# Parsing and Reverse Engineering Binary Files

## Lessons learned...
so far...

Kenneth Nielsen (CINF)

https://xkcd.com/99/

Python and beers presentation,
January 28, 2016

## About me

- My name is Kenneth Nielsen and I work at CINF
- Free and open source software (FOSS) enthusiast
- Use way too much time on Python conference presentations on youtube

**Slides and examples**

```
git clone https://github.com/KennethNielsen/presentations.git
```

https://github.com/KennethNielsen/presentations
http://bit.ly/1FclzDR

## Outline

## Introduction

- Parse binary file format, simpler format is unavailable
- E.g. pull data out of files created by instrument software
- With or without specification (emphasis is on without)
- More talk of techniques than code (though there is some at the end)
- Some talk about binary data in general

## Outline

## What do I mean by binary file?

### The narrow definition

- Not clear text e.g. XML, JSON or custom text format
- Simple binary files!
- Concatenation of binary representation of values
- No type information
- No third party serialization

## Binary representation of data

### It's all bits

- Computers only know about bits (1000101010111010101)
- The **type** of the data is what gives the bits meaning (converts it to a **value**)
- For the values, one or more bytes can be used, depending on the requirements

Examples of data types are:

- Signed and unsigned integers: short (int8), unsigned long (uint64)
- Booleans (commonly a special case of a 1 byte integer)
- Floats and doubles: 4 or 8 bytes floating point numbers
- Strings..!

## Strings

1 min in bytes and encodings (which is really a presentation on its own)

### We can only store bytes

- Text is stored in an encoding
- Many (all?) encoding contains ASCII at the first 128 positions
- Which means that it is recognizable
- After parsing, bytes should be decoded
- Some encodings are fixed width other variable
- Sometimes the string is prefixed with the number of bytes
- It's a pain

## Endianness

**The order of bytes in multi byte types**

To make matters worse, different “computer” architectures does not agree on in which order bytes in a multi-byte type should be stored!

**This is referred to as the “endianness”**

From wikipedia: https://en.wikipedia.org/wiki/Endianness

> *With big-endian the most significant byte of a word is stored at a particular memory address and the subsequent bytes are stored in the following higher memory addresses, the least significant byte thus being stored at the highest memory address. Little-endian format reverses the order and stores the least significant byte at the lower memory address with the most significant byte being stored at the highest memory address*

# One set of bytes, many meanings

```
The bytes ['0x50', '0x79', '0x74', '0x68', '0x6f', '0x6e', '0x20', '0xFF']
               char[] : b'Python \xff'
Not relevant     int8 : (80, 121, 116, 104, 111, 110, 32, -1)
Not relevant    uint8 : (80, 121, 116, 104, 111, 110, 32, 255)
Little-Endian   int16 : (31056, 26740, 28271, -224)
Big-Endian      int16 : (20601, 29800, 28526, 8447)
Little-Endian  uint16 : (31056, 26740, 28271, 65312)
Big-Endian     uint16 : (20601, 29800, 28526, 8447)
Little-Endian   int32 : (1752463696, -14651793)
Big-Endian      int32 : (1350136936, 1869488383)
Little-Endian  uint32 : (1752463696, 4280315503)
Big-Endian     uint32 : (1350136936, 1869488383)
Little-Endian   int64 : (-62928970010298032,)
Big-Endian      int64 : (5798793987111133439,)
Little-Endian  uint64 : (18383815103699253584,)
Big-Endian     uint64 : (5798793987111133439,)
Little-Endian   float : (4.6179809698425413e+24, -2.1324988332749138e+38)
Big-Endian      float : (16740622336.0, 7.369732216716093e+28)
Little-Endian  double : (-2.2536157545457443e+304,)
Big-Endian     double : (4.715928068041943e+79,)
```

There are 19 lines here! Luckily, often the endianness is known, which reduces it to 11

## Outline

## The hex editor

**Your friend, the hex editor**

- At first intimidating and difficult to make sense of
- Then ... **incredibly** useful

DEMO ghex

## The `struct` module

3 module functions:

- `unpack(fmt, string)`
- `pack(fmt, v1, v2, ...)`
- `calcsize(fmt)`

make up the core of the functionality

Most of the time we will just be using `unpack`.

DEMO in the terminal!

## With a format specification

Edited version of the specification of the Århus type STM file format: http://owww.phys.au.dk/spm/programs/

```
// Each image consists of a label record defined as follows:
la=
record
  //size in blocks, xch,ych are x,y pixels.
  nr,size,xch,ych,zch:smallint;
  //In Turbo Pascal format (year, month, day, hour, min, sec in 16
  //bit integers)
  time:datetime;
  xsize,ysize,xshift,yshift,zscale,tilt:smallint;
  speed:word; { now time in 10mS units 921221 S3 vers 6}
  // bias is -+10V for +-32 k  current is in 0.01 nA units
  bias,current:smallint;
  sample,title:string[20];
  postpr,postd1,constheight, { 910110 -constheight/postd2- }
  Currfac,R_Nr:smallint; { 930118 -Currfac/min- 931009 -R_Nr/max- }
  unitnr:smallint;  { 890612 }
  version:smallint;
  spare:array[48..63] of smallint;
end;
```

## With a format specification

- Depending on the implementation language, figure out what the integer names means
- By use of the specification, build up the formats
- Parse with `struct.unpack`

## Without a format specification

Look for information anywhere you can find it:

- (Half) implementations on the internet
- Possibly in other languages
- Online discussions

Some information is much better than no information.

**Other than that, hang on tight for the rest of the ride.**

## Outline

## Typical anatomy of a binary file

A binary file may have a layout as:

- For multipart files, an index
- A header (metadata)
- The data

```
Multipart      Single part

+---------+    +---------+
| Index   |    | Header  |
|---------|    |---------|
|Header 1 |    |         |
|Data 1   |    |  Data   |
|---------|    |         |
|Header 2 |    +---------+
|Data 2   |
+---------+
```

## Empty space

The header may be sparse, with padding (of zero bytes) in between information items

```
+-----------------+
|ID14....Comment.|
|....x-axis unit.|
|................|
|.Date...........|
|......95365.....|
+-----------------+
```

## Outline

## Get an export of the data file

### Get an export of any kind

Although the purpose of the exercise is to be able to read the binary files directly, having an export of both data and metadata for a few files to compare with is invaluable (maybe indispensable)

From this export extract information about the number of data points and as much metadata as possible

## Finding the data

- Use the hex editor
- In the case of a sparse header, look for when the data becomes dense and repeats in a pattern
- In the case of a dense header, without any clear start of data, calculate backwards from the end with the maximum number of bytes per point

```
+----------------+
|ID14....Comment.|
|....x-axis unit.|
|................|
|.Date...........|
|................|
|................|
|....3761....9578|
|....7A74....8943|
|....5679....9578|
|....6899....7536|
|....5478....1234|
+----------------+
```

## Analyze (Important)

- Measure how many bytes of data there is per point
- Is both x and y saved (xxxyyy or xyxyxy), or only y
- **NOTE:** That the data in the file, does not need to correspond to the expected data directly
- It may be scaled e.g. ADC integers instead of floats
- Compare with the export and see if there is a consistent scaling factor (or linear scaling relation)

## Quick detour on file objects

When working with text:

- `fileobject.read()` Read it all
- `for line in fileobject:` Iterate over lines in the file

But fileobject also have a “cursor”, which is really useful. When working with binary data:

- `fileobject.read(n)` Read n bytes
- `fileobject.seek(n, origin)` Seek to byte n from origin
- `fileobject.tell()` Get current position

## Read until there is no more

```
data = []
with open('FID1A.ch', 'rb') as file_:
    file_.seek(6144)
    raw = file_.read(8)
    while len(raw) == 8:
        data.append(unpack('<d', raw)[0])
        raw = file_.read(8)
points = numpy.array(data)
```

## Calculate number of points beforehand

```
with open('FID1A.ch', 'rb') as file_:
    file_.seek(0, 2)  # Seek to the end of the file
    end = file_.tell()
    num_points = (end - 6144) // 8
    file_.seek(6144)
    raw = file_.read(end - 6144)
    data = unpack('<{}d'.format(num_points), raw)
points = numpy.array(data)
```

## Parse directly into numpy array

```
with open('FID1A.ch', 'rb') as file_:
    file_.seek(0, 2)  # Seek to the end of the file
    end = file_.tell()
    num_points = (end - 6144) // 8
    file_.seek(6144)
    points = numpy.fromfile(file_, dtype='<d',
                            count=num_points)
```

DEMO data.py

## Outline

## The big “Can you spot the number in the big pile of numbers” game

Look for know metadata items from your export by:

- Use of a hex editor
- Searching for it by brute force
- Changing just one settings and look for differences in similar headers

## Brute force search

```
def find_number(value, filename, header_end, endiannes='>'):
    """Look for value in file named filename"""
    if isinstance(value, float):
        formats = 'fd'
    else:
        formats = 'bBhHiIqQ'

    with open(filename, 'rb') as file_:
        for index in range(header_end):
            for format_ in formats:
                file_.seek(index)
                raw = file_.read(calcsize(format_))
                value_at_index = \
                    unpack(endiannes + format_, raw)[0]
                if equal_or_close(value, value_at_index):
                    print('YATZY! Found value {} at index {}'.\
                          format(value, index))
```

DEMO header_tools.py

## Look for differences, part1

```
def bytes_diff(filename1, filename2, header_end):
    """Calculate bytes diff"""
    # Get the bytes
    byte_strings = []
    for filename in [filename1, filename2]:
        with open(filename, 'rb') as file_:
            byte_strings.append(file_.read(header_end))
```

## Look for differences, part2

```
start_pos = None
diff1, diff2 = b'', b''
for index, (byte1, byte2) in enumerate(zip(*byte_strings)):
    # In Python 3, iterating over bytes gives you numbers!
    if sys.version_info.major == 3:
        byte1, byte2 = bytes([byte1]), bytes([byte2])
    # Compare
    if byte1 != byte2:
        if start_pos is None:
            start_pos = index
        diff1 += byte1
        diff2 += byte2
    else:
        if start_pos is not None:
            print('At index:', start_pos)
            print(repr(diff1) + '\n' + repr(diff2) + '\n')
            start_pos = None
            diff1, diff2 = b'', b''
```

DEMO header_tools.py

## Parse the header, part 1, table of fields

```
fields = (
    ('sequence_line_or_injection', 252, UINT16),
    ('injection_or_sequence_line', 256, UINT16),
    ('start_time', 282, 'x-time'),
    ('end_time', 286, 'x-time'),
    ('version_string', 326, 'utf16'),
    ('description', 347, 'utf16'),
    ('sample', 858, 'utf16'),
    ('operator', 1880, 'utf16'),
    ('date', 2391, 'utf16'),
    ('inlet', 2492, 'utf16'),
    ('instrument', 2533, 'utf16'),
    ('method', 2574, 'utf16'),
    ('software version', 3601, 'utf16'),
    ('software name', 3089, 'utf16'),
    ('software revision', 3802, 'utf16'),
    ('units', 4172, 'utf16'),
    ('detector', 4213, 'utf16'),
    ('yscaling', 4732, ENDIAN + 'd')
)
```

## Parse the header, part 2

```python
def parse_header(file_):
    metadata = {}
    # Parse all metadata fields
    for name, offset, type_ in fields:
        file_.seek(offset)
        if type_ == 'utf16':
            metadata[name] = parse_utf16_string(file_)
        elif type_ == 'x-time':
            metadata[name] =\
                unpack(ENDIAN + 'f', file_.read(4))[0] / 60000
        else:
            metadata[name] =\
                unpack(type_, file_.read(calcsize(type_)))[0]

    # Convert date
    metadata['datetime'] = time.strptime(metadata['date'],
                                         '%d-%b-%y, %H:%M:%S')
    return metadata
```

DEMO header.py

## That's it

The file is now parsed and we know something about how to read binary files!

### Resist the temptation

Some of you may at this point feel an itching to try and *make* a binary format of your own i.e. write binary files instead of reading them, because it somehow feel like the *right* way to store stuff. **DON'T!!!**.

Wide use of custom format binary files is mostly a thing of the past. In the year 2016 custom format binary should **only** be used if it is absolutely necessary. Most things can be handled by simply using two files, one for the numpy data (which can write itself) and one of the metadate e.g. in JSON. If you really need a single file format with high space and read/write efficiency look into HDF5.

## Outline

## Summary, tools

- A hex editor
- Struct

## Summary, anatomy

- Header
- Data
- Possibly an index, for multipart files

## Summary, data

- Look for data with hex editor
- Calculate backwards from the known number of points
- Parse with struct
- Or directly into numpy.array

## Summary, header (metadata)

- Look for fields with hex editor
- Search for known numbers
- Look at differences
- Write out a table of known fields
- Parse with struct

Questions?