NXP Semiconductors
User's Guide

Document Number: MBOOTELFTOSBUG
Rev 3, 09/2018

elftosb User's Guide

by: NXP Semiconductors

1 Overview

The elftosb tool creates a binary output file that contains the user's application image along with a series of bootloader commands. The output file is known as a "Secure Binary" or SB file for short. These files typically have a .sb extension. The tool uses an input command file to control the sequence of bootloader commands present in the output file. This command file is called a "boot descriptor file" or BD file for short.

The elftosb tool is command line driven and can be separately built to run on Windows® OS, Linux® OS, and Apple Mac® OS. Currently, elftosb tool on Mac OS can support Kinetis devices but not i.MX devices. The MCU bootloader package contains the executable for all the three targets.

This document describes the usage of elftosb in terms of its command line parameters, input command file (.bd) structure, and contents of the output .sb file. The below block diagram describes the operation of elftosb at a high level, working on the inputs passed on command line such as image file, BD file, Key file, etc., and processing contents of the BD file to generate the output SB file.

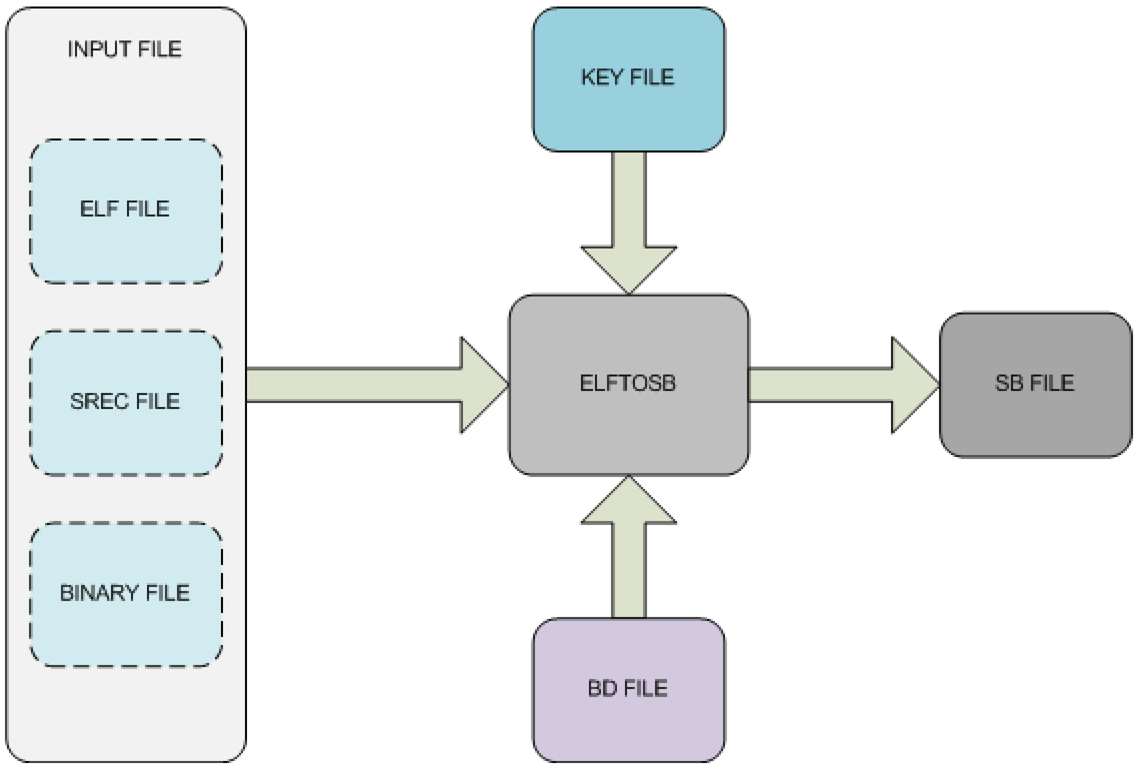

Figure 1. elftosb diagram

2 Command line interface

The elftosb has a set of command line options listed in the following table. Not all options are listed here, but 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 the "Sources" section).

The command line usage for the elftosb tool is:

```
elftosb [-?|--help] [-v|--version] [-f|--chip-family <family>]
           [-c|--command <file>] [-o|--output <file>]
           [-P|--product <version>] [-C|--component <version>]
           [-k|--key <file>] [-z|--zero-key] [-D|--define <const>]
           [-O|--option <option>] [-K|--keygen <option>] [-n|--number <int>]
           [-x|--extract] [-x|--sbtool] [-i|--index <int>] [-b|--binary]
           [-d|--debug] [-q|--quiet] [-V|--verbose]
           [-p|--search-path <path>]
           files...
```

Table 1. 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. For generating boot image for kinetis device specify "kinetis".
-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 zeros 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 and print list.
-?, --help	Show usage information.
-K/--keygen <option>	Generate AES-128 or AES-256 key file based on option value <128|256> (default <128>).
-n/--number <int>	Number of keys to generate per file (default=1)(valid only when -K is specified).
-x/--extract/--sbtool	Extract a specified section.
-i/--index <int>	Section index to extract (default=None Section) (valid only when -x is specified).

Table continues on the next page...

Table 1. Command line options (continued)

Option	Description
-b/--binary	Extract section data as binary (valid only when -x is specified) (Warning: -q is enabled implicitly if -b is specified).

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 the elftosb what format of the output *.sb* file to use. For generating boot image for kinetis device specify "kinetis". The case is ignored when comparing chip family names.

The output boot image is not encrypted by default. To encrypt the boot image, provide one or more keys. Use the -z switch to add a key that consists of all zeros. This is the default state of the hardware key in a chip that has not yet had its key programmed.

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 the command files, and the value can be any integer value allowed in the command files except for the multicharacter integer literals.

Before producing the output boot image, all constants set with the -D or --define options are set in the expression namespace inside the elftosb. These special constants override any normal constants with the same name that are specified in the command file. This enables 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 enables you to set or override the global option settings from the command line. The argument value is again an option name and the value is separated by an equals sign. The value can be any integer or string value allowed in the command file except for the multicharacter literals.

To extract the section content, use the -x/--extract/--sbtool option. Optionally, pass the index of the section you want with the -i/--index option. The section indices are printed under the "Section table" header in the normal output. The -x option alone causes a hex dump of the section contents to be printed inline with the normal output under the "Sections" header. If you additionally pass the -b/--binary option in the command line, then the binary contents of the section are instead echoed to stdout, enabling you to easily redirect the data to a file. In this mode, no other output is produced. In all cases, the section contents are unencrypted before being displayed.

To generate a random AES-128 or AES-256 key file in the format described in Chapter 4, elftosb key file format, use the -K/--keygen <128|256> option.

3 Command file

The command files are simply text files in any encoding that use ASCII for the lower 128 characters. In particular, this includes UTF-8. The line endings do not matter. The Unix, DOS, and Mac OS endings are all supported. Even the mixed line endings are accepted.

The standard extension for the command files (or the boot descriptor files as they are more commonly called) is *.bd*.

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. The elftosb command supports the ELF, S-record, and binary 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 the written command files that are fairly generic in purpose and reusable.

The command file declares a number of source files and assigns unique, easily referenced names to each. Each source can either explicitly call out the path to its file or let the user provide the path on the command line. When a user enters a path in a command line, the path can lead to any file and change each time the elftosb is called.

The command file then defines the sections required in the output *.sb* file. Each of these sections provides a definition for a sequence of operations (such as load and call) that refer to the contents of the source files or constant values present in the command file. These operations are mapped to the bootloader commands.

3.1 Blocks

The command file consists of several different blocks: options, constants, sources, keyblob, and sections. All blocks are optional, and there can be as many blocks of each type as the command file's 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 includes the contents enclosed in braces, as shown below.

Example 1. Basic block syntax

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

3.1.1 Block syntax

Blocks are arranged in two groups within a command file. The first are the configuration blocks: options, constants, sources, and the keyblob. 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.

The section definition blocks come after the configuration blocks. There can be any number of section blocks. Their lexical order in the command file determines the logical order of the sections in the output boot image.

3.1.1.1 Options

An options block contains zero or more name/value pairs and the option settings that assign values to global options used by the 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. The acceptable values depend on the particular option.

The option names are predefined by the 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 the following table.

Table 2. Option names for elftosb

Option name	Applies to	Description
alignment	Section	Power of 2 integer alignment requirement for start of the boot image section.
cleartext	Section	Integer Boolean value. Makes a section unencrypted, even in an encrypted image.
componentVersion	Boot image	Same format as the productVersion option.
driveTag	Boot image	Integer, sets drive tag field of the image header.

Table continues on the next page...

Table 2. Option names for elftosb (continued)

Option name	Applies to	Description
flags	Boot image	Integer value that is used for image-wide flags.
productVersion	Boot image	Version string in the form "xxx.yyy.zzz".
secinfoClear	GHS ELF source files	One of the "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 appendix describing the boot image format for the possible values of this field. The same applies to the sectionFlags option, except that it sets the flags field in the boot image section header.

3.1.1.2 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
  ;
```

The constant values retain the integer word size that they evaluated to when they are used in another expression.

The constant defined earlier in the constants block can be used in the definition of constants that follow it, as shown in the following example.

Example 2: Constants block

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

3.1.1.3 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. The individual source definitions are terminated with a semicolon.

The syntax for the source value depends on the type of the source definition. The two types are explicit paths and externally provided paths. The 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 enables you to easily vary the input file by changing the command line arguments.

The source 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
  ;
```

The 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 the options in an options block, but only few options apply to the sources. See Table 2 for the complete list of options.

3.1.1.4 Keyblob

There may be any number of keyblob blocks, but they must all come before any section block types within the command file. A keyblob block must be referenced in a keywrap statement to be useful. The grammar for a keyblob block is shown below.

```
keyblob_block    ::= 'keyblob' '(' int_const_expr ')' keyblob_contents
                 ;
keyblob_contents
                 ::= '{' ( '(' keyblob_options_list ')' )* '}'
                 ;
keyblob_options_list
                 ::= keyblob_option ( ',' keyblob_option )*
                 ;
keyblob_option   ::= ( IDENT '=' const_expr )?
                 ;
```

If the options list is empty, the corresponding keyblob entry is allocated but not populated. The supported keyblob option identifiers are:

- start: The start address of the encrypted region.
- end: The end address of the encrypted region.
- key: The AES-128 counter mode encrypted key for a region.
- counter: The initial counter value for a region.

Keyblob Block

```
keyblob (0)
{
 (
      start=0x68001000,
      end=0x68001fff,
      key="00112233445566778899AABBCCDDEEFF",
      counter="0011223344556677"
  )
  ()
  ()
  ()
}
```

NOTE

The region addresses that appear in the keyblob block must be supported by the underlying hardware. For example, the alias addresses may not be supported. See the corresponding chip reference manual.

3.1.1.5 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 3.12, "Statements".

```
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 in the following example, there are two forms of section contents. The first, with braces containing a sequence of statements, creates a bootable section with a number of bootloader commands as its content. Most sections are in this form. The syntax for statements in a bootable section are covered in detail in Section 3.12, "Statements".

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 or data sections. As examples, data sections can be used to hold resource files or a backing store for virtual memory paging.

An SB file created for a MCU ROM must start with a bootable section. The ROM stops processing at the end of this bootable section. Additional bootable and data sections are ignored.

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 is reported.

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 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 Table 2.

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 example, set the default section alignment to 2 K and then align one particular section to 4 K.

Sections are always created in the output .sb file in the order 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.

3.2 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, see the following table for examples of how tokens are written into the file.

Table 3. 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.

3.3 Whitespace

The 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.

3.4 Keywords

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

Table 4. 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
qspi	unsecure
ifr	jump_sp
enable	keyblob
start	end
key	counter
keywrap	reset
all	encrypt

3.5 Comments

The single-line comments are introduced at any point on a line with either a pound character ("#") or two slashes ("//") and run until the end of the line.

The multi-line comments work exactly the same as they do in ANSI C.

They begin with

"/*"

and end with

"*/"

Additionally, as with ANSI C, there is no support for the nested multi-line comments.

3.6 Identifiers

The identifiers are used for the 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.

3.7 Integers

The integer literals are of one of these three supported bases: binary, decimal, or hexadecimal. The decimal integers have no prefix. The hexadecimal integers must be introduced with '0x' and the binary integers must be introduced with '0b'.

The integer literals can optionally be followed by a metric multiplier character: "K", "M", or "G". The space characters are allowed between the last digit and the multiplier. Note that the 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. The lowercase "k", "m", or "g" are not allowed.

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

The integer constants can also be created with the 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 of 0x6f68 hex (the value of the characters "o" and "h" in ASCII) or 28520 decimal.

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

3.7.1 Integer expressions

An integer expression can be used in any place where 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. The following table lists the available operators.

Table 5. Integer expression operators

Operator	Description
+	add
-	subtract
*	multiply
/	divide
%	modulus
&	bitwise and
|	bitwise or
^	bitwise xor
<<	logical left shift

Table continues on the next page...

Table 5. Integer expression operators (continued)

Operator	Description
>>	logical right shift
.	set integer size
sizeof()	get size of a constant or symbol

In addition to the operators listed in the above table, the unary plus and minus are also supported. For the operator precedures, see the following table.

3.7.1.1 Operator precedence

This table 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 6. Operator precedence in increasing order

Operator	Description
|	bitwise or
^	bitwise xor
&	bitwise and
<< >>	left shift, right shift
+ -	add, subtract
* / %	multiply, divide, modulus
.	word size
unary + -	unary positive and negative

3.7.1.2 Word size operator

The 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. A 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 assumes the largest word size of two operands. So a byte-sized integer multiplied by a half-word-sized integer results 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 is truncated if necessary.

3.7.1.3 Sizeof operator

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

3.7.1.4 Constant references

Along with the integer literals, the expressions may refer to the 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 the sources share the same namespace as the constants, they cannot be used within an integer expression.

3.7.1.5 Symbol references

Just like constants, the symbol references may also be used in integer expressions. 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.

3.7.2 Boolean expressions

Unlike integer expressions, the Boolean expressions are limited in use. They are only used when defining a constant or an option, or as the conditional for the if and else-if statements described in Section 3.12.6. Specifically, you cannot use the Boolean expressions as the source or target of a load statement.

Table 7. 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?

There is a number of new operators that can be used in the Boolean expressions. In addition to those operators, the unary not operator (or the character "!") is supported. All of these operators evaluate to either 0 or 1. Like in 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()", returns true if the source file named inside the parentheses exists on the disk and was opened successfully. It is a syntax error to put a source name that was not 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 is 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 not important (because the expression has the same end value either way), the right hand operand is not evaluated. This is particularly useful in expressions 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 is always evaluated and "const" is not defined, an error is reported.

3.8 Strings

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

3.9 Section names

The named sections of the 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 the section name. The name is actually a standard glob-type expression that can match any number of ELF sections. The accepted characters include alphanumerics, underscore, period, asterisk, question mark, dashes, caret, and square brackets. Many of these characters are used only as a 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.

3.10 Symbol references

The 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.

If no source file is placed before the colon, the symbol comes from the default source file that is specified with a from statement. If the symbol reference is not within the context of a from statement, the source file name is required.

3.11 Binary objects

The 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 and 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.

3.12 Statements

Each statement within a bootable section block describes an operation that is performed by the bootloader when it processes the output *.sb* file. The individual statements correspond to at least one (and possibly more than one) boot command created in the output file. The intent is for the statements to describe what the user wants to happen, rather than exactly which boot commands are to be generated. It is the responsibility of the 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:

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

3.12.1 Load

The load statement is used for any operation where the user wants to store some form of data into the memory. In terms of the bootloader commands, this includes the 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 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 the elftosb determine how to best convert it into bootloader commands.

The load command is also used to write to the flash memory. When loading to the flash memory, the region being loaded to must be erased prior to the load. See the erase command for details of how to accomplish this.

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 composed 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. The data sources, and therefore segments, may or may not have a natural location in the memory associated with them. This natural location is the range of addresses in the memory where the data is 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 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 '>' symbol. The values for the target are either an address or an 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. The 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 the ELF sections to different addresses from which they were linked.

When the target is an address range or a symbol 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. Whenever the natural size of the source is larger than the target range, the source is truncated to the size of the range when loaded.

The 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 is useless to load each segment to the same address.

The most common form of a 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. Not all sections are loaded; only those sections whose type is SHT_PROGBITS or SHT_NOBITS are considered. All sections from the ELF files have natural locations and sizes.

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

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

```
load mySRecFile;
```

Binary file — The entire content of the file forms 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 that 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 the ELF sections using glob expressions. See Section 3.9 for more information about the 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 the following source name can be omitted if the load statement is within the from statement. These examples demonstrate the syntax:

Example load block

```
// 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, an explicit load target is not likely to be used. In fact, the elftosb only allows explicit targets for statements that select a single ELF section because it does not make sense to load multiple sections to the same target address. On the other hand, it can 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 example, the section name "$.sdram.*" matches every section that does not begin with ".sdram.".

As an example of how multiple section names in the list work, see the third example load statement above. The first section name "$.text.*" matches every ELF section that begins with ".text.". 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" is selected.

Integer value — The integer values are a unique type of load data, in that the value is used as a pattern to fill a region of memory. The 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;
```

If you load an integer value to a single address without a range, the load fills 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 a memory that contains a message to be displayed to the user or printed over a serial port. When 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;
```

3.12.1.1 Load IFR

An IFR option to the load command that specifies that the data in the data source should be programmed to the Flash IFR index indicated in the target location.

The grammar is as follows:

```
load_ifr_stmt   ::=    'load ifr' int_const_expr '>' int_const_expr
                ;
```

There are two forms of the load IFR statement, one to program to a 4-byte IFR location and another to program to an 8-byte IFR location.

4-byte load IFR statement

```
section (0)
{
    load ifr 0x1234567 > 0x30;
}
```

8-byte load IFR statement

```
section (0)
{
    load ifr {{11 22 33 44 55 66 77 88}} > 0x40;
}
```

3.12.2 Call

The call statement is used for inserting a bootloader command that executes a function from one of the files that are loaded into the 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, depending on the output boot image format. In general, the "call" commands are expected to return the bootloader and the "jump" commands are not. For the boot images, "call" produces the ROM_CALL_CMD and "jump" produces the 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 have their own syntax. All forms of the target boil down to just an address in the 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 uses 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 that does not support entry points or does not have one set is used, an error is reported.

```
# call the entry point
call myElfFile;

# same here
jump mySRecFile;

# this produces 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. 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 commands. 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.

3.12.3 From

More of a block than a true statement, the from statement has the simplest syntax. It produces no boot commands by itself. Instead, the from statement enables you to use simpler forms of the statements contained within it.

The simple grammar for the from statements follows this form:

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

The 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 the additional from statements, as they cannot be nested.

Certain forms of the load and call statements use an implicit source file. All the from statement does is setting this implicit source file for the statements found within it. This makes for cleaner and easier read command files.

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;
    }
}
```

The above example demonstrates how the from statement is used. The load and call statements inside the from statement do not have any source explicitly listed. Which file should the named sections be loaded from? Which file is the "_start" symbol located in? The from statement supplies the implicit source file for these statements.

The load statement loads all sections in the example source that have a name beginning with ".ocram.". The call statement generates a call boot command to the address of the "_start" symbol within the example source file.

3.12.4 Erase

The erase statement inserts a bootloader command to erase the flash memory.

Grammar for the erase statement:

```
erase_stmt ::= 'erase' address_or_range
  |       'erase' 'all'
  ;
```

There are two forms of the erase statement. The simplest form (erase all) creates a command that erases the available flash memory. The actual effect of this command depends on the runtime settings of the bootloader and whether the bootloader resides in the flash, ROM, or RAM.

The second form of the erase statement accepts an address or address range as an argument. It erases the flash sectors that are intersected by the address or range. To erase a single sector, provide a single address within that sector.

The erase statement

```
sources
{
    example = extern(0);
}

# create a section
section (0)
{
    erase all;
    load example;
}
```

3.12.5 Print

The print statement are 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 the standard out. The message is visible unless the caller enabled the quiet output feature. The warning statement does basically the same thing as the info statement, except that it prefixes the message with "warning:". Additionally, the message is always visible. Finally, the error statement stops the execution of the elftosb immediately and prints the message prefixed by "error:".

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 stops after this is printed
        error "Buffer size $(bufsize) is too small!";
    }
    else
    {
        info "Buffer size $(bufsize) is acceptable";
    }
    /* ...more... */
}
```

The three print statements support the substitution of constant values and source file paths using a syntax like that for the Unix shell variable substitution. A constant name or source file name placed in parentheses and prefixed with a dollar sign causes the appropriate value to be inserted before the message is printed to the standard out.

For the constant substitution, there is a limited control of the formatting of the constant's value. The 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. The "d" character formats the constant as decimal and the "x" character formats it as hexadecimal. For example, "$(x:floop)" formats the constant "floop" as hex.

3.12.6 If-Else

To make it easier to create reusable boot descriptor files, the elftosb has the if-else statement. These statements work just like the if statements in any other language you have used. Chain as many if-else statements as you like. 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 the ANSI C. No parentheses are required around the boolean expression after the "if" keyword. Additionally, curly braces are always required around the statements on both the if and else branches.

All types of statements are allowed inside the if-else statement, including the from statements. The converse is also true: the if-else statements may be placed inside the from statements.

3.12.7 Erase QuadSPI all statement

The erase QuadSPI all statement erases the entire external QuadSPI flash.

The grammar is:

```
erase_qspi_stmt ::=      'erase' 'qspi' 'all'
                ;
```

Erase QuadSPI All statement

```
section (0)
{
    erase unsecure all;
}
```

3.12.8 Erase Unsecure All statement

The erase unsecure all statement erases the entire internal flash, leaving the flash security disabled.

The grammar is:

```
unsecure_stmt    ::=     'erase' 'unsecure' 'all'
                 ;
```

Erase Unsecure All statement

```
section (0)
{
    erase unsecure all;
}
```

3.12.9 Enable QuadSPI statement

The enable QuadSPI statement initializes the external QuadSPI flash using a parameter block previously loaded to the RAM.

The grammar is:

```
enable_stmt     ::=    'enable' 'qspi' int_const_expr
                ;
```

Enable QuadSPI statement

```
section (0)
{
    # Load quadspi config block bin file to RAM, use it to enable QSPI.
    load myBinFile > 0x20001000;
    enable qspi 0x20001000;
}
```

3.12.10 Reset statement

The reset statement generates a booloader reset command that resets the target device. Any additional commands in the SB file after the reset command are ignored by the bootloader.

The grammar is:

```
reset_stmt      ::=     'reset'
                ;
```

Reset statement

```
section (0)
{
    reset;
}
```

3.12.11 Jump with stack pointer statement

The jump with stack pointer statement generates a booloader jump command that sets the stack pointer before jumping. Any additional commands in the SB file after the jump command are ignored by the bootloader. The first argument is the value of the stack pointer. The second argument is the jump address. The third (optional) argument is the argument to the function being jumped to.

The grammar is below. The call_target and call_arg elements are described in the regular elftosb documentation.

```
jump_sp_stmt    ::=    'jump_sp' sp_arg call_target call_arg?
                ;
sp_arg   ::=    int_const_expr
                ;
```

Jump with stack pointer statement

```
section (0)
{
    jump_sp 0x20000e00 0x1000 (0x5a5a5a5a);
}
```

3.13 Common usage example

The most common use of elftosb is to simply load a single ELF file and jump to its entry point, which is almost always the _start symbol defined by the C runtime library.

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
}
```

4 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 32/64 hexadecimal characters, for a total of 128/256 bits of key data. Multiple keys may appear in a key file. Each key is on a separate line. The line-ending format is not significant.

Example 16. Key file with two 128 bits keys

```
3F3CFBC001F399991035C3C6C7065924
1BA3CD4030FC4376B4AA8CB5E932432E
```

Example 17. Key file with one 256 bits key

```
AAB5CCFB687D378C93821E8793337EA8F98B48A0B596F36CDD169347322E8C87
```

The contents of a key file are in plaintext.

5 Appendix A: Command file grammar

The grammar for the command file format is shown below in the 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
   ;
```

```
keyblob_block ::= 'keyblob' '(' int_const_expr ')' keyblob_contents
  ;

keyblob_contents
  ::= '{' ( '(' keyblob_options_list ')' )* '}'
  ;

keyblob_options_list
  ::= keyblob_option ( ',' keyblob_option )*
  ;

keyblob_option ::= ( IDENT '=' const_expr )?
  ;

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
  ;

load_ifr_stmt ::= 'load ifr' int_const_expr '>' int_const_expr
  ;

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? ')'
  ;

jump_sp_stmt ::= 'jump_sp' sp_arg call_target call_arg?
  ;

sp_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
  ;

encrypt_stmt ::= 'encrypt' '(' int_const_expr ')' encrypt_stmt_list
  ;

encrypt_stmt_list
  ::= '{' ( statement )* '}'
  ;

erase_qspi_stmt ::= 'erase' 'qspi' 'all'
  ;

unsecure_stmt ::= 'erase' 'unsecure' 'all'
  ;

enable_stmt ::= 'enable' 'qspi' int_const_expr
  ;

reset_stmt ::= 'reset'
  ;

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
    ;
```

6 Appendix B: SB boot image file format

6.1 Glossary

AES-128 - Rijndael cipher with block and key sizes of 128 bits.

Block cipher - Encryption algorithm that works on blocks of N={64, 128, ...} bits.

CBC - Cipher Block Chaining, a cipher mode that uses the feedback between the ciphertext blocks.

CBC-MAC - A message authentication code computed with a block cipher.

Cipher block - The minimum amount of data on which a block cipher operates.

Ciphertext - Encrypted data.

DEK - Data encryption key, a one-time session key used to encrypt the bulk of the boot image.

ECB - Electronic Code Book, a cipher mode with no feedback between the ciphertext blocks.

Hash - Digest computation algorithm.

KEK - Key Encryption Key, used to encrypt a session key or DEK.

MAC - Message Authentication Code. Provides integrity and authentication checks.

Message digest - Unique value computed from the data using a hash algorithm. Provides only an integrity check (unless encrypted).

Plaintext - Unencrypted data.

Rijndael - Block cipher chosen by the US Government to replace DES. Pronounced "rain-dahl".

Session key - Encryption key generated at the time of encryption. Only ever used once.

SHA-1 - Hash algorithm that produces a 160-bit message digest.

6.2 Introduction

The entire boot image format is built around the requirements of AES-128, with its minimum block size of 128 bits or 16 bytes. AES-128 is the symmetric block cipher that is used for encrypted boot images. Using its block size as the base unit throughout the image makes it much easier to accommodate the encryption.

To support multiple executables within one image, the format has the concept of sections. Each section can contain a standalone bootable image, or may be a part of a larger sequence of sections. A boot command is provided that can be used to direct the bootloader to continue from another section at runtime.

There is a number of features of this format that are not useful for all applications or methods of reading. For instance, the section table is only useful if the random access to the boot image is available, while the boot tags are most useful when booting from a streaming media. The goal here is to provide a great deal of capability to the image, regardless of how it is accessed.

6.3 Basic types

Several basic C types are used throughout this document to represent cipher blocks, keys, and other important elements. The definitions for these types are shown below.

```
//! An AES-128 cipher block is 16 bytes.
typedef uint8_t cipher_block_t[16];

//! An AES-128 key is 128 bits, or 16 bytes.
typedef uint8_t aes128_key_t[16];

//! A SHA-1 digest is 160 bits, or 20 bytes.
typedef uint8_t sha1_digest_t[20];

//! Unique identifier for a section.
typedef uint32_t section_id_t;
```

6.4 Boot image format

The boot image format consists of five distinct regions. First, there is a plaintext header containing basic information about the image. A section table, also plaintext, comes afterwards. It describes each of the different sections within the image. For encrypted images, a key dictionary that is used to support multiple customer keys then follows. Next, each section has its data, which is prefixed with a tag used by the bootloader. Finally, the image terminates with an authentication code for the entire image. The figure below shows the basic layout of a boot image.

The image format is designed to be read from the streaming media without the support for random access while requiring the caching of as little data as possible. However, the format also includes features that are most useful when the random access to the image is possible. For example, the image ends with an authentication code computed from the entire rest of the image. This isn't particularly useful for the ROM, but can be used by the host-resident utilities to verify and authenticate the boot images before using them.

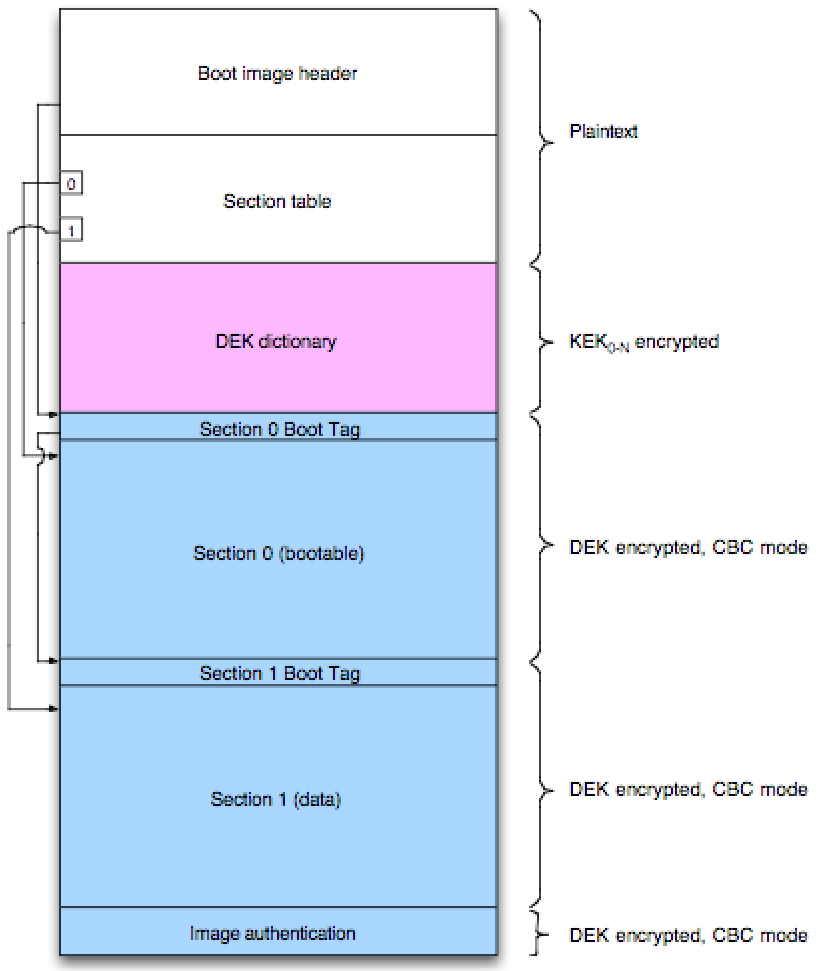

Figure 2. Boot image regions

The basic unit size of the format is that of an AES-128 cipher block, or 16 bytes. Every region in the file always starts on a cipher block boundary. Every field within the image is formatted in the little endian byte order.

6.4.1 Image header

The header of a boot image is always unencrypted. It provides the required information about the image as a whole, as well as some useful pointers to the other regions within the image.

The image header size is always a round number of cipher blocks. Any padding bytes that are necessary to fill out the structure are always set to random values. No padding is necessary if the header completely fills the last cipher block it occupies. The section table dictionary immediately follows.

The C structure definition for the image header is:

```
struct boot_image_header_t
{
    union
    {
        sha1_digest_t m_digest;
        struct
        {
            cipher_block_t m_iv;
            uint8_t m_extra[4];
        };
    };
    uint8_t m_signature[4];
    uint8_t m_majorVersion;
    uint8_t m_minorVersion;
    uint16_t m_flags;
    uint32_t m_imageBlocks;
    uint32_t m_firstBootTagBlock;
    section_id_t m_firstBootableSectionID;
    uint16_t m_keyCount;
    uint16_t m_keyDictionaryBlock;
    uint16_t m_headerBlocks;
    uint16_t m_sectionCount;
    uint16_t m_sectionHeaderSize;
    uint8_t m_padding0[2];
    uint8_t m_signature2[4];
    uint64_t m_timestamp;
    version_t m_productVersion;
    version_t m_componentVersion;
    uint16_t m_driveTag;
    uint8_t m_padding1[6];
};
```

The fields of `boot_image_header_t` have their descriptions in the following table. The flags defined for the `m_flags` field are shown in the second table.

Table 8. Image header fields

Field	Description
`m_digest`	SHA-1 digest of all fields of the header prior to this one. The first 16 bytes (of 20 total) also act as the initialization vector for CBC-encrypted regions.
`m_signature`	Always has the value 'STMP'.
`m_majorVersion`	Major version of the boot image format, currently 1.
`m_minorVersion`	Minor version of the boot image format, currently 1 or 2.
`m_flags`	Flags associated with the entire image.
`m_imageBlocks`	Size of the entire image in blocks.
`m_firstBootTagBlock`	Offset from the start of file to the cipher block containing the first boot tag.
`m_firstBootableSectionID`	Unique identifier of the section to start booting from.
`m_keyCount`	Number of entries in the DEK dictionary.
`m_keyDictionaryBlock`	Starting block number, from the beginning of the image, for the DEK dictionary.
`m_headerBlocks`	Size of the entire image header in blocks.
`m_sectionCount`	Number of sections.
`m_sectionHeaderSize`	Size in blocks of a section header.
`m_padding0`	Two bytes of padding to align `m_signature2` to a word boundary. Set to random values.
`m_signature2`	Always set to 'sgtl'. This second signature is only present in files with a minor version greater or equal to 1.
`m_timestamp`	Timestamp in microseconds size 1-1-2000 00:00 when the image was created.

Table continues on the next page...

Table 8. Image header fields (continued)

Field	Description
`m_productVersion`	Product version.
`m_componentVersion`	Component version.
`m_driveTag`	Identifier for the disk drive or partition containing this image.
`m_padding1`	Eight bytes of padding to fill out the cipher block. Set to random values.

Table 9. Boot image fields

Constant	Bit	Description
`ROM_DISPLAY_PROGRESS`	0	Turns on the progress reports of executed commands.
`ROM_VERBOSE_PROGRESS`	1	Prints the extra information in reports about the executed commands. Applies only if `ROM_DISPLAY_PROGRESS` is also enabled.

The `m_majorVersion` and `m_minorVersion` fields describe the version of the boot image format, not the version of the ROM (as in the previous boot image formats). The major version field is currently 1. Any time this field is changed, the format is no longer backwards compatible with the previous versions and a new bootloader is required. The minor version field should be incremented for any format changes that are backwards compatible with the previous bootloader versions. For instance, adding a new field to the end of the image header is backwards compatible due to the presence of the `m_headerBlocks` field. In this case only `m_minorVersion` should be incremented. However, if the image header fields are reordered, the current bootloader can no longer read the image and the `m_majorVersion` field must be incremented. See the file format versions table at the end of this document for more version details.

If the value of the `m_keyCount` is zero, then the boot image is fully unencrypted. The image is always encrypted if there is at least one key in the dictionary.

The SHA-1 digest of the header provides a basic integrity check for unencrypted images. It does not provide any extra security because it can simply be updated along with any changes made to the header.

Throughout the rest of the file, any time something is encrypted using the CBC mode, the first 16 bytes of the `m_digest` field are used as the initialization vector. The digest is random enough because the header differs for all boot images. The `m_timestamp` field, in addition to its nominal purpose, serves to guarantee that the plaintext header is different for every boot image created. In addition to improving the randomness of the header digest, this is important because the header is authenticated with the customer key.

The `m_keyDictionaryBlock` field is also used to help the boot ROM speed up its processing of the header. This value can be calculated from other header fields, but having it pre-calculated allows the ROM code to keep track of fewer header fields.

The `m_productVersion` and `m_componentVersion` fields contain version values that describe the firmware within the boot image. These fields use the following C structure defintion:

```
struct version_t
{
    uint16_t m_major;
    uint16_t m_pad0;
    uint16_t m_minor;
    uint16_t m_pad1;
    uint16_t m_revision;
    uint16_t m_pad2;
};
```

Within each of the major, minor, and revision fields of the `version_t` structure, the version number is in the right-aligned BCD format. The default value for both versions is 999.999.999.

The `m_padding0` and `m_padding1` fields are used to align other fields and round out the structure size to an even cipher block. These bytes are set to random values when the image is created to add to the “whiteness” of the header for cryptographic purposes.

6.4.2 Section table

The section table is basically an index of the starting block and length for each section within a boot image. It also contains flags that apply solely to that section.

The table is always unencrypted and comes immediately after the plaintext image header and before the DEK dictionary, if the dictionary is present.

The C type definition for the section table and its entries are as follows:

```
struct section_header_t
{
    section_id_t m_identifier;
    uint32_t m_offset;
    uint32_t m_length;
    uint32_t m_flags;
};
struct section_table_t
{
    section_info_t m_sections[1];
};
```

The fields of `section_header_t` are described in the following table. The flags defined for the `m_flags` field of `section_header_t` are as shown in the second table.

Table 10. Section header fields

Field	Description
`m_identifier`	Unique 32-bit identifier for this section.
`m_offset`	The starting cipher block for this section's data from the beginning of the image.
`m_length`	The length of the section data in cipher blocks.
`m_flags`	Flags that apply to the entire section.

Table 11. Section flags

Constant	Bit	Description
`ROM_SECTION_BOOTABLE`	0	The section is bootable and contains a sequence of bootloader commands.
`ROM_SECTION_CLEARTEXT`	1	The section is unencrypted. Applies only if the rest of the boot image is encrypted.

The length of each entry in the section table comes from the `m_sectionHeaderSize` field of the image header. The entries are always a round number of cipher blocks long, being padded if necessary. All entries in the table are of the same length. In version 1 of the file format, the section table entries are a single cipher block long and have no padding.

The total number of sections (and thus the number of entries in the section table) is given in the `m_sectionCount` field of the image header. This should always be at least 1 for a valid bootable image. If it is 0, then the image contains no boot commands and is considered invalid. In addition, there must be at least one section with the `ROM_SECTION_BOOTABLE` flag set for an image to be valid.

The size of the section table is either `(header.m_sectionCount * header.m_sectionHeaderSize) cipher blocks or (header.m_sectionCount * header.m_sectionHeaderSize * 16)` bytes.

6.4.3 DEK dictionary

The key dictionary always follows the image header in the next cipher block in the encrypted images. The unencrypted images do not have a DEK dictionary.

Its purpose is to allow a single boot image to work with any number of customer keys. This is accomplished by generating a new key, the Data Encryption Key (DEK), every time a boot image is generated. Except for this dictionary, the rest of the image is encrypted with this DEK. The dictionary is used to map from any given customer key to the DEK in a secure manner, by encrypting the DEK with each customer key to be supported. Thus, the DEK is never available without a valid customer key.

Each entry in the dictionary consists of two pieces of data: the Message Authentication Code (MAC) and the encrypted DEK itself. The MAC acts as a check code (a known value that can be searched for). Otherwise, there is no way to tell a valid decryption of the DEK from garbage.

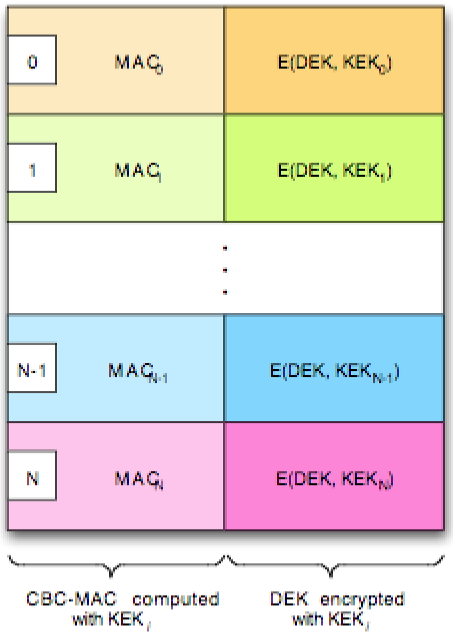

Figure 3. DEK dictionary

The MAC is generated using a technique called CBC-MAC. The header of the boot image and the section table, which are both always plaintext, are encrypted in the CBC mode using the KEK for the given dictionary entry. The initialization vector for this encryption is always zero. Only the last cipher block is retained throughout this process. The authentication code is the last cipher block.

The C type definition for the DEK dictionary is as follows:

```
struct dek_dictionary_entry_t
{
    cipher_block_t m_mac;
    aes128_key_t m_dek;
};
struct dek_dictionary_t
{
    dek_dictionary_entry_t m_entries[1];
};
```

The `m_dek` field in each entry is encrypted using the KEK in the CBC mode using the IV from the image header. The CBC-MAC result, held in the `m_mac` field, is not encrypted. This is not necessary because it is generated from the secret OTP key.

The number of entries in the dictionary is determined from the m_keyCount field of the image header. The dictionary size is always `header.m_keyCount * 2 cipher blocks, or header.m_keyCount * 32` bytes. If `m_keyCount` is zero, then the DEK dictionary occupies no cipher blocks in the image and the entire image is unencrypted.

The only realistic limit on the size of the dictionary is the boot time. The more dictionary entries, the longer it takes to boot the device. At least the algorithm to search for the DEK should be O(n).

6.4.4 Section boot tags

Before each section data region, there is a special tag cipher block that describes the following section. These tags are called boot tags because the boot ROM uses them to search for sections without having to maintain a copy of the entire section table in the memory or re-read portions of the image from the storage. Boot tags are always paired with a section data region—there is never one without the other. Another way to think of boot tags is as a section header local to the section contents.

The actual structure of a boot tag is that of the `ROM_TAG_CMD` bootloader command. Reusing the boot command structure for the boot tag simplifies the ROM code. The tag command contains duplicates of some of the fields from the section table entry for the section data region with which it is paired. The most important of these are the section identifier and the section length (in blocks).

Because there is no padding allowed between the sections, the section length effectively points to the next boot tag. This allows the boot ROM to easily search for section data regions by comparing the identifiers and following the chain formed by the boot tags. The last boot tag in an image always has its `ROM_LAST_TAG` flag set to help the ROM know at what point to stop searching.

6.4.5 Section data regions

There are two types of section data regions. The first is a bootable region that contains a sequence of boot commands. Second is any non-bootable region that can contain arbitrary data that is not processed by the boot ROM. These regions may contain resources or other data to be used by customer applications.

The contents of a bootable region are simply a number of bootloader commands sequenced one after another. Bootable sections must always begin with a `ROM_TAG_CMD` bootloader command. See section 9 for more details about the structure of bootloader commands and the details of individual commands.

An SB file created for a MCU ROM must start with a bootable section. The ROM stops processing at the end of this bootable section. Additional bootable and data sections are ignored.

Section data regions must be ordered in the same sequence as they appear in the section table. That is, the data region for section number 1 must come after the data region for section number 0 within the boot image. Also, there must be no pad blocks inserted before or after section data regions, even though the format implicitly supports this by the use of cipher block pointers. These restrictions are intended to make the processing of the boot image by the ROM easier.

6.4.6 Image authentication code

Every boot image ends with an authentication code that is computed from the entire contents of the image (excluding the authentication code, of course). This code is a SHA-1 digest encrypted with the DEK using the CBC mode. The authentication code consumes two cipher blocks in the image, with three words of padding added after the last word of the SHA-1 digest (because the SHA-1 digest is 160 bits and the cipher blocks are 128 bits). The padding bytes are set to random values.

The digest is computed from the following components, in this order: plaintext header, plaintext section table, DEK dictionary, plaintext section contents.

The hash algorithms themselves do not provide authentication, only an integrity check. However, if the digest is encrypted with a secret key, then it can be used to provide authentication.

In an unencrypted boot image, the image authentication code is of course also unencrypted. The code no longer provides authentication, but does still provide an integrity check over the entire image.

The authentication code always starts at the cipher block number (`header.m_imageBlocks - 2`).

6.5 Encryption details

6.5.1 Encryption process

The process of encryption takes place solely within the elftosb utility, because it converts the ELF or S-record binaries into a boot image. The sequence below shows the steps that the elftosb takes to encrypt an image.

1. Build plaintext image header
    a. Generate IV
    b. Compute SHA-1 over image header
2. Generate plaintext section table
3. Generate DEK
4. For every KEK:
    a. Read KEK key file
    b. Compute CBC-MAC over plaintext image header with IV=0
    c. Encrypt DEK with KEK in CBC mode with IV from header
    d. Combine unencrypted CBC-MAC and encrypted DEK into dictionary entry
5. For every section:
    a. Generate a `ROM_TAG_CMD` as the boot tag for this section
    b. Encrypt the boot tag using CBC mode with IV from header
    c. Generate plaintext section contents
    d. Encrypt the section contents using CBC mode with IV from header
6. Compute SHA-1 digest of image
7. Encrypt image digest using CBC node with IV from header

6.5.2 Decryption process

The decryption process takes place within the ROM. In addition, there is a host utility program that can decrypt a boot image for testing purposes.

1. Read the first cipher block of the image header. The `m_keyCount` field in the first cipher block tells if the image is encrypted or not. If the image is encrypted, the m_keyCount is going to be non-zero.
2. As the image header is read, compute the CBC-MAC over it using the customer key.
3. For each entry in the DEK dictionary:
    a. does the `m_mac` field match the computed CBC-MAC? If not, jump to the next entry.
    b. if the `m_mac` field matches, decrypt the DEK using the customer key and exit the loop.
4. For each section table and any section data regions that are to be read:
    a. decrypt the region using the DEK in the CBC mode with the IV from the header.

6.5.3 Boot commands

A bootable section in an image contains a sequence of boot commands and any data required by those commands. The commands are processed in a linear sequence starting with the first. Each boot command occupies a single cipher block, plus any cipher blocks required for data associated with that command. The C structure definition for a boot command is as follows:

```
struct boot_command_t
{
    uint8_t m_checksum;
    uint8_t m_tag;
    uint16_t m_flags;
    uint32_t m_address;
    uint32_t m_count;
    uint32_t m_data;
};
```

The commands described in this section are chosen to allow for the greatest flexibility in construction of boot images using the least number of command types. For the most part, the individual fields of boot_command_t vary in exact meaning between each command and are described below.

Because the m_checksum field is always calculated in the same way for every command, it deserves a special mentioning here. This field provides a cheap and easy way to verify that the cipher block contains a valid bootloader command. While eight bits are certainly not enough to act as a solid defense against either corruption or intended changes, it is still better than nothing.

The checksum is computed in the following manner:

```
boot_command_t bootCommand;
uint8_t * bytes = reinterpret_cast<uint8_t *>(&bootCommand);
uint8_t checksum = 0x5a;
int i;

// Unroll this loop for better optimization.
for (i = 1; i < sizeof(bootCommand); ++i)
{
    checksum += bytes[i];
}
```

Note that the checksum is computed only over bytes 1 through 15 of the boot_command_t structure for each boot command. Put another way, any additional cipher blocks of data following a command are not included in the checksum. Also note that the initial checksum value is 0x5a instead of 0. This is to prevent an all-zero command from also having a zero checksum.

The m_tag fields of each boot command contain a unique byte value that identifies which command the structure describes. The list of boot command tag values is shown in the following table.

Command tag value	Command tag mnemonic
0x00	ROM_NOP_CMD
0x01	ROM_TAG_CMD
0x02	ROM_LOAD_CMD
0x03	ROM_FILL_CMD
0x04	ROM_JUMP_CMD
0x05	ROM_CALL_CMD
0x06	Reserved
0x07	ROM_ERASE_CMD

Table continues on the next page...

Table continued from the previous page...

Command tag value	Command tag mnemonic
0x08	`ROM_RESET_CMD`
0x09	`ROM_MEM_ENABLE_CMD`
0x10	`ROM_PROG_CMD`

Any values of `m_tag` that do not match those listed in the previous table are invalid. If encountered, the bootloader stops and reports an error.

ROM_NOP_CMD

The `ROM_NOP_CMD` command is a no-operation. The bootloader simply skips it. All fields except the `m_tag` fields are ignored by the bootloader and may contain any value. However, until other uses are documented for these fields, they should contain the values presented in the following table.

Table 12. No-op command fields

Field	Description
`m_checksum`	Simple checksum, which comes to 0x5a when all other fields are zeros.
`m_tag`	0x00 or `ROM_NOP_CMD`
`m_flags`	0
`m_address`	0
`m_count`	0
`m_data`	0

Any values of `m_tag` that do not match those listed in the previous table are invalid. If encountered, the bootloader stops and reports an error.

ROM_TAG_CMD

The `ROM_TAG_CMD` is used as a kind of “key frame” that describes a section, or a local section header. It contains most of the fields from the section’s entry in the section table.

This command in not expected to appear within the command stream in a bootable section, and the bootloader just ignores it if it is present. The purpose of this command definition is to describe the structure of the boot tag cipher block. Boot tags use the exact same structure as the boot commands to make the bootloader’s job much easier.

Table 13. Hint Tag command fields

Field	Description
`m_checksum`	Simple checksum of the other fields of `boot_command_t`.
`m_tag`	0x01 or `ROM_TAG_CMD`
`m_flags`	Bit 0: `ROM_LAST_TAG`
`m_address`	The `m_tag` field from the section header.

Table continues on the next page...

Table 13. Hint Tag command fields (continued)

Field	Description
`m_count`	The number of cipher blocks that the data for this section occupies. This is also the number of cipher blocks until the next boot tag (except for the last one).
`m_data`	The `m_flags` field from the section header.

ROM_LOAD_CMD

This command is followed by an arbitrary number of cipher blocks that contain the data to be loaded into the memory, starting at the location specified by the `m_address` field of `boot_command_t`. The `m_count` field contains the number of bytes to be loaded into this location in the memory.

Table 14. Load command fields

Field	Description
`m_checksum`	Simple checksum of the other fields of `boot_command_t`.
`m_tag`	0x02 or `ROM_LOAD_CMD`
`m_flags`	Bit 0: `Reserved`
`m_address`	Memory address to which the data is stored.
`m_count`	Number of bytes to load. This is also the number of valid bytes in the data cipher blocks following this command.
`m_data`	CRC-32 over the data to be loaded.

The number of cipher blocks following the command is `(m_count + 15) / 16`. This means that there may be up to 15 bytes of padding in the last data cipher block. The pad bytes are always filled with random data. See the following figure for an example of how the cipher blocks are arranged for a load command with a data size of 18 bytes.

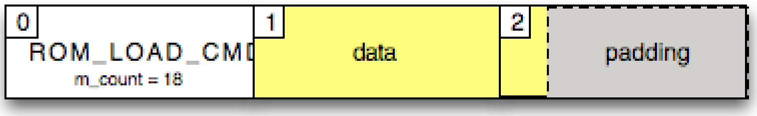

Figure 4. Load comment cipher blocks

There are no restrictions on the alignment for the `m_address` or m_count fields. It is up to the ROM implementation to decide how to best optimize the loading of data. Thus there is no guarantee on the order in which the data is written to the memory.

The `m_data` field contains a CRC-32 value computed over the data following the command header block. Any pad bytes in the last data cipher block are included in the CRC-32 calculation.

ROM_FILL_CMD

This bootloader command is used to fill the regions of memory with a bit pattern. The fill pattern is always a full 32 bits wide, but the byte aligned fill length and target address are fully supported.

Table 15. Fill command fields

Field	Description
`m_checksum`	Simple checksum of the other fields of `boot_command_t`.
`m_tag`	0x03 or `ROM_FILL_CMD`

Table continues on the next page...

Table 15. Fill command fields (continued)

Field	Description
`m_flags`	Always 0.
`m_address`	The starting memory address to which the fill pattern is written.
`m_count`	Number of bytes to fill.
`m_data`	The fill pattern. Always replicated across the word, regardless of the pattern size.

The fill pattern, regardless of its actual size, must be spread across the entire `m_data` field. A pattern that is a byte wide must be replicated four times across `m_data`, and twice for the half-word patterns.

When filling, the pattern is adjusted so that the most significant byte is aligned with the first byte to be filled. The following figure demonstrates what this looks like.

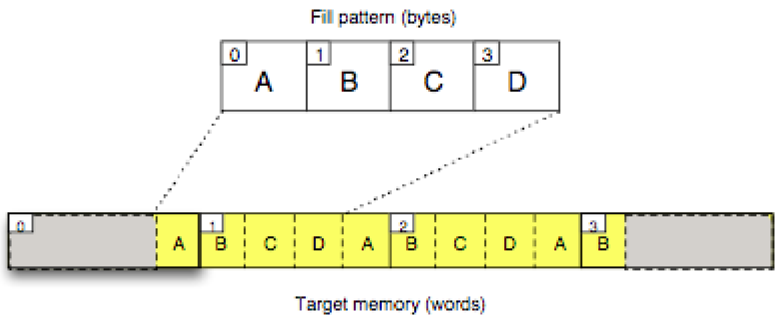

Figure 5. Fill pattern alignment

Note that this command is guaranteed to use the word writes between any unaligned ragged edges. This enables the use of the fill command as a word poke operation to write to the registers.

ROM_JUMP_CMD

When the bootloader encounters this command, the bootloading stops and the CPU control is transferred to the function residing at `m_address`. The contents of `m_data` is passed as a single argument to the function. The ROM does not expect to regain control of the CPU after this command is executed.

Table 16. Jump command fields

Field	Description
`m_checksum`	Simple checksum of the other fields of `boot_command_t`.
`m_tag`	0x04 or `ROM_JUMP_CMD`
`m_flags`	Bit 0: Reserved.
`m_address`	Address that the PC is set to.
`m_count`	Initial stack pointer if `m_flags` bit 1 is set, otherwise 0.
`m_data`	Argument to pass to the entry point in R0.

The prototype of the function executed by `ROM_JUMP_CMD` is as follows.

```
void jump_function( uint32_t arg );
```

Note the void result. If the function does return, the bootloader fails with the ERROR_ROM_LDR_JUMP_RETURNED error.

If bit 1 of `m_flags` is set, the m_count field contains the initial stack pointer register value to set before the jump is executed.

ROM_CALL_CMD

Like the `ROM_JUMP_CMD`, the `ROM_CALL_CMD` also invokes a function residing at `m_address` and passes the value `m_data` as its argument. The first and most important difference between the two commands is a semantic one (the function invoked by `ROM_CALL_CMD` is expected to relinquish the control and return to the ROM to allow the bootloading to continue). In addition, this command adds a second optional argument to the function prototype. This second argument can be used in combination with the function's return value to tell the bootloader to jump to another section in the current boot image or prepare for an entirely new boot image.

Table 17. Call command fields

Field	Description
`m_checksum`	Simple checksum of the other fields of `boot_command_t`.
`m_tag`	0x05 or `ROM_CALL_CMD`
`m_flags`	Bit 0: Reserved.
`m_address`	Address for the function to call.
`m_count`	0
`m_data`	Argument to pass to the function in R0.

The full prototype of the function executed by `ROM_CALL_CMD` is as follows:

```
int call_function( uint32_t arg, uint32_t * resultId );
```

The value of the `m_data` field is passed in the first argument to the function. The second argument is a pointer to a word that the function can modify to return a section or image ID.

The return value determines what happens when `call_function()` returns and whether the *resultId is examined. The possible return values are shown in the following table.

Table 18. Call command return values

Return value	Action
< 0	Negative values as errors.
0=SUCCESS	Success. Continue executing the commands in the current section.
1=ROM_BOOT_SECTION_ID	Switch to the section with the ID of *resultId.
2=ROM_BOOT_IMAGE_ID	Restart the bootloader in expectance of a new boot image. The *resultId value is passed to the driver when its initialization function is called again.
> 2	Ignored, same as SUCCESS.

The two positive return codes have special meanings. If the function returns ROM_BOOT_SECTION_ID, then the bootloader begins searching for a section of the current image that has an ID equal to the value returned through resultId. This section must follow the current section in the image or it is not going to be found because the bootloader only searches forward through the image. If no section with a matching unique identifier is found, the boot fails with an error.

If the function returns ROM_BOOT_IMAGE_ID, then the bootloader prepares itself to start reading an entirely new boot image file and signals this to the current boot driver by calling its initialization function again. The value returned through resultId is the ID of a boot image; the meaning of the image ID is specific to each boot driver and not all boot drivers support switching to new image files. The behavior is undefined when switching boot images with a driver that does not support this functionality.

Only when the return value is ROM_BOOT_SECTION_ID or ROM_BOOT_IMAGE_ID is the value pointed to by the resultId examined when the bootloader resumes execution. Because of this and how the ARM® ABI works, the functions that do not expect to return ROM_BOOT_SECTION_ID or ROM_BOOT_IMAGE_ID can shorten their prototype to the following:

```
int call_function_short( uint32_t arg );
```

ROM_ERASE_CMD

The erase command applies only to devices with an internal flash memory array (i.e., Kinetis devices). It executes a flash erase command for either the entire flash array or the range of memory specified in the command fields.

Table 19. Erase command fields

Field	Description
`m_checksum`	Simple checksum of the other fields of `boot_command_t`.
`m_tag`	0x07 or `ROM_ERASE_CMD`
`m_flags`	See the following table.
`m_address`	Start address of flash to erase.
`m_count`	Number of bytes of flash to erase. The end address is `m_address + m_count - 1`.
`m_data`	0

Table 20. Erase command flag bits

Bit	Flag	Description
0	`ROM_ERASE_ALL_MASK`	If set, erase all flash instead of only the specified range. If cleared, the `m_address` and `m_count` fields are used to determine the range of flash to erase.
1	`ROM_ERASE_ALL_UNSECURE_MASK`	If set, erase all flash and set the flash security state to disabled (erase-all-unsecure).
11:8	0x00 `kLdrMemoryCtrl_InternalFlash` 0x01 `kLdrMemoryCtrl_QSPI0`	Memory controller ID. Value 0x0 (default) indicates internal flash. Value 0x01 indicates external QSPI0 on devices that support QSPI0. If set to 0x01, then bit 1 (`ROM_ERASE_ALL_UNSECURE_MASK`) is ignored.

Bit 0 of the `m_flags` field determines whether the entire flash array is erased, or if only a subset is erased. If bit 0 is set, the command erases the whole flash. In this case, the m_address and m_count fields are ignored.

If bit 0 of `m_flags` is cleared, then the range of flash memory to erase is specified by the `m_address` and `m_count` command fields. Because the flash memory can only be erased on a whole-sector basis, all flash sectors that are intersected by the address range are erased. This applies even if the address range does not begin or end on an aligned sector boundary.

If bit 1 of the `m_flags` field is set, the flash security state is set to disabled after the flash is erased. See the specific chip reference manual for details on the flash erase-all-unsecure command.

Bits 11:8 indicate the memory controller ID of the flash device to erase. Value 0x0 (default) indicates the internal flash. Value 0x01 indicates the external QSPI0 on devices that support QSPI0.

ROM_RESET_CMD

The target is reset.

Table 21. Reset command fields

Field	Description
`m_checksum`	Simple checksum, which comes to 0x5a when all other fields are zeros.
`m_tag`	0x08 or `ROM_RESET_CMD`
`m_flags`	0
`m_address`	0
`m_count`	0
`m_data`	0

ROM_MEM_ENABLE_CMD

Enable (configure) the external memory. The m_flags field bits 11:8 indicate the memory controller ID. The `m_address` field contains the address in the RAM where the config block was previously written, and the `m_count` field contains the size of the config block. The format of the configuration block depends on the memory space.

Note that this command does not actually write the config block to the external media, but simply uses the config block to configure the interface.

Table 22. Memory enable command fields

Field	Description
`m_checksum`	Simple checksum, which comes to 0x5a when all other fields are zeros.
`m_tag`	0x09 or `ROM_MEM_ENABLE_CMD`
`m_flags`	See the Memory controller ID table.
`m_address`	Address of the existing config block in the RAM.
`m_count`	Size of the config block.
`m_data`	0

Table 23. Memory enable command flag bits

Bits	Value	Description
11:8	0x01 `kLdrMemoryCtrl_QSPI0`	Memory controller ID. Value 0x01 indicates external QSPI0 on devices that support QSPI0. No other values are supported.

ROM_PROG_CMD

Write to the program-once persistent bits. Bits 11:8 of the `m_flags` field contain the memory space ID (only `kLdrMemorySpace_IFR0` is supported). Bit 1 of the m_flags field indicates a 8-byte write (if set) or 4-byte write (if clear). The `m_address` field contains the IFR index. The `m_count` field contains the first four bytes to be programmed. The `m_data` field (optionally) contains the next 4 bytes to be written (if bit 1 of the flags field is set).

Table 24. Program command fields

Field	Description
`m_checksum`	Simple checksum, which comes to 0x5a when all other fields are zeros.
`m_tag`	0x0a or `ROM_PROG_CMD`
`m_flags`	See the Program command flags bits table.
`m_address`	IFR index.
`m_count`	First four bytes to be programmed.
`m_data`	Second four bytes to be programmed (if 1 of `m_flags` is set.

Table 25. Program command flags bits

Bit(s)	Flag/Value	Description
1	`ROM_PROG_8BYTE_MASK`	If set, write eight bytes, otherwise write four bytes.
11:8	0x04 `kLdrMemorySpace_IFR0`	Memory space. Value 0x04 indicates internal IFR flash. No other values are supported.

6.6 File format versions

The versions are listed as Major.minor.

Table 26. File format versions

Version	Description
1.3	Support for Kinetis-specific features.

7 Revision history

The following table contains a history of changes made to this user's guide.

Table 27. Revision history

Revision number	Date	Substantive changes
0	09/2015	Initial release
1	04/2016	Kinetis Bootloader v2.0 release
2	05/2018	MCU Bootloader v2.5.0 release
3	09/2018	MCU Bootloader v2.6.0 release