32" type="text" value="IDAT" />
  <field name="data" length="${data length:} * 8" type="hex" />

```

```
<field name="crc" length="32" type="hex" />
</sequence>
```

### 2.6.3 Text entries

Text entries are used to hold things such as author, description, comments, etc. The png specification defines the data as being in the format:

```
Keyword:      1-79 bytes (character string)
Null separator: 1 byte
Text:         n bytes (character string)
```

This is a little difficult, as we have a variable length field whose length we don't know. We can use the 'end-sequenceof' to detect the end of the keyword, and a variable length text string to read the value. eg:

```
...
<sequenceof name="chunks">
  <choice name="chunk">
    <reference name="image data" />
    <reference name="text chunk" />
  ...
  <sequence name="text chunk">
    <field name="data length:" length="32" type="integer" />
    <field name="type" length="32" type="text" value="tEXt" />
    <sequenceof name="keyword">
      <choice name="char">
        <field name="null" length="8" value="0x0"><end-sequenceof /></field>
        <field name="character" length="8" type="text" />
      </choice>
    </sequenceof>
    <field name="value" type="text" length="${data length:} * 8 - len{keyword}" />
    <field name="crc" length="32" type="hex" />
  </sequence>
```

Note that we use the 'len{...}' *expression* to reference the length of another entry that has already been decoded.

## 2.7 Using the specification

While the specification is interesting, and decoding to xml can be useful in certain situations, native libraries are the preferred way of accessing the decoded data. As such, bdec supports either *generating C language decoders* or using the specification from *within python code*.

## 2.8 Where to go from here

There are many other chunk types in the png specification; try decoding sRGB (very easy) or PLTE (more difficult; use the 'length' attribute of a sequenceof). Read the *tips* section for useful tips on improving your specification.

One thing to realise is that the bdec specification will only take you so far; except for trivial file formats, code will still need to be written before you have a fully functional decoder (for example, decompression). The important fact is that this is exactly the non-trivial code that you have to think about; all the drudgery of normal loading and validation is already taken care of.

Offload as much possible into the specification, and it will make your code easier to read, and future maintenance much more pleasant.

Have fun!



# BDEC SPECIFICATIONS

The bdec specification is machine readable, and is capable of describing multiple existing binary formats.

## 3.1 Xml specification format

Bdec specifications are written in an xml markup format. The specification allows for many of the common (and not so common) features found in most binary file formats.

### 3.1.1 Protocol element

The protocol element is the outer element in a specification. No attributes can be set within the protocol block. The protocol block can have two child elements;

- A single *common element* (optional)
- A single *protocol entry*

The single protocol entry will be the default entry used when decoding.

### 3.1.2 Common element

The common element can contain multiple *entries* that can be referenced using *reference entries*.

### 3.1.3 Example

An extremely simple protocol specification:

```
<protocol>
  <sequence name="packet">
    <reference name="header" />
    <reference name="payload" />
  </sequence>

  <common>
    <sequence name="header">
      <field name="id" length="16" value="0x863f" />
      <field name="date" length="16" type="integer" />
    </sequence>
```

```
<sequence name="init">
  <field name="id" length="8" value="0x0" />
  <field name="name length" length="16" />
  <field name="name" length="{name length} * 8" type="text" />
</sequence>

<sequence name="data message">
  <field name="id" length="8" value="0x1" />
  <field name="data length" length="16" />
  <field name="data" length="{data length} * 8" type="hex" />
  <field name="trailer" type="integer" length="32" />
</sequence>

<choice name="payload">
  <reference name="init" />
  <reference name="data message" />
</choice>
</common>
</protocol>
```

## 3.2 Field entries

Field entries are the core part of a specification; they represent all the data that is found in the binary file (entries other than fields are responsible for the ordering and repetition of the field entries). The type of the field's data is specified in the field's attributes;

- **Type:** The data on disk can be many types, such as text (in any encoding), integer (little endian / big endian), and raw binary data.
- **Length:** The length of the data may be a fixed length, or it may be *variable length*. The data's length may be less than a byte, and it may not be byte aligned.

### 3.2.1 Specification

Bdec fields have the following attributes;

- A name (optional)
- A *length* in bits
- A *type* (optional)
- An encoding (optional)
- A *value* (optional)
- *Constraint* attributes (optional)
- An *if* (optional)

### 3.2.2 Field types

There are currently four available field types. If the type is not specified, it is assumed to be *binary*.

## Integer

Integer fields represent numeric values. There are two types of encodings supported, [little endian](#) and [big endian](#). The ‘integer’ type can be prefixed with ‘signed’ to specify that the integer can be negative. eg:

```
<!-- Unsigned big endian -->
<field name="a" type="integer" length="32" />

<!-- Unsigned big endian (note that 'unsigned' is redundant) -->
<field name="a" type="unsigned integer" length="32" />

<!-- Signed big endian (note that 'big endian' is redundant) -->
<field name="a" type="signed integer" encoding="big endian" length="16" />

<!-- Signed little endian -->
<field name="a" type="signed integer" encoding="little endian" length="64" />
```

The default encoding is unsigned big endian.

## Text

Text fields represent textual data in the binary file (ie: data that can be printed). It can use any unicode encoding, and defaults to `ascii`.

## Hex

Hex fields represent binary data whose length is a multiple of whole bytes. It has no encoding.

## Binary

Binary fields represent binary data of any length. It has no encoding.

### 3.2.3 Expected value

The ‘value’ attribute is another name for the ‘expected’ *constraint*.

### 3.2.4 Examples

A numeric field that is 2 bytes long, in big endian format:

```
<field name="data" length="16" type="integer" encoding="big endian" />
```

A utf-16 text field that is 8 bytes long:

```
<field name="name" length="64" type="text" encoding="utf16" />
```

A single bit boolean flag:

```
<field name="is header present" length="1" />
```

A two byte field that has an expected value:

```
<field name="id" length="16" value="0x00f3" />
```

A single numerical character (eg: characters '0'..'9'):

```
<field name="number" length="8" min="48" max="57" />
```

### 3.3 Sequence entries

Sequence entries are simple containers for other entries (see the [object composition](#) entry in wikipedia). Type are similar in function to structs in C.

#### 3.3.1 Specification

Bdec sequence entries can have 3 attributes;

- A name (optional)
- A `length` (optional)
- A `value` (optional)
- An *if* (optional)

Sequence attributes can contain any other type of entry (ie: *fields*, sequences, *sequenceofs*, *choices*, and *references*).

#### 3.3.2 Sequence length

The sequence length is an optional *expression* that sets the length of the sequence in bits. It is used to validate the combined length of all of the items contained within the sequence.

#### 3.3.3 Sequence value

A sequence can be assigned a value that can be referenced in expressions. The value itself is an *expression* that can reference entries contained within the sequence.

Sequence values can be used for several purposes, including:

- Converting text lengths to referencable lengths
- Joining non-adjacent numeric fields into one value
- Adding custom integer encodings

If a sequence has a value, that value can be limited using *constraints*.

#### 3.3.4 Examples

A simple sequence of fields:

```
<sequence name="data">
  <field name="a" length="8" type="integer" />
  <field name="b" length="64" type="text" />
</sequence>
```

Validation of the length of child fields using *expressions*:

```
<field name="total length" length="32" encoding="little endian" />
<sequence name="payload" length="{total length} * 8">
  <field name="a" length="8" />
  ...
  <field name="data length" length="32" encoding="little endian" />
  <field name="data" length="{data length} * 8" />
</sequence>
```

## 3.4 Choice entries

Choice entries allow one of many options to be present in the data stream. If the first option fails to decode, it will try the second (and so on). Only when all options have failed to decode will the choice entry fail.

### 3.4.1 Specification

Choice entries can have up to two attributes.

- A name (optional)
- A *length* (optional)
- An *if* (optional)

The choice must have multiple child entries specifying the different possible options (ie: *fields*, *sequences*, *sequence-ofs*, other choices, or *references*).

#### Choice length

The choice *length* is an optional attribute that specifies the number of bits the entry will contain. It is only used for validation; if the amount of data decoded by the successful option doesn't match the expected length the choice will fail to decode.

The length value can reference other entries using *expressions*.

### 3.4.2 Examples

Only allow fields with certain values:

```
<choice name="whitespace">
  <field name="space" length="8" value="0x20" />
  <field name="tab" length="8" value="0x9" />
</choice>
```

Choose between two entry types:

```
<choice name="entry">
  <sequence name="text entry">
    <field length="8" value="0x0" />
    <field name="length" length="8" />
    <field name="value" length="{length} * 8" type="text" />
  </sequence>
  <sequence name="integer entry">
    <field length="8" value="0x1" />
  </sequence>
</choice>
```

```
        <field name="value" length="32" type="integer" />
    </sequence>
</choice>
```

Allow only hex characters (eg: characters ‘a’ .. ‘f’, ‘A’ .. ‘F’, ‘0’ .. ‘9’):

```
<choice name="char">
    <field name="lowercase char" length="8" min="97" max="102" />
    <field name="uppercase char" length="8" min="65" max="70" />
    <field name="number" length="8" min="48" max="57" />
</choice>
```

## 3.5 SequenceOf entries

SequenceOf entries are repetitions of another element a given number of times. The repeated entry may be repeated a given number of times, it may be repeated until a buffer is empty, or it may repeat until a condition is met.

### 3.5.1 Specification

Bdec sequenceof entries can have 3 attributes;

- A name (optional)
- A count (optional)
- A length (optional)
- An *if* (optional)

A SequenceOf entry contains entries that are to be repeated (ie: a *field*, a *sequence*, a sequenceof, a *choice*, or a *references*). If more than one entry is found, an intermediate *sequence* will be created to contain all of the child entries.

A sequenceof should have either a count, a length, or have an embedded *end-sequenceof* to indicate when to stop repeating the element. If none of these are present, it is a *greedy* sequenceof.

### 3.5.2 Count loops

The count attribute is an *expression* that indicates the number of times the embedded item should repeat.

### 3.5.3 Length loops

The length attribute is an *expression* that indicates the size of the buffer in bits. The embedded entry will be repeated until the buffer is empty.

### 3.5.4 End-sequenceof loops

By embedding an ‘end-sequenceof’ tag somewhere in the child, when that child item is decoded, the sequenceof will end. This is useful when loops are terminated with an ‘end’ tag in the buffer. It is usually used with a *choice* entry.

For example, a *null terminated string* can be defined by a sequence of characters ended by a null character.

It is possible to use an ‘end-sequenceof’ with a count and length loop, but the decode will fail if the last entry is not an end-sequenceof.

### 3.5.5 Greedy sequenceof

A ‘greedy’ sequenceof doesn’t have ‘count’, ‘length’, or ‘end-sequenceof’ entries; it will continue to decode until all of the available data has been decoded. If its child entry fails to decode while data is still available, the sequenceof decoding fails.

### 3.5.6 Examples

A block of 100 name/value pairs:

```
<sequenceof name="data" count="100">
  <sequence name="entry">
    <field name="name" length="64" type="text" />
    <field name="value" length="32" type="integer" encoding="little endian" />
  </sequence>
</sequenceof>
```

A null terminated string is a repeated block of characters that is terminated by a ‘null’ character. This can be represented by:

```
<sequenceof name="null terminated string">
  <choice name="entry">
    <field name="null" length="8" value="0x0"><end-sequenceof /></field>
    <field name="char" length="8" />
  </choice>
</sequenceof>
```

A repeated loop of entries that has a known length (eg: the header in the wma/wmv specification) can be defined as:

```
<field name="buffer length" length="32" type="integer" />
<sequenceof name="items" length="{buffer length} * 8">
  <sequence name="item">
    <field name="a" length="8" />
    <field name="b" length="16" />
  </sequence>
</sequenceof>
```

## 3.6 Reference entries

Reference entries are used to reference a *top level entry* at the current location in the specification. This allows parts of the specification to be re-used in multiple locations, and also allows the specification to be broken up into logical boundaries.

References are identified by name, and cannot include modifications to the top level entry (eg: no changing the type of fields, changing the children of a sequence, adding an end-entry, ...).

It is possible however to give the referenced entry another name, and this can help in defining common types (such as defining a default integer encoding).

The top level entry doesn’t need to be specified in the file before it is referenced (although it must be specified before the specification has finished loading). Referenced entries can be used to decode specifications with recursive data structures (eg: pdf).

### 3.6.1 Specification

A bdec reference can have one or two attributes;

- `name` (optional)
- `type` (optional)

Either a name or a type must be specified for a reference.

#### Reference name

The name attribute specifies the name to be used for this entry. If no `type` is specified, this is assumed to be the name of the common entry.

#### Reference type

If specified, identifies the name of common entry to be referenced. If it is not specified, the reference name is used to find the referenced entry.

### 3.6.2 Examples

Reuse a 'dword' type and a null terminating string type:

```
<common>
  <field name="dword" length="32" encoding="little endian" />

  <sequenceof name="null terminating string">
    <choice name="entry">
      <field name="null" length="8" value="0x0" ><end-sequenceof /></field>
      <field name="char" length="8" type="text" />
    </choice>
  </sequenceof>

  <sequence name="address">
    <reference name="flat number" type="dword" />
    <reference name="street number" type="dword" />
    <reference name="street" type="null terminating string" />
    <reference name="city" type="null terminating string" />
  </sequence>
</common>
```

## 3.7 Expressions

Expressions are used to represent an integral value; for example, the length of a *field*. Expressions can contain numbers, perform numerical operations, reference the value of previously decoded fields, and reference the length of a previous decoded entry.

### 3.7.1 Numbers

The simplest expression is just a decimal number. For example, a field that is one byte long can have an length expression of "8" (ie: 8 bits).

### 3.7.2 Value references

Many fields in a data file need to reference the values of other fields. For example, variable length fields are typically stored as a length field, followed by a data field. Expressions can reference the value of the previously decoded length field by referring to it by [name](#):

```
<field name="data length" length="8" />
<field name="variable length data" length="${data length}" />
```

### 3.7.3 Numerical operations

It is often necessary to perform simple numerical operations in an expression. In the [previous example](#) the length was stored in bits, but it will often be stored in bytes:

```
length="${data length} * 8"
```

Supported numerical operations include the use of brackets, addition, subtraction, multiplication, and division. The operator precedence matches that found in C.

### 3.7.4 Length references

It is sometimes necessary to refer to the length of a previously decoded item. For example, the length value may include a header, followed by a variable length data field. The length of the data field is the total length minus the length of the header entry. The length of the header can be referenced by [name](#):

```
length="${data length} * 8 - len{header}"
```

### 3.7.5 Resolving names

Names are resolved by looking for a previously defined entry with the given name.:

```
<sequence name="data">
  <field name="length" length="16" type="integer" />
  <field name="value" length="${length} * 8" type="text" />
  ...
</sequence>
```

If there isn't an entry with the matching name, the parent entry will be checked.:

```
<field name="length" length="16" type="integer" />
<sequence name="data">
  <field name="value" length="${length} * 8" type="text" />
  ...
</sequence>
```

It is possible to look inside entries by separating names with the '.' character.:

```
<sequence name="header"
  <field name="length" length="16" type="integer" />
  ...
</sequence>
<field name="data" length="${header.length}" />
```

### 3.7.6 Examples

A variable length string:

```
<field name="length" length="32" type="integer" />
<field name="text" length="{length} * 8" type="text" />
```

A variable length data object with a header:

```
<sequence name="header">
  <field length="8" value="0x38" />
  <field name="index" length="32" type="integer" />
  <field name="length" length="32" type="integer" />
  ...
</sequence>
<field name="data" length="{header.length} * 8 - len{header}" />
```

A variable length sequence of entries:

```
<field name="num items" length="8" />
<sequenceof name="items" count="{num items}">
  <sequence name="item">
    ...
  </sequence>
</sequenceof>
```

### 3.7.7 Boolean expressions

All entries can use an optional 'if' attribute; this attribute contains a boolean expression. If that expression evaluates to true, the entry will be decoded, if not, it will be skipped. Boolean expressions can contain all standard expressions (such as [value references](#), [numerical operations](#), etc), as well as boolean comparisons (>, ==, &&, ||, etc).

Optional entries can be very useful when the presence of an entry depends on values that came significantly before it in the specification, such as flags indicating the presence of a footer:

```
<sequence name="packet">
  <sequence name="header">
    ...
    <field name="has footer:" length="8" />
    ...
  </sequence>
  <sequence name="body">
    ...
  </sequence>
  <sequence name="footer" if="{has footer:}">
    ...
  </sequence>
</sequence>
```

When it is possible to use either a *choice* or several optional entries, always prefer the choice; it will result in a clearer spec, and will generate nicer code. For example:

```
<!-- This is the bad way to do it -->
<field name="type:" length="8" />
<sequence name="a" if="{type:} == 1">
  ...
</sequence>
<sequence name="b" if="{type:} == 2">
```

```

...
</sequence>
<sequence name="c" if="{type:} == 3">
...
</sequence>

```

can be much better specified with a *choice*:

```

<!-- This is the good way to do it -->
<choice name="packet">
  <sequence name="a">
    <field name="type:" length="8" value="1" />
    ...
  </sequence>
  <sequence name="b">
    <field name="type:" length="8" value="2" />
    ...
  </sequence>
  <sequence name="c">
    <field name="type:" length="8" value="3" />
    ...
  </sequence>
</choice>

```

## 3.8 Constraints

We can place constraints on what values are considered valid for a given entry. The constraints are;

- An [expected value](#)
- A [minimum or maximum value](#)

### 3.8.1 Expected values

Entries can have an expected value, and will fail to decode if the data found doesn't match the expected value.

The expected value attribute can be either in hex (eg: value="0xf3"), or in the type of the entry:

```

<field name="null" length="8" value="0x0" />
<field name="identifier" length="16" type="integer" value="18" />
<field name="name" length="120" value="expected string" />

```

### 3.8.2 Value ranges

Entries that can be converted to integers can have a minimum and a maximum value. For example, you may want a field that decodes all numerical text entries '0' to '9'.

Both min and max are inclusive; ie:

```
min <= value <= max
```

If the decoded value falls outside the minimum or maximum, the entry fails to decode. eg:

```
<field name="digit" length="8" type="text" min="48" max="57" />
```

will fail to decode bytes with value 0-47, and 58-255.

### **3.8.3 Advanced usage**

Usually constraints will only ever be used with fields, but they can also be used with other entries (that have a value, such as sequences with a 'value' attribute, or choices where all options are of the same type). In these cases the expected value attribute must be named 'expected', not 'value'). There are five basic entries in a bdec protocol specification.

- A *field*
- A *sequence*
- A *choice*
- A *sequenceof*
- A *reference*

Once the basic entries have been mastered, there are some *tips* that can be used to improve the readability of the specification.

## TIPS

Being able to effectively read and modify the specification is crucial to maintainability.

### 4.1 Hiding entries

Some entries are useful to name in the specification, but aren't very useful to see in the decoded output. For example;

- When using a choice, the options are often distinguished by the first few bytes found in each option. These bytes aren't 'interesting' to the user, as they are already known.
- When there is a variable length field, it is often represented as an entry containing a length, followed by the entry containing the data. The 'length' entry isn't interesting to the user.
- There are often sections of a file whose data contains nothing; essentially, filler sections of the file.

To hide these entries from the xml output, and from the compiled structures, the names of these entries can be either left blank, or postfixed with a ':' character. These entries will be hidden from output.

eg:

```
<field name="length:" length="8" type="integer" />
<field name="data" length="${length:} * 8" type="hex" />
```

or:

```
<choice name="entry">
  <sequence name="triangle">
    <field length="8" value="0x13" />
    ...
  </sequence>
  <sequence name="rectangle">
    <field length="8" value="0x14" />
    ...
  </sequence>
</choice>
```

While it is usually useful to hide these entries, when use the bdecode command line decoder, the '-verbose' option can be used to print these hidden entries.

## 4.2 Named references

Specifications often contain entries with a consistent format. For example, there might be a lot of 32 bit little endian integers. Instead of writing:

```
<common>
  <sequence name="header">
    <field length="32" value="0x12345678" />
    <field name="year" length="32" type="integer" encoding="little endian" />
    <field name="month" length="32" type="integer" encoding="little endian" />
    <field name="day" length="32" type="integer" encoding="little endian" />
    <field name="version" length="32" type="integer" encoding="little endian" />
  </sequence>
  ...
</common>
```

use a common entry that defines types that are used regularly, such as:

```
<common>
  <field name="dword" length="32" type="integer" encoding="little endian" />

  <sequence name="header">
    <field length="32" value="0x12345678" />
    <reference name="year" type="dword" />
    <reference name="month" type="dword" />
    <reference name="day" type="dword" />
    <reference name="version" type="dword" />
  </sequence>
  ...
</common>
```

## 4.3 Optional entries

Often the header of a format has a flag which indicates whether an optional block of data is present. While it is possible to create a choice with the two options, this isn't clear or convenient, eg:

```
<choice name="header">
  <sequence name="header with footer">
    <field name="footer present" length="8" value="0x1" />
    ...
    <reference name="footer" />
  </sequence>
  <sequence name="header without footer">
    <field name="footer not present" length="8" value="0x0" />
    ...
  </sequence>
</choice>
```

This is verbose and difficult to follow. It is possible instead to make the entry conditional upon an *expression*:

```
<sequence name="header">
  <field name="footer present:" length="8" />
  ...
  <sequence name="footer" if="{footer present:}">
    ...
  </sequence>
</sequence>
```

As a rule of thumb, if an entry can be present zero or one times, and the entry depends on a previous flag, use a conditional. If one of several possibilities can be used (eg: differing payloads in a message, etc), use a choice.

## 4.4 Avoid duplication within choice options

Often specifications have a field representing the type of a payload that is followed by several common entries before the different options diverge. eg:

```
<choice name="packet">
  <sequence name="type a">
    <field name="data length:" length="8" />
    <field name="type:" length="8" value="0x0" />
    <reference name="header" />
    ... 'type a' specific entries
  </sequence>
  <sequence name="type b">
    <field name="data length:" length="8" />
    <field name="type:" length="8" value="0x1" />
    <reference name="header" />
    ... 'type b' specific entries
  </sequence>
</choice>
```

While it is possible to put the common entries into every choice option, this involves significant repetition, and often makes the compiled decoders awkward to use (as the common fields are duplicated in different structures). It's much better to decode the 'type:' field without an expected value, and reference it within the payload. eg:

```
<sequence name="packet">
  <field name="data length:" length="8" />
  <field name="type:" length="8" />
  <reference name="header" />
  <choice name="payload">
    <sequence name="type a">
      <sequence value="{type:}" expected="0" />
      ... 'type a' specific entries
    </sequence>
    <sequence name="type b">
      <sequence value="{type:}" expected="1" />
      ... 'type b' specific entries
    </sequence>
  </choice>
</sequence>
```



## COMPILING TO C

*Specifications* can be compiled to efficient, readable C language decoders by running:

```
bcompile png.xml
```

The generate source includes a sample 'main.c' that can be used as for decoding png files to xml. See the alternate main.c that prints the image width and height, and displays all png 'text' sections.



---

# PYTHON INSTANCE DECODER

*Specifications* can be loaded directly from python code, and can be used to decode directly to python instances:

```
from bdec.data import Data
from bdec import DecodeError
from bdec.spec.xmlspec import load
from bdec.output.instance import decode

spec = load('test.xml')[0]
data = Data(open('data.bin', 'rb').read())
try:
    values = decode(spec, data)
except DecodeError, err:
    print 'Oh oh...', err
```

The 'values' object now represents the decoded data in python instances;

- *Fields* decode to their natural type; eg: integers, strings, or for 'binary' types, to a Data instance.
- *Sequences* act as objects; access child entries with '.' (eg: a.b).
- *SequenceOf* entries decode to a list.
- *Choice* entries are like a sequence, but only the child that was actually decoded will be present.

For example, given the following spec:

```
<protocol>
  <sequence name="a">
    <field name="b" length="16" type="text" />
    <field name="count:" length="8" />
    <sequenceof name="values" count="{count:}">
      <field name="c" length="8" type="integer" />
    </sequenceof>
  </sequence>
</protocol>
```

You can use the decoded instance like:

```
result = decode(spec[0], data)
print result.a.b
for c in result.a.values:
    print c
```