Synalyze It!

User's Guide

Synalyze It!: User's Guide

Andreas Pehnack

Table of Contents

List of Figures

Chapter 1. Welcome to Synalyze It!

> No man can reveal to you nothing but that which already lies half-asleep in the dawning of your knowledge.
>
> —Khalil Gibran

Thank You

Thank you for taking the time to read this manual. Here you'll find not only how to use Synalyze It! but also essential knowledge about the analysis of binary files.

The idea behind Synalyze It! is to support you in all the tasks that are related to analysis of binary files. Likewise, this manual is intended to help you make the most out of the application.

In any case I'm interested in your feedback. Be it positive, if you miss something or any other improvement.

There are many clickable references in this manual to Wikipedia or the glossary at the end of the manual that explains the most important terms related to Synalyze It!

Features only available in Synalyze It! Pro are marked with 

.

Subscribe to the Synalyze It! Newsletter

Learn about the latest news, get relevant hints and tips about how to make most of the application. Subscribe today:

- Go to the Synalysis web site http://www.synalysis.net/
- Enter your email address in the box on the left side
- Click *Subscribe*

Chapter 2. What is Synalyze It!

> That is strength, boy! That is power! What is steel compared to the hand that wields it?
>
> —Thulsa Doom

At first glance the application will look mainly like a regular hex editor, however a powerful one that supports many text code pages, allows finding not only text but also numbers, masks or all strings in a file or displays a histogram.

But what really sets it apart from all the other hex editors is a the support of grammars. Grammars? Yes, every binary file has a layout that enables certain applications to read and interpret them. These layouts are called grammars in Synalyze It! because of the similarities to the structure of human languages. Grammar files are stored as plain XML files and describe all the structures and data fields that comprise certain formats.

If a grammar is applied to a binary file Synalyze It! highlights all elements of the file and makes the analysis much easier. Even non-experts become able to decode the contents of files they have a grammar for. Many grammars already exist at http://www.synalysis.net/formats.xml and can be downloaded for free.

With Synalyze It! you are able to

- Display and edit files of unlimited size
- Analyze unknown binary file formats
- Apply the grammar you created to any similar file
- Compare a sequence of bytes in different text encodings
- See in a histogram how often different bytes occur in a file
- Get a list all strings in a file
- Do much much more, especially with the Pro version

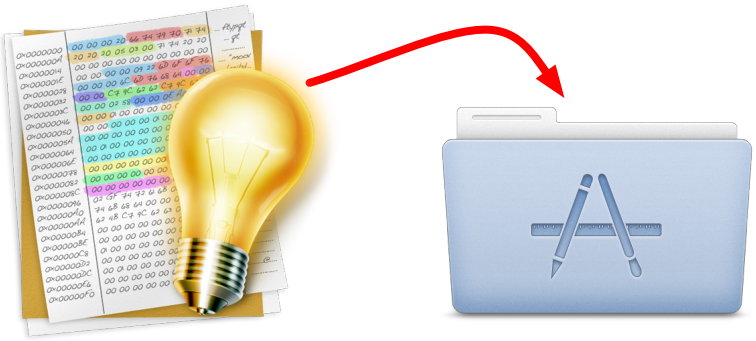
The scripting support in the Pro version allows to write custom Python routines that process the parsing results, import, export or modify grammars, manipulate files or fill gaps of the generic parser.

Chapter 3. Installation

> I don't necessarily think that installation is the only way to go.
>
> It's just a label for certain kinds of arrangements.
>
> —Barbara Kruger

If you bought Synalyze It! via the Mac App Store, the installation is done for you automatically. Users who downloaded the software from the web site simply drag Synalyze It! after uncompressing to their application folder.

Figure 3.1. Installation of the application



If you install grammar files via the application they are stored in the path

```
~/Library/Containers/com.synalyze-it.SynalyzeItPro/Data/Library
/Application Support/Synalyze It! Pro/Grammars
```

by Synalyze It! Pro and

```
~/Library/Containers/net.synalysis.SynalyzeIt/Data/Library
/Application Support/SynalyzeIt/Grammars
```

by Synalyze It! Those grammars are suggested automatically for appropriate files you open.

Scripts are stored in

```
~/Library/Containers/com.synalyze-it.SynalyzeItPro/Data/Library
/Application Support/Synalyze It! Pro/Scripts
```

and will be embedded in grammars if you reference them.

Chapter 4. Synalyze It! explained

> The cause is hidden; the effect is visible to all.
>
> —Ovid

In Synalyze It! you mainly work with two types of windows: the actual files you're analysing or using as a reference to build a grammar and the grammar editor that lets you make up the structures and elements of grammars. The Pro version features additionally a scripting editor.

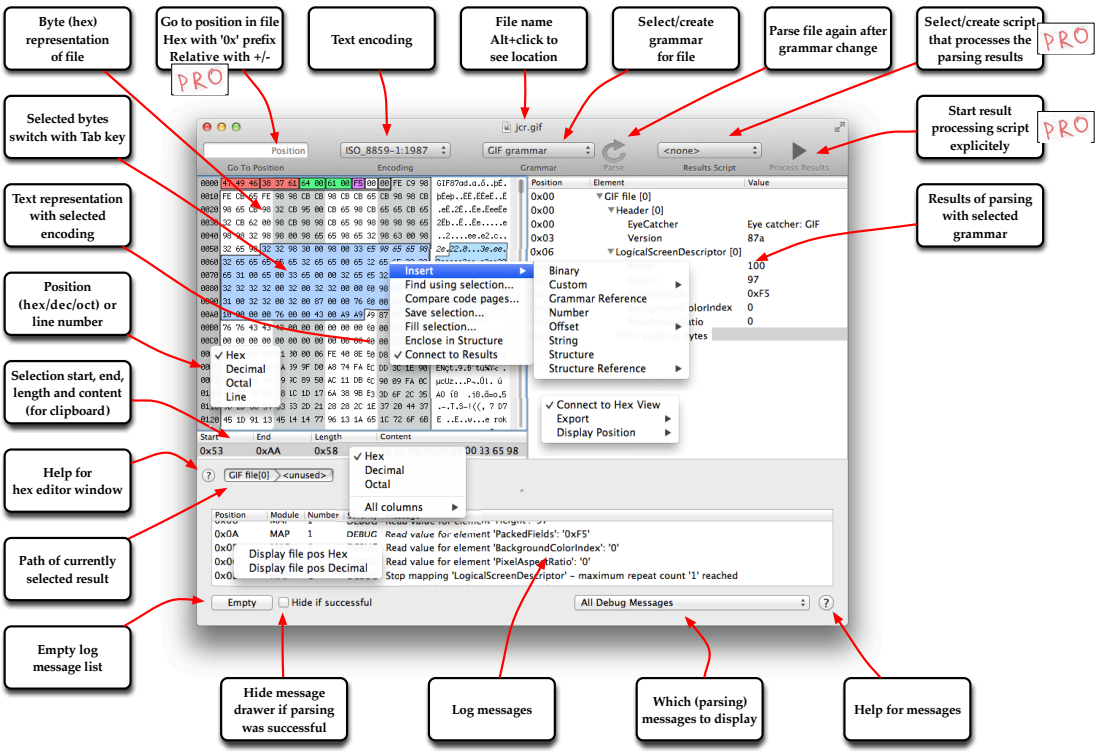
The Reference Document

The first thing you see when opening an arbitrary file is a hex dump and a text representation of the bytes. The editing functions work the same like in a text editor - you can overwrite, insert and remove bytes, select and copy bytes or text to the clipboard and so on.

The hex editor window is the starting point when exploring the details of a file. Much of the appearance can be customized like colors, position and selection number formats. There are plenty of text encodings that can be selected to decode not only ASCII-encoded text but also Unicode or EBCDIC as still found on IBM systems like z/OS or in formats like IJPDS.

There is a primary and a secondary selection for hex bytes and text. Per default the primary selection is displayed in darker blue than the secondary selection. The contents of the primary selection are displayed in the table below the hex editor and are copied to the clipboard when you press **Cmd**+**C** (copy) or **Cmd**+**X** (cut). Switch between the selections with the **Tab** key and toggle insert/overwrite mode with **Cmd**+**K**.

Figure 4.1. Parts of the Reference Document



The contextual menu of the hex view allows you to search in the file, compare text in different encodings or save the selected bytes to disk. The Pro version additionally allows to fill the selection with text or bytes.

Once you selected a grammar for the file in the toolbar, the window is split and on the right side you see the the parsing results. The contextual window of the hex view offers now some more options: you can add a new element or structure to the grammar and link the hex view to the results view. This means that wherever you click in the hex view, the corresponding result will be selected on the right-hand side.

The parsing results are not only displayed, you can edit the values and they will be translated back to the file. For all editing in the file unlimited undo and redo are available.

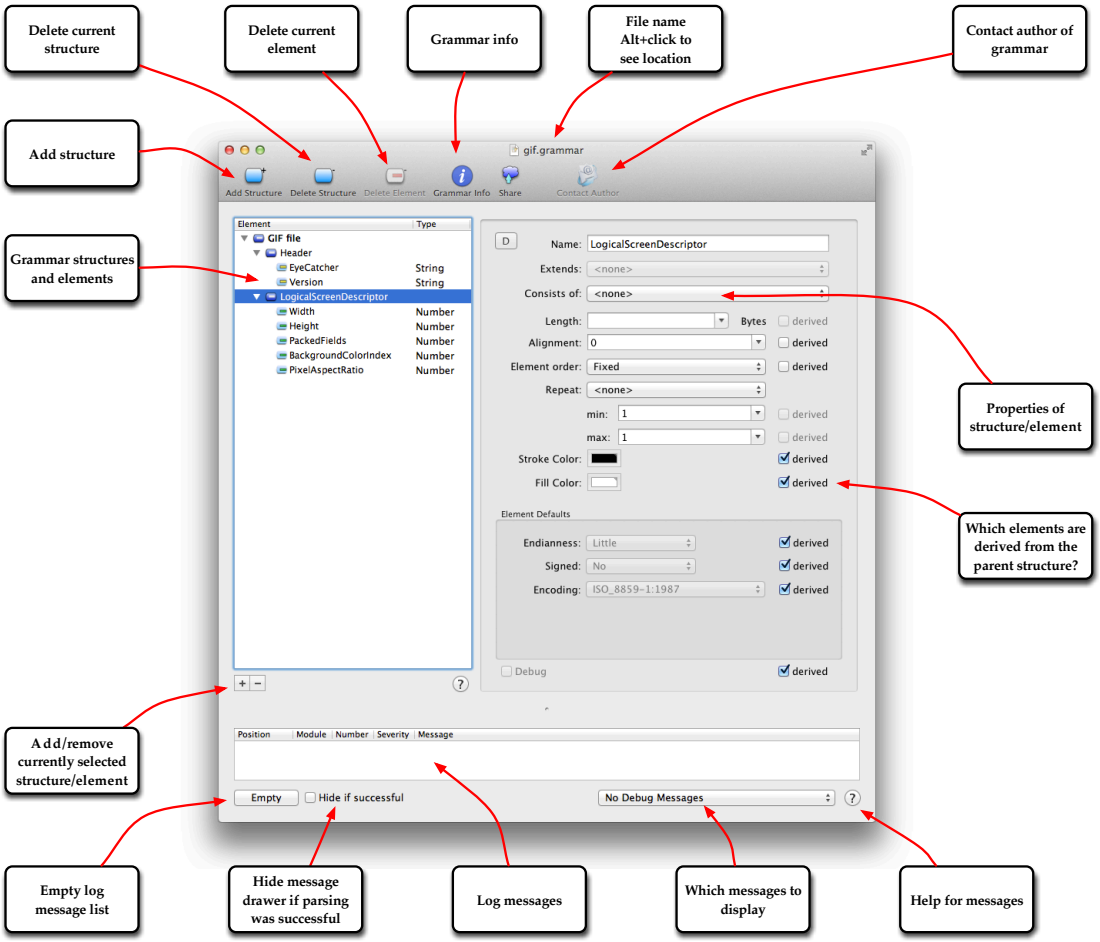
The Pro version provides more means to work with the parsing results. You can save them as an XML or text file and even process them with a custom script. Some sample scripts are available on http://synalysis.net/scripts.html.

The Grammar Editor

When starting to create a new grammar you will mostly do the first steps in a reference file that serves as a *model*. There you can select the bytes that should be interpreted as a structure, number, string or another element. This immediate feedback — per default the grammar is applied after each change — lets you quickly set up a basic grammar.

However, a good grammar avoids redundancy and makes use of the powerful inheritance feature. The grammar editor lets you craft elegant grammars that represent file formats as abstract as possible.

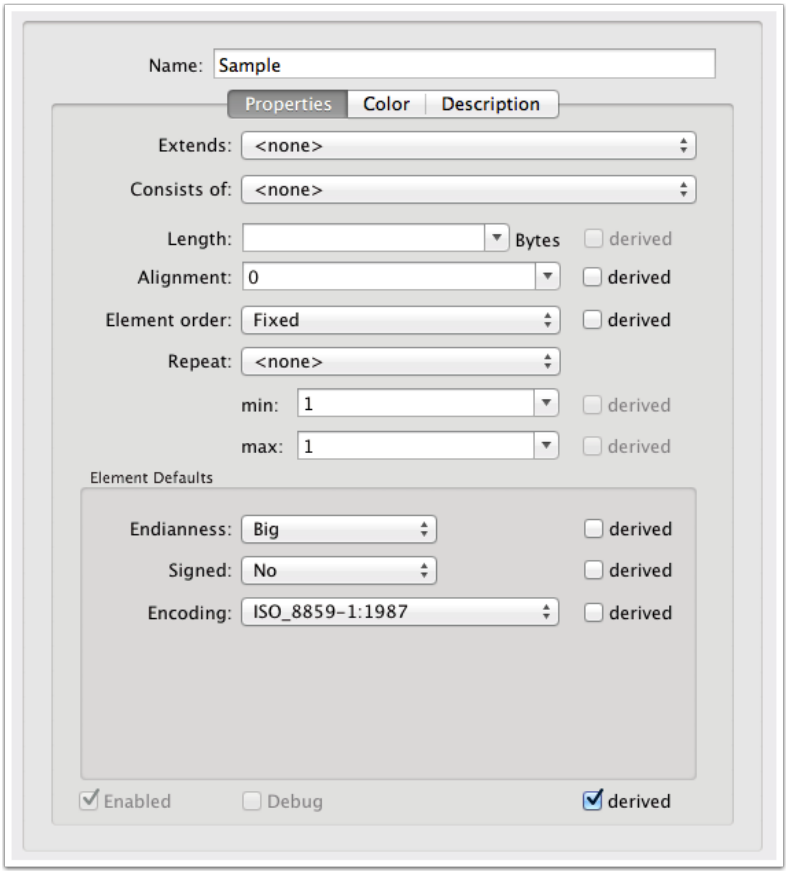
Figure 4.2. Parts of the Grammar Editor



You can easily rearrange structures and their elements by drag and drop, pressing the **Alt** key duplicates them.

Structure Properties

Figure 4.3. Structure properties



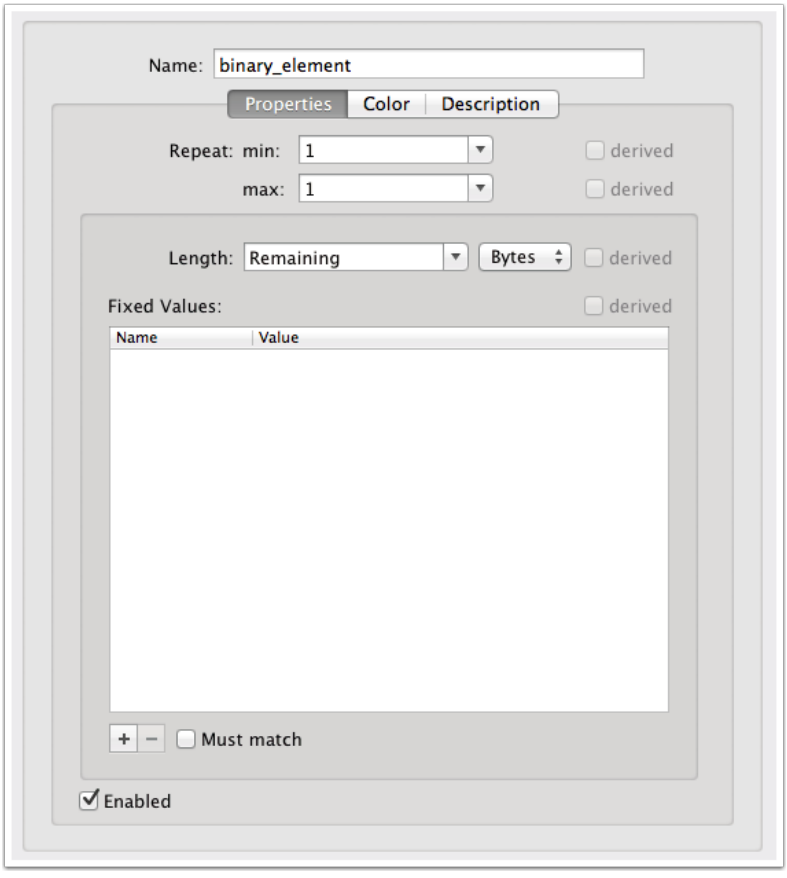
- *Extends* - Select here the structure to inherit from. Only top-level structures can inherit from other top-level structures
- *Consists of* - Select here a parent structure if the structure consists of multiple similar records.
- *Length* - The structure length in bytes. You can also select here the name of an integer number element inside the structure or which was parsed before.
- *Alignment* - If a structure must start at a multiple of *n* bytes, use the alignment field.
- *Element order* - Choose a *fixed* element order if all elements in the structure have to appear in a fixed order. If only a single element of many is expected, choose *variable*.
- *Repeat* - The name of an integer number element that specifies how often to repeat this structure. Make sure the max repeat count is at least the highest possible repeat count.
- *min* - The minimum repeat count. Parsing fails if that number is not reached.
- *max* - The maximum repeat count. Parsing stops if that number is reached. Select *unlimited* if the structure should fill the remaining space (determined by the enclosing structure).
- *Endianness* - The default endianness of elements in this structure.

- *Signed* - The default "signedness" of elements in this structure.
- *Encoding* - The default encoding of strings in this structure.
- *Stroke Color* - The color of the path drawn around this structure in the hex view.
- *Fill Color* - The background color of this structure in the hex view.
- *Description* - Description of the structure. This is displayed in a tooltip in the results tree view.

Binary Element Properties

Binary elements are used for bit or byte sequenced that shouldn't be analyzed in more detail.

Figure 4.4. Binary element properties



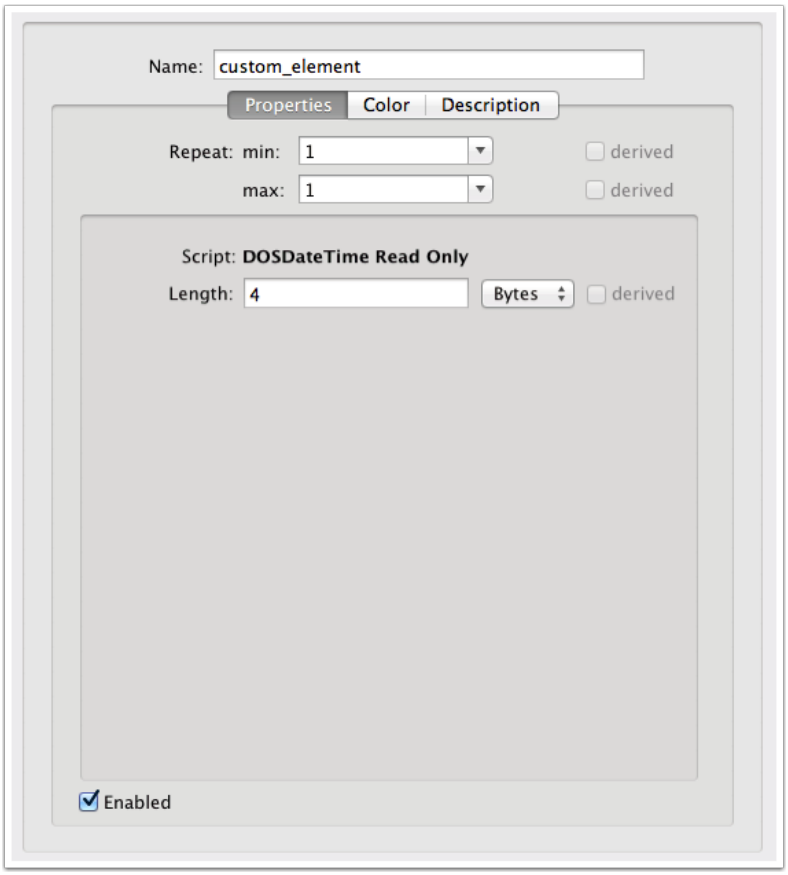
- *Repeat min* - The minimum repeat count. Parsing fails if that number is not reached.
- *Repeat max* - The maximum repeat count. Parsing stops if that number is reached. Select *unlimited* if the element should fill the remaining space (determined by the enclosing structure).
- *Length* - The element length in bits or bytes. You can also write here an expression that may contain element names of elements parsed before.
- *Fixed Values* - If *Must Match* is set one of these values must occur in the file to be parsed. Write the values as hex bytes.

- *Must match* - If this flag is set the enclosing structure is only parsed successfully if one of the *fixed values* is found.

Custom Element Properties 

The script used to parse or translate back to the file is chosen when the custom element is created. The script is copied to the grammar so it doesn't depend on the scripts stored on your disk.

Figure 4.5. Custom element properties

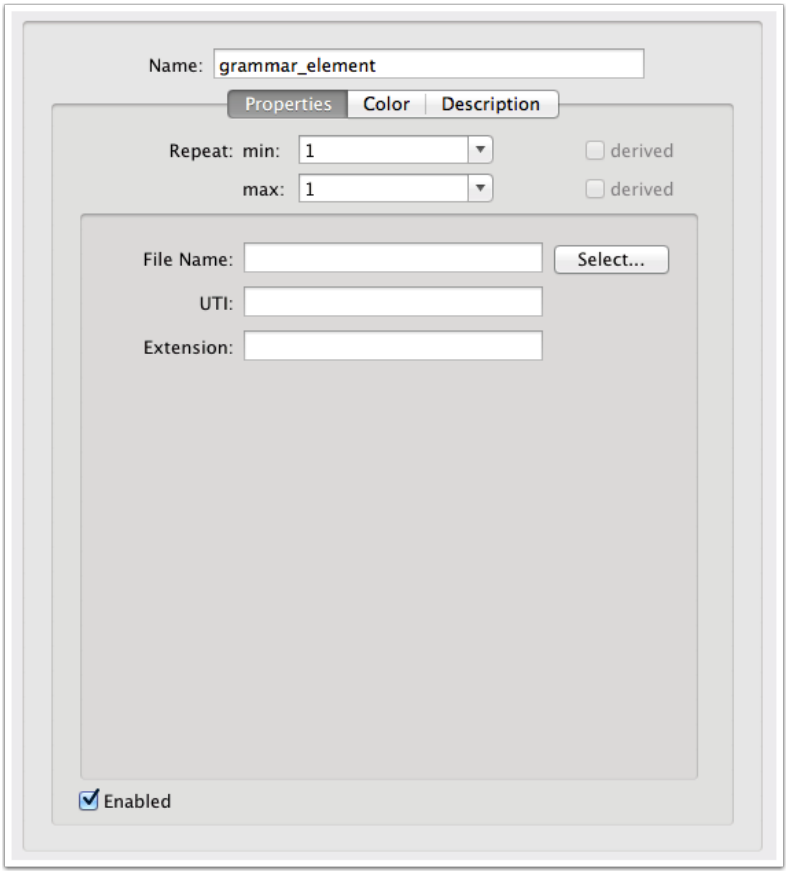


- *Repeat min* - The minimum repeat count. Parsing fails if that number is not reached.
- *Repeat max* - The maximum repeat count. Parsing stops if that number is reached. Select *unlimited* if the element should fill the remaining space (determined by the enclosing structure).

Grammar Element Properties 

The grammar element parses parts of a file using an external grammar file. This makes sense for file formats like Exif that can occur inside other file formats.

Figure 4.6. Grammar element properties

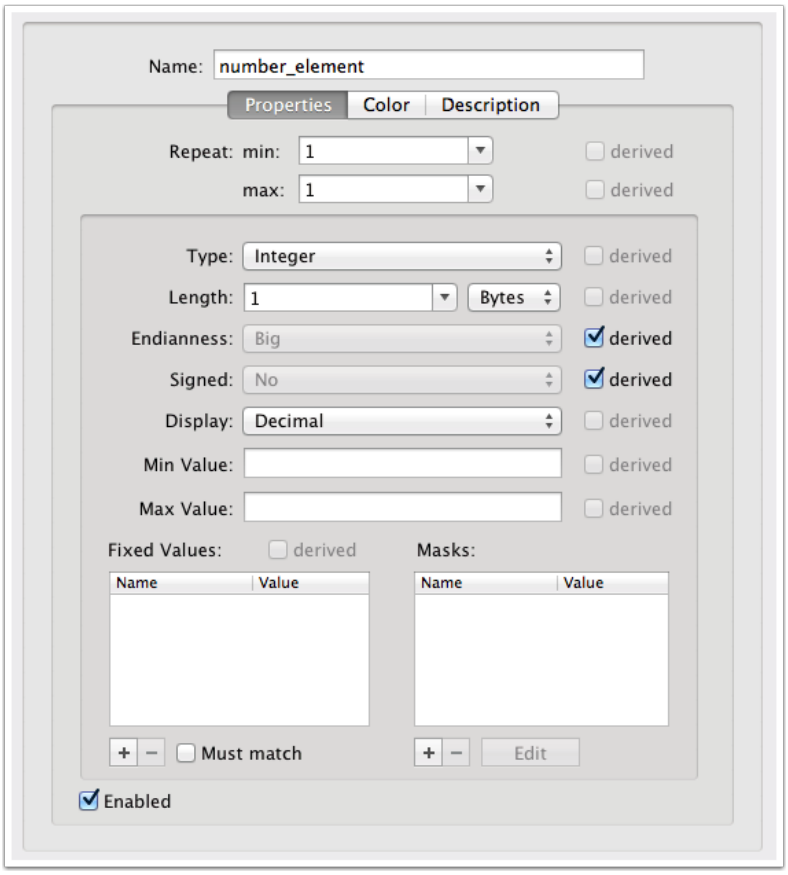


- *Repeat min* - The minimum repeat count. Parsing fails if that number is not reached.
- *Repeat max* - The maximum repeat count. Parsing stops if that number is reached. Select *unlimited* if the element should fill the remaining space (determined by the enclosing structure).
- *File Name* - Name of the grammar file to be used. If an absolute path is specified it is always used. If only the grammar file name is given the grammar is searched first in the directory of the referencing grammar, next in the directory where installed grammars are stored.

Number Element Properties

Number elements are used for any kind of numbers - float or integer.

Figure 4.7. Number element properties



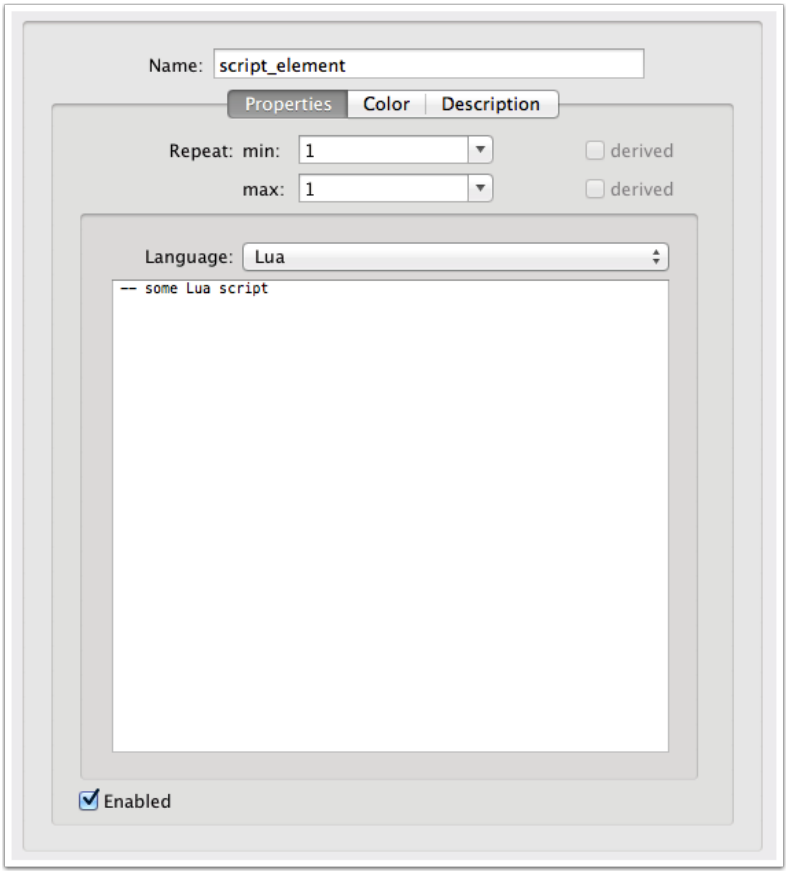
- *Repeat min* - The minimum repeat count. Parsing fails if that number is not reached.
- *Repeat max* - The maximum repeat count. Parsing stops if that number is reached. Select *unlimited* if the element should fill the remaining space (determined by the enclosing structure).
- *Type* - Number type (Integer/Float).
- *Length* - The element length in bits or bytes (up to 64 bits). You can also write here an expression that may contain element names of elements parsed before.
- *Endianness* - Byte order of number in file. Read more about this in the glossary
- *Signed* - Should the bytes in the file be interpreted as signed or unsigned number? Signed numbers are read as two's complement.
- *Display* - Select here how to display the number in the results tree view besides the hex editor.
- *Min Value* - The lowest value this number can have. If *Must match* is set parsing of the enclosing structure fails if this constraint is violated.
- *Max Value* - The highest value this number can have. If *Must match* is set parsing of the enclosing structure fails if this constraint is violated.
- *Fixed Values* - If *Must Match* is set one of these values must occur in the file to be parsed. The values are interpreted depending on the number display format.

- *Masks* - If you want to show in the results view that certain bits or bit combinations do match you cann add masks and and different values for each of them.
- *Must match* - If this flag is set the enclosing structure is only parsed successfully if one of the *fixed values* is found and the number is within *Min Value* and *Max Value*.

Script Element Properties 

The script element allows to inject little scripts that are executed while a file is parsed. A typical usage of it is to set the endianness to be used depending on certain bytes in a file (e. g. in the TIFF file format). Scripts can be written in Lua or Python, whatever you like more.

Figure 4.8. Script element properties

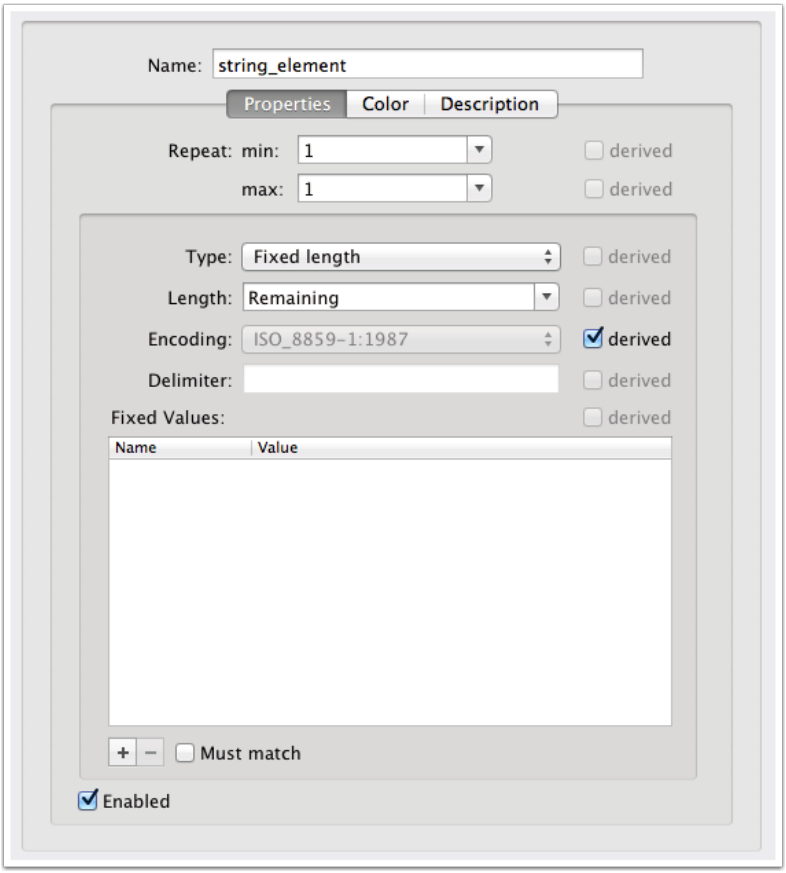


- *Repeat min* - The minimum repeat count. Parsing fails if that number is not reached.
- *Repeat max* - The maximum repeat count. Parsing stops if that number is reached. Select *unlimited* if the element should fill the remaining space (determined by the enclosing structure).

String Element Properties

The string element is used for different kinds of strings.

Figure 4.9. String element properties

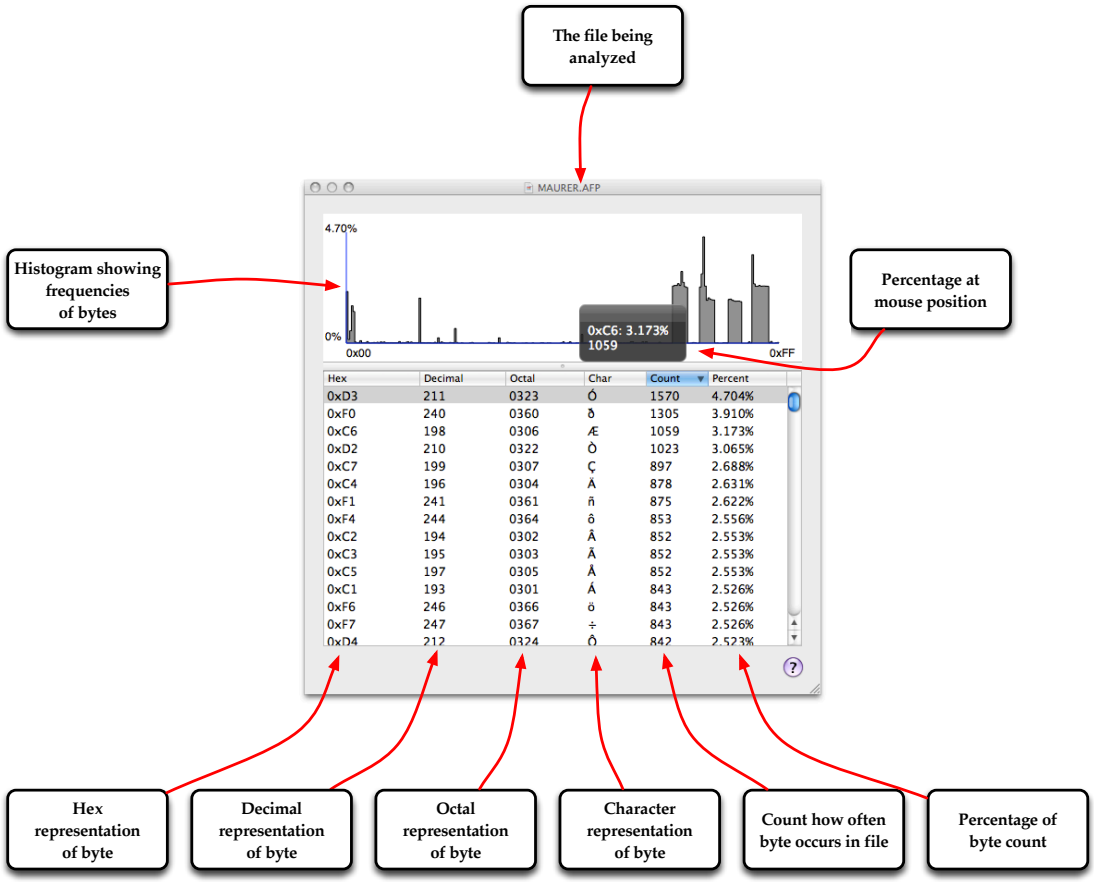


- *Repeat min* - The minimum repeat count. Parsing fails if that number is not reached.
- *Repeat max* - The maximum repeat count. Parsing stops if that number is reached. Select *unlimited* if the element should fill the remaining space (determined by the enclosing structure).
- *Type* - Strings can be interpreted as *Fixed length*, *Zero terminated*, *Delimiter terminated* or *Pascal*. Fixed-length strings are expected to consist of exactly as many bytes as specified in *Length*. Zero-terminated strings are a special case of delimiter-terminated strings. They are expected to end with a character of value zero (also strings of multi-byte characters). For pascal strings the first character is interpreted as the actual string length. You can also specify a length that the pascal string consumes in any case, independently of the actual string length.
- *Length* - Length of the string in bytes. Here an expression can be used that contains names of elements parse previously.
- *Encoding* - The encoding of the string in the file.
- *Delimiter* - Delimiter of a delimiter-terminated string. Specify here the byte sequence that ends a string in hex.
- *Fixed Values* - If *Must Match* is set one of these values must occur in the file to be parsed. The values are translated automatically depending on the encoding.

The Histogram

When beginning to analyze a binary file, especially if you don't know which format it has, a histogram can be quite useful. Histograms in Synalyze It! show you at a glance the frequency of all bytes and provide an impression of the characteristics of a file. In many file formats you'll see that certain bytes are more frequent than others; usually those bytes are an essential part of the basic format structure like record separators. An equally-leveled histogram is mostly evidence of compressed or encrypted files.

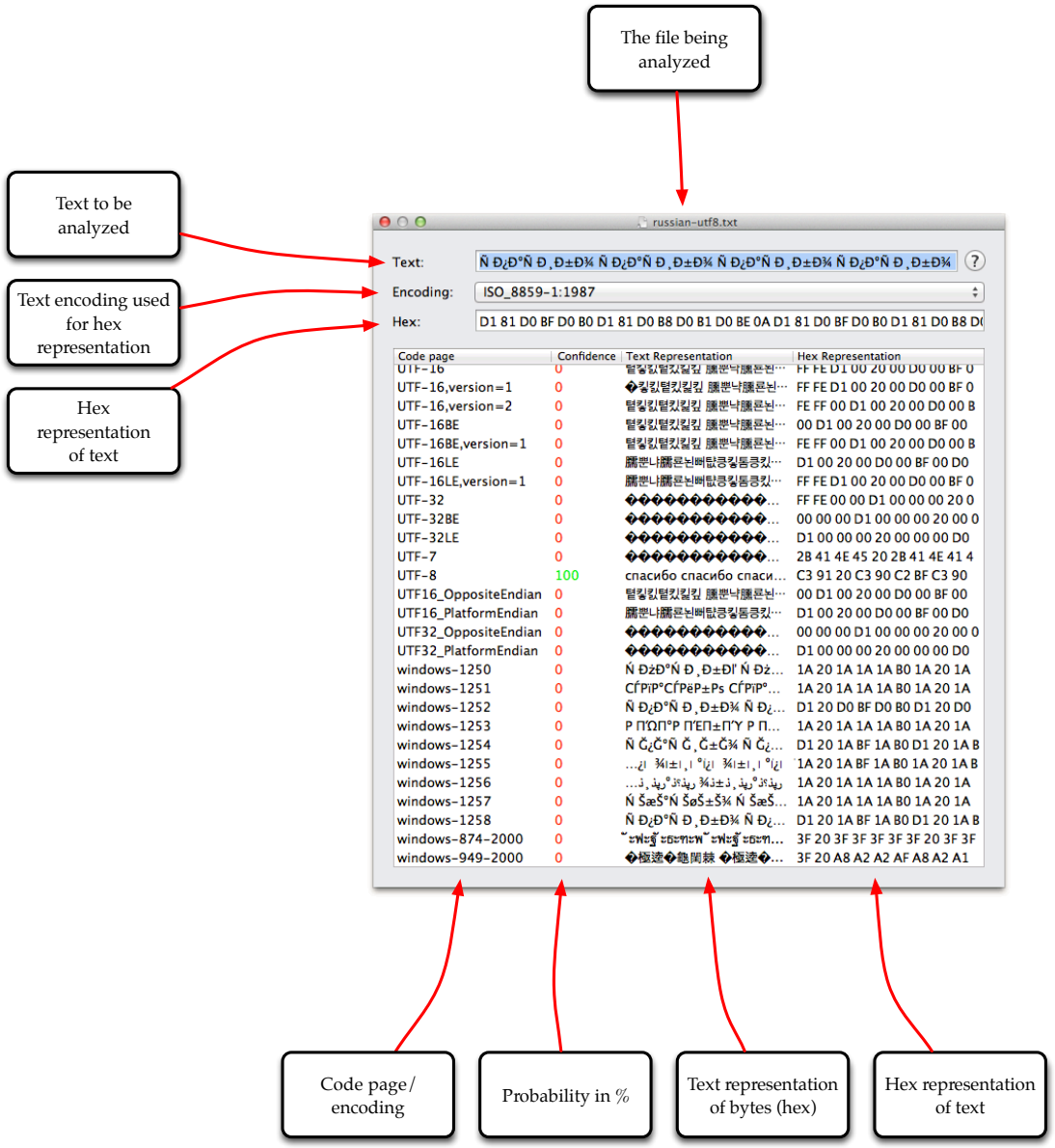
Figure 4.10. Parts of the Histogram dialog



Compare Text Encodings

In cases where you are not sure how text is encoded in a certain file the code page comparison dialog can be an indispensable help. It displays a sequence of bytes translated to text via dozens of encodings. Additionally a confidence value is computed that tells you the probability that an encoding matches. The table shows both a translation of the text at the top to bytes and a translation of the bytes at the top to text with all available encodings. For more information about text or character encodings, see Text encodings in the glossary.

Figure 4.11. Encoding comparison dialog



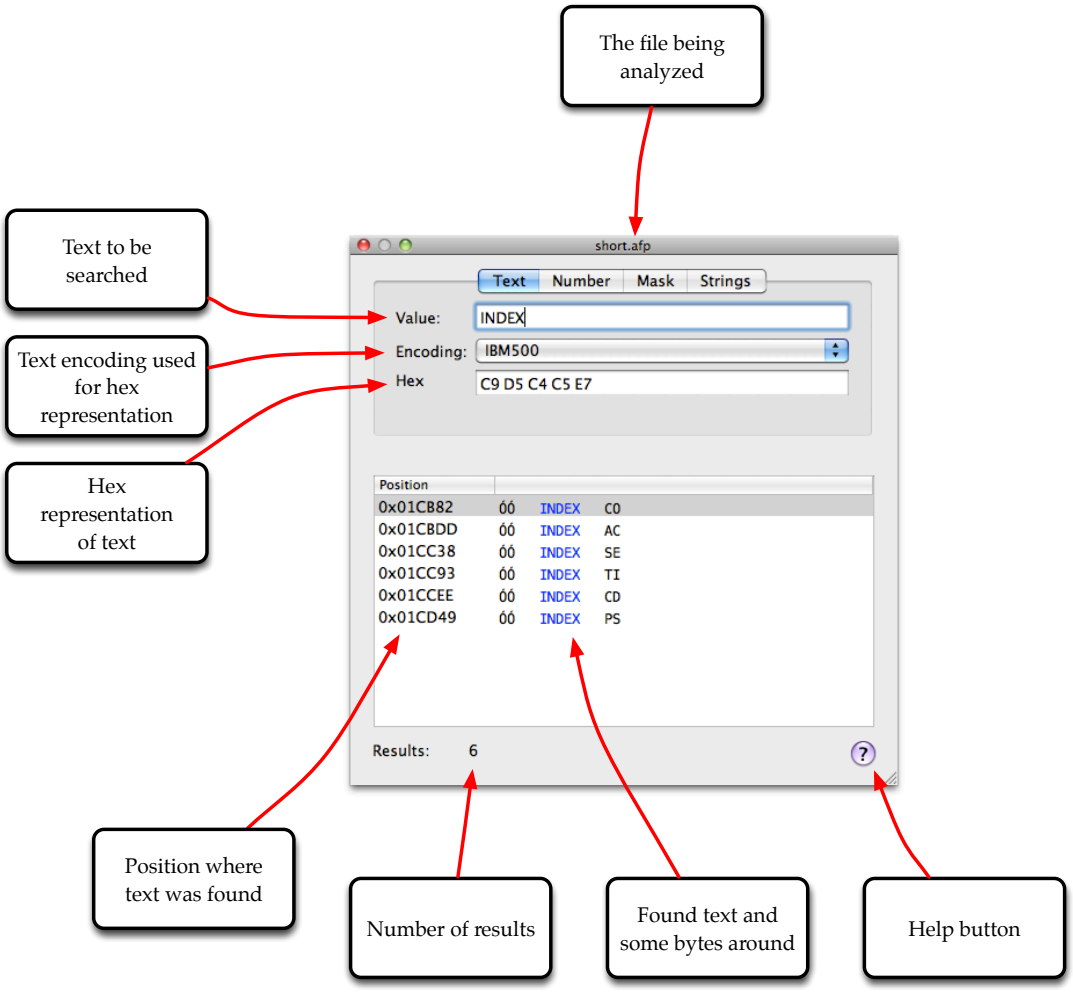
Find Dialog

Synalyze It! lets you search not only for text but also numbers, masks and display all strings in a file. You can open the find dialog in the contextual menu of selected text or by pressing **Cmd**+**E** (find selected text). **Cmd**+**F** opens the search dialog with the text from the find pasteboard which may be filled by the search dialog in another application. The search results are updated while you type. Double click a result in the find dialog to jump to the file position of the search result. You may use the find dialog not only to search but also to convert text or numbers to bytes or to get a binary representation of a few bytes.

Text Search

Unlike most other hex editors Synalyze It! lets you select one of many text encodings to have full control over the bytes that are actually searched. You can edit either the text or the hex representation of the searched bytes.

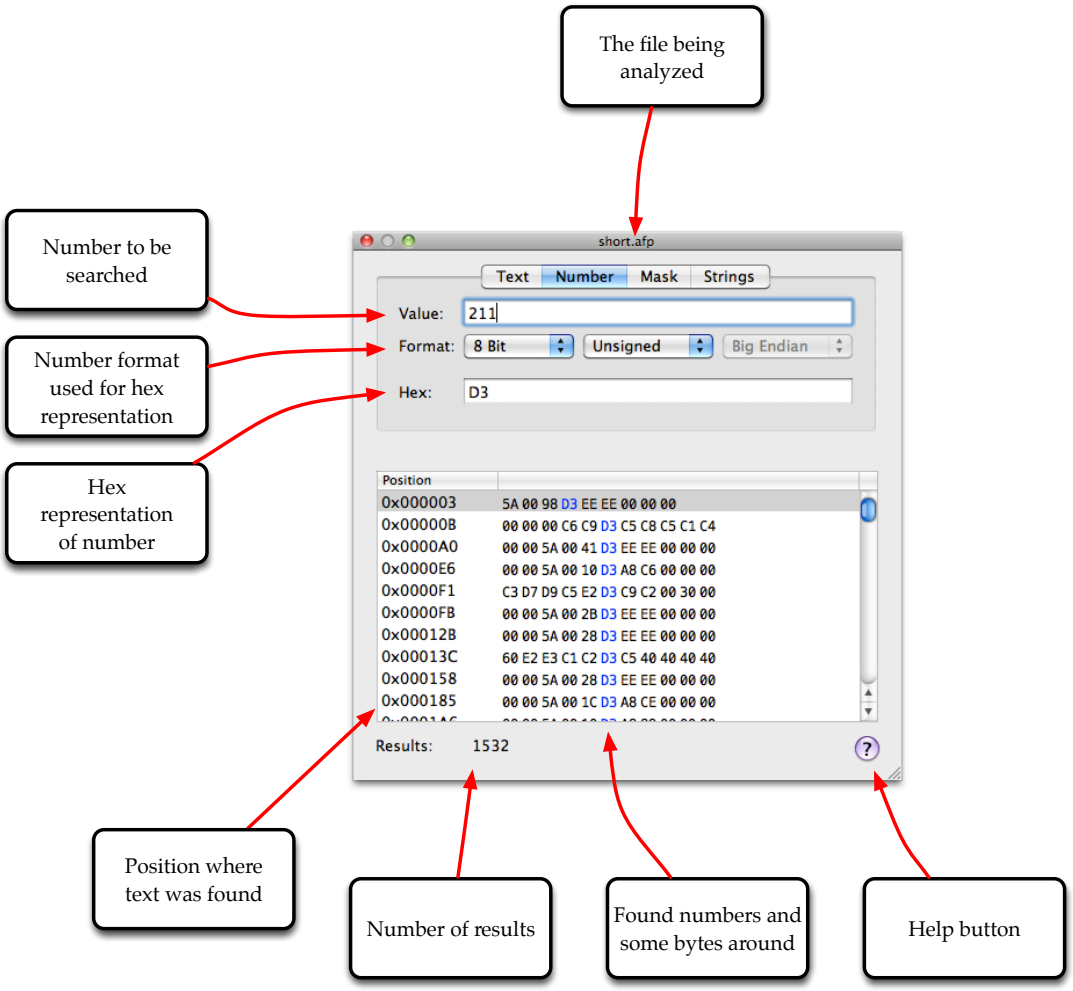
Figure 4.12. Text search dialog



Number Search

The number search feature makes it easy to search for an integer number in a file. Not only you can define the number length but also if it is represented in little or big endian format.

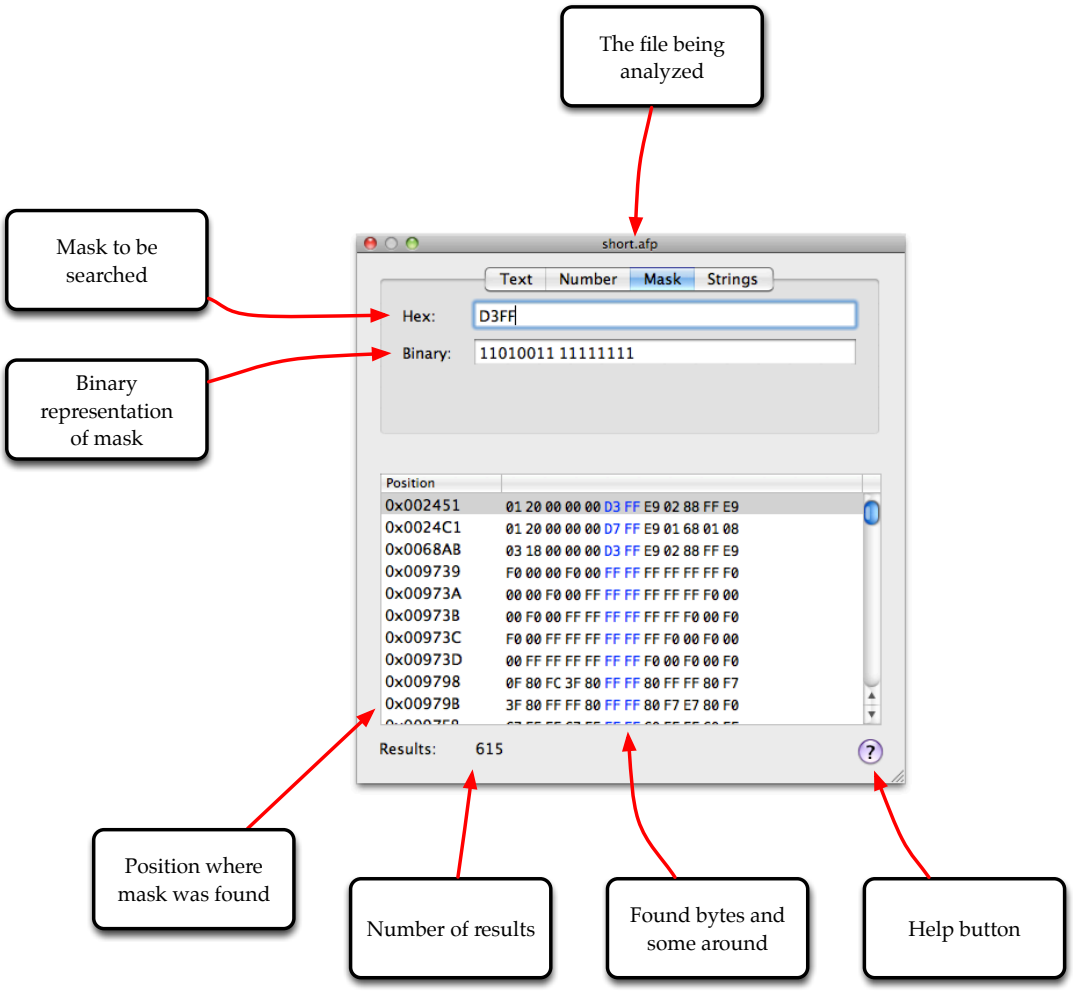
Figure 4.13. Number search dialog



Mask Search

If you want to find a sequence of bytes with certain bits set, the mask search was developed for you.

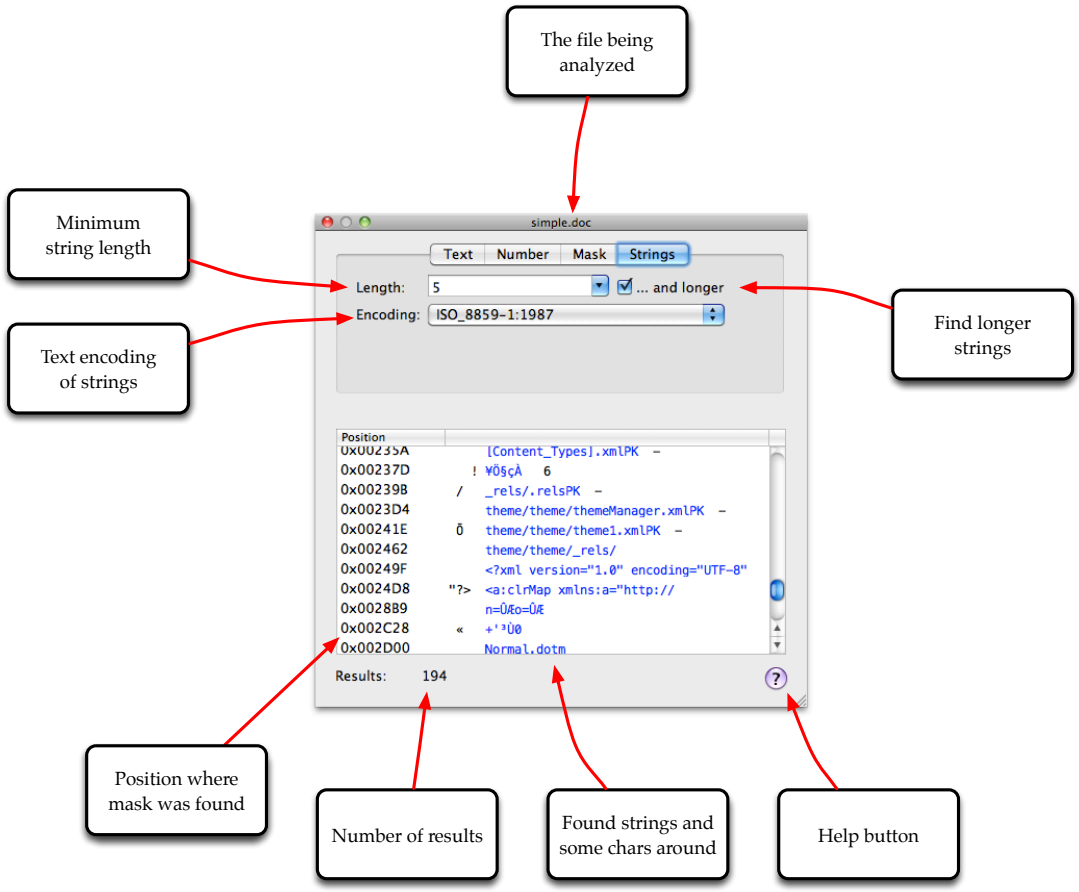
Figure 4.14. Mask search dialog



Strings

There's a Unix tool that offers the same functionality on the command line however in Synalyze It! you can even select in which text encoding the strings should be found.

Figure 4.15. Strings dialog



Checksums dialog

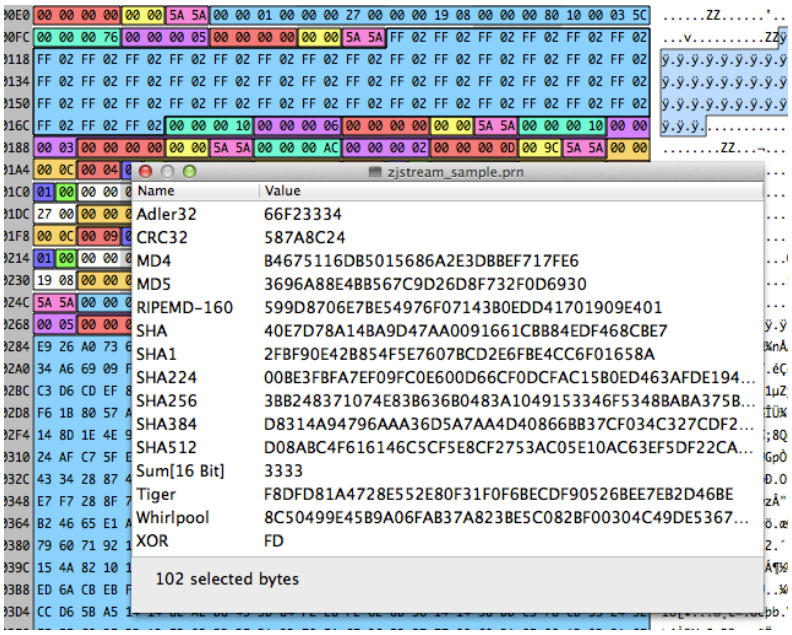
Binary files often contain check sums to detect or even correct unwanted modifications. Synalyze It! Pro lets you compute them on the currently selected bytes. All supported hash algorithms are immediately recomputed if you change the selection.

Supported checksums/hash values:

- *Adler-32* - Used for example in zlib
- *CRC32* - Cyclic Redundancy Check
- *MD4* - MD4 Message-Digest Algorithm
- *MD5* - MD5 Message-Digest Algorithm
- *RIPEMD-160* - RACE Integrity Primitives Evaluation Message Digest
- *SHA* - Secure Hash Algorithm
- *SHA-1* - Secure Hash Algorithm 1

- *SHA-224* - Secure Hash Algorithm 2 with 224 bits
- *SHA-256* - Secure Hash Algorithm 2 with 256 bits
- *SHA-384* - Secure Hash Algorithm 2 with 384 bits
- *SHA-512* - Secure Hash Algorithm 2 with 512 bits
- *Sum[16 Bit]* - Sum of all bytes in an unsigned 16-bit integer
- *Tiger* - Tiger hash value with length 192 bits. Optimized for 64-bit platforms
- *Whirlpool* - Whirlpool cryptographic hash function
- *XOR* - All selected bytes XOR'ed

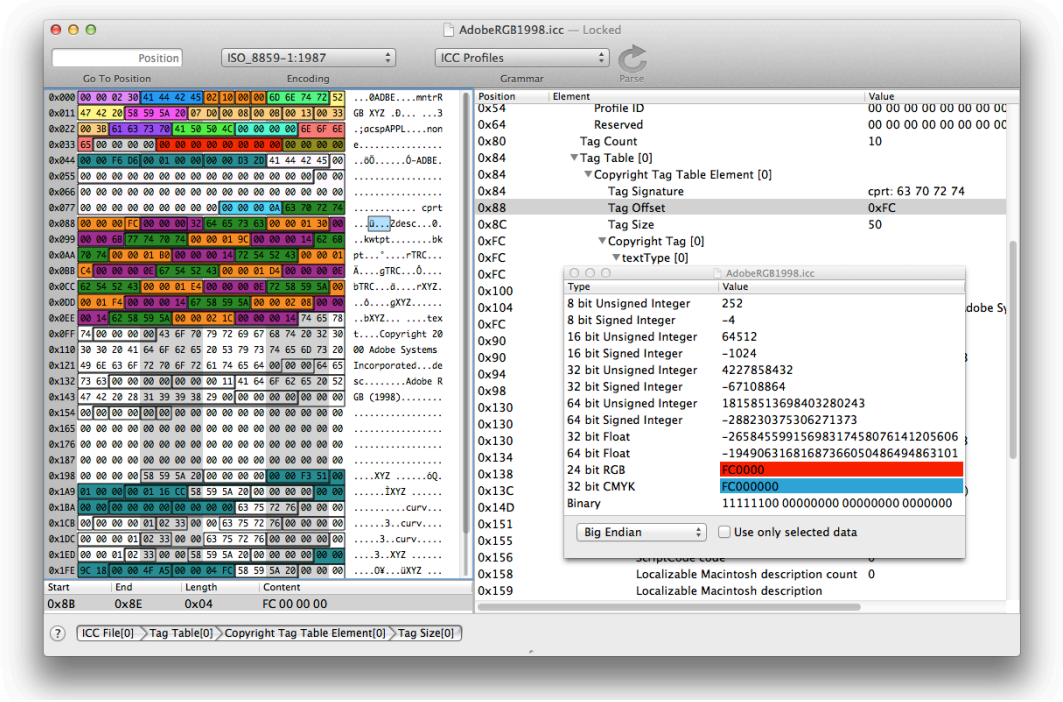
Figure 4.16. Checksums dialog



Data Panel Dialog 

Mostly you'll work in binary files with a hexadecimal representation of the contents. However humans are more familiar with decimal numbers. Synalyze It! Pro displays selected bytes in common variable sizes (8, 16, 32 and 64 bit), signed and unsigned. Additionally RGB and CMYK colors are shown for the selected bytes as well as a binary translation. The bytes can be interpreted alternatively in little or big endian.

Figure 4.17. Data Panel



Chapter 5. Scripting 

> I can't remember what my first script was.
>
> —Tom Stoppard

Even without scripting Synalyze It! is quite a useful tool that allows to analyze files of many file formats. However, there are rare cases that are better handled by custom scripts so the user interface doesn't have to become more complex. Additionally, the scripting features of Synalyze It! Pro let you automize various tasks.

You can write scripts

- in *script elements*, for example to control the endianness of a file
- in the *script editor* for custom data types or for automatization purposes

There are different types of scripts:

- *Generic* - can be started from the menu in any context. Use this for helper functions or the like
- *Grammar* - works on grammars. Useful for importing into, exporting from or modifying grammars
- *File* - works on files. Can be used to modify opened files
- *Data type* - scripts to be used by *custom elements*
- *Process Results* - processes parsing results. Handy for exporting to an own format
- *Selection* - processes only the selected bytes in the hex editor

Scripts can be available globally or in the context of a grammar. The scripting editor shows a separate list of global scripts and for each opened grammar.

For all scripts in Synalyze It! Pro you can choose Lua or Python depending on your language preferences. See *The Script Page* for useful sample scripts. Of course, if you develop a script that may help another user, it would be great if you share it!

The scripting reference has detailed information about all available classes and methods.

Using the function `logMessage("Message")` you can debug your scripts.

The Script Element

Sometimes the standard grammar structures and elements are not enough to parse a file format. For example, in a ZIP file it is best to start at the end but Synalyze It! usually at the first byte. A script element can continue the parsing at another file offset.

```
-- Lua script that continues parsing at end of file

-- get byte view of analyzed file
byteView = currentMapper:getCurrentByteView()

-- get file length
fileLength = byteView:getLength()

-- query grammar applied to file
currentGrammar = currentMapper:getCurrentGrammar()

-- get the structure we want to apply
structure = currentGrammar:getStructureByName
                                  ("ZIP end of central directory record")

-- parse at file offset fileLength-22 the structure queried above
bytesProcessed = currentMapper:mapStructureAtPosition
                                          (structure, fileLength-22, 22)
```

Another common application of script elements is to select dynamically if the number elements should be parsed in little or big endian mode.

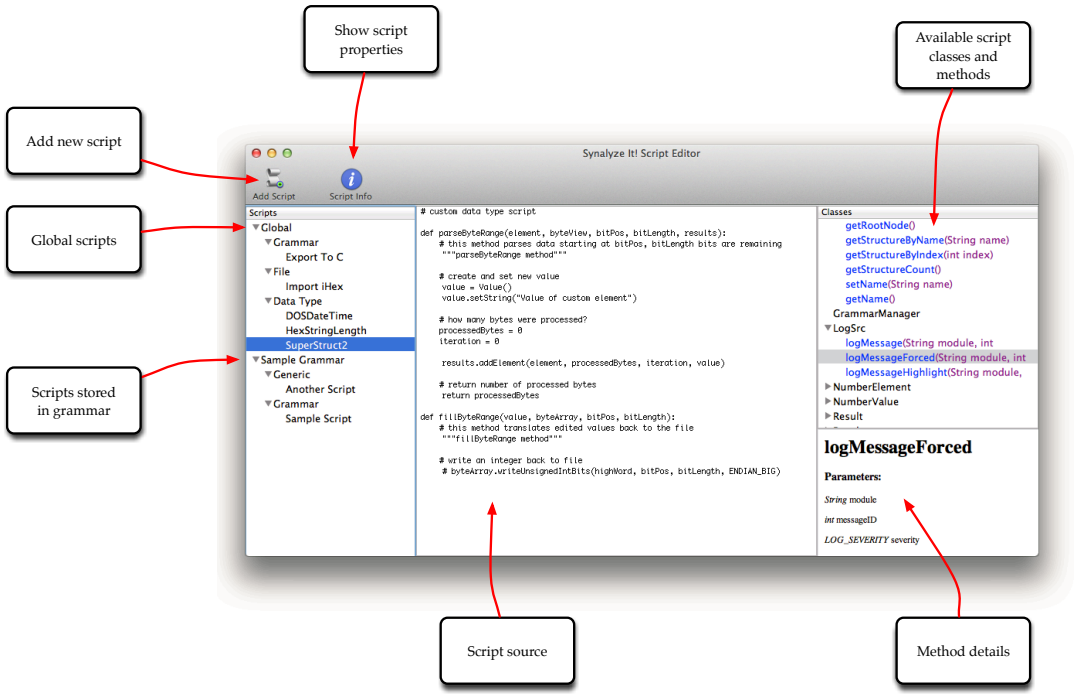
```
-- Lua script that sets endianness depending on value of previous element

-- get collection with results so far
results = currentMapper:getCurrentResults()

-- get latest added result
lastResult = results:getLastResult()

-- access the parsed value
value = lastResult:getValue()

-- get the value parsed
signature = value:getString()

if (signature == "JZJZ") then
    currentMapper:setDynamicEndianness(synalysis.ENDIAN_BIG)
else
    currentMapper:setDynamicEndianness(synalysis.ENDIAN_LITTLE)
end
```

In order to make this work the endianness of the number elements in the grammar has to be set to *dynamic*.

The Script Editor

All scripts apart from the ones in scripting elements are edited in the script editor.

Figure 5.1. Script editor window



The Custom Element

The script that implements the logic of a custom element is edited in the script editor. In the custom element you only select which script should be used so the code doesn't have to be copied. (You also don't like redundancy, right?)

There are two tasks every structure element has to perform, be it a number, string or custom element:

1. Parse data from a file and create a representation that can be displayed on screen

2. Translate an edited value back to file

The following two script functions should be implemented accordingly (Python syntax):

```
def parseByteRange(element, byteView, bitPos, bitLength, results):
    """parseByteRange method"""

def fillByteRange(value, byteArray, bitPos, bitLength):
    """fillByteRange method"""
```

Please check out http://synalysis.net/scripts.html to find illustrated samples.

Generic Scripts

There may be scripts you want to run comfortably via the Script menu that are not related to grammars or other files. Generic scripts don't implement a certain script function, the whole code is executed once you run them.

Grammar Scripts

While you can create grammars in the grammar editor and add elements via the hex editor there are cases where scripts make your life much easier.

Grammar scripts are intended for mainly three tasks:

- Create or extend a grammar from an external source, for example an XML or .h header file
- Modify a grammar
- Export a grammar to some other representation

In Synalyze It! Pro there's already an export to .dot GraphViz files built-in however with scripts only your programming skills are the limit ;-)

Grammar scripts can contain three method but only the `processGrammar(grammar)` method is required (Python syntax).

```
def init():
 print "init"

def processGrammar(grammar):
 print "grammar"

def terminate():
 print "terminate"
```

File Scripts

Laziness is one of the main incitements that motivates people to automize work with their computers. Since you're a computer expert you probably don't want to perform tedious tasks when editing a file — be it binary or text.

File scripts allow you to create or manipulate files in any possible way. The `processByteArray(byteArray)` method must be implemented, `init()` and `terminate()` are optional.

```
def init():
 print "init"

def processByteArray(byteArray):
 print "byteArray"

def terminate():
```

```
 print "terminate"
```

On http://synalysis.net/scripts.html you see how a file script can be implemented.

Result Processing Scripts

Now if you created your grammar and can see the beautiful tree that shows all the structures and elements of your files, what comes next?

In Synalyze It! Pro you can export the whole tree as an XML or text file but your own script could do so much more!

An obvious application is an export to C structures as shown on http://synalysis.net/scripts.html however I'm sure there are many other use cases.

There are three methods you can implement. `init()` is called first, then `processResult(result)` for every single result, finally `terminate()` can be used to clean up.

```
def init():
 print "hello init"

def processResult(result):
 print "hello result"

   type = result.getType()

 if type == RESULT_STRUCTURE_START_TYPE:
  print("Structure Start")
 else:
  print("other")

 level = result.getLevel()
  print (level)

def terminate():
 print "hello terminate"
```

Selection Scripts

Often it's useful to process a script only on the bytes selected in the hex editor. Selection scripts are only available if there is a selection.

The `processByteRange(byteView, byteArray, bytePos, byteLength)` method is mandatory.

```
def processByteRange(byteView, byteArray, bytePos, byteLength):
 print "process byte range here..."
```

On http://synalysis.net/scripts.html you see how a selection script can be implemented.

Chapter 6. How Do I...

> I feel very adventurous.
>
> There are so many doors to be opened, and I'm not afraid to look behind them.
>
> — Elizabeth Taylor

Even if you have some experience in binary file formats it may not be obvious how to translate this to grammars in Synalyze It!

This chapter will cover common cases and questions asked by users. Feel free to contact me in case you miss something here.

Structure Inheritance

In many file formats like PNG there are structures that comprise equal as well as differing elements (see also inheritance in the glossary). In Synalyze It! grammars you first create the parent structure with all similar elements. This structure and its child structures must be separate from the main structure that encloses the whole file.

Figure 6.1. Example of inherited structures (PNG chunks)

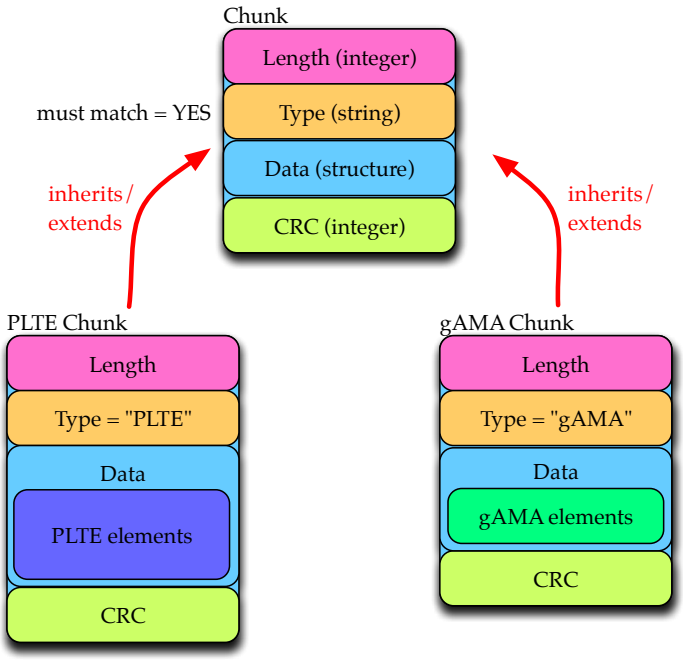
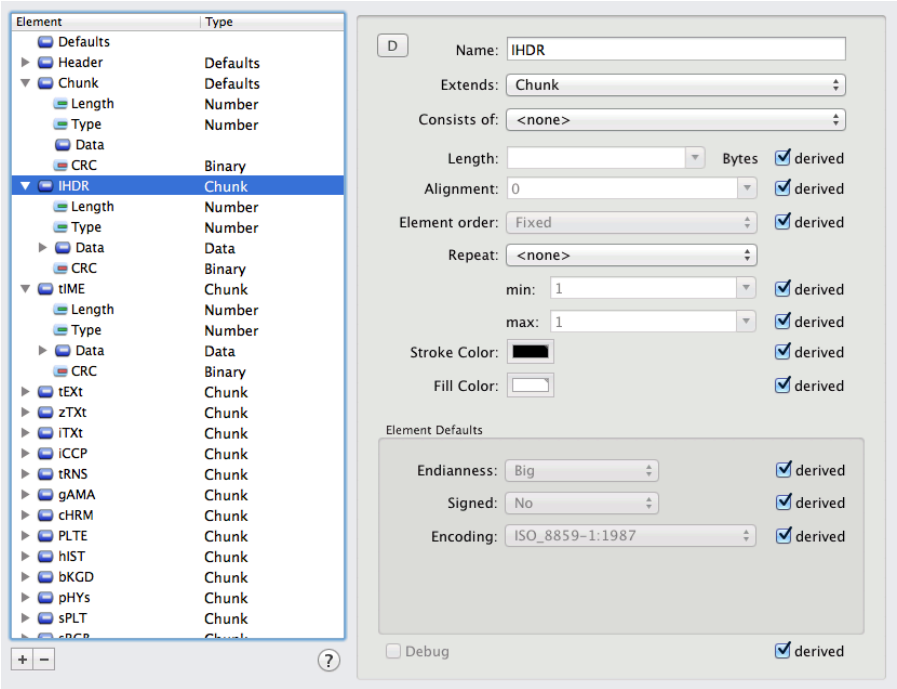


Figure 6.2. Screenshot of inherited Chunk structure



Step by Step

1. Create the parent structure by clicking *Add Structure* in the toolbar. Give the structure a meaningful name.

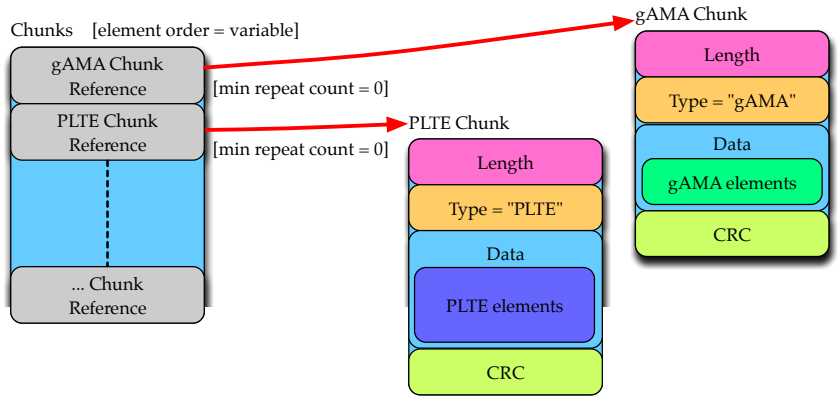
2. Create all elements that are common among all child structures. Set all properties like *must match*, colors, or endianness.

3. Create the child structures by clicking *Add Structure* in the toolbar. Give the structures meaningful names, add the elements that are special for each of them and set the correct properties.

Match the right Structure

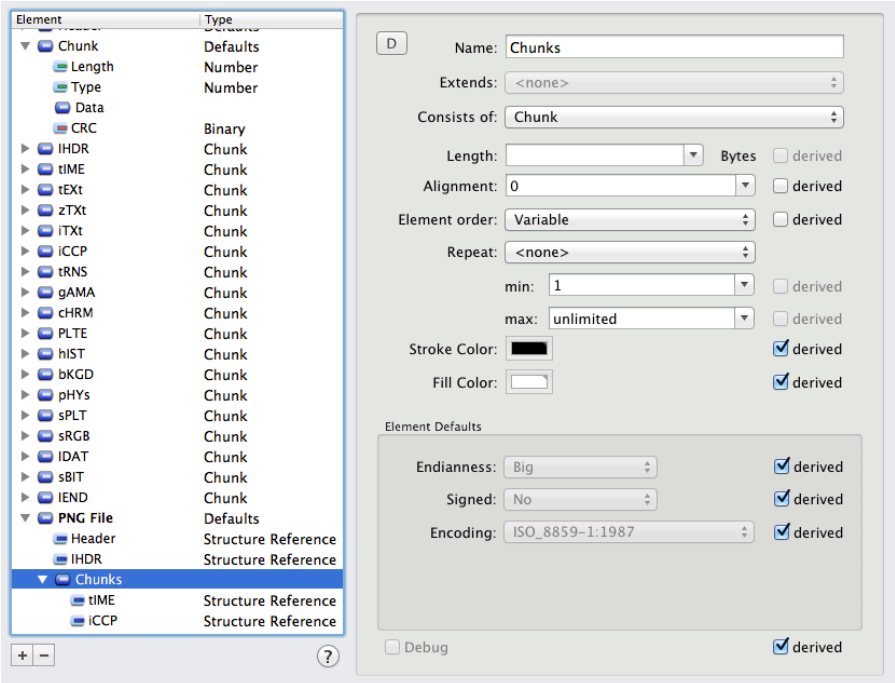
In the section above you learned how to design structures with minimal effort. The next question is how to select the right structure automatically. The chunk structures in figure Figure 6.1, “Example of inherited structures (PNG chunks)” are already prepared for automatic selection by the Synalyze It! parser. The *type* element of the PLTE and gAMA chunk structures contains one so-called fixed value that must be present in the file at a certain file position. The *must-match* flag is derived from the parent structure.

Figure 6.3. Example of automatically matched structures



The automatic structure mapping can be compared to switch/case constructs in programming languages. There must be a criteria that determines which structure to apply. This criteria lies in the elements of the structures themselves, the *must-match* flag lets structures only be mapped if any of the specified *fixed values* is found in the file.

Figure 6.4. Screenshot of Chunks structure



Step by Step

1. Create a structure that will select one of multiple structures. Set the element order to *variable* to use the switch/case logic instead of sequential processing. In the screenshot this is the *Chunks* structure. Set *Repeat max* to *unlimited* if multiple structures should be parsed

2. Create all the structures you want to select from by clicking *Add Structure* in the toolbar. Set the *must match* check box for all elements which have fixed values or min/max values that decide if a structure should match at a certain position in the file.

3. Create structure references in the structure you created first. The *Repeat min* field is automatically set to zero for these references which indicates they are optional.

Often you'll use inherited structures for selection of one-of-many structures because mostly the criteria that decides which structure should be applied is in the same structure element like *Type* in the PNG example.

In case you want to parse files which contain structures you didn't define in the grammar you can add a reference to the parent structure to the "select structure" because it doesn't have the constraints of the child structures and matches always.

Chapter 7. Support

> I'm so lucky. I have such a great support system. All I have to do is run.
>
> —Cathy Freeman

If you still face problems after reading this manual, there are different ways to ask for help.

Fogbugz allows to enter issues directly into the Synalyze It! issue tracking system.

Emails can be sent to `<support@synalysis.com>` or via the web form on synalysis.net.

I'm open for any feature requests or any other suggestions, please send a note to `<ideas@synalysis.com>` to help improve the application.

Thanks for using Synalyze It! We look forward to hearing from you :-)

Tip

For further information about Synalyze It! visit http://www.synalysis.net

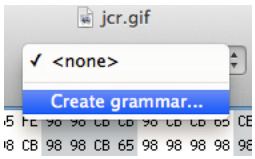
Chapter 8. Reverse Engineering

> The hidden harmony is better than the obvious.
>
> —Pablo Picasso

The term reverse engineering connotes usually something forbidden that only hackers do. However there are situations where reverse engineering is totally legal, fun and useful.

Let's assume there is a file you're interested in and you don't know much about its format. The first step you can do is to look at the histogram and check if there are bytes that are more frequent than others. Often those bytes play a special role in the file format. Zero bytes for example are often used to end strings or to fill the unused space of elements.

Figure 8.1. Create a grammar from the file to be analyzed

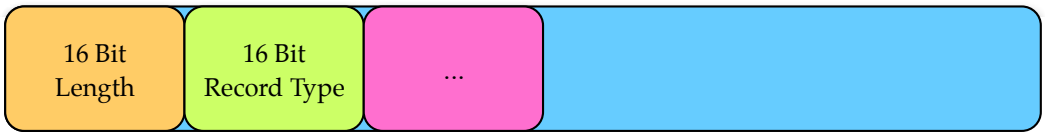


To start creating an own grammar for your file format you simply click "Create grammar..." in the grammar selection toolbar item. By this a new grammar document is created that contains already information about file extension and/or type of the file to be analyzed.

If you are in the fortunate position that you have some control over the generation of the file, you can try to save the file with little changes. For example, if the file is a saved score of some game, produce files with as little difference as possible. In many cases this will lead you to the relevant bytes and fields. Often it's easy to map the data you know from the generating program to the bytes in the file. With this approach you try to work from the inside out. First you identify single data elements, then the structures around them.

The alternative way is to find repeating patterns in the file that correspond with the record structure of the file. Sometimes it helps to scroll quickly though the file and let the eye detect sections of different content in the file. The next step is to search for bytes that could hold the lengths or file offsets of those sections.

Figure 8.2. A sample record



To let Synalyze It! parse the file it's necessary to learn incrementally how the file is constructed. Sometimes you'll find some hints in newsgroups, however even without prior knowledge there are chances to analyze formerly completely unknown file formats. You should be aware of common ways how binary file formats are built. In many cases you'll see a hierarchical structure that consists of elements called records or chunks. Those records often contain a field that holds the

length of the record and an identifier that tells the reading program which type of record is to be read.

When you search for certain lengths or file offsets in the file you definitely have to understand that the bytes can occur in reverse order. See also the explanation of endianness in the glossary.

Chapter 9. Expressions

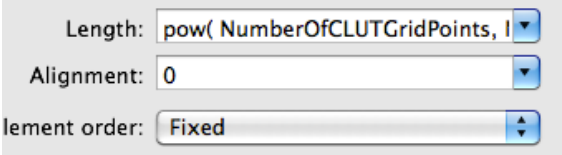
There are different places where you can use expressions instead of single numbers. The expression parsing uses primarily the ExprEval library (license).

One of the few modifications to the original expression parsing library is that expressions don't need to be ended with a semicolon in Synalyze It!. Additionally hex numbers are accepted if they are prefixed with `0x`.

Lengths of structure elements

Structure, string and binary element lengths in the grammar can be computed by an expression. The lengths of structures allow to contain the name of number variables that are inside the structure so the expression is computed when all needed variables are read.

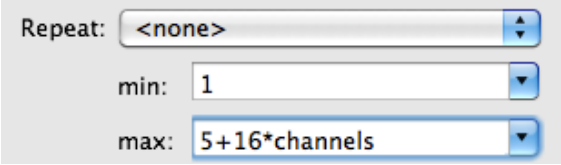
Figure 9.1. Length expression



Repeat Counts

The repeat counts of structures and structure elements can comprise expressions including variables parsed before the structure or element.

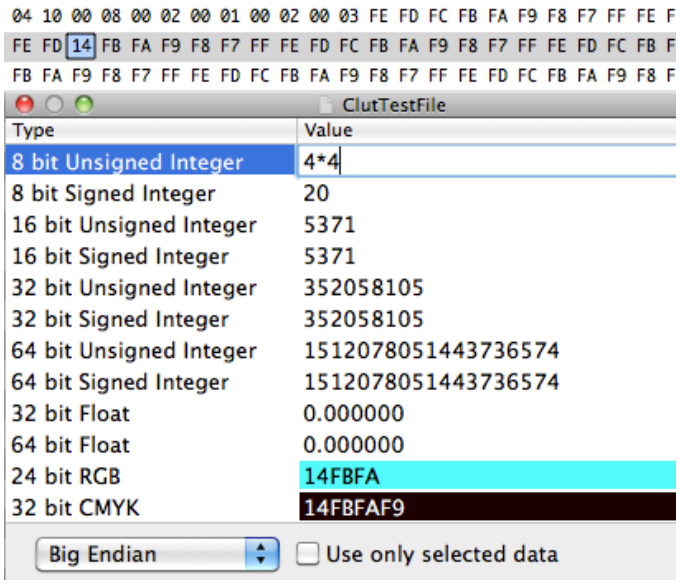
Figure 9.2. Repeat count expression



Data Panel

Normally you use the data panel to quickly see decimal values for the bytes in a file. However, not only can you enter new values but even expressions that are resolved to the correct byte representation in the file. 

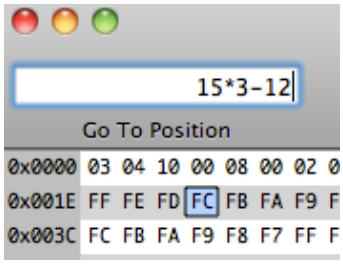
Figure 9.3. Data Panel



Jump to File Offset

The jump to file offset entry field in the toolbar is handy when you frequently jump to new file offsets. You can enter there simple hex or decimal numbers as well as expressions. 

Figure 9.4. Go to position with expression



Expression Syntax

Expressions have pretty much the same syntax as they would have on paper, with the following exception:

- The asterisk '*' must be used to multiply. *Examples:*
  - 5*6 *Valid*
  - (x+1)*(x-1) *Valid*
  - (x+1)(x-1) *Invalid*

Some functions may take reference parameters. These parameters are references to other variables. You can mix reference parameters with normal parameters. The order of the normal para-

meters must remain the same and the order of the reference parameters must remain the same. *Examples:*

- min(1,2,3,4,&mval); *&mval is a reference to a variable mval*
- min(1,2,&mval,3,4); *You may mix them inside like this.*
- min(1,2,(&mval),3,4); *You may not nest reference parameters in any way*

Expressions may also be nested with parenthesis. *Examples:*

- sin(x-cos(5+max(4,5,6*x)));
- 6+(5-2*(x+y));

If a variable is used in an expression, but that variable does not exist, it is considered zero. If it does exist then its value is used instead.

Notice: An expression can *NOT* assign to a constant and an expression can *NOT* use a constant as a reference parameter.

Order of operators

The order of operators are processed correctly in ExprEval. The parameters to functions may be evaluated out of order, depending on the function itself.

The following illustrates the order of operators:

Operator	***Direction***	***Example***
Functions and Parenthesis	N/A	(x + 5) * sin(d)
Negation	Right to Left	y = -2
Exponents	Left to Right	y = x ^ 2
Multiplication and Division	Left to Right	x * 5 / y
Addition and Subtraction	Left to Right	4 + 5 - 3
Assignment	Right to Left	x = y = z = 0

Internal Functions

The following functions are provided with ExprEval:

Function	***Min. Args***	***Max. Args***	***Min. Ref Args***	***Max. Ref Args***	***Result/Comment***
abs(v)	1	1	0	0	Absolute value of v. abs(-4.3) returns 4.3
mod(v,d)	2	2	0	0	Remainder of v/d. mod(5.2,2.5) return 0.2
ipart(v)	1	1	0	0	The integer part of v. ipart(3.2) returns 3
fpart(v)	1	1	0	0	The fractional part of v. fpart(3.2) returns 0.2
min(v,...)	1	None	0	0	The minimum number passed. min(3,2,-5,-2,7) returns -5

Function	*Min. Args*	*Max. Args*	*Min. Ref Args*	*Max. Ref Args*	*Result/Comment*
max(v,...)	1	None	0	0	The maximum number passed. max(3,2,-5,-2,7) returns 7
pow(a,b)	2	2	0	0	The value a raised to the power b. pow(3.2,1.7) returns $3.2^{1.7}$
sqrt(a)	1	1	0	0	The square root of a. sqrt(16) returns 4
sin(a)	1	1	0	0	The sine of a radians. sin(1.5) returns around 0.997
sinh(a)	1	1	0	0	The hyperbolic sine of a. sinh(1.5) returns around 2.129
asin(a)	1	1	0	0	The arc-sine of a in radians. asin(0.5) returns around 0.524
cos(a)	1	1	0	0	The cosine of a radians. cos(1.5) returns around 0.0707
cosh(a)	1	1	0	0	The hyperbolic cosine of a. cosh(1.5) returns around 2.352
acos(a)	1	1	0	0	The arc-cosine of a in radians. acos(0.5) returns around 1.047
tan(a)	1	1	0	0	The tangent of a radians. tan(1.5) returns around 14.101
tanh(a)	1	1	0	0	The hyperbolic tangent of a. tanh(1.5) returns around 0.905
atan(a)	1	1	0	0	The arc-tangent of a in radians. atan(0.3) returns about 0.291
atan2(y,x)	2	2	0	0	The arc-tangent of y/x, with quadrant correction. atan2(4,3) returns about 0.927
log(a)	1	1	0	0	The base 10 logarithm of a. log(100) returns 2
pow10(a)	1	1	0	0	10 raised to the power of a. pow10(2) returns 100
ln(a)	1	1	0	0	The base e logarithm of a. ln(2.8) returns around 1.030
exp(a)	1	1	0	0	e raised to the power of a. exp(2) returns around 7.389
logn(a,b)	2	2	0	0	The base b logarithm of a. logn(16,2) returns 4
ceil(a)	1	1	0	0	Rounds a up to the nearest integer. ceil(3.2) returns 4
floor(a)	1	1	0	0	Rounds a down to the nearest integer. floor(3.2) returns 3
rand(&seed)	0	0	1	1	Returns a number between 0 up to but not including 1.

Function	*Min. Args*	*Max. Args*	*Min. Ref Args*	*Max. Ref Args*	*Result/Comment*
random(a,b,&seed)	2	2	1	1	Returns a number between a up to and including b.
randomize(&seed)	0	0	1	1	Seed the random number generator with a value based on the current time. Return value is unknown
deg(a)	1	1	0	0	Returns a radians converted to degrees. deg(3.14) returns around 179.909
rad(a)	1	1	0	0	Returns a degrees converted to radians. rad(180) returns around 3.142
recttopolr(x,y)	2	2	0	0	Returns the polar radius of the rectangular co-ordinates. recttopolr(2,3) returns around 3.606
recttopola(x,y)	2	2	0	0	Returns the polar angle (0...2PI) in radians of the rectangular co-ordinates. recttopola(2,3) returns around 0.588
poltorectx(r,a)	2	2	0	0	Returns the x rectangular co-ordinate of the polar co-ordinates. poltorectx(3,1.5) returns around 0.212
poltorecty(r,a)	2	2	0	0	Returns the y rectangular co-ordinate of the polar co-ordinates. poltorecty(3,1.5) returns around 2.992
if(c,t,f)	3	3	0	0	Evaluates and returns t if c is not 0.0. Else evaluates and returns f. if(0.1,2.1,3.9) returns 2.1
select(c,n,z[,p])	3	4	0	0	Returns n if c is less than 0.0. Returns z if c is 0.0. If c is greater than 0.0 and only three arguments were passed, returns z. If c is greater than 0.0 and four arguments were passed, return p. select(3,1,4,5) returns 5
equal(a,b)	2	2	0	0	Returns 1.0 if a is equal to b. Else returns 0.0 equal(3,2) returns 0.0
above(a,b)	2	2	0	0	Returns 1.0 if a is above b. Else returns 0.0 above(3,2) returns 1.0
below(a,b)	2	2	0	0	Returns 1.0 if a is below b. Else returns 0.0 below(3,2) returns 0.0
avg(a,...)	1	None	0	0	Returns the average of the values passed. avg(3,3,6) returns 4

Function	*Min. Args*	*Max. Args*	*Min. Ref Args*	*Max. Ref Args*	*Result/Comment*
clip(v,min,max)	3	3	0	0	Clips v to the range from min to max. If v is less than min, it returns min. If v is greater than max it returns max. Otherwise it returns v. clip(3,1,2) returns 2
clamp(v,min,max)	3	3	0	0	Clamps v to the range from min to max, looping if needed. clamp(8.2,1.3,4.7) returns 1.4
pntchange(side1old, side2old, side1new, side2new, oldpnt)	5	5	0	0	This is used to translate points from different scale. It works no matter the orientation as long as the sides are lined up correctly. pntchange(-1,1,0,480,-0.5) returns 120 (x example) pntchange(-1,1,480,0,-0.5) returns 360 (y example)
poly(x,c1,...)	2	None	0	0	This function calculates the polynomial. x is the value to use in the polynomial. c1 and on are the coefficients. poly(4,6,9,3,1,4) returns 2168 same as $6*4^4 + 9*4^3 + 3*4^2 + 1*4^1 + 4*4^0$
and(a,b)	2	2	0	0	Returns 0.0 if either a or b are 0.0 Else returns 1.0 and(2.1,0.0) returns 0.0
or(a,b)	2	2	0	0	Returns 0.0 if both a and b are 0.0 Else returns 1.0 or(2.1,0.0) returns 1.0
not(a)	1	1	0	0	Returns 1.0 if a is 0.0 Else returns 0.0 not(0.3) returns 0.0
for(init,test,inc,a1,...)	4	None	0	0	This function acts like a for loop in C. First init is evaluated. Then test is evaluated. As long as the test is not 0.0, the action statements (a1 to an) are evaluated, the inc statement is evaluated, and the test is evaluated again. The result is the result of the final action statement. for(x=0,below(x,11),x=x+1,y=y+x) returns 55.0 (if y was initially 0.0)
many(expr,...)	1	None	0	0	This function treats many subexpressions as a single object (function). It is mainly for the 'for' function. for(many(j=5,k=1),above(j*k,0.001),many(j=j+5,k=k/2),0)

Internal Constants

The following constants are provided with ExprEval:

Constant	*Math Form*	*Value*
M_E	e	2.7182818284590452354
M_LOG2E	$\log_2(e)$	1.4426950408889634074
M_LOG10E	$\log_{10}(e)$	0.43429448190325182765
M_LN2	$\ln(2)$	0.69314718055994530942
M_LN10	$\ln(10)$	2.30258509299404568402
M_PI	π	3.14159265358979323846
M_PI_2	$\pi/2$	1.57079632679489661923
M_PI_4	$\pi/4$	0.78539816339744830962
M_1_PI	$1/\pi$	0.31830988618379067154
M_2_PI	$2/\pi$	0.63661977236758134308
M_1_SQRTPI	$1/\sqrt{(\pi)}$	0.56418958354776
M_2_SQRTPI	$2/\sqrt{(\pi)}$	1.12837916709551257390
M_SQRT2	$\sqrt{(2)}$	1.41421356237309504880
M_1_SQRT2	$1/\sqrt{(2)}$	0.70710678118654752440

Chapter 10. Scripting Reference

This chapter contains all classes and methods you can use in the scripting functions. If you feel there's something missing, please contact me.

Class ByteArray

A byte array object represents mostly a larger memory chunk. The actual storage is handled by ByteStorage objects.

Methods of `ByteArray`:

```
long getLength();
```

get length

```
deleteRange(long position,
            long length);
```

Delete range in byte array.

Parameters:

`position`	Position where to delete
`length`	Number of bytes to delete

```
fillRange(long position,
          long length,
          byte[] fillBytes);
```

Fill range in byte array.

Parameters:

`position`	Position where to fill
`length`	Number of bytes to fill
`fillBytes`	An array of bytes to fill in range

```
writeSignedInt(long position,
               long length,
               ENDIAN_TYPE endianType);
```

Write signed integer to byte array.

Parameters:

`position`	Position where to write (bytes)
`length`	Number of bytes to write

Parameters:

endianType	Endianness of number to write

```
writeSignedIntBits(long position,
                   long length,
                   ENDIAN_TYPE endianType);
```

Write signed integer to byte array (on bit level)

Parameters:

position	Position where to write (bits)
length	Number of bits to write
endianType	Endianness of number to write

```
writeUnsignedInt(long position,
                 long length,
                 ENDIAN_TYPE endianType);
```

Write unsigned integer to byte array.

Parameters:

position	Position where to write
length	Number of bytes to write
endianType	Endianness of number to write

```
writeUnsignedIntBits(long position,
                     long length,
                     ENDIAN_TYPE endianType);
```

Write unsigned integer to byte array (on bit level)

Parameters:

position	Position where to write (bits)
length	Number of bits to write
endianType	Endianness of number to write

```
insertByte(long position,
           char byte);
```

Insert byte into byte array.

Parameters:

position	Position where to insert

Parameters:

byte	The byte to insert

```
replaceByte(long position,
            char byte);
```

Replace byte in byte array.

Parameters:

position	Position where to replace
byte	The byte to replace

Class ByteView

A byte view object is a proxy to a

Methods of `ByteView`:

```
long getLength();
```

Get length of byte view.

```
byte readByte(long position);
```

Read byte from byte view (position in bytes)

Parameters:

position	Position where to read the byte

```
int readSignedInt(long position,
                  int length,
                  ENDIAN_TYPE endianType);
```

Read signed integer from byte view.

Parameters:

position	Position where to read the number
length	Length of the number in bytes
endianType	Little/big endian

```
uint readUnsignedInt(long position,
```

```
                    int length,
                    ENDIAN_TYPE endianType);
```

Read unsigned integer from byte view.

Parameters:

position	Position where to read the number
length	Length of the number in bytes
endianType	Little/big endian

```
String readString(long position,
                  int length,
                  String encoding);
```

Read string from byte view.

Parameters:

position	Position where to read the string
length	Length of the string in bytes
encoding	Encoding of the string

Class Element

An element object represents one item in a structure.

Methods of `Element`:

```
Element Element(ELEMENT_TYPE type,
                String name,
                bool setDefaults);
```

Constructor.

Parameters:

type	The type of the element
name	The name of the element
setDefaults	Set defaults for element?

```
String getName();
```

Get name.

```
 setName(String name);
```

Set name.

Parameters:

name	The new name of the element

```
String getDescription();
```

Get description.

```
setDescription(String name);
```

Set description.

Parameters:

name	The new description of the element

```
Structure getEnclosingStructure();
```

Get enclosing structure.

```
String getLength();
```

Get length.

```
LENGTH_UNIT getLengthUnit();
```

Get length unit.

```
setColorRgb(float red,
            float green,
            float blue);
```

Set fill color (RGB).

Parameters:

red	Red color component between 0.0 and 1.0
green	Green color component between 0.0 and 1.0
blue	Blue color component between 0.0 and 1.0

```
setLength(String length,
          LENGTH_UNIT lengthUnit);
```

Set length.

Parameters:

length	The new length of the element
lengthUnit	The new length unit (bits/bytes) of the element

```
ELEMENT_TYPE getType();
```

Get type.

```
bool mustMatch();
```

Get "must match" flag.

```
setMustMatch(BOOL mustMatch);
```

Set "must match" flag.

```
Value getMinValue();
```

Get minimum value.

```
Value getMaxValue();
```

Get maximum value.

```
Element getParent();
```

Get parent.

Class Grammar

A grammar with all structures and their elements

Methods of `Grammar`:

```
String getEncoding();
```

Get encoding.

```
String getDescription();
```

Get grammar description.

```
setDescription(String description);
```

Set grammar description.

Parameters:

description The description of the grammar

```
void addStructure(Structure structure);
```

Add structure.

Parameters:

structure The structure to be appended

```
void insertStructureAtIndex(Structure structure,
                            int index);
```

Insert structure at index.

Parameters:

structure The structure to be inserted

index Index where structure should be inserted

```
void deleteStructureAtIndex(int index);
```

Delete structure at index.

Parameters:

index Index where structure should be deleted

```
Structure getRootNode();
```

Get root node (structure)

```
void setStartStructure(Structure startStructure);
```

Set start structure.

```
Structure getStructureByName(String name);
```

Get structure by name.

Parameters:

name	Name of the structure to get

```
Structure getStructureByIndex(int index);
```

Get structure by index.

Parameters:

index	Index of the structure to get

```
int getStructureCount();
```

Get number of structures.

```
setName(String name);
```

Set grammar name.

Parameters:

name	The name of the grammar

```
String getName();
```

get grammar name

```
setUTI(String UTI);
```

Set UTI grammar is valid for.

Parameters:

UTI	The UTI

```
String getUTI();
```

Get UTI grammar is valid for.

```
setFileExtension(String fileExtension);
```

Set file extension grammar is valid for.

Parameters:

fileExtension	The file extension to be set

```
String getFileExtension();
```

Get file extension grammar is valid for.

Class GrammarManager

The

Methods of `GrammarManager`:

Class LogSrc

A log source object allows to write log messages to a log target, e. g. the messages window.

Methods of `LogSrc`:

```
logMessage(String module,
           int messageID,
           LOG_SEVERITY severity,
           String message);
```

Write a log message.

Parameters:

module	A domain of message IDs. Can be any string that lets you identify your messages
messageID	An ID to identify the message
severity	Severity of the message
message	The actual message

```
logMessageForced(String module,
                 int messageID,
```

```
                    LOG_SEVERITY severity,
                    String message);
```

Write a log message. It will be displayed no matter which severity is specified.

Parameters:

module	A domain of message IDs. Can be any string that lets you identify your messages
messageID	An ID to identify the message
severity	Severity of the message
message	The actual message

```
logMessageHighlight(String module,
                    int messageID,
                    LOG_SEVERITY severity,
                    String message);
```

Write a log message. The first message written with this method will be selected.

Parameters:

module	A domain of message IDs. Can be any string that lets you identify your messages
messageID	An ID to identify the message
severity	Severity of the message
message	The actual message

Class Mask

A

Methods of `Mask`:

```
String getName();
```

Get name.

```
String getDescription();
```

Get description.

```
unsigned int getValue();
```

Get value.

Class NumberElement

A number element object represents one number item in a structure.

Methods of `NumberElement`:

```
NUMBER_DISPLAY_TYPE getNumberDisplayType();
```

Get number display.

```
NUMBER_TYPE getNumberType();
```

Get number type.

```
ENDIAN_TYPE getEndianness();
```

Get endianness.

```
bool isSigned();
```

Is number element of type signed?

Class NumberValue

Methods of `NumberValue`:

```
ulong getUnsigned();
```

Get unsigned number.

```
void setUnsigned(ulong number);
```

Set unsigned number.

Parameters:

`number`	The unsigned number to be set

```
long getSigned();
```

Get signed number.

```
void setSigned(long number);
```

Set signed number.

Parameters:

number The signed number to be set

```
ulong getFloat();
```

Get floating-point number.

```
void setFloat(double number);
```

Set floating-point number.

Parameters:

number The number to be set

Class Result

objects are created during the structure mapping process. Depending on their type they refer to a structure or struct element and a value.

Methods of `Result`:

```
Value getValue();
```

Get value.

```
Mask getMask();
```

Get mask.

```
ByteView getByteView();
```

byte view.

```
int getLevel();
```

Get level.

```
int getIteration();
```

Get iteration.

```
int getStartBytePos();
```

Get start (byte).

```
int getStartBitPos();
```

Get start (bit).

```
int getByteLength();
```

Get length (bytes).

```
int getBitLength();
```

Get length (bits).

```
String getName();
```

Get name.

```
Structure getStructure();
```

Get structure.

```
Element getElement();
```

Get structure element.

Class Results

A results object contains the results of the structure mapping process

Methods of `Results`:

```
Result addStructureStart(Structure structure,
                         long startPos,
                         int iteration,
                         String name,
                         bool addSizeToEnclosing);
```

Add start of a structure to the results.

Parameters:

`structure`	The structure that was mapped
`startPos`	Where in the file was the structure mapped?
`iteration`	How often was this structure mapped consecutively? (Array of structures)
`name`	Name to show for the result
`addSize-ToEnclosing`	Add size to the enclosing structure result? Set this to true if the structure is actually contained in the enclosing structure in the result tree.

```
Result addStructureStartAtPosition(Structure structure,
                                   long startPos,
                                   int iteration,
                                   String name);
```

Add start of a structure to the results.

Parameters:

`structure`	The structure that was mapped
`startPos`	Where in the file was the structure mapped?
`iteration`	How often was this structure mapped consecutively? (Array of structures)
`name`	Name to show for the result

```
Result addStructureEnd(long endPos);
```

Add end of a structure to the results.

Parameters:

`endPos`	Where in the file did the structure end? Padding bytes are calculated automatically

---

```
Result addElement(Element element,
                  long length,
                  int iteration,
```

```
                    Value value);
```

Add a structure element to the results. Length is specified in bytes.

Parameters:

element	The structure element that was mapped
length	Length of the element in bytes
iteration	How often was this structure element mapped consecutively? (Array of structures)
value	The value resulting of the element being mapped to the file

---

```
Result addElementBits(Element element,
                      long length,
                      int iteration,
                      Value value);
```

Add a structure element to the results. Length is specified in bits.

Parameters:

element	The structure element that was mapped
length	Length of the element in bits
iteration	How often was this structure element mapped consecutively? (Array of structures)
value	The value resulting of the element being mapped to the file

---

```
cut(Result result);
```

Cut results.

Parameters:

result	First result

```
Result getLastResult();
```

Get last result.

```
Result getPrevResult(Result result);
```

Get previous result.

Parameters:

result	The result you want the predecessor for

```
Result getResultByName(String name);
```

Get result by name.

Parameters:

name	Name of the result you're looking for

Class String

The

Methods of `String`:

Class StringElement

A string element object represents one binary item in a structure.

Methods of `StringElement`:

```
String getEncoding();
```

Get encoding.

```
setEncoding(String encoding);
```

Set encoding.

Parameters:

encoding	The new string encoding of the element

```
int getBytesPerChar();
```

Get bytes per character.

```
STRING_LENGTH_TYPE getLengthType();
```

Get string length type.

Class StringValue

Methods of `StringValue`:

```
String getString();
```

Get string.

```
setString(String string);
```

Set string.

Parameters:

`string`	The string to be set

Class Structure

A structure object represents a structure in a grammar.

Methods of `Structure`:

```
Grammar getGrammar();
```

Get grammar.

```
setAlignment(int alignment);
```

Set alignment.

Parameters:

`alignment`	The alignment value

```
int getAlignment();
```

Get alignment.

```
String getDescription();
```

Get structure description.

```
setDescription(String description);
```

Set structure description.

Parameters:

description The description of the structure

Parameters:

description The default encoding

```
String getName();
```

Get name.

```
setName(String name);
```

Set name.

Parameters:

name The new name of the structure

```
setDisabled(BOOL disabled);
```

Disable/Enable structure for parsing.

Parameters:

disabled Disable/enable the structure

```
setLength(String length,
          LENGTH_UNIT lengthUnit);
```

Set length of structure.

Parameters:

length The new length of the structure

lengthUnit The new length unit of the structure

```
setRepeatMin(String repeatMin);
```

Set minimum repeat count of structure.

Parameters:

repeatMin The new min repeat count of the structure

```
setRepeatMax(String repeatMax);
```

Set maximum repeat count of structure.

Parameters:

repeatMax	The new max repeat count of the structure

```
int getElementCount();
```

Get element count.

```
Element getElementByIndex(int index);
```

Get element by index.

```
Element getElementByName(String name);
```

Get element by name.

```
setDefaultEncoding(String defaultEncoding);
```

Set default encoding.

Parameters:

defaultEncoding	The default encoding

```
String getDefaultEncoding();
```

Get default encoding.

```
setElementOrder(ORDER_TYPE order);
```

Set element order.

Parameters:

order	The element order in the structure

```
int appendElement(Element * element);
```

Append element.

Parameters:

element	The element to be appended

```
void insertElementAtIndex(Element * element,
                          int index);
```

Insert element at certain index.

Parameters:

element	The element to be inserted
index	The index where to insert

```
void deleteElementAtIndex(int index);
```

Delete element at index.

Parameters:

index	The index where to delete

Class StructureElement

A structure element object represents one structure item inside another structure.

Methods of `StructureElement`:

```
Structure getStructure();
```

Get structure.

Class StructureMapper

A structure mapper object maps the structures of a grammar to a file (

Methods of `StructureMapper`:

```
long mapStructure(Structure structure);
```

Map a structure at the current position to a file.

Parameters:

structure	The structure to apply

```
long mapStructureAtPosition(Structure structure,
                            long position,
                            long size);
```

Map a structure at the given position to a file.

Parameters:

structure	The structure to apply
position	Where to apply the structure
size	Maximum space the structure can consume

```
long mapElementWithSize(Element element,
                        int maxSize);
```

Map an element at the current position to a file.

Parameters:

element	The element to be applied
maxSize	The maximum size the element may have

```
setDynamicEndianness(ENDIAN_TYPE endianness);
```

Set dynamic endianness.

Parameters:

endianness	The endianness to use from now on

```
ENDIAN_TYPE getDynamicEndianness();
```

Get endianness set currently.

```
ByteArray getCurrentByteArray();
```

Current byte array being processed.

```
ByteView getCurrentByteView();
```

Current byte view being processed.

```
Structure getCurrentGrammar();
```

Current grammar being processed.

```
LogSrc getCurrentLogSrc();
```

Current log source used for output.

```
Results getCurrentResults();
```

Current results used while mapping structures.

```
Structure getCurrentStructure();
```

Current structure being mapped.

```
Element getCurrentElement();
```

Current structure element being mapped. This is of course the scripting element...

```
long getCurrentOffset();
```

Current file offset of the mapping operation.

```
setCurrentOffset(unsigned long offset);
```

Set current file offset of the mapping operation.

Parameters:

offset	New offset to continue processing after script

```
long getCurrentRemainingSize();
```

Current remaining size of the mapping operation.

Class Value

A

Methods of `Value`:

```
String getName();
```

Get name.

```
setName(String name);
```

Set name.

Parameters:

name	The alignment value

```
VALUE_TYPE getType();
```

Get value type.

Glossary

Here you find some terms explained in the context of this manual.

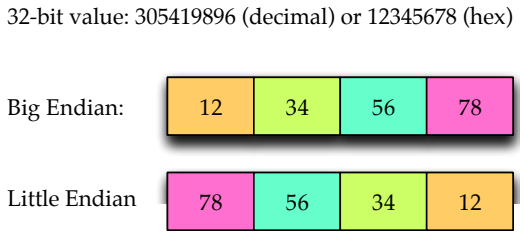
E

Endianness

When you develop in high-level languages like Java or C you often don't notice that the variables you work with are stored in a different byte order in memory, depending on the machine you work on. Only if you display a memory dump of structures or variables you see that the bytes may appear in a different order than what you expected. This reverse byte ordering is called *little endian*. *Big endian* means that the bytes of a variable in memory are ordered as if you write the value on paper. There are CPUs that can work both in little and big endian mode but usually you'll find little endian on PC architectures while big endian is found on platforms like AIX or Solaris (SPARC).

As mentioned the endianness is normally hidden from the casual programmer however if you dump structures or variables directly to a file or transmit them via a TCP connection, it does play a role. Many file format specifications explicitly define the endianness of the data fields. There are file formats that allow as well big as little endian interpretation for the number elements. Synalyze It! supports such formats with a feature called *dynamic endianness* — a script can define for a certain file if the elements marked with dynamic endianness should be interpreted as little or big endian numbers.

Figure 30. Litte/big endian example



G

Grammar

Grammar in the context of Synalyze It! means a definition of the structure of a certain file format. Just as spoken languages also binary files must follow a set of rules to be able to be understood - be it by humans or by computers. The definition of a grammar for binary files allows to parse them by the generic parser in Synalyze It!. Those grammars are stored on disk in XML format.

I

Inheritance

The term inheritance is used in Synalyze It! as in object-oriented programming languages. In record-oriented binary file formats you often find similar records that start with the same elements like record length or an identifier that identifies the record. Defining a parent structure once that holds the elements which are shared by all child structures saves time, avoids mistakes and makes the grammar easier to understand.

P

Pascal strings

There are different concepts in the various programming languages how text strings are stored in memory. In C-based programming languages the length of a string is only determined by a byte with value zero after the last character while in Pascal the first byte contains the length of the following characters. Accordingly you find in binary files both types of text string representations plus such of fixed-length.

T

Text encodings

A good part of the information computers process is text. Since computers only know how to handle and store numbers, characters have to be represented by numbers. In the early days of computers storage was expensive so characters were assigned to as least bits as possible. ASCII is still the code page most people know however the 7 bits are only enough to represent 128 characters, including control characters like line feed or carriage return. To represent text in non-English languages, more code points were needed so many 8-bit code pages exist that base on ASCII or the EBCDIC code invented by IBM. Nowadays memory is much cheaper and the hassle of translating different code pages can be easily avoided by encodings that