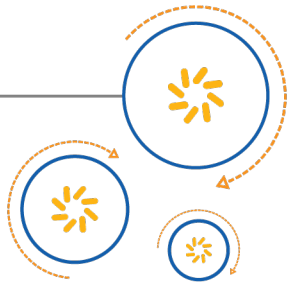
Qualcomm Technologies International, Ltd.



Qualcomm Kalimba DSP Assembler

User Guide

80-CT425-1 Rev. AH

October 24, 2017

Revision history

Revision	Date	Description
1	APR 2011	Initial release. Alternative document number CS-00212259-UG
2	JAN 2012	Updated to latest CSR™ style
3	APR 2014	Updated to new CSR style
4	AUG 2014	
5	MAR 2015	Added documentation of `.VAR8`, `.VAR16`, `.MINIM` and `.MAXIM` directives and other differences for 32-bit Kalimba targets.
6	AUG 2016	Documented the internal representation of floating point constants and large integer constants, and how this representation differs for 32-bit targets as of KCC 46/kalasm3r80. Updated document references and added guidelines on how to link programs directly using kld, the binutils linker Updated to conform to QTI standards.
7	DEC 2016	Repaired Broken Cross references. No change to Technical Content.
AH	AUG 2017	Added to the System Content System. Update DRN to use Agile number. No changes were made to the technical content.

Contents

Tables

Figures

1 Introduction to KalAsm 3

KalAsm 3 is a suite of tools used to produce applications for the Qualcomm® Kalimba™ digital signal processor (DSP) core. QTIL recommends the following documents as a source of further information:

- *Qualcomm Kalimba Architecture 3 DSP User Guide*: Describes the Kalimba DSP programming model and instruction set for the CSR8670 target.
- *Qualcomm Kalimba Architecture 4 DSP User Guide*: Describes the Kalimba DSP programming model and instruction set for 32-bit Kalimba targets such as CSRA68100.
- *Qualcomm Kalimba Architecture 5 DSP User Guide*: Describes the Kalimba DSP programming model and instruction set for the CSR8675 target.
- *Qualcomm Kalimba C Compiler User Guide*: Describes how to use the C compiler and the kcc driver tool. This guide is useful if writing an application containing a mixture of C code and assembly code.

The KalAsm 3 suite is a heavily modified version of the GNU Binary Utilities, also known as binutils.

2 Overview of KalAsm 3

The KalAsm 3 suite consists of several tools that you can use together to build assembler source files into an application image that can be loaded and run on the Kalimba DSP. This section describes some of the tools and how they fit together to build a complete DSP application.

2.1 Kalsam 3Tools

Table 2-1 KalAsm 3 tools

Name	Purpose
kas	Preprocessor and assembler: takes a single assembler source file and produces one object file.
kar	Library archiver: combines multiple object files into a single library archive.
klink	Linker wrapper: interfaces with the kld linker to combine libraries and object files into a single Executable and Linkable Format (ELF) image. It is responsible for arranging code and data in memory, which it does according to rules read from a linker control file. The format of this linker control file is much simpler than a binutils format linkscript, but offers less control. For the latest Kalimba Architecture 4 targets such as CSRA68100, it is necessary to link using the kld utility directly.
kld	The **binutils** linker program. As well as resolving symbols and placing code and data, the linker also performs simple optimisations (called *relaxation*) to reduce the size of certain DSP instructions to reduce the overall memory usage. kld is controlled using a linkscript, which offers complete control over placement of code, data and metadata in the final image. However, unlike klink, it does not arrange code and data on behalf of the user.
elf2kap	Converts ELF binary images into the KAP format suitable for loading onto Qualcomm® BlueCore™ devices.

2.2 KalAsm 3 build process overview

The tools can be manually invoked at a command line, or with a custom GNU make script. This section describes the intended flow. Figure 2-1 shows a sample build process overview for the assembly code.

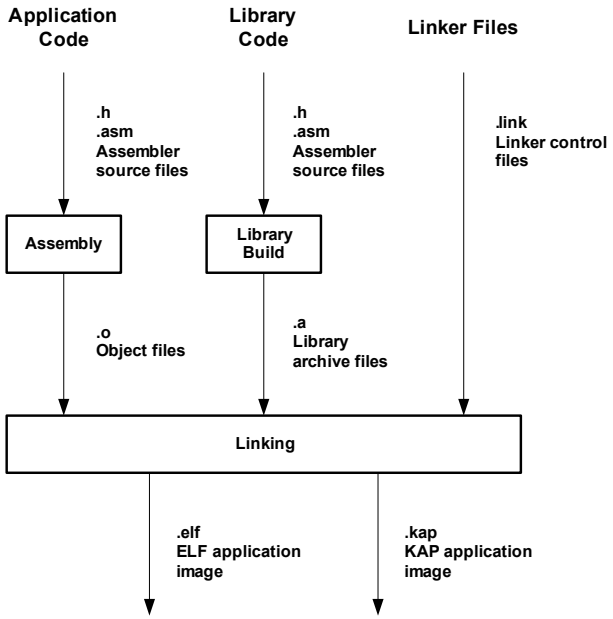


Figure 2-1 Build process overview

2.2.1 KalAsm 3 build process assembly

Figure 2-2 shows how source files are assembled into object files.

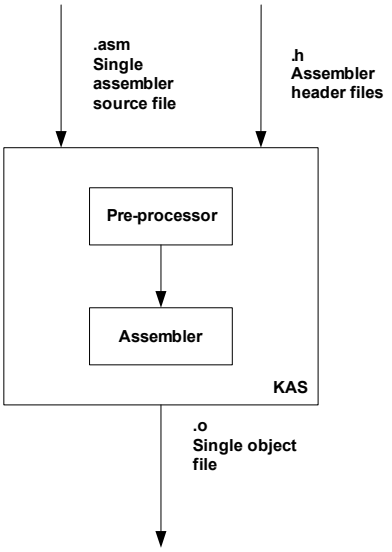


Figure 2-2 Assembly

The Kalimba assembler, **kas**, runs once for each assembler source, producing a corresponding object file for each one.

2.2.2 KalAsm 3 build process library creation

Figure 2-3 shows how library archives are created.

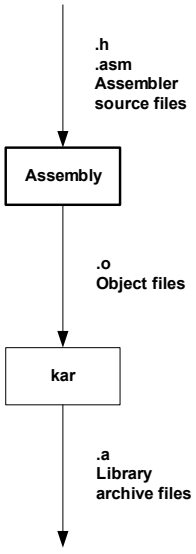


Figure 2-3 Library build

The archiver, **kar**, packages a group of object files into a single library archive. The process is repeated for each library.

2.2.3 KalAsm 3 build process link using klink, the linker wrapper

Figure 2-4 shows the linking process using **klink**.

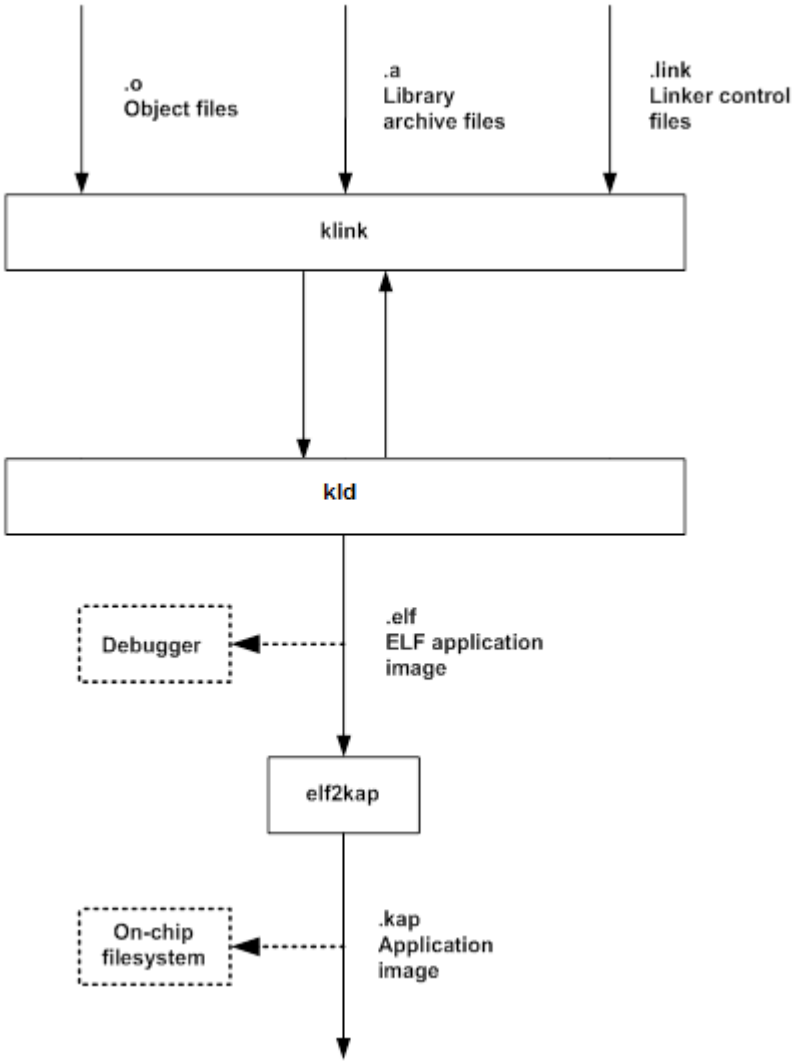


Figure 2-4 Linking using klink

The linker wrapper, **klink**, invokes **kld** to combine code and data from object files and library archives into an ELF application image, according to rules supplied in one or more linker control files. It handles the specific placement of code and data into memory regions. For targets which support KAP-based image loading, such as CSR8670 and CSR8675, the ELF image is translated to `.kap` format by the **elf2kap** utility. See Linking in KalAsm 3 for more information about using **klink**.

2.2.4 KalAsm 3 build process linking directly using kld

On 32-bit targets such as CSRA68100, klink cannot be used, as it is not flexible enough to allow all contents of non-volatile memory, including ROM, to be placed. Therefore, **kld** must be invoked directly. Figure 2-5 shows the linking process if linking directly using **kld**:

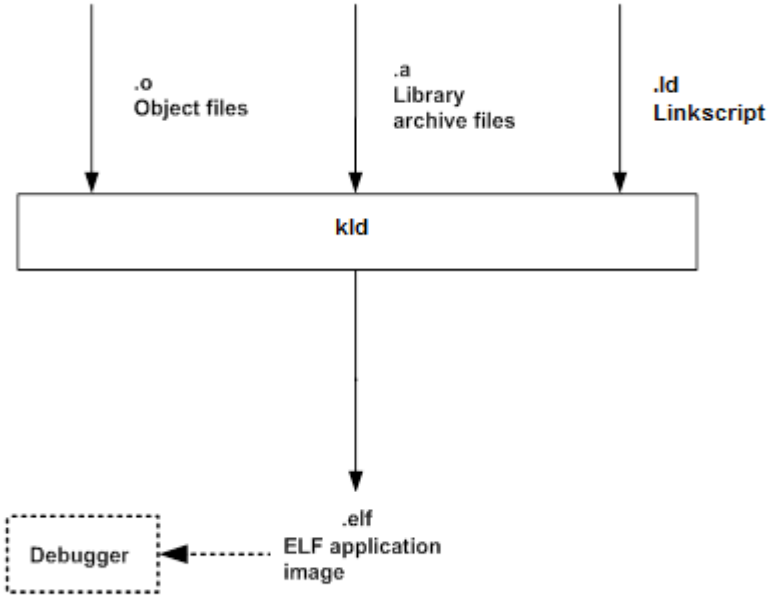


Figure 2-5 Linking directly using kld

The software development kit for your target should include an appropriate linkscript for use when linking.

See Linking directly using kld - overview for more information on using **kld**.

2.3 KalAsm 3 files and formats

File Content	Type	Description
Assembler sources	`.asm, .s`	Contain program code written in Kalimba DSP assembly language. As well as DSP instructions, these files include: ■ Assembler Directives that control the structure of the code. Examples include dividing the source into modules and allocating space for variables. See KalAsm 3 numbers and expressions. ■ Assembler Expressions: mathematical and logical expressions that are evaluated by the assembler itself to produce values to use in DSP instructions and variables. See KalAsm 3 keywords and reserved words. ■ Preprocessor commands to direct the preprocessor to modify the source before assembly, for example by including other files or conditionally excluding parts of the file. The KalAsm 3 tools use a C-style preprocessor. See KalAsm 3 conversions and encoding. ■ Comments: sections of free-form text ignored by the assembler that can be used for adding documentation and other annotations to the code.
Headers and data	`.h, .dat`	Assembler sources intended to be included in other sources using the #include preprocessor command. See KalAsm 3 conversions and encoding.
Object files	`.o`	Assembler output files, one for each assembler source file. Object files use standard ELF format, and contain: ■ Partially encoded instructions and data for the DSP ■ The names and offsets of variables and code labels ■ A relocation table to allow the linker to fix up instructions to reflect the final layout of code and data in memory ■ Dwarf 2 debug information for consumption by a debugger (if generation of debugging information is enabled)
Preprocessor output	`.pp`	This contains the result of pre-processing the assembler source.
Library archives	`.a`	This is a collection of object files, combined into a single file for convenience. Uses the standard Unix archive format with GNU extensions.
Private library archives	`.pa`	This is a library archive that has been processed to remove source information and disable disassembly.
klink linker control files	`.link`	This describes the available memory regions to the klink linker wrapper, and defines segments to help control how code and data are placed within these regions. See Linking in KalAsm 3 .
kld (binutils format) linkscripts	`.ld`	This allows complete control over the placement of code, data and metadata in both RAM and non-volatile memory. However, the linkscript format is more complex than klink linker control files. See Linking directly using kld - overview.
Map files	`.kmap`	Report of the result of linking, including how code and data were laid out in memory, disassembly of all code and a summary of memory usage and fragmentation.

File Content	Type	Description
Application	`.elf`	The linked application, in the form of a standard ELF image. Includes: ■ Complete instructions and data encoded for the DSP ■ Names and final addresses of variables and code labels ■ Dwarf 2 debug information for consumption by a debugger (if generation of debugging information is enabled)
Loadable Image	`.kap`	A loadable version of the application supported on the csr8670 and csr8675 targets. The KAP format is a simple binary script which is processed by the DSP application loader in the BlueCore firmware, or by a bootstrap program. The KAP file is stored in a text format on the PC, but is converted to binary before being stored on the target device. In particular, this means that the size of the KAP file is not indicative of the space that will be required on the device.

3 KalAsm 3 assembler syntax reference

Assembler sources are plain text files consisting of preprocessor commands, comments, assembler directives, labels, and instructions. Assembler code is structured as a number of modules, delimited by the `.MODULE` and `.ENDMODULE` directives. Each module can contain instructions, code labels, variables (using `.VAR`), and constant definitions (using `.CONST`). In addition, global variables and constants may be declared outside a module. A source file may contain multiple modules.

Preprocessor

The preprocessor modifies the input file, according to the preprocessor commands it contains, before it is assembled. The assembler uses a complete C-language preprocessor, so the syntax will be familiar to C programmers. Use it to:

- Include other files (such as common constant definitions)
- Conditionally include or exclude portions of the file
- Define simple substitutions and macros.

Section KalAsm 3 preprocessor directives describes the preprocessor.

Preprocessor directives begin with the `#` character.

Comments

Comments enable you to include portions of free-form text in the assembler source to provide explanation, documentation, or other annotations. The assembler accepts C and C++ style comments.

Table 3-1 Example comments

Example	Details

`/* This is a`
`block comment`
`over multiple lines`
`*/`
	Block comment: these comments may span one or more lines, and they may occur anywhere that whitespace is valid.

`// Single line comment`	Line comment: from the double-slash to the end of the line is treated as a comment.

Modules

Use modules to organize code and data. They operate in two ways:

- A module specifies that a particular block of instructions form a contiguous sub-program, and indicates a segment that controls where in program memory those instructions are stored.
- A module provides a namespace for labels and variables. See more information in Section KalAsm 3 constants, labels, variables, and blocks.

See Section KalAsm 3 .MODULE and .ENDMODULE assembler directives for more information on the `.MODULE` and `.ENDMODULE` directives.

Variables

Define variables with the `.VAR` directive. Use them to provide constant tables, scratch memory, buffers, counters, etc. A variable is a contiguous block of one or more words of data memory with a name. Variables are always allocated statically when the application starts; their size and location in memory is fixed at application build time. See section KalAsm 3 .VAR assembler directive for more information about the `.VAR` directive.

Constants

Define numerical constants with the `.CONST` directive. Constants are simply a convenience for programmers, and have no representation in the running program. Both floating point and integer constants are permitted. See for more information about the `.CONST` directive. See for information about the use of constants in numerical expressions.

Labels

Labels are symbolic names for particular program locations. They consist of a symbol name followed by a colon. They are used to name and reference sub-programs, and to control program flow within sub-programs. See for more information about labels.

Instructions

Instructions follow the syntax described in the *Kalimba Architecture 3 DSP User Guide,* or the equivalent for the target being used. In addition to the syntax described there, several shortcuts allow a more natural programming style. These are translated by the assembler into equivalent instructions. For example, to copy the contents of one register to another you can write `r0 = r1`. This is treated identically to `r0 = r1 + Null;` by the assembler.

Appendix KalAsm 3 instruction shortcut syntax describes permitted shortcuts.

Numbers and expressions

You can use a numerical expression in most places where a numerical value is permitted, such as in instructions, variable initializers, and constant definitions. These expressions are evaluated by the assembler to a floating point or integral value and, in the case of instructions and variables, further converted to a suitable format for the DSP. See for full details of number formats and expressions.

Preprocessor commands may also contain numerical expressions. However, they are evaluated by the preprocessor and not the assembler and so obey a slightly different set of rules.

3.1 KalAsm 3 constants, labels, variables, and blocks

The names of constants, labels, variables and blocks are symbols. Symbol names obey these rules:

- The first character is a letter (`A-Z` and `a-z`), underscore (_) or dollar (`$`)
- The following characters, if any, may be any mixture of letters, underscores, dots (`.`), dollar ($), and numbers (`0-9`)
- Symbols are case-sensitive, i.e. the symbol `$foo` is distinct from the symbols `$FOO` and `$Foo`

As a special case, the symbol consisting of a single dot (`.`) is an automatically defined label referring to the address of the current instruction. For example, the instruction `jump .;` is a branch-to-self; effectively an infinite loop.

Symbols share a single namespace. For example, it is not valid to have a constant and a label with the same name, nor constant and variable, nor variable and label.

Global and local symbols

Most symbols begin with a dollar character (`$`). This means that they are considered global: they can be referenced anywhere in the program under the same name. Without a dollar, the symbol is local. By convention, local symbols are intended for reference from within the module that defines them. However, they can also be referenced from outside that module by prefixing them with the name of the module followed by dot. For example, a label named `MY_LABEL` defined within a module named `$MY_MODULE` can be referenced outside that module as `$MY_MODULE.MY_LABEL`. Global symbols may be defined anywhere, whereas local symbols must be defined within modules.

Example

```
.VAR/DM $global_one;
.MODULE $my_module;
    .DATASEGMENT DM;
    .CODESEGMENT PM;
    .VAR $global_two;
    .VAR local_one;
    r0 = M[$global_one];
    r1 = M[$global_two];
    r2 = M[local_one];
    r3 = M[$my_module.local_one];  // Equivalent to the previous line.
.ENDMODULE;
```

3.2 KalAsm 3 module names

Module names are not symbols, but they follow similar rules:

- Must be valid symbol names as described in Section KalAsm 3 constants, labels, variables, and blocks
- Must begin with a dollar character (`$`)

Module names do not share a namespace with the other symbols described in this section. Therefore, you can create a module with the same name as a label or variable, although Qualcomm® does not recommend this.

3.3 KalAsm 3 segment names

As with module names, segment names are not symbols, but they do follow the same syntactic rules. They do not share a namespace with either symbols or modules, so in theory a segment may share the same name with a module, variable, block, constant, or label. Segments do not conventionally begin with a dollar character, and there is no concept of local or global segments.

See and Linking algorithm for details on using segments.

3.4 KalAsm 3 keywords and reserved words

Assembler keywords and reserved words may not be used as symbol, module, or segment names. Examples of keywords are register names, condition codes, and words used in instructions (e.g `IF`, `MAX`, `ASHIFT`). KalAsm 3 instruction shortcut syntax gives a complete list.

NOTE Although directive names such as `.VAR` are assembler keywords, they do not clash with possible symbol names because of the leading dot. Similarly, many symbols begin with dollar (`$`), which distinguishes them from any keyword. Unlike symbols, however, keywords are not case-sensitive. So, for example, `MAX`, `Max` and `max` are all keywords and so cannot be used as symbol names.

3.5 KalAsm 3 .MODULE and .ENDMODULE assembler directives

Use these directives to start and end modules. Modules are the basic unit of DSP assembly code. They provide a contiguous sub-program of instructions and a namespace for symbols.

Syntax

```
.MODULE moduleName ;
.ENDMODULE ;
```

Where `moduleName` is a valid module name that is unique in your program, following the rules described in KalAsm 3 constants, labels, variables, and blocks. Each `.MODULE` directive must be matched with a corresponding `.ENDMODULE`. Modules may not be nested.

A module does not in itself provide a code label, so if the module contains DSP instructions, it must begin with a label to be useful. One possible convention is:

```
.MODULE $M.incrementR1;
    .CODESEGMENT PM;
    $incrementR1:
        r1 = r1 + 1;
        rts;
.ENDMODULE;
```

Here, the module is named `$M.incrementR1`, but the code it contains is called by the label `$incrementR1`. An easy alternative is to give the module and label the same name.

3.6 KalAsm 3 .CODESEGMENT and .DATASEGMENT assembler directives

These directives control where the code and data within a particular module are placed in DSP memory.

Syntax

```
.CODESEGMENT segmentName ;
.DATASEGMENT segmentName ;
```

Where `segmentName` is the name of a segment. KalAsm 3 segment names gives the rules for naming segments. Segment directives are only valid within modules, and most modules will contain either one or both of these directives. Each module can have, at most, one of each of these directives. These directives conventionally appear at the start of each module because they must occur before any labels or instructions in the case of `.CODESEGMENT`, or before any variables or blocks in the case of `.DATASEGMENT`.

Any module containing DSP instructions must have a `.CODESEGMENT` directive to specify where the code in that module is to be placed in memory. If a module contains no DSP instructions, the `.CODESEGMENT` directive is not required. Variables defined with `.VAR <name>` use the default data segment for the module, which is defined using the `.DATASEGMENT` directive.

Segment names correspond to the segments defined in the linker control files. See KalAsm 3 defaults control files for more details on linker control files.

3.7 KalAsm 3 .MINIM and .MAXIM assembler directives

These directives are used to choose the instruction set encoding for all code defined in the current module.

NOTE The `.MINIM` directive is only accepted when building for Kalimba targets that support the MiniMode instruction set. The default for all Kalimba targets is the MaxiMode (full 32-bit instruction word) encoding.

3.8 KalAsm 3 .VAR assembler directive

Use this directive to declare and optionally initialize data variables. Variables may be a single memory location, or an array of contiguous locations.

Syntax

```
.VAR[/segmentName] variableName [[[length]]] [= initialiser] ;
```

Most elements of the syntax are optional, with defaults.

Table 3-2 Example VAR syntax

Example	Description
`.VAR example1;`	A single, uninitialized memory location, named example1, located in the default segment specified with `.DATASEGMENT`.
`.VAR/DM1 example2;`	Located in segment DM1, rather than the default.
`.VAR arrayExample1 [4];`	An uninitialized array of 4 data memory locations.
`.VAR arrayExample2 [3] = 1, 2, 3;` `.VAR arrayExample3 [] = 1, 2, 3;`	Two equivalent initialized array variables; the length field is optional if a complete initializer is given.
`.VAR arrayExample4 [4] = 1 ...;` `.VAR arrayExample5 [4] = 1, 2 ...;`	`...` can be used to fill the last part of a variable with a repeated value. The initializers in this example are equivalent to `1, 1, 1, 1` and `1, 2, 2, 2` respectively.
`.VAR fracExample1 [] = 1.0, 0.5, -1.0;` `.VAR fracExample2 [] = 0.5, 1, 10;`	Variables can also be initialized with fractional values, or a mixture of fractionals and integers. **NOTE** Integer literals such as `1` are always encoded as integers, and floating point literals (e.g. `1.0`) are encoded as fractionals. Number formats, conversions, and encodings are explained in Section KalAsm 3 numbers and expressions.

Variable names follow the rules described in KalAsm 3 constants, labels, variables, and blocks. Variables may be declared either within or outside of a module. Variables declared outside of a module must have names beginning with a dollar character (`$`) to indicate that they have global scope, and must explicitly specify a target segment.

Separate data variables within a single source file or module are not usually allocated contiguous areas of memory, nor are they necessarily stored in the order they appear in the source file. However, you can use the `.BLOCK` directive (see KalAsm 3 .BLOCK assembler directive) to arrange this.

You can use a more complex expression anywhere that a number is used in the examples in Table 3-2. See KalAsm 3 numbers and expressions.

KalAsm 3 .VAR8, .VAR16 assembler directives

If developing for a 32-bit Kalimba target, which supports octet-addressable data, it is possible to use the `.VAR8` and `.VAR16` directives to define and initialize 8-bit and 16-bit variables, respectively.

These directives allow the same syntax as a normal `.VAR` directive, as shown in Table 3-2. This includes the ability to declare and initialize arrays, and to use expressions in place of literal values.

NOTE Initialization values that will not fit into an 8-bit or 16-bit type will be truncated and a warning displayed. This means that fractional initializers will not work.

String initializers

Variables may also be initialized using string literals, using this syntax:

```
string("stringliteral" [/option] [/option] … )
```

Where `stringliteral` is a string of characters and backslash-escapes, and each `option` is as shown in Table 3-3.

Table 3-3 String options

Option	Meaning
`pack16`	Pack two characters in the least significant 16 bits of each word (the default is a single character in the least significant 8 bits of word). The characters are packed in the most significant octet first. For example, `string("abcd"/pack16)` is stored as `0x6162 0x6364`.
`pack24`	Pack three characters in each word. The characters are packed in the most significant octet first.
`Nonnul`	Do not terminate the string with a null character. By default strings are null terminated.

String literals follow similar escaping rules to the C language.

NOTE When developing for a 32-bit Kalimba target with octet-addressable data, there is no need to select a packing option. A `VAR8` array supplied with a `string("literal")` initializer will cause characters to be packed optimally.

3.9 KalAsm 3 .BLOCK assembler directive

You can use the `.BLOCK` directive to group multiple variables so that they are contiguous and in a particular order in memory.

Syntax

```
.BLOCK[/segmentName] [blockName] ;
    /* One or more .VAR directives. */
.ENDBLOCK ;
```

You can optionally name blocks, following the same rules as for variables. As with variables, blocks may either specify a particular segment, or may use the default specified for the containing module if they are declared within a module. Variables declared within a block may not specify an explicit segment; they are placed according to the block's segment.

Variables in a block may be initialized, uninitialized or, a mixture of both.

NOTE A block is not a namespace in the same way that a module is. If a block is defined outside a module, then the variables it contains must begin with a dollar character to indicate that the names are global. Blocks that are inside modules may contain variables that have module-local scope (i.e. do not begin with a dollar); those variables have a fully qualified name based on the name of the module, not on the name of the block.

You can place blocks in a section appropriate for circular buffers. However, because the variables in a block are contiguous in memory, this only guarantees the correct alignment for the first variable in the block.

Example

```
.BLOCK/DM1 $exampleBlock;
    .VAR $first;
    .VAR $second [] = 4, 3, 2, 1;
.ENDBLOCK;
```

3.10 KalAsm 3 .CONST assembler directive

The `.CONST` directive enables you to define useful constants to enhance readability of code.

Syntax

```
.CONST constName expression ;
```

You can define constants either in or outside modules. In either case, the `constName` must follow the rules specified in KalAsm 3 constants, labels, variables, and blocks. `expression` is a numeric literal or another valID Expression as described in KalAsm 3 numbers and expressions. It must be possible to evaluate `expression` to a numeric value at the point the constant is defined. In particular, it may not contain references to constants defined later or defined in a different source file, and it may not contain references to symbols such as variables and labels that are resolved by the linker rather than the assembler. This means that you cannot use `.CONST` to create aliases for symbols.

Conventionally, constants are defined before they are used, and are shared between source files by defining them in header files that are included using the `#include` preprocessor command. To support this, you can define the same constant multiple times, either in the same or different source files, so long as the value is the same every time.

Several module-local constants, i.e. those defined within a module and without a leading dollar character, may have the same name but different values, so long as they are defined in different modules. This is because their fully qualified names, which include the name of the containing module, are different.

This way of sharing constants using header files and `#include` is encouraged as a useful convention, but it is not strictly necessary in many cases. Constants are symbols similar to code labels and variables, so they can be referenced from source files that have not defined them or be referenced before their definition. However, because they are then evaluated by the linker rather than the assembler, the evaluation rules are slightly different in this case.

Constants retain the full precision of the internal representation including whether the value they hold is a floating point or integer value. Floating point values are converted to fractional values at the point they are used to provide a value for the DSP: e.g. for a DSP instruction, or in a variable initializer.

Example

```
.CONST $PI 3.1415927;
.CONST $EXAMPLE_ADDRESS 0xff8000;
.CONST SAMPLE_COUNT 1024;
.CONST $RADIUS 10;
.CONST $AREA ($RADIUS ** 2) * $PI;
```

Constants fill a similar role to preprocessor macros set using `#define`, but their behavior is quite different. You must define macros in each source file that uses them, or in a common header file included with `#include`, and they may have different definitions in different source files. Macro

names are replaced during pre-processing, and their values are not recorded in the ELF image. You can define constants as a single source file and reference them from any other source file in the application (including in libraries). If constants with the same name are defined in more than one source file (or included into more than one source file), then the linker checks that they have the same value, and reports an error otherwise. Finally, the names and values of constants defined in the application are stored in the ELF image and reported in the KMAP file.

Internal representation of .CONST values

Values specified by the `.CONST` directive can be a normal integer value, a floating point value, or a value larger than the normal integer representation allows. Floating point or large literals are converted into a special representation that is understood internally, and by the kmapper tool.

If viewing `.CONST` values as entries in the `ABS` section via kobjdump, users may occasionally encounter these special representations. For completeness, this section describes the format.

On 24-bit targets, and for 32-bit targets prior to KCC 46/KalAsm 3 v80, the following special symbol values apply:

```
#define SYM_FLOAT_MAGIC   (1<<27)   /* 0x08000000 */
#define SYM_INT64_MAGIC   (1<<28)   /* 0x10000000 */
```

On 32-bit targets using KCC 46/kalasm3 v80 and later, the special symbol values were changed to minimize the risk of clashing with genuine addresses in the memory map:

```
#define SYM_FLOAT_MAGIC_K32 (0x0ffffffe)
#define SYM_INT64_MAGIC_K32 (0x0fffffff)
```

A `.CONST` floating point symbol `foo` will take the appropriate magic value listed above. The actual floating point value can be deduced using the following symbols:

`foo?exp` exponent part

`foo?mant_lo` least significant word of mantissa bits

`foo?mant_hi` most significant word of mantissa bits

A `.CONST` integer symbol `bar`, which does not fit into a 24-bit range (if targeting a 24-bit Kalimba), or which does not fit into a 32-bit signed representation (if targeting a 32-bit Kalimba), takes the appropriate magic value listed above. The true value can be deduced using the following symbols:

`bar?int64_lo` least significant word of large integer constant

`bar?int64_hi` most significant word of large integer constant

Assembly code does not need to be aware of the above conventions, since the linker is aware of the format and bare symbols work in the expected way. This information is only presented in case it is necessary for third party tools to be able to parse kobjdump symbol table output.

4 KalAsm 3 numbers and expressions

In most places in assembler source where a simple numerical value is required, you can use an expression instead. For example, use them in constant definitions and instruction immediates, and to size and initialize variables:

```
.CONST $ONE 10 - 9;
.VAR $VALUES[$ONE * 2] = 1 + 2, sin(8) / 2;
r0 = r1 + log(4 - 2);
```

These expressions are evaluated by the assembler when the application is built and therefore are not subject to the limitations of the DSP instruction set. For the same reason, they cannot include references to values that do not exist until runtime, such as registers and memory locations.

Intermediate values in expressions are either integer or floating point. Integer values are represented internally as signed 64 bit two's complement integers. Floating point values are represented using the C language double type, with at least 53 bits of mantissa. Unless otherwise described in this document, the behavior on overflow is not defined.

Some of the operations described in this section are only defined for either integers or floating point values, or for particular combinations. Some implicitly convert between value types.

4.1 KalAsm 3 numeric literals

Numbers can be written in several formats.

Table 4-1 Number formats

Examples	Description
`0, 1, 123, 654321`	Decimal integers.
`0.5, 123.456, 1.5e6, 123.0e-4`	Decimal floating point numbers.
`0x0, 0xfaff, 0x123abc`	Hexadecimal integers.
`0x0.1, 0xab.12`	Hexadecimal floating point. Exponents are not permitted.
`0b0, 0b1010`	Binary integers.
`0b0.1, 0b101.01`	Binary floating point.

- Underscores: can be add between any digits in numeric literals to add clarity. For example `0b1001_1001,100_000`.
- Octal format is not permitted. Decimal numbers may not have leading zeros, to avoid confusion with the C language's octal format.

- Integers, and hex or binary floating point literals that cannot be exactly represented in the internal representation cause the assembler to report an error.
- Decimal floating point literals only cause an error if they overflow or underflow to zero in the internal representation.

4.2 KalAsm 3 symbols

Constants that have been defined with `.CONST` in an earlier line in the same file, or in a file included using the preprocessor on an earlier line, may be used directly in expressions without limitations. They evaluate to the value assigned to the constant. For example:

```
.CONST $DIAMETER 5.0;
.CONST $RADIUS $DIAMETER / 2;
.CONST $PI 3.1415927;
.CONST $AREA ($RADIUS ** 2) * $PI;
```

Labels and variable names may be used in expressions. Their value is the final address of the code or data referenced. However, you can only use expressions using labels and variables that can be trivially simplified to a symbol plus or minus a value. For example:

```
.VAR $my_variable;
r0 = $my_variable; // Valid
r1 = $my_variable + 1; // Valid
r2 = 10 + ($my_variable - 5) - 4; // Valid
.CONST $offset 1;
r3 = $my_variable + $offset; // Valid
r4 = 1 - $my_variable; // Invalid
r5 = $my_variable * 2; // Invalid
```

The same restrictions apply to any values that cannot be calculated until variables and code are located in memory. In particular, `.CONST` constants that are referenced before they are defined, or in a different file to the file that defines them, are subject to these restrictions.

One result of these restrictions is that you generally cannot write an expression that calculates the difference between the addresses of two labels or variables. As an exception to this rule, you can calculate the difference between the addresses of two variables that are both in the same `.BLOCK`, if the block is defined in the same source file as the expression, or in an included source file:

```
.VAR $regular_variable_one;
.VAR $regular_variable_two;
.BLOCK $my_block;
  .VAR $block_variable_one;
  .VAR $block_variable_two;
.ENDBLOCK;
r0 = $block_variable_one - $block_variable_two;  // Valid
r0 = $regular_variable_one - $regular_variable_two;  // Invalid
```

4.2.1 KalAsm 3 arithmetic operators

Table 4-2 Arithmetic operators

Examples	Notes
`a + b` `a – b`	Addition and subtraction.
`a * b`	Multiplication.
`a / b` `a % b`	Division and remainder. Integer by integer division rounds towards zero (as is common in C language implementations). Remainder is defined such that `A % B == A - B * (A / B)` Division or remainder by zero causes an error. Remainder only operates on integers.
`-a` `+a`	Negation and unary plus. Unary plus performs no operation and is allowed for adding clarity to expressions.
`a ** b`	Exponentiation, i.e. raise a to the power b. Integers may not be raised to negative integer powers. For floating point values, this operator is implemented with the C language standard library `pow` function.

- Operations on two integer operands yield an integer result.
- Operations involving a floating point operand first convert any integer operands to floating point.

4.3 KalAsm 3 relational operators

Table 4-3 Relational operators

Examples	Notes
`a < b`	One if `a` is strictly less than `b`, zero otherwise.
`a <= b`	One if `a` is less than or equal to `b`, zero otherwise.
`a > b`	One if `a` is strictly greater than `b`, zero otherwise.
`a >= b`	One if `a` is greater than or equal to `b`, zero otherwise.
`a == b`	One if `a` is exactly equal to `b`, zero otherwise.
`a != b`	One if `a` is not equal to `b`, zero otherwise.

The result of these operations is always an integer one or zero.

- Comparing an integer with a floating point number is an error.

4.4 KalAsm 3 logical operators

Table 4-4 Logical operators

Example	Notes
`a && b`	One if `a` and b are both non-zero, zero otherwise.
`a || b`	One if either one of `a` or `b` are non-zero, zero otherwise.
`! a`	One if `a` is exactly zero, zero otherwise.

The result of these operations is always an integer one or zero.

These operators only operate on integer operands.

`&&` and `||` are implemented as shortcut operators, i.e. the left-hand operand is evaluated first and if it determines the result of the operation, the right-hand operand is not evaluated. In particular, this means that the right-hand operand may contain undefined symbols or evaluation errors such as division by zero.

4.5 KalAsm 3 bitwise operators

Table 4-5 Bitwise operators

Example	Notes
`a & b`	Bitwise AND.
`a | b`	Bitwise OR.
`a ^ b`	Bitwise XOR.
`~a`	Bitwise NOT.

- These operators only operate on integer operands.

4.6 KalAsm 3 shift operators

Table 4-6 Shift operators

Example	Notes
`a << b` `a >> b`	Logical shift left and right. Shift `a` by `b` bits.
`a <<< b` `a >>> b`	Arithmetic shift left and right. Shift `a` by `b` bits.

- These operators only operate on integer operands.
- Logical shifts shift in zero bits, as does arithmetic left shift.
- Arithmetic right shift duplicates the most significant bit (the sign bit).

- Negative shifts reverse the shift direction. For example `x << -1` is equivalent to `x >> 1` and `x <<< -1` is equivalent to `x >>> 1`.
- Shifts by 64 bits or more evaluate to zero in the case of logical shifts and arithmetic left shift, but fill with sign bits in the case of arithmetic right shift.
- Shifts do not saturate in any case.

4.7 KalAsm 3 conversion operators

Example	Notes
`float(a)`	Convert integer a into a floating point value. It is an error if the integer is not representable as a floating point value. The operand may be a floating point value already, in which case the same value is returned unchanged.
`ceil(a)`	Return the smallest integral value not less than `a`.
`floor(a)`	Return the largest integral value not greater than `a`.
`trunc(a)`	Return a rounded to the nearest integer not larger in absolute value.
`round(a)`	Return a rounded to the nearest integer, rounding halfway cases away from zero.

4.8 KalAsm 3 Mathematical operators

Example	Notes
`sin(a)` `cos(a)`	Sine and cosine of `a`. The argument is taken to be in radians.
`sqrt(a)`	The non-negative square root of `a`. An error is reported if `a` is negative.

`log(a)`
`log10(a)`
`log2(a)`
	Logarithms base `e`, `10` and `2` respectively. An error is reported if `a` is less than or equal to zero.

- The argument may be an integer or floating point value. Integer arguments are first converted to floating point.
- These operations always return a floating point value.

4.9 KalAsm 3 conditional operator

Example	Notes
`a ? b : c` `if a then b else c`	If `a` is non-zero, the result is `b`, otherwise `c`. This operator shortcuts; the unused argument is not evaluated. This means that the unused operand may contain undefined symbols or evaluation errors such as division by zero. The condition a must evaluate to an integer.

4.10 KalAsm 3 miscellaneous operators

Example	Notes
`length(symbol)`	The length of the variable or block named by symbol. The argument must be a valid symbol identifier. The result is an integer. If symbol has not been defined at an earlier line in the same file, the length function has similar restrictions to those described in KalAsm 3 symbols for referencing variables and labels. The length returned is expressed in terms of minimum addressable data units for the given target. For a 24-bit Kalimba processor, this means that the length of a variable is measured in terms of 24-bit data words. For a 32-bit Kalimba processor, the length is instead measured in octets (8-bit units).
`min(a, b)` `max(a, b)`	Respectively the minimum and maximum of `a` and `b`. These are simple shortcuts for `x < y ? x : y` and `x > y ? x : y`.
`abs(a)`	The magnitude of `a`.

4.11 KalAsm 3 operator precedence

Precedence and associativity rules match those in the C language. Table 4-7 summarizes the precedence ordering, going from tightest to loosest binding.

Table 4-7 Operator precedence

Operators	Description
`(..)`	Grouping
`f(..)`	Function-like
`**`	Exponentiation
`~ ! -`	Unary
`* / %`	Multiplicative
`+ -`	Additive
`<< >> <<< >>>`	Bit shifts
`> >= < <=`	Relational
`== !=`	Equality and inequality
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`&&`	Logical AND
`||`	Logical OR
`?:`	Conditional

- All operators with two operands associate to the left, except for the exponentiation operator (**) which associates to the right.
- Operators with a single argument and the conditional operator associate to the right.

NOTE As in the C language, parts of the operator precedence can be confusing. For example:

```
foo & bar == baz    means        foo & (bar == baz)
high << 4 + low     means       high << (4 + low)
```

4.12 KalAsm 3 conversions and encoding

Values are kept in high precision internal format until they are used in a variable initializer or an instruction. At that point, they are converted to a form usable by the DSP appropriate to the context. In general, integer values are truncated, giving an error if they cannot be represented in the space available.

On 24-bit Kalimba targets, floating point values are converted to either signed or unsigned 24-bit fractional numbers, depending on context. An error is given if the value overflows a 24-bit fractional. As a special case, 1.0 saturates to `0x7fffff` signed, or `0xffffff` unsigned.

On 32-bit Kalimba targets, floating point values are converted to either signed or unsigned 32-bit fractional numbers, depending on context. An error is given if the value overflows a 32-bit fractional. As a special case, 1.0 saturates to `0x7fffffff` signed, or `0xffffffff` unsigned.

There is an exception to the above encoding rules. In a normal 24-bit or 32-bit multiply instruction with the `(frac)` selector used to denote a fractional multiplication, it is possible to supply a fractional number as an operand to the multiplication. In this case, the 16-bit fractional value will have the least significant bits zero padded to form either a 24-bit or 32-bit fractional, depending on the selected Kalimba target. For example, 0.5 is represented as `0x4000` in the multiply instruction, but is interpreted as `0x400000` on a 24-bit Kalimba target, and `0x40000000` on a 32-bit Kalimba target.

4.13 KalAsm 3 preprocessor directives

This topic provides reference information about the KalAsm 3 preprocessor commands, including syntax and usage examples. Preprocessor command syntax must conform to the following rules:

- The command must be the first non-white space character on its line
- The command cannot be more than one line in length unless the backslash character (`\`) is inserted

NOTE The preprocessor included in KalAsm 3 is a modified version of the C preprocessor distributed with the GNU Compiler Collection (GCC).

Table 4-8 Preprocessor commands

Command/Operator	Description
`#include`	Includes the contents of a file
`#define`	Defines a macro
`defined`	Checks if a symbol has been defined
`#undef`	Removes macro definition
`#if`	Begins an `#if`/`#endif` pair
`#elif`	Sub-divides an `#if`/`#endif` pair

Table 4-8 Preprocessor commands (cont.)

Command/Operator	Description
`#else`	Identifies alternative instructions within a `#if`/`#endif` pair
`#endif`	Ends an `#if`/`#endif` pair
`#ifdef`	Begins an `#if`/`#endif` pair and tests if macro is defined
`#ifndef`	Begins an `#if`/`#endif` pair and tests if macro is defined
`#warning`	Reports a warning message
`#error`	Reports an error message
`#line`	Allows the user to set the filename for errors and messages

The following topics describe the preprocessor commands in detail.

#include directive

#define directive

#undef directive

Conditional assembly

Diagnostics directives

#line directive

4.13.1 #include directive

The `#include` directive directs the preprocessor to insert the text from a file at the command location. The preprocessor searches for the file in the order:

1. The current directory
2. The directories you specify using the `-I` command line option

Syntax

```
#include "fileName"     // include a user file
```

Unlike standard C, include commands are also subject to macro expansion, so the filename may be supplied by a macro. For example:

```
#define HEADER "header.h"
#include HEADER
```

4.13.2 #define directive

The `#define` directive has two functions:

- Defining symbolic constants
- Defining macros

When you define a symbolic constant in your source code, the preprocessor substitutes each occurrence of the constant with the defined text or value.

NOTE Defining a macro has a similar effect to using the find-and-replace feature of a text editor, although it does not perform a replace in between pairs of double quotation marks (″) in literals.

When you define a macro in your source code, the preprocessor replaces all subsequent occurrences of the macro reference with its definition. Therefore, unlike other symbol definitions, it is global without requiring a dollar sign (`$`) prefix.

For multi-statement macro definitions, terminate all the statements or all but the last statement with a backslash (`\`).

NOTE It is good programming practice to end the last macro statement without a semicolon and to use a trailing semicolon at the macro invocation.

When ending a statement macro with a semicolon, ensure that its macro invocation does not add another terminating semicolon.

Arguments can be added to the macro definition by appending a comma separated list of arguments within parentheses after the macro name.

Syntax

```
#define macroSymbol replacementText
#define macroSymbol(arguments) replacementText
```

Where:

- `macroSymbol`: Macro identifying symbol.
- `arguments`: An optional comma-separated list of argument names.
- `replacementText`: Series of instructions or a constant definition to substitute each occurrence of `macroSymbol` in your source code.

Examples

```
#define BUFFER_SIZE 1020
// Defines a constant named BUFFER_SIZE and sets its
// value to 1020.
// The constant is used as:
.VAR $input_buffer[BUFFER_SIZE];
// Becomes identical to:
.VAR $input_buffer[1020];
// Likewise the following definition:
#define REG_SWAP(rA, rB, rDUMMY) rDUMMY = rA; rA = rB; \
rB = rDUMMY
// Is used as:
REG_SWAP (r1, r2, r4);
// And is identical to:
r4 = r1; r1 = r2; r2 = r4;
```

4.13.3 #undef directive

The `#undef` command directs the preprocessor to undefine a macro.

Syntax

```
#undef macroSymbol
```

Where:

- `macroSymbol`: is the macro created with the #define command.

Example

```
#undef BUFFER_SIZE // undefines a macro named BUFFER_SIZE
```

4.13.4 Conditional assembly

A conditional is a directive that enables you to include or exclude a chunk of code or not depending on the evaluation of an arithmetic expression or the existence of a macro definition or combination of both.

The #if and #endif directives

The `#if` directive starts a block source ending in a `#endif` directive. The body of the conditional is only processed if the condition of the `#if` directive evaluates to true.

Syntax

```
#if expression
controlled text
#endif /* expression */
```

Where:

- `expression`: C expression of integer type. See Numerical expressions directives.
- `controlled text`: included text if the expression evaluates to non-zero, otherwise it is skipped.

The #else directive

You can add the `#else` directive to a conditional to provide an alternative text to be used if the condition fails.

Syntax

```
#if expression
controlled text
#else
alternative text
#endif /* expression */
```

Where:

- `expression`: C expression of integer type. See Numerical expressions directives.
- `controlled text`: Included text if the expression evaluates to non-zero, otherwise it is skipped.
- `alternative text`: Included text if the expression evaluates to zero, otherwise it is skipped.

The #elif directive

One common case for conditionals is nested conditionals. The `#elif` directive simplifies the writing of a conditional of the form:

```
#if X == 1
...
#else /* X != 1 */
#if X == 2
...
#else /* X != 2 */
...
#endif /* X != 2 */
#endif /* X != 1 */
Use the #elif directive to rewrite this more simply:
#if X == 1
...
#elif X == 2
...
#else // Where X != 1 and X != 2
...
#endif
```

Syntax

```
#if expression1
expression 1 text
#elif expression2
expression 2 text
#endif // condition
```

Where:

- `expression1`/`expression2`: C expression of integer type. See .
- `expression 1 text`: Included text if the expression 1 evaluates to non-zero, otherwise it is skipped.
- `expression 2 text`: Included text if the expression 1 evaluates to zero and expression 2 evaluates to non-zero, otherwise it is skipped.

The “defined” operator

You can use the special operator `defined` in `#if` and `#elif` expressions to test whether a certain name is defined as a macro.

Syntax

```
defined NAME
defined(NAME)
```

Where:

- NAME is a macro name. Both forms evaluate to 1 if NAME is defined as a macro in the current point in the program, and 0 otherwise.

The #ifdef and #ifndef directives

The macros #ifdef and #ifndef are shortcuts:

- #ifdef MACRO is equivalent to #if defined(MACRO)
- #ifndef MACRO is equivalent to #if defined(MACRO) == 0

Example

```
#if defined(ARCH32) && BITS_PER_UNIT == 16
// include archi1.h at this point if ARCH32 is defined
// and BITS_PER_UNIT is 16
#include "arch1.h"
#elseif BITS_PER_UNIT == 8
// include arch2.h and define constant ARCH_ALTERNATIVE if
// BITS_PER_UNIT is 8
#include "arch2.h"
#define ARCH_ALTERNATIVE 1
#else
#error "Unsupported architecture."
#endif
```

4.13.5 Diagnostics directives

The preprocessor provides directives that you can use to cause a warning or error to be reported.

Syntax

```
#error messageText
#warning messageText
```

Where:

- messageText: Is user-defined text. Long messages can be broken up by escaping new-lines with a backslash character (\) at the end of each line except the last.

The #error directive causes the preprocessor to report the supplied message and exit immediately. The #warning directive reports the supplied message and continues pre-processing.

Example

```
#if BUFFER_SIZE < 1024
    #error "MyError: BUFFER_SIZE must be at least 1024."
```

```
#endif
#ifdef OLD_OPTION
    #warning "MyWarning: OLD_OPTION is deprecated, use NEW_OPTION
instead."
    #define NEW_OPTION
#endif
```

NOTE Neither `#error` nor `#warning` macro-expands its argument. It is conventional to write the argument of these directives as a single string constant, surrounded with double quote marks (`"`), although this is not strictly required.

4.13.6 #line directive

Use the `#line` directive to control the source line reported by software using the preprocessor. This is usually only relevant if the input to the preprocessor is the output of some other program.

Syntax

```
#line linenum
#line linenum filename
```

Where:

- `Linenum`: Is a non-negative decimal integer constant. It specifies the line number for the following line of input. If `filename` is present all the subsequent lines are reported to come from the specified file.

Example

```
// Header comment on bit.asm
#warning "First Warning"
#line 10 "arch.h"
#warning "Second Warning"
#line 22
#warning "Third Warning"
```

Displays:

```
bit.asm:2: warning: First Warning
arch.h:10: warning: Second Warning
arch.h:22: warning: Third Warning
```

4.13.7 Numerical expressions directives

Expressions, such as conditions to `#if` directives, are C expressions of integer type subject to restrictions. They may contain:

- Integer constants
- Character constants

- Arithmetic operations involving addition, subtraction, multiplication, division, bitwise operations, shifts, comparisons, and logical comparisons (`&&` and `||`, following the normal short-circuit rules of standard C)
- Macros that are expanded before evaluation begins
- Uses of the `defined` operator

5 Linking in KalAsm 3

Linking is the process of combining the objects and libraries generated by the assembler into a single ELF application image. The 'klink' linker wrapper arranges the code and data in memory according to rules given in the linker control file. It also fixes up instructions that reference labels or variables with their final addresses. It does this by repeatedly invoking the **kld** binutils linker program, having converted klink linker control files into a kld-format linkscript. This allows klink to automatically control placement of code and data.

A klink linker control file declares a number of regions, overlays, and segments. Regions and overlays describe a range of addresses in the processor's memory map. Regions are non-overlapping, and are typically used to describe the location of hardware features such as RAM banks and windows for mapping non-volatile memory. Overlays are used for code or data that is stored in non-volatile memory and mapped into the memory map or copied into RAM at run time. Because overlays can be switched in and out at run time, it is possible for multiple overlays to have the same address range. Segments describe where code modules and variables may be placed, in terms of regions and overlays.

5.1 KalAsm 3 defaults control files

For applications built for 24-bit Kalimba targets such as CSR8670 and CSR8675, the default control files are sufficient without modification. They provide several `REGION` and `OVERLAY` definitions to describe the RAM and non-volatile memory access windows in the memory map, and `SEGMENT` rules for allocating code and data to those regions.

The default linker control files for 32-bit Kalimba targets are only suitable for simple test programs, because they only describe portions of the memory map covering data and program RAM. It is not possible to describe non-volatile memory regions. To do so requires direct use of **kld**, the binutils linker, and creation of a detailed linkscript to describe the layout of all code and data both in RAM and non-volatile memory.

5.1.1 KalAsm 3 default regions and overlays

Regions are defined for each of the RAM blocks the Kalimba DSP can access. Table 5-1 lists the regions defined for CSR8670 to describe RAM.

Table 5-1 CSR8670 regions

Region	Description
PM_REGION	Main program RAM.
PM_CACHE_REGION	Cache region of program RAM. In most Kalimba applications, this is used automatically by the DSP to improve the performance of code running from non-volatile memory, so it is not available for direct use by the program.
DM1_REGION DM2_REGION	First and second data memory regions.

Table 5-2 lists the regions defined to describe the non-volatile memory access windows in the DSP memory map.

Table 5-2 Non-volatile memory regions

Region	Description
PMFLASH_REGION	Window for executing code stored in non-volatile memory.
DMFLASHWIN1_REGION DMFLASHWIN2_REGION DMFLASHWIN3_REGION	Three windows allowing data access to non-volatile memory.
DMFLASH_WIN1_LARGE_REGION DMFLASH_WIN2_LARGE_REGION DMFLASH_WIN3_LARGE_REGION	Larger windows allowing data access to non-volatile memory. These are configured to be a large view on the same data visible in the corresponding small windows.

Overlays must be defined to make use of the non-volatile memory windows. Whereas the regions define an area of the memory map, the overlay defines a block of initialized data that may be mapped into a particular region at run time. Table 5-3 lists the overlays that are defined by default.

Table 5-3 Default overlays

Overlay	Description
flash.code	Code stored in non-volatile memory for execution. Mapped to PMFLASH_REGION.
flash.data16	16-bit data, mapped to DMFLASHWIN1_LARGE_REGION.
flash.data24	24-bit data, mapped to DMFLASHWIN2_LARGE_REGION.
flash.windowed_data16	16-bit data using the virtual region DMFLASH_REGION. This region can only be accessed using the non-volatile memory access routines provided in the core library. This is not recommended for use on CSR8670 and later targets. When used, data in this region is mapped into DMFLASHWIN3_REGION.

5.1.2 KalAsm 3 default segments

Table 5-4 lists the segments defined by the default linker control files.

Table 5-4 Default segments

Segment	Description
DM1 DM1CIRC	Target DM1_REGION, the first data RAM bank. DM1CIRC targets the same region, but guarantees the correct alignment for circular buffers.
DM2 DM2CIRC	Target DM2_REGION, the section data RAM bank.
DM DMCIRC	Target either DM1_REGION or DM2_REGION.
DMCONST16	For 16-bit read-only data. Targets flash.data16.
DMCONST	For 24-bit read-only data. Targets flash.data24.
DMCONST_WINDOWED16	Targets flash.windowed_data16. Data in this section is not directly accessible. It must be accessed using non-volatile memory routines provided by the core library. It is not recommended for use on CSR8670 and later targets. For 16-bit, read-only data.
PM_RST PM_ISR	Special regions for use by the core library, to guarantee that the reset and interrupt routines are located at the correct addresses.
PM_RAM	For code that must be allocated to RAM. Targets PM_REGION.
PM	For code that may be allocated to either RAM or non-volatile memory. Code in this segment is placed in RAM (PM_REGION) if possible, but otherwise uses non-volatile memory (flash.code).
PM_FLASH	Code that must be placed in non-volatile memory. Targets flash.code.

5.2 Format of the klink linker control files

Linker control files are text files, following similar conventions to assembler source files. Comments are C or C++ style, see KalAsm 3 build process link using klink, the linker wrapper. The files are preprocessed according to the same rules as assembler source, see KalAsm 3 build process assembly. Linking rules are given with a number of directives, all following the format:

```
DIRECTIVE [argument ...] ;
```

Where DIRECTIVE is one of the directives described in the following sections. Directives may be split over multiple lines or multiple directives may be given on a single line. Directives are not case-sensitive.

The linker does not provide an expression evaluator. Where numeric values are required either hexadecimal or decimal literals must be used.

Multiple control files may be passed on the linker command line. In this case, the files are considered in order, as if they were concatenated. However, the files are pre-processed independently. So, for example, a macro defined in the first file is not available in subsequent files. An alternative is to provide a single control file that uses the preprocessor #include directive to include other control files. In that case, the files are preprocessed together.

5.2.1 CHIP directive

Syntax

```
CHIP chipName ;
```

This declares that this control file targets the Kalimba DSP on chip `chipName`. This directive is mandatory.

Example

```
CHIP CSR8670;
```

5.2.2 START directive

Syntax

```
START codeLabel ;
```

Specify the code label that is at the program entry point. This is required to allow the linker to discard unreferenced code and variables. When using the default command line options, this directive is mandatory. In most programs `codeLabel` is a label in the core library.

Example

```
START $reset;
```

5.2.3 REGION directive

Syntax

```
REGION regionName startAddress endAddress CODE ;
REGION regionName startAddress endAddress DATA ;
```

Declare a memory region named `regionName`, with the given address range, of the specified type. `startAddress` is the first address within the region and `endAddress` should be the last. As described above, the addresses must be given as decimal or hexadecimal numeric literals. Expressions are not permitted.

By convention, `regionName` is uppercase, and is given the suffix `_REGION` to distinguish region names from segment names; this is purely for clarity, and is not required by the linker.

When a region is defined, it may not be re-defined.

A klink region is analogous to a binutils output section.

Example

```
// Program memory is 0x2c00 words starting at 0
REGION PM_REGION 0x000000 0x002bff CODE;
// Data memory bank 1 is 0x8000 words starting at 0
REGION DM1_REGION 0x000000 0x007fff DATA;
```

5.2.4 SEGMENT directive

Syntax

```
SEGMENT segmentName [optionList] linkOrder regionList ;
OVERRIDE SEGMENT segmentName [optionList] linkOrder regionList ;
```

Creates a segment, and specifies rules for allocating code or variables assigned to that segment to particular memory regions. Any segment used in the assembler sources (including the library sources), in directives such as `.CODESEGMENT`, `.DATASEGMENT`, `.VAR` or `.BLOCK`, must be handled with a segment rule in the linker control file. The code and data placed in a segment are not necessarily contiguous, and do not usually appear in the order they appear in the source or on the command line.

The `OVERRIDE` keyword allows segments to be redefined. Without this keyword, redefinition of segments causes the linker to terminate with an error.

A klink region is analogous to a binutils output section.

Regions

`regionList` is a comma separated list of one or more regions where code and data placed in the named segment may be located. You must have previously defined the regions must with the `REGION` directive. If more than one region is specified, by default the regions are considered to be in order of preference. That is, a given variable or code module is placed in the first region in the list that has sufficient space for it. You can override this behavior with the `BALANCED` option, see Linking algorithm.

Link Order

`linkOrder` is an integer that controls the order in which variables are located in regions. Segments with lower link orders are allocated first. By convention, the default control files use multiples of 10 to ensure that it is possible to add segments to the link order between the default segments without making excessive changes.

For a more detailed description of how code and data is located in regions, see .

Options

optionList is an optional comma separated list of options that control how the linker handles the segment. Options are not case-sensitive.

Table 5-5 Segment options

Option	Meaning
BALANCED	Balance the contents of this segment between the regions in regionList instead of placing them in the first region with space remaining.
CIRCULAR	Give every variable or block in the segment suitable alignment for use as a circular buffer. For more details about circular buffers, see the *Kalimba Architecture 3 DSP User Guide* or equivalent for your target. The CIRCULAR option is only required for buffers with special alignment requirements. **NOTE** Slignment requirements can be avoided on circular buffers by setting the base registers.
KEEP	When used with the default options, the linker discards any code or data that is not referenced. This option causes the linker to keep everything in this segment, regardless of whether it is referenced or not.

Example

```
SEGMENT DM1CIRC     circular 10 DM1_REGION;
SEGMENT MY_DATA              20 DM1_REGION;
OVERRIDE SEGMENT MY_DATA     20 DM2_REGION;
SEGMENT DM          balanced 40 DM1_REGION, DM2_REGION;
SEGMENT PM                   40 PM_REGION, flash.code;
```

5.2.5 SUBREGION directive

Syntax

```
SUBREGION subregionName startAddress endAddress parentRegion ;
```

Sub-regions allow you to allocate segments within a particular range of addresses within a previously defined region (parentRegion). Sub-regions only restrict the allocation of segments that are explicitly assigned to the sub-region. Segments may still be assigned to the parent region, and those segments can still be assigned to the range of addresses covered by the SUBREGION.

Example

```
REGION DM1_REGION 0x000000 0x007fff DATA;
SUBREGION DM1_LOW_POWER_REGION 0x000000 0x001000 DM1_REGION;
```

5.2.6 OVERLAY directive

Syntax

```
OVERLAY overlayName regionName width;
```

This defines a block of code or initialized data that will be stored in non-volatile memory and either mapped or possibly copied into the region `regionName` at runtime.

So that the application can access the overlay data, the linker creates a variable that is populated with the address in non-volatile memory by the application loader (which is initiated by a call to `KalimbaLoad()` in the VM application). The variable is named by prefixing the overlay name with a dollar and appending `.address`. So for the `flash.code` overlay, the variable is named `$flash.code.address`. These variables are placed according to a special segment named, (`overlay info`).

The `width` is the word size in bits of code or data stored in the region. For code overlays, it must be 32, the size of an instruction. For data overlays, it may be either 16 or 24.

You can have multiple overlay blocks that all map into the same region.

Example

```
OVERLAY flash.code PMFLASH_REGION 32;
SEGMENT (overlay info) balanced  40  DM1_REGION, DM2_REGION;
```

5.3 Linking algorithm

The linker uses the declarations made in the linker control file to pack contiguous sections of code and data into the available memory. It considers the sections one by one, placing each at the lowest available valid address in the first valid region (or for balanced segments, the region with the least usage so far). In detail, the algorithm is as follows:

1. Match a `SEGMENT` rule to each section, and report an error if there are any sections with no match.
2. For each value of `linkOrder` ascending from zero
    a. While there are sections remaining with this link order
        i. Find the most constrained section
        ii. If that section's segment is balanced sort the segment's `regionList` by decreasing space used so far
        iii. Otherwise
        iv. Leave the segment's `regionList` in the original order
    b. Place the section at the lowest address possible in the first region with sufficient space available

At each step, the algorithm chooses the *most constrained* section remaining to place. This heuristic helps to maximize the use of memory. The most constrained section is defined to be:

- The section with the fewest valid regions that have sufficient space available with the correct alignment
- If there is more than one, then the section with the strictest alignment constraints
- If there is still more than one, then the largest section

This algorithm allows efficient use of memory with only minimal intervention from you. However, it does mean that the order in which variables and code are placed in memory does not correspond to the order they appear in the source or on the linker command line. So, for example, variables that are

adjacent in the source are not contiguous in memory unless they are grouped with the `.BLOCK` directive.

5.4 klink linker control file preprocessor

KalAsm 3 preprocessor directives describes how kliink's linker preprocessor works. However, it works per input linkscript and not on their concatenation.

Example

```
klink –o final.elf file1.o file2.o –T script1.link script2.link
script3.link
```

Each of the scripts on the command line are preprocessed independently. If `script1.link` defines a macro, this macro is not seen by either `script2.link` or `script3.link`. A possible solution is to add the definitions you want all the link scripts to see to a header file that is then included by all of them.

6 Linking directly using kld - overview

kld, the binutils linker, places code, data and metadata strictly according to output section specifications written in the supplied linkscript. It does not attempt to pack memory in an automatic or balanced way. Indeed, if an input section is encountered which is not assigned to an output section, kld will create a new output section and append it to the existing contents.

Documentation for ld describing command line options and the linkscript format can be found at:

`https://sourceware.org/binutils/docs-2.18/ld/`

6.1 Invoking kld

Many of the command line options supported by **kld** match the ones provided by **klink**. For example, `--gc-sections` enables garbage collection of unused input sections, therefore saving space on unused code and data. However, there are some important differences:

- Relaxation must be enabled using `--relax` if you wish to generate MiniMode code
- For 32-bit Kalimba targets such as csra68100, the endianness of the generated ELF image must be manually specified using the `-EL` option

If you are used to creating an ELF image by invoking the kcc driver tool described in the *Kalimba C Compiler User Guide*, it is necessary to split the build and link stages into separate commands. This is because kcc always attempts to perform the link using **klink**. For example, given this command to build `hello_world.elf`:

```
kcc -kcsra68100_audio hello_world.c assembly.s -g -o hello_world.elf
```

Linking directly using **kld** might instead require:

```
kcc -kcsra68100_audio hello_world.c -g -c -o hello_world.o -mark-code-section-type
kcc -kcsra68100_audio assembly.s -g -c -o assembly.o -mark-code-section-type
kld --relax -EL --gc-sections crt0.o hello_world.o assembly.o -T linkscript.ld -L<library path> -lc -lfle -lcrt -o hello_world.elf
```

Here, the kcc driver handles transforming C and assembly files into individual object files and, due to the `-c` option (“do not link”), stops after the assembly stage. **kld** is then invoked directly. The user must pass paths to any libraries required by the link, such as libc, using the `-L` option.

6.2 More on linkscripts

The most difficult aspect of switching to linking using **kld** is getting the linkscript correct. Many of the details handled automatically by **klink** must be specified explicitly in a **kld** linkscript.

6.2.1 Section placement directives and wildcards in linkscripts

A **kld** linkscript is mainly a series of placement directives telling the linker how the contents of input sections are mapped to various output sections in the finished ELF image. Compiled code or data will be assigned to a suitable default input section, although this can be overridden using section placement pragmas – see the *Kalimba C Compiler User Guide* for more information. For hand-written assembly, the input section name corresponds with the segment specified using the `.CODESEGMENT` or `.DATASEGMENT` directive for the current module. To ease the placement process for code and data, wildcards are supported. A wildcard can be used on input section names, whole object file names, or a mixture of both. A small number of directives can cover the placement of a lot of code or data.

The wildcarding facility can be used to split apart MiniMode and MaxiMode code sections. This is necessary because the instruction encoding used is a property of the ELF section containing the code – it is not possible to mix instruction encodings in the same section. Both the C compiler and the assembler support the `-mark-code-section-type` command line option, which automatically appends `__maxim` or `__minim` to code section names to assist this process.

6.2.2 Image checksum facility in linkscripts

As an extension to the standard ld linkscript format, the `CHECKSUM_DATA_SECTION` directive instructs the linker to generate a checksum of all code and data found within the final, linked ELF image. The user specifies the name of an input section to hold the generated section. This can then be placed in the desired output section just like any other code or data. The checksum can be used by a debugger to distinguish between ELF images in a project, and to validate that an image loaded on-chip matches the image loaded in a debug session.

6.2.3 Address spaces in linkscripts

Another important concept is the difference between a VMA (Virtual Memory Address, conventionally referring to the address of an entity at run-time) versus the LMA (Load Memory Address, conventionally referring to the address of an entity at load-time). A useful way to think about the difference is to consider some initialisation data located in ROM. The LMA would correspond with the data's position in the ROM image. The VMA would be the address in RAM at run-time, presuming a loader has arranged to copy the data to a suitable location in data memory at start of day. Exact loading schemes vary depending on the target.

Flat address spaces pose a problem for Kalimba, which has a Harvard architecture, and therefore maintains separate address spaces for code and data. By default, it would be impossible to distinguish between code at PM address `0x0` and data at DM address `0x0`. To work around this problem, code VMAs should have the most significant bit of the address set. Therefore, code located at PM address 0x0 would actually have a VMA of `0x80000000`.

6.2.4 Sample linkscript

Figure 6-1 is a sample linkscript showing a basic code and data layout suitable for use with both C code and hand-written assembly. This demonstrates several of the previously described concepts.

Figure 6-1 Sample kld linkscript with basic code and data layout

```
OUTPUT_ARCH(csra68100_audio)
ENTRY($reset)
CHECKSUM_DATA_SECTION(dm);
/* $CODE_RAM_START_ADDRESS has the top bit set. This tells the linker
that this is in
 * the program address space rather than the data address space */
$CODE_RAM_START_ADDRESS = 0x84000000;
$CODE_ROM_START_ADDRESS = 0x04000000;
$DATA_RAM_START_ADDRESS = 0x00000000;
$DATA_ROM_START_ADDRESS = 0x00000000;
SECTIONS
{
    . = $CODE_RAM_START_ADDRESS;
    .text_reset : AT($CODE_ROM_START_ADDRESS)
    {
        *(PM_RST)
    }
    .text_maxim :
    {
        *(*PM*__maxim)
        *(*pm*__maxim)
    }
    .text_minim :
    {
        *(*__minim)
    }
    . = $DATA_RAM_START_ADDRESS;
    .data : AT($DATA_ROM_START_ADDRESS)
    {
        *(DM*)
        *(dm*)
    }
    .bss (NOLOAD) :
    {
        *(dmzi*)
    }
    .const :
    {
        *(dmconst)
```

```
        }
}
```

A suitable linkscript is normally provided as part of a software development kit.

A KalAsm 3 instruction shortcut syntax

In addition to the instruction syntax given in the processor user guides, the assembler allows certain shortcut forms to be used. This is intended to help you keep your code clear and readable:

- In all cases, a single shortcut translates to a single instruction.
- Where the full instruction may be conditional, the shortcut forms may also be conditional.
- Likewise, where the full instruction accepts indexed memory accesses in parallel (i.e. type A and C instructions), the shortcut forms also accept that.
- In the descriptions below, unless otherwise noted, `r0` and `r1` should be taken to represent arbitrary bank 1 registers and similarly, `I0` and `I1` represent arbitrary bank 2 registers. The word `value` represents any valid numeric value or expression.

A.1 Assignments

Shortcut	Full Instruction
`r0 = r1;`	`r0 = r1 + Null;`
`r0 = r1 + Carry;`	`r0 = r1 + Null + Carry;`
`r0 = M[r1];`	`r0 = M[r1 + Null];`
`M[r0] = r1;`	`M[r0] = r1 + Null;`
`r0 = value;` `r0 = 0;`	`r0 = Null + value;` `r0 = r0 - r0;`
`r0 = value + Carry;`	`r0 = Null + value + Carry;`
`I0 = r0;`	`I0 = Null + r0;`
`r0 = I0;`	`r0 = Null + I0;`
`I0 = I1;`	`I0 = Null + I1;`
`I0 = value;` `I0 = 0;`	`I0 = Null + value;` `I0 = I0 - I0;`

A.2 Replacing constant zero with the null register

Shortcut	Full Instruction
`M[value] = 0;`	`M[value] = Null;`
`M[value + r0] = 0;`	`M[value + r0] = Null;`

Shortcut	Full Instruction
`M[value] = r0 + 0;`	`M[0] = Null + r0;`
`M[value] = 0 + r0;`	`M[0] = Null + r0;`
`M[r0 + r1] = 0;`	`M[r0 + r1] = Null;`

A.3 Negation

Shortcut	Full Instruction
`r0 = -r1;`	`r0 = Null - r1;`
`r0 = -M[r1];`	`r0 = Null - M[r1];`
`M[r0] = -r1;`	`M[r0] = Null - r1;`
`r0 = -r1 - Borrow;`	`r0 = Null - r1 - Borrow;`
`r0 = - M[value];`	`r0 = Null - M[value];`
`M[value] = -r1;`	`M[value] = Null - r1;`
`I0 = -r0;`	`I0 = Null - r0;`
`r0 = -I0;`	`r0 = Null - I0;`
`I0 = -I1;`	`I0 = Null - I1;`

A.4 NOP

Shortcut	Full Instruction
`nop;`	`Null = Null + Null;`

A.5 Indexed memory accesses

Shortcut	Full Instruction

`M[I0, M0] = r0, r1 = M[I4, M1];`	`Null = Null + Null,`
	`M[I0, M0] = r0,`
	`r1 = M[I4, M1];`

Any type A or C combination of indexed memory accesses can be written in this way without specifying a main opcode. The assembler considers the main opcode to be `Null = Null + Null;`.

A.6 Reordering commutative operands

Table A-1 Addition and logical operators

Shortcut	Full Instruction
`r0 = M[value] + r1;`	`r0 = r1 + M[value];`

`r0 = value AND r1;`
`r0 = value OR r1;`
`r0 = value XOR r1;`
	`r0 = r1 AND value;`
	`r0 = r1 OR value;`
	`r0 = r1 XOR value;`

`I0 = value + r0;` `r0 = value + I0;`	`I0 = r0 + value;` `r0 = I0 + value;`

Table A-2 Multiply and multiply-accumulate

Shortcut	Full Instruction
`r0 = value * r1 (int);` `r0 = value * r1 (frac);`	`r0 = r1 * value (int);` `r0 = r1 * value (frac);`

`rMAC = value * r1 (UU);`
`rMAC = value * r1 (SS);`
`rMAC = value * r1 (US);`
`rMAC = value * r1 (SU);`
	`rMAC = r1 * value (UU);`
	`rMAC = r1 * value (SS);`
	`rMAC = r1 * value (SU);`
	`rMAC = r1 * value (US);`

`rMAC = rMAC + value * r0 (UU);`
`rMAC = rMAC + value * r0 (SS);`
`rMAC = rMAC + value * r0 (US);`
`rMAC = rMAC + value * r0 (SU);`
	`rMAC = rMAC + r0 * value (UU);`
	`rMAC = rMAC + r0 * value (SS);`
	`rMAC = rMAC + r0 * value (SU);`
	`rMAC = rMAC + r0 * value (US);`

`rMAC = rMAC - value * r0 (UU);`
`rMAC = rMAC - value * r0 (SS);`
`rMAC = rMAC - value * r0 (US);`
`rMAC = rMAC - value * r0 (SU);`
	`rMAC = rMAC - r0 * value (UU);`
	`rMAC = rMAC - r0 * value (SS);`
	`rMAC = rMAC - r0 * value (SU);`
	`rMAC = rMAC - r0 * value (US);`

Table A-3 Load and store

Shortcut	Full Instruction
`M[-1 + r0] = r1;` `M[r0 - 1] = r1;`	`M[r0 + -1] = r1;` `M[r0 + -1] = r1;`
`r1 = M[-1 + r0];` `r1 = M[r0 - 1];`	`r1 = M[r0 + -1];` `r1 = M[r0 + -1];`

Table A-4 Stack operations

Shortcut	Full Instruction
rMAC0 = M[r0 + FP];	rMAC0 = M[FP + r0];

SP = SP + r0;
r0 = SP + r0;
r0 = FP + r0;
	SP = r0 + SP;
	r0 = r0 + SP;
	r0 = r0 + FP;

rMAC0 = M[FP - *value*];
rMAC0 = M[-*value* + FP];
rMAC0 = M[SP - *value*];
rMAC0 = M[-*value* + SP];
	rMAC0 = M[FP + -*value*];
	rMAC0 = M[FP + -*value*];
	rMAC0 = M[SP + -*value*];
	rMAC0 = M[SP + -*value*];

M[FP - *value*] = rMAC0; M[-*value* + FP] = rMAC0; M[SP - *value*] = rMAC0; M[-*value* + SP] = rMAC0;	M[FP + -*value*] = rMAC0; M[FP + -*value*] = rMAC0; M[SP + -*value*] = rMAC0; M[SP + -*value*] = rMAC0;

r0 = *value* + SP;
r0 = *value* + FP;
SP = *value* + SP;
FP = *value* + FP;
	r0 = SP + *value*;
	r0 = FP + *value*;
	SP = SP + *value*;
	FP = FP + *value*;

r0 = SP - *value*; r0 = FP - *value*; SP = SP - *value*; FP = FP - *value*;	r0 = SP + -*value*; r0 = FP + -*value*; SP = SP + -*value*; FP = FP + -*value*;
pushm <r0>, SP = *value* + SP;	pushm <r0>, SP = SP + *value*;
push r0 – *value*;	push r0 + -*value*;

A.7 The (SS) modifier is optional

Shortcut	Full Instruction
rMAC = r0 * r1;	rMAC = r0 * r1 (SS);

A.8 Push

Shortcut	Full Instruction
push *value*;	push Null + *value*;

B KalAsm 3 keywords and reserved words

This topic lists all the keywords and reserved words that would otherwise be syntactically valid names. You cannot use these words as the names for constants, labels, variables, blocks, modules or, segments. See KalAsm 3 assembler syntax reference for more information on naming rules and conventions. The words are listed in alphabetical order.

A	**e**	**i**	**m**	**r**	
A	ELSE	I0	M0	R0	RLINK
ABS	EQ	I1	M1	R1	RMAC
AL		I2	M2	R10	RMAC0
AND		I3	M3	R2	RMAC1
ASHIFT		I4	MAX	R3	RMAC12
		I5	MIN	R4	RMAC2
		I6	MOD24	R5	RMACB
		I7	MOD3	R6	RMACB0
		IF		R7	RMACB1
				R8	RMACB12
				R9	RMACB2
				RFLAGS	RTI
				RINTLINK	RTS
B	f	j	n	s	
B	FP	JUMP	NB	SE16	
B0			NC	SE8	
B1			NE	SIGNDET	
B4			NEG	SLEEP	
B5			NONUL	SP	
BLKSIGNDET			NOP	STRING	
BORROW			NULL	t	
BREAK			NV	THEN	
			NZ	TWOBITCOUNT	
C	g	k	o	u	
C	GE	KALCODE	ONEBITCOUNT	USERDEF	
CALL	GT		OR		
CARRY					
D	h	l	p	v	

A	e	i	m	r
DEFINED	HI	L0	PACK16	V
DIV		L1	PACK24	x
DIVREMAINDER		L4	PLOOK	XOR
DIVRESULT		L5	POP	z
DO		LE	POPM	Z
DOLOOPEND		LS	POS	
DOLOOPSTART		LSHIFT LT	PUSH PUSHM	

Document references

Document	Reference
Qualcomm Kalimba Architecture 3 DSP User Guide	80-CE519-1/CS-00202067-UG
Qualcomm Kalimba Architecture 4 DSP User Guide	80-CT886-1/CS-00333144-RM
Qualcomm Kalimba Architecture 5 DSP User Guide	80-CT525-1/CS-00318059-UG
Qualcomm Kalimba C Compiler User Guide	80-CT240-1/CS-00124812-UG

Terms and definitions

Term	Definition
CSR	Cambridge Silicon Radio
DSP	Digital Signal Processor
ELF	Executable and Linkable Format
e.g.	*exempli gratia*, for example
etc	*et cetera*, and the rest, and so forth
i.e.	*Id est*, that is
PC	Personal Computer
QTIL	Qualcomm Technologies International, Ltd.
RAM	Random Access Memory
ROM	Read Only Memory