elftosb documentation

elftosb 2.2 Operation and Command File Format

Table of Contents

- Scope
- Requirements and assumptions
- Overview
- Command file
  - Basics
  - Blocks
  - Lexical elements
    - Whitespace
    - Keywords
    - Comments
    - Identifiers
    - Integers
    - Strings
    - Section names
    - Symbol references
    - Binary objects
  - Integer Expressions
    - Operator precedence
    - Word size operator
    - Sizeof operator
    - Constant references
    - Symbol references
  - Boolean Expressions
  - Block syntax
    - Options
    - Constants
    - Sources
    - Sections
  - Statements
    - Load
    - Call
    - From
    - Mode
    - Print
    - If-Else
- Options
- Command line interface
- elftosb key file format
- Common usage example
- Appendix: Command file grammar
- Revision History

replace section # with anchored links or remove

add version 2.6.x features and update grammar appendix to match

Scope

This document is the original document describing elftosb 2.x, the new command file format used by it, and other aspects of elftosb's operation. Also covered is some of the reasoning behind the switch to this format, a description of how to use the command files, details about the grammar, and some notes on implementation.

Requirements and assumptions

The basic set of features that must be supported by the command file format come from those features that were supported through command line options in elftosb 1.x. The most important of these features are:

- Different ways to load ELF and S-record files
    - Only load the file, do not execute
    - Load and jump to entry point
    - Load and call entry point
- Exclude sections from a source ELF file
- Pass an arbitrary word-sized argument to functions when executed
- Specify product and component version numbers

In addition, a important new requirement is that the user must be able to have elftosb load any given ELF section to an address other than the one to which it was linked. This is used to support certain virtual memory implementations.
And finally, the command file format must be able to support the following features of the boot image format:

- Create multiple boot image sections from one or more source files
- Support any number of encryption keys

Overview

With version 2.0 elftosb becomes much more like a linker than it was before. Whereas previous versions used command line options to tell the tool what output to produce, this version uses a file written in a special syntax to do the same. This file is called the "elftosb command file", or just command file for short. All invocations of elftosb must provide a command file in addition to the input files. A more common term for elftosb command files is "boot descriptor files". These files typically will have a .bd extension.

The elftosb command file works very much like a linker command file. It describes the output file (the .sb file) in terms of the input file or files. As before, elftosb supports both ELF and S-record input files. The command file can either explicitly declare the input files' paths, or it can let the user provide the paths on the command line. This feature enables command files to be written that are fairly generic in purpose and reusable.

The general idea is that the command file declares a number of source files and assigns unique names to them so they are easy to refer to. As mentioned above, each source can either explicitly call out the path to its file or let the user provide the path on the command line. When the path comes from the command line it can refer to any file and can change each time elftosb is called.
The command file then defines the sections that are required in the output .sb file. Within each of these section definitions a sequence of operations, such as load and call, are listed that refer to the contents of the source files or to constant values present in the command file. These operations are mapped to bootloader commands. This way the contents and structure of the .sb file are described.

Command file

Basics

Command files are simply text files in any encoding that uses ASCII for the lower 128 characters. In particular, this includes UTF-8. Line endings do not matter; Unix, DOS, and Mac OS endings are all supported. Even mixed line endings are accepted.
The standard extension for command files, or boot descriptor files as they are more commonly called, is .bd.

Blocks

The command file is broken into several different blocks: options, constants, sources, and sections. All blocks are optional, and there can be as many blocks of each type as the command file author likes. The only rule is that all section blocks must come after all other block types. Each block in a command file is introduced with a block type keyword and has contents enclosed in braces, as demonstrated in Example 1. The different block types are described in detail in Section 6.6.

Example 1. Basic block syntax

```
# define the options block
options {
    # content goes here
}
```

Lexical elements

This section describes the various textual components that go into a command file, their syntax, and how they are used. While reading the sections below, refer to Table 1 for examples of how tokens are written in the file.

Table 1. Example token values

Token	Description
10000	Integer literal
0x200	Integer with value of 512
256K	Integer with value of 262144
0b001001	Integer with value of 9
'q'	Byte-sized integer with value of 0x71 or 113
'dude'	Word-sized integer with value of 0x64756465 or 1685415013
"this is a test"	String literal
$.text	Section name matching ".text"
$*	Section name matching all sections
$*.bss	Another section name matching all .bss sections, such as ".sdram.bss"
appElfFile:main	Symbol reference with explicit source file
:printMessage	Symbol reference using default source file
`{{ 01 02 03 0b }}`	A four byte long binary object.

Whitespace

Whitespace in the form of space characters, tabs, newlines, or carriage returns is ignored throughout the command file, except within a string. Any form of line ending is allowed.

Keywords

Table 2 lists every keyword that is used in elftosb command files. These identifiers are not available for use as source file or constant names. Not all of the keywords are actively used in command files yet, but they are set aside for features that are intended for the future.

Table 2. Command file keywords

call	no
constants	options
extern	raw
false	section
filters	sources
from	switch
jump	true
load	yes
mode	if
else	defined
info	warning
error	sizeof

Comments

Single-line comments are introduced at any point on a line with either the pound character ("#") or two slashes ("//") and run until the end of the line.
Multi-line comments work exactly the same as they do in ANSI C. They begin with

"/*"

and end with

"*/"

. And like ANSI C, there is no support for nested multi-line comments.

Identifiers

Identifiers are used for option names, constants, and source names. They follow the familiar ANSI C rules for identifiers. They can begin with an underscore or any alphabetic character and may contain any number of underscores and alphanumeric characters.

Integers

Integers literals are of one of three supported bases: binary, decimal, or hexadecimal. Decimal integers have no prefix. Hexadecimal integers must be introduced with '0x', and binary integers must be introduced with '0b'.
Integer literals can optionally be followed by a metric multiplier character: "K", "M", or "G". Space characters are allowed between the last digit and the multiplier. Note that binary multiplier values are used, not the standard metric multipliers. This means that "K" multiplies the integer by 1024, "M" by 1048576, and "G" by 1073741824. Lower case "k", "m", or "g" are not allowed.

All integer values within a command file are unsigned and have an associated size. Supported integer sizes are byte (8-bits), half-word (16-bits), and word (32-bits). Integer literals are by default all word-sized values. To change the word size, the "word size" operator is used in an expression.

Integer constants can also be created with character sequences contained in single quotes. One, two, or four character sequences are allowed. These correspond to byte, half-word, and word sized integers. For example: 'oh' is equal to a half-word with the value 0x6f68 hex (the value of the characters "o" and "h" in ASCII) or 28520 decimal.

Several keywords are set aside for built-in integer constants for Boolean values. These are "yes", "no", "true", and "false". The "yes" and "true" keywords evaluate to 1, while "no" and "false" evaluate to 0. These keywords can be used anywhere that accepts an integer value, including the command line.

Strings

All string literals are contained within double quote characters. They may not extend beyond the end of a line. C-style escape sequences are not support so that the backslash character can be used as-is in file paths. Unfortunately this makes it impossible to insert a double quote, newline, or other special character in the middle of a string.

Section names

Named sections of ELF files are selected with a section name literal. These special literals begin with a dollar-sign character ('$') and continue until the first character that is not allowed in a section name. The name is actually a standard glob-type expression that can match any number of ELF sections. Accepted characters include alphanumerics, underscore, the period, asterisk, question mark, dashes, caret, and square brackets. Many of these characters are used only as part of the glob expression.
The supported glob sub-expressions are:

*	Matches any character, zero or more times in a row.
?	Matches any single character.
[set]	Matches any character in the set.
[^set]	Matches any character not in the set.

In the above list, a set is any combination of single characters and ranges. Ranges are formed as two characters separated by a hyphen: "a-z" inclusively matches all characters from "a" to "z".
When used in the section list of a load statement, you can prefix a section name with a tilde ("~") character to invert the set of matched ELF sections.

Symbol references

Source files in the ELF format have a symbol table embedded in them. A symbol reference is used to refer to a particular symbol in an ELF file by its name. When used in an integer expression the symbol reference has the symbol's value, which is usually its address.

The syntax for a symbol reference is quite simple, consisting of an optional source file name followed by a colon and then the symbol's name. The symbol name is not placed in quotes or any sort of delimiters and has the same character set as a regular identifier (see section 6.3.4).

If no source file is placed before the colon then the symbol will come from the default source file that is specified with a from statement (see section 0 for details). If the symbol reference is not within the context of a from statement, then the source file name is required.

Binary objects

Binary object values, known as "blobs", are simply a sequence of hexadecimal bytes that form an object. Double curly braces open and close a blob. Every two hexadecimal characters form one byte in the blob; all whitespace is ignored. Case does not matter for the hex characters. Non-hex characters are illegal, and comments are not allowed within a blob.

Integer Expressions

An integer expression can be used in any place an integer constant value is required in the grammar. These expressions for the most part follow the style of standard C expressions, with a few extensions. Table 3 lists the available operators.

Table 3. Integer expression operators

Operator	Description
+	add
-	subtract
*	multiply
/	divide
%	modulus
&	bitwise and
|	bitwise or
^	bitwise xor
<<	logical left shift
>>	logical right shift
.	set integer size
sizeof()	get size of a constant or symbol

In addition to those operators listed in Table 3, unary plus and minus are also supported. For operator precedence see Table 4.

Operator precedence

Table 4 lists the expression operators grouped in their order of precedence. The first row in the table is the lowest and the last row is the highest precedence.

Table 4. Operator precedence in increasing order

Operators	Description
|	Bitwise Or
^	Bitwise Xor
&	Bitwise And
<< >>	Left shift, right shift
+ -	Add, subtract
* / %	Multiply, divide, modulus
.	Word size
unary + -	Unary positive and negative

Word size operator

The one operator that needs extra discussion is the integer size operator ("."), which is fairly unique. It consists of a period followed by one of the characters "w", "h", or "b". These characters are case-sensitive; "W", "H", and "B" are not accepted. Whitespace is allowed between the period and the following character. This operator changes the word size for the expression to its left. The "w" character sets the size to a 32-bit word, the default, "h" to a 16-bit word, and "b" to an 8-bit word.
For any given binary operation, the result will assume the largest word size of the two operands. So a byte-sized integer multiplied by a half-word-sized integer will result in a half-word. It does not matter which side of the operation the two differently sized operands are located. The actual operation is always performed as 32-bit words and the result truncated if necessary.

Sizeof operator

To take the size of either a symbol or constant there exists the sizeof operator. This operator's syntax is simply the keyword "sizeof" followed by either a symbol reference or constant identifier in parentheses. The parentheses are required, unlike the sizeof operator in ANSI C. Sizes are always 32-bit values.

Constant references

Along with integer literals, expressions may refer to constants defined in the constants blocks by their name. A constant name is simply a standard identifier. Placing a constant name in an expression is equivalent to inserting that constant's integer value. Although sources share the same namespace as constants, they cannot be used within an integer expression.

Symbol references

Symbol references may also be used in integer expressions, just like constants. A symbol reference has the value of the symbol's value in the ELF file and is a 32-bit value. Usually a symbol's value is its address, although some special symbols can have other values. If the referenced symbol does not exist in the source file then the symbol reference has a value of 0.

Boolean Expressions

Unlike integer expressions, Boolean expressions are limited in use. They are only allowed to be used when defining a constant or option, or as the conditional for the if and else-if statements described in section 6.7.6. Specifically, you can't use Boolean expressions as the source or target of a load statement.

Table 5. Boolean expression operators

Operator	Description
&&	Boolean and
||	Boolean or
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to
exists(src_file)	does a source file exist?
defined(const)	is a constant defined?

As shown in Table 5, there are number of new operators that can be used in Boolean expressions. In addition to those operators in Table 5 the unary not operator, the character "!", is supported. All of these operators evaluate to either 0 or 1. Like ANSI C, a value of 0 means false and any non-zero value means true.
There are two function-like operators that can be used in a Boolean expression. The first, "exists()", will return true if the source file named inside the parentheses exists on disk and was opened successfully. It is a syntax error to put a source name that hasn't been defined in a sources block inside an exists operator.
The second special operator is "defined()". It takes the name of a constant between the parentheses. The operator has a value of true if the named constant has been assigned a value, either within the boot descriptor file or from the command line.
The && and || binary operators are short-circuit operators. This means that if the left-hand operand is equal to a value that makes the value of the right-hand operand unimportant (because the expression would have the same end value either way), then the right hand operand is not evaluated. This is particularly useful in an expression such as "if defined(const) && const > 10…". Here the right-hand greater than expression is only evaluated if the constant "const" is defined. If the right-hand expression were always evaluated and "const" happened to not be defined, then an error would be reported.

Block syntax

Blocks are arranged in two groups within a command file. First come the configuration blocks: options, constants, and sources. There can be any number of these and they can be placed in any order. All configuration block types are optional. But usually at least one sources block is necessary for a useful command file.
After the configuration blocks come the section definition blocks. There can be any number of section blocks. Their lexical order in the command file determines the logical order of sections in the output boot image.

Options

An options block contains zero or more name/value pairs, the option settings, that assign values to global options used by elftosb to control the generation of the output file.
Each entry in the options block takes the following form:

```
option_def  ::= IDENT ‘=’ const_expr
   ;
const_expr  ::= bool_expr
   | STRING_LITERAL
   ;
```

Within the block, each option definition must be terminated by a semicolon. The value of an option can be either a string or any integer or Boolean expression. Acceptable values depend on the particular option.
The option names are predefined by elftosb itself and are not used anywhere else in the command file. Thus, it is possible to have a source with the same name as one of the options, although that might be confusing. The complete list of available options is in Section 7.

Constants

Similar to the options block, the constants block contains a sequence of zero or more constant definition statements, each followed by a semicolon. Each constant definition statement is simply a name/value assignment. The right hand side is the constant's name, a standard identifier, while the left hand side is an integer or Boolean expression.
The constant definition grammar looks like this:

```
constant_def  ::= IDENT ‘=’ bool_expr
   ;
```

Constant values retain the integer word size that they evaluated to when they are used in another expression.
A constant defined earlier in the constants block can be used in the definition of constants that follow it, as shown in Example 2.

Example 2. A constants block

```
# this is an example constants block
constants {
     ocram_start = 0;
     ocram_size = 256K;
     ocram_end = ocram_start + ocram_size -- 1;
}
```

Sources

The sources block is where the input files are listed and assigned the identifiers with which they are referenced throughout the rest of the command file. Each statement in the sources block consists of an assignment operator (the "=" character) with the source name identifier on the left hand side and the source's path value on the right hand side. Individual source definitions are terminated with a semicolon.
The syntax for the source value depends on the type of source definition, of which there are two types: explicit paths and externally provided paths. Sources with explicit paths simply list the path to the file as a quoted string literal.
The external sources use an integer expression to select one of the positional parameters from the command line. This type of source allows the user to easily vary the input file by changing the command line arguments.
The sorce definition grammar follows this form:

```
source_def ::= IDENT ‘=’ source_value ( ‘(‘ source_attr_list? ‘)’ )?
  ;

source_value ::= STRING_LITERAL
  | ‘extern’ ‘(‘ int_const_expr ‘)’
  ;

source_attr_list
  ::= source_attr ( ‘,’ source_attr )*
  ;

source_attr ::= IDENT ‘=’ const_expr
  ;
```

There source definition can optionally have a list of source attributes contained in parentheses at the end of the definition. These attributes are the same as options in an options block but only a few options apply to sources. See section 7 for the complete list of options.
Each source file has a dictionary of options associated with it, created from the contents of its option list. This dictionary has the global options as its parent. Therefore, if a source file uses a particular option and it is not contained in its local dictionary, the global options are searched. Conversely, the local dictionary can override a global option value just for that source file.
Note that section-specific options are not applied to source files used within that those sections.

Example 3. Source definitions

```
# set global options
options
{
 toolset = “GHS”;
}

sources
{
 # a source with an explicit path
 player = “output/player/player.elf”;

 # an external source that overrides the global toolset option
 another = extern(0) ( toolset=”GCC” );
}
```

ELF files generated by different toolsets are not entirely compatible with each other. elftosb needs to know which toolset produced each ELF file so it can read it properly. This is where the *toolset* option comes in. It's value is the name of one of three ARM toolsets. Acceptable values are "GHS", "GCC" or "GNU", and "ADS". Case does not matter. Specifying the toolset in the global options block applies that toolset as the default for all ELF files. This can be overridden on a per-file basis by setting the *toolset* option only for that source file when it is defined.

The default toolset for ELF files is GHS. For GHS ELF files, the *secinfoClear* option determines how .bss and similar sections are converted to bootloader commands. In most cases you won't have to set this option, but it important to understand how it works. The name of the *secinfoClear* option comes from the .secinfo section generated by the Green Hills linker that contains a list of regions of memory that must be cleared before the program can be executed.

When *secinfoClear* is set to "ignore", the .secinfo section is ignored and all sections are cleared as described in the ELF file. If set to "default" or "c" or left unset, a section will only be cleared if it is *not* listed in the .secinfo table. In this case it is assumed that the C runtime startup code will itself clear these sections and there is no need for the bootloader to do so. Finally, if *secinfoClear* is set to "rom" then sections will be cleared only if they *are* listed in .secinfo. This case is useful if you do not have a C runtime startup for some reason. In any case, if there is no .secinfo section present in the ELF file the behavior defaults to act like the "ignore" option. This is also how non-GHS ELF files work.

Sections

There may be any number of section blocks, but they must all come after the other block types within the command file. Each section block corresponds directly to a section created in the output .sb file. Section blocks also have a slightly different opening syntax than other blocks, in that you specify the section's unique identifier value and any options specific to that section.
The grammar for a section block is shown below. The *statement* non-terminal is described in detail in Section 6.7.

```
section_block ::= ‘section’ ‘(‘ int_const_expr section_options? ‘)’ section_contents
  ;

section_options ::= ‘;’ section_option_list
  ;

section_option_list
  ::= source_option ( ‘,’ source_option )*
  ;

source_option ::= IDENT ‘=’ const_expr
  ;

section_contents
  ::= ‘{‘ statement* ‘}’
  | ‘<=’ SOURCE_NAME ‘;’
  ;
```

As is demonstrated by Example 4, there are two forms of section contents. The first, the one with braces containing a sequence of statements, creates a bootable section with a number of bootloader commands as its content. Most sections will be of this form. The syntax for statements in a bootable section are covered in detail in Section 6.7.
The second form creates an arbitrary data section. The raw binary contents of the listed source file are copied wholesale into that section of the

output file. There is no predetermined format for data sections. As examples, data sections can be used to hold resource files or a backing store for virtual memory paging.

Example 4. Two section blocks

```
# create a bootable section
section (32) {
 # statements...
}

# create a data section
section (64) <= my_source_file;
```

The section identifier number that appears in the parentheses must be unique for that section. If two sections have the same identifier, an error will be reported.
You can set options that apply only to a single section by inserting them after the section's unique identifier, separated by a semicolon. In the grammar above, options are described by the *section_options* non-terminal. If there is more than one option, they are separated by commas instead of semicolons as in an options block.
These are the important options that apply to sections in output files that you should be aware of. They are also listed in section 7.

alignment
This option takes an integer power of two as its value. The offset within the output .sb file to the first byte of a section with a special alignment is guaranteed to be divisible by the alignment value. Alignments equal to or below 16 is ignored, as that is the minimum alignment guaranteed by the cipher block size of an .sb file. Note that the section itself is aligned, not the boot tag for that section. Any padding inserted to align a section consists of "nop" bootloader commands.

cleartext
Set this option to a Boolean value. The keywords "yes", "no", "true", and "false" are accepted, as is any integer expression that evaluates to zero or non-zero. The default is false. When set to true, and if the output file is encrypted, the section to which the *cleartext* option applies is left unencrypted. Beware that the ROM does not currently support unencrypted bootable sections in an encrypted file. So this option is most useful for data sections.
As with all options, these can be set globally using an options block instead of individually per section. You can also set a global default and override it with a section-specific option. For instance, you could set the default section alignment to 2K and then align one particular section to 4K.
Sections are always created in the output .sb file in the order in which they appear in the command file. In addition, the first bootable section that is defined in the command file becomes the section that the bootloader starts processing first, after it examines the .sb file headers.

Statements

Each statement within a bootable section block describes an operation that will be performed by the bootloader when it processes the output .sb file. Individual statements correspond to at least one, and possible more than one, boot command created in the output file. The intent is for statements to describe what the user wants to happen rather than exactly which boot commands are to be generated. It is the responsibility of elftosb to ensure that valid boot commands are generated.
All statements except the from and if-else statements must end with a semicolon.
For all of the inline examples below, assume the following definitions:

```
options {
 toolset = “GHS”;
}

sources {
 myElfFile = “app.elf”;
 mySRecFile = “utility.s37”;
 myBinFile = “data.bin”;
}
```

Load

The load statement is used for any operation where the user wants to put some form of data into memory. In terms of bootloader commands this includes data loads, pattern fills, and word pokes. The goal is for the load statement to be extremely flexible. These statements can be very simple in syntax but very complex underneath. In other words, a short, sweet load statement can produce a large sequence of boot commands. On the other hand, a long and complex load statement may produce a single boot command. The idea is to abstractly describe the desired

operation and let elftosb determine how to best convert it into bootloader commands.
The grammar for a load statement is:

```
load_stmt ::= ‘load’ load_data ( ‘>’ load_target )?
  ;

load_data ::= const_expr
  | SOURCE_NAME
  | section_list ( ‘from’ SOURCE_NAME )?
  ;

section_list ::= section_ref ( ‘,’ section_ref )*
  ;

section_ref ::= ( ‘~’ )? SECTION_NAME
  ;

load_target ::= ‘.’
  | address_or_range
  ;

address_or_range
  ::= int_const_expr
  | int_const_expr ‘..’ int_const_expr
  ;
```

As shown in the grammar, all load statements are introduced with the "load" keyword. Each load statement is comprised of a data source and a target location. The source is always required, but the target can be implicit, in which case it is based on the source itself. Not all combinations of source and target types are allowed.
The source is represented by the *load_data* non-terminal in the grammar above. There are four types of sources that are allowed: integer values, string literals, a source file, or one or more named sections of a source file. These diverse sources boil down to one or more segments of data, depending on the type of source. Data sources, and therefore segments, may or may not have a natural location in memory associated with them. This natural location is the range of addresses in memory where the data would be placed by default. They also may have a natural size in bytes. For instance, a section of an ELF file is linked to a certain address and has a length. These combine to form the section's natural address and size. For another example, the content of a binary file has a natural size but not an address.
The target of the load statement determines the address in memory at which the source is loaded and the length of the load. For certain source types that have a natural location, the target is optional and can be excluded from the statement. If explicitly listed, the target follows a '>' symbol after the source data. An alternative, equivalent form for an implicit target is to put a dot (period) after the '>'. Values for the target are either an address or address range. When a target is a single address it does not have a length associated with it. In this case the length of the load comes from the source data itself. References to symbols from an ELF file can also be used as a load target. They are equivalent to an address range, from the symbol's start address to its end address.
When the target is a single address, the entire data source is loaded to that address. This is true even if the source has a natural address. This allows the user to, for instance, load ELF sections to different addresses from which they were linked.
When the target is an address range, or a symbol since they is equivalent to an address range, the source is both located and potentially truncated. The load address is the start of the target range. This works the same as with a single target address. If the natural size of the data source is equal to or smaller than the size of the target range (the end address minus the start address) then the entire source is loaded. When the source's size is smaller than the target range, the leftover bytes are not modified in any way. In the case where the natural size of the source is larger than the target range, the source is truncated to the size of the range when loaded.
Data sources that are composed of multiple segments, such as ELF files with multiple sections, must be loaded to their natural location. This is because only one target address or range can be specified, and it would be useless to load each segment to the same address.
The most common form of load statement is to simply load a source file by name. This can produce quite different data sources, depending on the source file's type. The specific features of each data source type are described below.

ELF file — Using an entire ELF file as a data source causes all sections within the file to be loaded. Actually, not all sections are loaded; only those sections whose type is SHT_PROGBITS or SHT_NOBITS are considered. All sections from ELF files have natural locations and sizes.
ELF files produced by the Green Hills MULTI toolset are treated specially. In these files the .secinfo section is used to determine which sections should actually be filled, as controlled by options in the link descriptor file. See the *toolset* and *secinfoClear* options described in section 7 for more information.

```
# these two loads are completely equivalent
load myElfFile;
load myElfFile > .;
```

S-record file — The contents of the file are turned into an in-memory image where contiguous regions of data are found by coalescing the individual load commands. Load segments are created from each of the contiguous regions. These segments do have natural addresses.

```
load mySRecFile;
```

Binary file — The entire contents of the file form one load segment that does not have a natural address. However, a binary file does have a natural length.

```
// load an entire binary file to an address
load myBinFile > 0x70000000;

// load part of a binary file
load myBinFile > 0x70000000..0x70001000;
```

Binary object — Almost like a binary file except the data is listed inline in the boot descriptor file. Again, raw binary data has no natural address but does have a natural length.

```
// load an eight byte blob
load {{ ff 2e 90 07 77 5f 1d 20 }} > 0xa0000000;
```

ELF section list – If you want to load only certain sections of an ELF file, a syntax is supported that lets you select ELF sections using glob expressions. See section 6.3.7 for more information about section names. The data source syntax is a list of one or more section names followed by the "from" keyword and a source name for an ELF file. The "from" keyword and following source name are allowed to be omitted if the load statement is within a from statement. These examples demonstrate the syntax:

```
// inclusive section name
load $.text from myElfFile;

// exclusive section name
load ~$.mytext from myElfFile;

// example load inside a from statement
from myElfFile {
 load $.text.*, ~$.text.sdram;
}
```

Because all sections of an ELF file have a natural location and size, and the code in those sections expects to be at that location, you will probably never use an explicit load target. In fact, elftosb only allows explicit targets for statements that select a single ELF section. This is because it doesn't make sense to load multiple sections to the same target address, while it could be useful to relocate a single section to a new address in memory.
The actual comma separated list of ELF section name expressions that follows the "load" keyword progressively filters the selected ELF sections. Each section name in the list can optionally be preceded by a tilde character (i.e., "~"), in which case the set of matched sections is inverted. For instance, the section name "~$.sdram.*" will match every section that does *not* begin with ".sdram.".
As an example of how multiple section names in the list works, consider the third example load statement above. The first section name "$.text.*" matches every ELF section that begins with ".text.". Then the second name in the list matches every ELF section *but* the one named ".text.sdram" *out of those sections matched by the previous section name*. If the source file contains ".text.ocram", ".text.sdram", ".bss", and ".data" then only ".text.ocram" will be selected.
Integer value — Integers values are a unique type of load data, in that the value is used as a pattern to fill a region of memory. Integer sources do not have a natural address but they do have a natural length.

```
# pattern fill
load 0x55.b > 0x2000..0x3000;

# load two bytes at an address
load 0x1122.h > 0xf00;
```

So if you load an integer value to a single address without a range, the load will fill as many bytes as the integer value is long. The second load

statement in the example above loads two bytes to 0xf00 because the integer value is a half-word.
If you instead load an integer to an address range, only those bytes that are included in the range are filled. This is true even if the integer value's size is larger than the address range's length.
String literal — Using string literals as the load data source is very similar to loading a binary file. One interesting use for this ability is to fill a buffer in memory that contains a message to be displayed to the user or printed over a serial port. Once the buffer is set you can invoke the print routine with a call statement.

```
# load a string at the address of a symbol
load “hello world!” > myElfFile:szMessage;
```

Call

The call statement is used for inserting a bootloader command that executes a function from one of the files that has been loaded into memory. The type of function call is determined by the introductory keyword of the statement.
The grammar for these statements looks like this:

```
call_stmt ::= call_type call_target call_arg?
  ;

call_type ::= ‘call’
  | ‘jump’
  ;

call_target ::= SOURCE_NAME
  | symbol_ref
  | int_const_expr
  ;

call_arg ::= ‘(‘ int_const_expr? ‘)’
  ;
```

As with the load statement, the call statement begins with a special keyword. But instead of a single keyword there are two possibilities. The keyword selects which specific boot command is produced by the statement, dependant upon the output boot image format. In general, "call" commands are expected to return the bootloader and "jump" commands are not. For boot images, "call" produces a ROM_CALL_CMD and "jump" produces a ROM_JUMP_CMD. See the boot image format design document for specific details about these commands, such as the function prototypes they expect.
After the introductory keyword comes the call target, of which there are three forms that each have their own syntax. All forms of the target boil down to just an address in memory. The different forms are described in detail below.

Source file — If a source file name is used as the call target, the call statement will use the entry point to that source file as the target address. This implies that the source file must have an entry point; if a source file is used that either does not support entry points or does not have one set then an error is reported.

```
# call the entry point
call myElfFile;

# same here
jump mySRecFile;

# this will produce an error because binary files
# do not have an entry point
call myBinFile;
```

Integer expression — Using an integer expression is the most straightforward call target. The expression simply evaluates to the address of the function that is invoked by the call or jump boot command.

```
# jump to a fixed address
jump 0xffff0000;
```

Symbol — Although it is just another form of integer expression, it is important to point out that a reference to a symbol in an ELF file can be used

as the call target. Both the form where the source file is explicit and the form where it is implicit are supported. The implicit form uses the source file from the enclosing from statement (see section 0). It is an error to use the implicit form outside of a from statement. It is also an error to list a symbol that is not present in the source file, or to use a source file with a type other than ELF.

```
# call a function by name and pass it an arg
call myElfFile:initSDRAM (32);

# this is the implicit form of symbol usage
from myElfFile {
 call :reboot();
}

# this is an error because Srecords do not have symbols
jump mySRecFile:anEntryPoint();
```

Note that the file the symbol comes from does not actually have to be loaded by the same command file. It is only used to find an address, whether or not the function actually exists at that location.
The final part of a call statement is the optional argument value. It is just an integer expression wrapped in parentheses. The expression determines what value is passed as the first argument to the call or jump boot command. If the expression is excluded from the statement then the argument value defaults to zero. Using empty parentheses is equivalent to completely excluding the parentheses.

From

More of a block than a true statement, the from statement is the simplest as far as syntax. It also produces no boot commands by itself. Instead, a from statement allows the user to use simpler forms of the statements contained within it.
The simple grammar for from statements follows this form:

```
from_stmt  ::= ‘from’ SOURCE_NAME ‘{‘ statement* ‘}’
   ;
```

So basically a from statement consists of the "from" keyword, a source identifier, and a sequence of statements enclosed in braces. There is no terminating semicolon after the closing brace. Any type of statement is allowed between the braces, except for additional from statements---they cannot be nested.
Certain forms of the load and call statements use an implicit source file. All a from statement does is set this implicit source file for the statements found within it. This makes for cleaner and easier to read command files.

Example 5. The from statement

```
# name our input file
sources {
 example = extern(0);
}

# create a section
section (0) {
 from example {
  # load from example and call a function inside it
  load $.ocram.*;
  call :_start;
 }
}
```

Example 5 demonstrates how the from statement is used. The load and call statements inside the from do not have any source explicitly listed. Which file should the named sections be loaded from? Which file is the symbol "_start" located in? The from statement supplies the implicit source file for these statements.
So the load statement loads all sections in the example source that have a name beginning with ".ocram.". And the call statement generates a call boot command to the address of the "_start" symbol within the example source file.

Mode

The mode statement is a special purpose statement that is intended to be used only for unique circumstances. One of these statements will

produce a bootloader command that restarts the bootloader and potentially changes the boot mode. The statement accepts a new boot mode value, one of the same values that can be presented to the boot mode pins or the boot mode bits in OTP.
The grammar is extremely simple:

```
mode_stmt  ::= ‘mode’ int_const_expr
   ;
```

A mode statement simply consists of the "mode" keyword followed by any integer expression. The expression must evaluate to a boot mode value as would be seen on the boot mode pins of the device.

Example 6. The mode statement

```
# some constants
constants {
 USB_MODE = 0;
}

# create a section
section (0) {
 mode USB_MODE;
}
```

Print

The print statement is actually three very similar statements that are used to print different categories of messages to the user. The three types of print statement are *info*, *warning*, and *error*. All print statements begin with a keyword corresponding to their type, as seen in the grammar here:

```
print_stmt ::= ‘info’ STRING
  | ‘warning’ STRING
  | ‘error’ STRING
  ;
```

The info statement simply prints the message to standard out. The message will be visible unless the caller has enabled the quiet output feature.
The warning statement does basically the same thing as the info statement except it prefixes the message with "warning:". Also, the message is always visible.
Finally, the error statementstops the execution of elftosb immediately and prints the message prefixed by "error:".

Example 7. The print statement

```
sources
{
 # give the ELF file a name
 afile = “file.elf”;
}

constants
{
 # create a constant that is the size of a symbol
 bufsize = sizeof(afile:_my_buf);
}

# create a section
section (0)
{
 if bufsize < 128
 {
  # elftosb will stop after this is printed
  error “Buffer size $(bufsize) is too small!”;
 }
 else
 {
  info “Buffer size $(bufsize) is acceptable”;
 }

 /* ...more... */
}
```

The three print statements support substitution of constant values and source file paths using a syntax like that for Unix shell variable substitution. A constant name or source file name placed in parentheses and prefixed with a dollar sign will cause the appropriate value to be inserted before the message is printed to standard out.
For constant substitution, you have limited control of the formatting of the constant's value. Formatting options are placed before a colon that prefixes the name of the constant inside the parentheses. The two supported formatting options are the characters "d" and "x", only one of which is allowed at a time. "d" formats the constant as decimal and "x" formats it as hexadecimal. For instance "$(x:floop)" formats the constant "floop" as hex.

If-Else

To make it easier to create reusable boot descriptor files, elftosb has the if-else statement. These statements work just like if statements in any other language you've used. You can chain as many else-if statements as you like. And, of course, the final else branch is optional and may be excluded.
The grammar looks like this:

```
if_stmt  ::= ‘if’ bool_expr ‘{‘ statement* ‘}’ else_stmt?
  ;

else_stmt ::= ‘else’ ‘{‘ statement* ‘}’
  | ‘else’ if_stmt
  ;
```

There are several differences in syntax from ANSI C. No parentheses are required around the Boolean expression after the "if" keyword. And curly braces are always required around statements on both the if and possible else branch.
All types of statements are allowed inside an if-else statement, including from statements. The converse is also true: if-else statements may be placed inside from statements.

Options

This section lists the names and allowed values for options that can be set in the options block of a command file. Options can also be applied to individual source files.

Table 6. Option names

Option name	Applies To	Description
alignment	section	Power of 2 integer alignment requirement for start of boot image section.
cleartext	section	Integer Boolean value. Makes a section unencrypted even in encrypted image.
componentVersion	boot image	Same format as the productVersion option.
driveTag	boot image	Integer, sets drive tag field of image header.
flags	boot image	Integer value that is used for the image-wide flags.
productVersion	boot image	Version string in the form "xxx.yyy.zzz".
secinfoClear	GHS ELF source files	One of "default", "ignore", "rom", or "c" where "default" is equal to "c".
sectionFlags	section	Integer value used to set flags for boot image sections. Or-ed with implicit flags.
toolset	ELF source files	One of "GHS", "GCC", or "ADS".

The two version options are used to set the default product and component version numbers. Either version can be overridden from the command line.
The *flags* option sets the flags field in the header of a boot image file. See the document that describes the boot image format for the possible values of this field. The same applies to the *sectionFlags* option, except it sets the flags field in the boot image section header.

Command line interface

elftosb has the set of command line options listed in Table 7. Not all options are listed here; only those that directly interface with the things described in this document are described. Note that a space is required between both the short or long form option and any value. Any arguments listed after the options are the positional source files utilized by the *extern()* syntax (see section 6.6.3).

Table 7. Command line options

Option	Description
-p PATH, --search-path PATH	Adds a path to the end of the list of search paths.
-f CHIP, --chip-family CHIP	Selects output boot image format.
-c FILE, --command FILE	Specify the command file to use. This option is required.
-o FILE, --output FILE	Set the output file path. Also required.
-P VERS, --product VERS	Set product version.
-C VERS, --component VERS	Set component version.
-k FILE, --key FILE	Add a key file and enable encryption.
-z, --zero-key	Add a key of all zeroes and enable encryption.
-D NAME=INT, --define NAME=INT	Override or set a constant value.
-O OPTION=VALUE, --option NAME=VALUE	Set a global option value.
-V, --verbose	Print more detailed output.
-q, --quiet	Print only warnings and errors.
-d, --debug	Enable debug output.
-v, --version	Display tool version.
-?, --help	Show usage information.

Two required command line options are used to set the command file and the output file paths.

The -f or --chip-family switch is used to tell elftosb what format of output .sb file to use. The argument should be one of "37xx", "377x", "378x", "mx23", "imx23", "i.mx23", "mx28", "imx28", or "i.mx28". Case is ignored when comparing chip family names. The only actual difference is that the i.MX28 family will enable HAB related commands.
The output .boot image is by default not encrypted. To encrypt the boot image you need to provide one or more keys. Use the -z switch to add a key that consists of all zeroes. This is the default state of the hardware key in a chip that has not yet had its OTP key burned.
One very useful option is -D or --define, used to set and potentially override a constant value. The argument to the option is an identifier and an integer value separated by an equals sign. The constant name identifier can be any constant name allowed in command files, and the value can be any integer value allowed in command files except multi-character integer literals.
Before producing the output boot image, all constants set with -D or --define options are set in the expression namespace inside elftosb. These special constants override any normal constants with the same name that are specified in the command file. This allows you to put a default value for a constant in the command file and very easily change it with each invocation of elftosb.
Similar to -D is the -O or --option switch that lets you set or override global option settings from the command line. The argument value is again an option name and value separated by an equals sign. The value can be any integer or string value allowed in the command file except multi-character literals.

elftosb key file format

The key files provided to elftosb with the ~~k~~ -key command line switch have a very simple format. Each line of a key file contains one key, which is an uninterrupted string of 32 hexadecimal characters, for a total of 128 bits of key data. Any number of keys may appear in a key file, each on a separate line. The line ending format does not matter.

Example 8. Key file with two keys

```
3F3CFBC001F399991035C3C6C7065924
1BA3CD4030FC4376B4AA8CB5E932432E
```

As can be seen in Example 8, the contents of a key file are in plaintext. A future version of elftosb will use key files that are encrypted, making storage and transport of keys much more secure.

Common usage example

The most common use of elftosb is to simply load a single ELF file and jump to its entry point, which will almost always be the _start symbol defined by the C runtime library. This is very simple to do, but it does require a boot descriptor file, as does all use of elftosb 2.x.

Basic reusable boot descriptor file

```
// Define one input file that will be the first file listed
// on the command line. The file can be either an ELF file
// or an S-record file.
sources {
  inputFile = extern(0);
}

// create a section
section (0) {
  load inputFile; // load all sections
call inputFile; // jump to entry point
}
```

Nice and easy.

Appendix: Command file grammar

The grammar for the command file format is presented below in Extended Backus-Naur Format (EBNF).

```
command_file ::= pre_section_block* section_def*
```

```
  ;

pre_section_block
  :: options_block
  | constants_block
  | sources_block
  ;

options_block ::= 'options' '{' option_def* '}'
  ;

option_def ::= IDENT '=' const_expr ';'
  ;

constants_block
  ::= 'constants' '{' constant_def* '}'
  ;

constant_def ::= IDENT '=' int_const_expr ';'
  ;

sources_block ::= sources '{' source_def* '}'
  ;

source_def ::= IDENT '=' source_value ( '(' source_attr_list? ')' )? ';'
  ;

source_value ::= STRING_LITERAL
  | 'extern' '(' int_const_expr ')'
  ;

source_attr_list
  ::= option_def ( ',' option_def )*
  ;

section_block ::= 'section' '(' int_const_expr section_options? ')'
              section_contents
  ;

section_options
  ::= ';' source_attr_list?

section_contents
  ::= '{' statement* '}'
  | '<=' SOURCE_NAME ';'
  ;

statement ::= basic_stmt ';'
  | from_stmt
  | if_stmt
  ;

basic_stmt ::= load_stmt
  | call_stmt
  | mode_stmt
  | message_stmt
  ;

load_stmt ::= 'load' load_data ( '>' load_target )?
  ;

load_data ::= int_const_expr
  | STRING_LITERAL
  | SOURCE_NAME
  | section_list ( 'from' SOURCE_NAME )?
  ;

section_list ::= section_ref ( ',' section_ref )*
  ;
```

```
section_ref ::= ( '~' )? SECTION_NAME
  ;

load_target ::= '.'
  | address_or_range
  ;

address_or_range
  ::= int_const_expr
  | int_const_expr '..' int_const_expr
  ;

symbol_ref ::= SOURCE_NAME? ':' IDENT
  ;

call_stmt ::= call_type call_target call_arg?
  ;

call_type ::= 'call'
  | 'jump'
  ;

call_target ::= SOURCE_NAME
  | symbol_ref
  | int_const_expr
  ;

call_arg ::= '(' int_const_expr? ')'
  ;

from_stmt ::= 'from' SOURCE_NAME '{' in_from_stmt* '}'
  ;

in_from_stmt ::= basic_stmt ';'
  | if_stmt
  ;

mode_stmt ::= 'mode' int_const_expr
  ;

message_stmt ::= message_type STRING_LITERAL
  ;

message_type ::= 'info'
  | 'warning'
  | 'error'
  ;

if_stmt  ::= 'if' bool_expr '{' statement* '}' else_stmt?
  ;

else_stmt ::= 'else' '{' statement* '}'
  | 'else' if_stmt
  ;

const_expr ::= bool_expr
  | STRING_LITERAL
  ;

int_const_expr ::= expr
  ;

bool_expr ::= int_const_expr
  | bool_expr '<' bool_expr
  | bool_expr '<=' bool_expr
  | bool_expr '>' bool_expr
  | bool_expr '>=' bool_expr
  | bool_expr '==' bool_expr
  | bool_expr '!=' bool_expr
  | bool_expr '&&' bool_expr
```

```
  | bool_expr '||' bool_expr
  | '!' bool_expr
  | IDENT '(' SOURCE_NAME ')'
  | '(' bool_expr ')'
  | 'defined' '(' IDENT ')'
  ;

expr  ::= INT_LITERAL
  | IDENT
  | symbol_ref
  | expr '+' expr
  | expr '-' expr
  | expr '*' expr
  | expr '/' expr
  | expr '%' expr
  | expr '<<' expr
  | expr '>>' expr
  | expr '&' expr
  | expr '|' expr
  | expr '^' expr
  | unary_expr
  | expr '.' INT_SIZE
  | '(' expr ')'
  | 'sizeof' '(' symbol_ref ')'
  | 'sizeof' '(' IDENT ')'
  ;

unary_expr ::= '+' expr
  | '-' expr
  ;
```

Revision History

REVISION	DATE	DESCRIPTION
0.0	04/17/06	Created document
0.1	05/22/06	First draft
0.2	08/04/06	Updating and filling in missing pieces, changed scope to cover elftosb 2.x operation instead of just the command file format
0.3	12/14/06	Filled in missing documentation for ELF section load statements
0.4	07/05/07	Documented new features of version 2.2
0.5	04/01/11	Moved documentation to wiki