

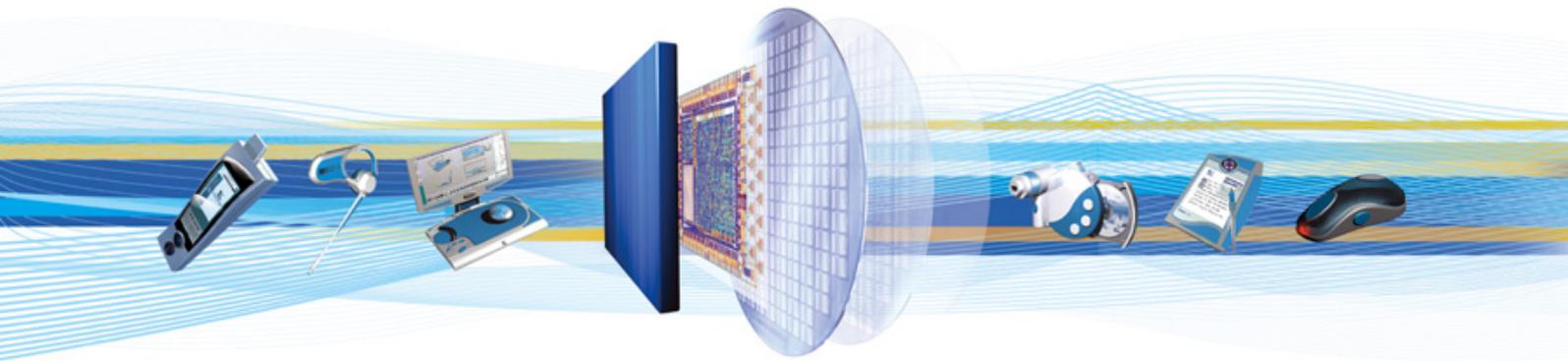


BlueLab™

Kalimba DSP Assembler

User Guide

May 2005



CSR

Churchill House
Cambridge Business Park
Cowley Road
Cambridge CB4 0WZ
United Kingdom

Registered in England 3665875

Tel: +44 (0)1223 692000
Fax: +44 (0)1223 692001

www.csr.com

Contents

1	Introduction	4
2	Overview	5
3	Overall Assembler Process Flow	6
3.1	Input files	8
3.1.1	Default and Groups files	8
3.1.2	Assembler Input file (.asm)	8
3.1.3	Data Files (.dat)	8
3.1.4	Library Files (.klib)	8
3.2	Input / Output files	9
3.2.1	Object Files (.kobj)	9
3.3	Output files	10
3.3.1	Pre-processor Files (.kpp)	10
3.3.2	Listing Files (.klist)	10
3.3.3	Linker Output Files (.klo)	11
3.3.4	Linker Map Files (.kmap)	11
3.4	Program Content	14
3.4.1	Instructions Set	14
3.4.2	Assembler Directives	14
3.4.3	Pre-Processor Commands	15
4	Assembler DOS Prompt Interface	16
5	Assembler Syntax Reference	17
5.1.1	Assembler Keywords and Symbols	18
5.2	Formats and Conventions	19
5.2.1	Numeric Formats	19
5.2.2	Fractional Type Support in Q Format	19
5.2.3	Comment Conventions	19
5.2.4	Global Symbols	20
5.3	Special Assembler Operators	21
5.3.1	LENGTH Operator	21
5.3.2	BITREVERSE Operator	22
5.4	Pre-Processor Commands	23
5.4.1	.include	24
5.4.2	.define	25
5.4.3	defined	26
5.4.4	.undef	26
5.4.5	.if	27
5.4.6	.elif	28
5.4.7	.else	29
5.4.8	.endif	29
5.4.9	.ifdef	30
5.4.10	.elifdef	31
5.4.11	.ifndef	32
5.4.12	.elifndef	32
5.4.13	.message	33
5.4.14	.warning	33
5.4.15	.error	34
5.4.16	.line	34
5.4.17	.file	35
5.5	Assembler Directives	36
5.5.1	.VAR, Declare a Variable	37
5.5.2	.CONST, Declare a Constant	38
5.5.3	.BLOCK & .ENDBLOCK, Declare a variable grouping	39
5.5.4	.MODULE & .ENDMODULE, Declare a Program Module	41
5.5.5	.DATASEGMENT and .CODESEGMENT	42
6	Accompanying Tools Provided with kalasm2	43
6.1	The Kalimba Packer Utility (kalpac2)	43

7	Some Background on DSPs & Number Representation	44
7.1	Signed Fractional Numbers	44
7.2	Some Implications of Q Format Representations	45
7.2.1	Interpretation	45
7.2.2	Accuracy of 1	46
7.2.3	Out of Range Numbers	46
8	Document References	47
Appendix A	kalasm2 switches	48
	Terms and Definitions	49
	Document History	50

List of Figures

Figure 3.1	Assembler Input and Output Files	6
Figure 3.2	An example linker map file	13

List of Tables

Table 3.1	Kalasm2 Input and Output files	7
Table 4.1	File Extension for Kalasm	16
Table 5.1	Numeric Formats	19
Table 5.2	Comment Conventions	19
Table 5.3	Pre-Processor Commands	23
Table 5.4	Assembler Directives	36
Table 5.5	Example Memory Type Section	40

List of Equations

Equation 5.1	Fractional Type Conversion Formula	19
Equation 7.1	Q.23 Number format	44

1 Introduction

The **BlueCore™3-Multimedia** Kalimba DSP Assembler User Guide documents the kalasm2 assembler. It is intended as a user guide for developers of software applications and algorithms on the Kalimba digital signal processor (DSP) core.

This document is intended to be read in conjunction with other Kalimba DSP documents:

- BlueCore3-Multimedia Kalimba DSP User Guide (bc3-ug-001Pc) which looks into the programming model, including the instruction set mnemonics
- BlueLab xIDE User Guide (CSR reference blab-ug-002Pa) which covers xIDE, the software debugging tool

2 Overview

This User Guide describes the process of developing new programs in assembly language for the Kalimba DSP in the BlueCore3-Multimedia.

This guide should be read and used in conjunction with the Kalimba DSP User Guide, consequently only information about the assembler and its impact on program design and use is covered in this guide, including:

- Input and Output files used by the kalasm2 assembler, their uses and structure
- Assembly Instructions
- Pre-processor commands
- Assembler directives and special commands

The kalasm2 executable is called by xIDE, the integrated development environment that forms part of BlueLab. These tools supersede Kalasm and kaldbg respectively, the notable difference being the removal of any dependency upon Matlab®.

From this point onwards the term "assembler" refers to the kalasm2 assembler only.

The assembler processes input files written in Kalimba assembly language and converts them into instruction code which is run on the Kalimba DSP. Further inputs to the assembler include:

- data files, used to initialise variables
- header files, typically defining constants
- libraries, used to define commonly used routines.

3 Overall Assembler Process Flow

Assembly language programs should be written using an editor that produces text files.

Note:

Word processors that embed special control codes in the text are not suitable.

Source files should be saved with a .asm extension to identify them as kalasm2 assembly source files.

The source files are assembled by executing kalasm2. The two typical methods by which kalasm2 can be executed are:

- Directly, e.g. from the DOS prompt, with relevant switches
or
- Indirectly, via xIDE, with switches applied.

It is envisioned that most users will use kalasm2 through the xIDE environment.

Once executed kalasm2 processes the input files to produce output files, completing each of the stages shown in Figure 3.1.

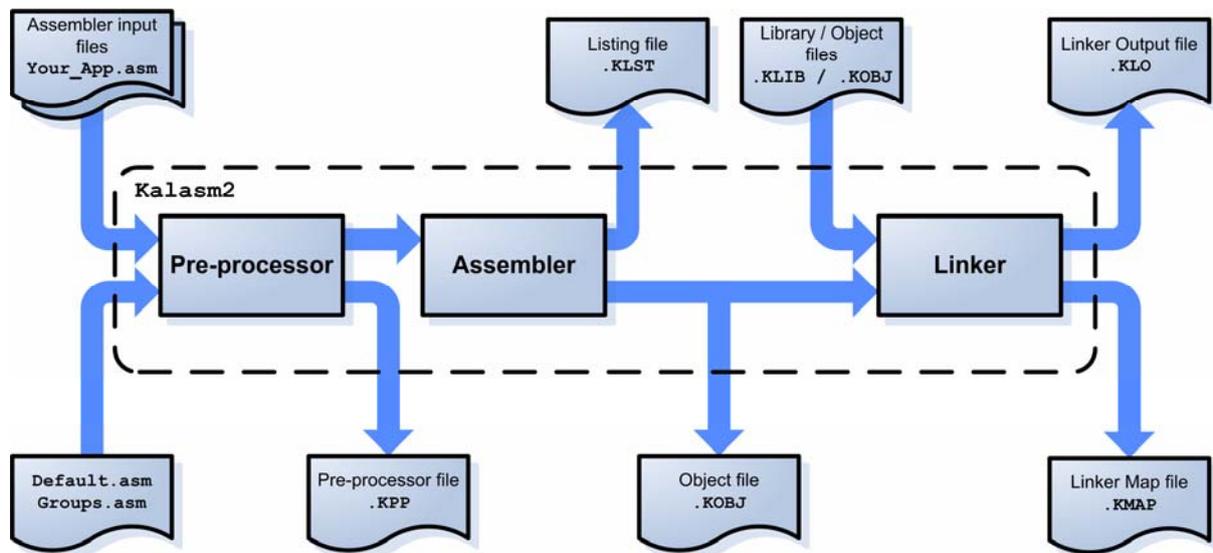


Figure 3.1 Assembler Input and Output Files

Control over kalasm2 options, including the files that are produced and how input files are handled, is achieved through application of relevant switches, a complete list of which can be found in Appendix A .

Table 3.1 below lists the files used in the kalasm2 assembly process with a brief description of their purpose.

File Type	File	Description	Direction	Section
Default file	default.asm	Defines where code and data are stored. Generally this does not need to be altered from the supplied file.	Input	3.1.1
Groups file	groups.asm	Defines the size of program and data memory. Generally this does not need to be altered from the supplied file.	Input	3.1.1
Assembler Input File	your_app.asm	Contains your program written in the Kalimba assembly language	Input	3.1.2
Data file	data_values.dat	Contains initialisation parameters for variables rather than embedding them in your assembler input file.	Input	3.1.3
Pre-processor file	your_app.kpp	Parsed code confirming pre-processor operation	Output	3.3.1
Object file	your_app.kobj	Minimal representation of assembler output, can be passed directly to linker	Both	3.2.1
Library file	lib_app.klib	Identical to an object file but contains library routines, only the required routines are passed to the linker	Input	3.1.4
Listing file	your_app.klst	A verbose representation of assembler output, including comments, and assembly code	Output	3.3.2
Linker output file	your_app.klo	Output of overall assembly and linking process, comprising of instruction code, data memory values and symbol information	Output	3.3.3
Linker map file	your_app.kmap	A verbose output file, containing instruction codes (and disassembly), data memory values and symbol information	Output	3.3.4

Table 3.1 Kalasm2 Input and Output files

3.1 Input Files

This section briefly describes the different types of Input files used in the kalasm2 assembly process.

3.1.1 Default and Groups Files

As well as the kalimba code to be assembled, the assembler requires a `default.asm` file and a `groups.asm` file. These files contain information relating to the allocation of data and code in kalimba.

The `groups.asm` file names a number of program and data memory groups. A group consists of a legal range of addresses for either data or code.

The `default.asm` file defines data and code segments and associates them with particular groups already defined in the `groups.asm` file.

The `groups.asm` and `default.asm` files are automatically included during the assembly process and are located at the head of the assembled file.

3.1.2 Assembler Input File (.asm)

The Assembler Input file contains the program code written in the Kalimba DSP assembly language. As well as assembly code, it may include:

- Pre-processor commands – which direct the pre-processor to perform specific operations, such as including external files and defining macros, for further details see section 5.4
- Assembler directives – instructions which control the assembly process, they do not produce code to run on the DSP, for further details see section 5.5
- Assembler commands – a rather smaller group of functions performed by the assembler, not the DSP. They are not run by the DSP but affect the code which is, for further details see section 5.3

3.1.3 Data Files (.dat)

Data files are used to initialise variables. Typically large variables, where the values are generated by an external program, for example the weights of filter taps or the coefficients for a calculation. The files are used when the assembler initialises the relevant variable; see section 5.5 Assembler Directives.

3.1.4 Library Files (.klib)

The Library files provide the user with a method to create libraries of common functions and routines. These functions can then be included in different routines simply by including the relevant library file.

The format of a library file is fundamentally identical to that of an object file and they appear in the assembly process in the same place.

However, as library files are created for general use and not specific applications they may well include functions that are not needed in every instance they are called. Therefore, the assembler will only include a routine from a library file if there is a reference to it in the main code. This allows large libraries of standard routines to be used without creating large DSP programs every time the library is 'linked in'.

3.2 Input / Output Files

3.2.1 Object Files (.kobj)

The kalasm2 pre-processor and assembler stages result in the production of an object file.

An object file contains all the information from the Input files, (including the `groups.asm` file and the `default.asm` file) in a compressed format that is easily read by the Linker (but not by humans).

kalasm2 can output object files for subsequent use, for example as library files.

More commonly the object file can be passed directly on to the Linker for creation of the DSP application. In this case the Linker will resolve any library references for common functions or routines and include the necessary code, allocate memory addresses and output the final application code.

Where required object files can be used as the initial input files when creating a Kalimba DSP application using kalasm2. This may be desirable when wishing to distribute application programs without revealing the original source code.

3.3 Output Files

3.3.1 Pre-processor Files (.kpp)

A kalimba Pre-processor file is an output text file that contains the code generated by the pre-processor. That is with the code stripped of comments, with any defined macros resolved and with any ".include" files added.

The file is headed by a copy of the `groups.asm` file and `default.asm` file followed by a parsed version of the assembler code.

The Pre-processor file allows the user to confirm that pre-processing has been completed correctly and that any constants defined as numbers or by macros have been replaced with their associated value.

3.3.2 Listing Files (.klst)

A kalimba Listing file is an output text file that lists the results of the assembly process.

Listing files display information in four columns:

- **Hex-Address** - this column contains the absolute address in program memory of the current line of assembly source.
- **Hex-Instruction** - this column contains the hexadecimal instruction code that the assembler generates for the line of assembly source. If the instruction references a variable or program memory address, an asterisk (*) is inserted after the instruction code. This indicates that the instruction is not complete as memory mapping is determined at the linker stage so variable addresses will not be available.
- **Assembly Source Line Number** - this column contains the assembly source line number
- **Assembly Source** - this column contains the assembly source line from the file.

3.3.3 Linker Output Files (.klo)

The Linker output files are the final result of the assembly process. This file is then passed on to Kalpac2 (see section 6.1) which finally generates the file stored in Kalimba.

The Linker output file is effectively a list of the data constituting the program and the data memory sections that will be allocated in Kalimba, along with some debugging symbol information, it is only created when kalasm2 executes without error.

3.3.4 Linker Map Files (.kmap)

A Kalimba Linker map file is an output text file that lists the results of the linking process. It is only generated when the assembler has run without error.

The information in a Linker map files is separated into three sections:

- Code address space
- Data address space
- Symbols

A brief description of the information within each area is given below.

Code address space

The Code address space consists of the following columns:

- Address column: this identifies the address in program memory where the instruction is located
- Module column: this identifies which module of code the instruction belongs to
- Instruction type (T) column: labels each instruction as a type A, B or C (CR and CC) instruction
- Code column: gives the instruction code corresponding to the instruction,

Note:

Type B instructions are generally represented by two lines, the first corresponding to a possible prefix instruction and the second the 'actual' instruction.

In most cases, the prefix instruction is not required and so is automatically removed by the linker. However when constants are used which exceed the 16 bits available in the actual instruction, the prefix instruction is used to describe the larger value.

- Assembler code column: the assembler code representing the instruction as found in the .asm file.

Data address space

The Data address space consists of the following columns:

- Address column: identifies the address of the variable in memory.

Note:

For variables representing a single element this is simply the address of that element. For arrays, the address corresponds to the first element of the array.

- Module column: the module the variable is declared in.
- Value column: this contains the name of the variable and the data stored; with the first row containing the name of the variable and the subsequent row(s) containing the data stored.

Symbols

The Symbols section consists of the following columns:

- Address column: the location the symbol can be found in memory;

Note:

This table includes all symbols for data memory and program memory.

- Type column: the type of symbol, either 'CODE' for data from program memory – address labels – or 'CONST' and 'DATA' otherwise.
- Size column: indicates the size of the symbol, for program and constant symbols, this is zero, for data memory this is the size of the symbol from one up to the size of the array.
- Module column: the module the symbol is defined in.
- Name column: the name of the symbol and the data stored, with '-----' representing an un-initialised variable value.

The contents of the Linker map file are useful when debugging code.

They can be used as a key to program code stored in Kalimba program memory, giving locations of variables in data memory and the initial starting value of variables in data memory.

An example of a Linker map file is given in Figure 3.2.

Kalimba linker listing file exam_map.kmap

```
Code address space
Addr Module      T Code
  $M.Main          $Main:
0000 $M.Main          // PREFIX(0x0);
0000 $M.Main      B  51000000    I0 = NULL + 0x0;
0001 $M.Main          // PREFIX(0x0);
0001 $M.Main      B  51C00006    L0 = NULL + 0x6;
0002 $M.Main          // PREFIX(0x0);
0002 $M.Main      B  51400006    I4 = NULL + 0x6;
0003 $M.Main      A  50E0000F    L4 = NULL + NULL    NULL = M[I0,M0];
0004 $M.Main          // PREFIX(0x0);
0004 $M.Main      B  5110000D    I1 = NULL + 0xD;
0005 $M.Main      A  50D0000F    L1 = NULL + NULL    NULL = M[I0,M0];
  $M.Main          $M.Main.Loopdata:
0006 $M.Main      CC 83102131    rMAC = rMAC AND NULL    R0 = M[I0,1]    R1 = M[I4,1];
0007 $M.Main          // PREFIX(0x0);
0007 $M.Main      B  01C00005    R10 = NULL + 0x5;
0008 $M.Main      B  E5F0000A    DO 0xA;
0009 $M.Main      CC AF232131    rMAC = rMAC + R0 * R1 (ss)    R0 = M[I0,1]    R1 = M[I4,1];
  $M.Main          $M.Main.Notchfilter:
000A $M.Main      CC AF230000    rMAC = rMAC + R0 * R1 (ss);
000B $M.Main          // PREFIX(0x0);
000B $M.Main      B  79440005    I4 = I4 - 0x5;
000C $M.Main      CC 03009500    NULL = NULL + NULL    M[I1,1] = rMAC;
000D $M.Main          // PREFIX(0x0);
000D $M.Main      B  69011B5E    NULL = I1 - 0x1B5E;
000E $M.Main      B  DD100006    if NE JUMP 0x6;

Data address space
Addr Module      Value
  $M.Main          $M.Main.filter:
0000 $M.Main          0E147B,0E147B,1C28F6,1C28F6,0E147B,0E147B
  $M.Main          $M.Main.data:
0006 $M.Main          000000,000001,000002,000003,000004,000005,000006
  $M.Main          $M.Main.output:
000D $M.Main          -----,-----,-----,-----,-----,-----

Symbols
Addr Type  Size Module      Name
FF00 CONST 0000 $M.Main      $regBase
FF00 CONST 0000 $M.Main      $reg1
FF01 CONST 0000 $M.Main      $reg2
0000 CODE 0000 $M.Main      $Main
0006 CODE 0000 $M.Main      $M.Main.Loopdata
000A CODE 0000 $M.Main      $M.Main.Notchfilter
0000 DATA 0006 $M.Main      $M.Main.filter
0006 DATA 0007 $M.Main      $M.Main.data
000D DATA 0007 $M.Main      $M.Main.output
```

Figure 3.2 An Example Linker Map File

3.4 Program Content

Statements within an assembly source file comprise of:

- assembly instructions
- assembler directives
- preprocessor commands

Instructions assemble to executable code, while directives and commands modify the assembly process. The syntax of each of the statement types is described in section 3.4.

3.4.1 Instructions Set

Instructions follow the DSP instruction set syntax, documented in the BlueCore3-Multimedia Kalimba DSP User Guide (bc3-ug-001Pc).

Each instruction begins with a keyword and ends with a semicolon (;). The location of an instruction can be marked by placing an address label at the beginning of an instruction line or on the preceding line. Address labels are marked with a colon (:).

At linking, the linker replaces any label references with the absolute address of that label. There is no length restriction when defining labels.

Note:

Labels are case sensitive, the assembler treats “loop” and “Loop” as unique labels.

Example:

```
if NZ jump non_zero;
    ...
    jump continue;
non_zero:
    ...
continue:
```

3.4.2 Assembler Directives

Assembler Directives instruct the assembler to perform in a particular way in certain situations. For example an assembler directive could be used to instruct the assembler to allocate a certain amount of memory for a particular variable.

Directives can also be used to control the operation of other directives. For example a block command can be used to tell the assembler to allocate contiguous memory locations for selected variables.

Assembler Directives begin with a period (.) and end with a semicolon (;). The period must be the first non-whitespace character on the line containing your directive.

Note:

The assembler allows for directives in both lowercase and uppercase. By convention uppercase is used to distinguish directives from other assembly statements.

Example:

```
.BLOCK/DM sin_cos_lookup_coefs;
.VAR sin_coef = sin_coef.dat;
.VAR cos_coef = cos_coef.dat;
.ENDBLOCK;
```

For a complete description of the Assembler Directives see 5.4.

3.4.3 Pre-Processor Commands

The assembler uses a purpose made pre-processor, which is run by the assembler before the assembly process begins. The assembler then uses the Pre-processor file, to create an Object file.

If required kalasm2 can be configured to output the Preprocessor file (.kpp) by setting the relevant switch when kalasm2 is called, see appendix 1.

Note:

Users familiar with Kalasm (ie the older Matlab version), should note that command names no longer lead with a hash (#) but rather with a period (.), otherwise the commands are the same.

Pre-processor commands are useful for modifying assembly code. For example:

- You can use the `.include` command to fill memory, load configuration registers and separate subroutines from your main code.

Or

- You can use the `.define` command to define constants and aliases for frequently used instruction sequences. The pre-processor replaces each occurrence of the macro reference with a corresponding value or series of instructions.

For more information about the pre-processor command set, see 5.4.

Pre-processor commands begin with a period (.) and end with a carriage return. The period sign must be the first non-whitespace character on the line containing the command.

If the command is longer than one line, use a backslash (\) and a carriage return to continue the command on the next line.

It is important that no characters are inserted between the backslash and the carriage return. Unlike assembly directives, pre-processor commands are case-sensitive and must be lowercase.

For a list of the pre-processor commands, see 5.4

Example:

```
.include "sbc.h"  
.define MAX 100
```

4 Assembler DOS Prompt Interface

This section describes the kalasm2 command-line interface and option (switch) set. Switches control certain aspects of the assembly process. A full list of the options can be found in Appendix A .

To run the assembler from a DOS command prompt, type kalasm2 followed by the arguments (including switches):

```
kalasm2 sourceFile.asm
```

- Where sourceFile.asm is the source file to assemble (including extension)

The assembler supports relative and absolute path names. When you provide an input or output file name as a parameter, use the following guidelines for file names:

- Include the drive letter and path string if the file is not in the current project directory.
- Include the file name extension from each file.

Extension	File Description
.asm	Assembler source file
.kpp	Pre-processor output file, all comments have been removed and ".include" and ".define" have been replaced, this file is always produced whether code assembles or not
.kfst	Output of the assembler – fed on to the linker and is always produced whether code assembles or not
.kobj	Output of the assembler, can be fed directly into the linker in a future call, only produced when code assembles
.klib	Additional code to be incorporated with assembler source code at linking, whilst the code is identical to that in object files, the linker will only include modules from the library if a label is referenced, whilst object files are always included in their entirety
.klo	Linker output file, only produced when the code assembles
.kmap	Linker map file, again this is only produced the code assembles

Table 4.1 File Extension for Kalasm2

The assembler command line switches are case-insensitive. The following command lines, for example:

```
kalasm2 main.asm
kalasm2 MAIN.AsM
```

runs the kalasm2 assembler on the file main.asm and produces the standard output files. The output files will have names: main.klo/kmap.

5 Assembler Syntax Reference

When source programs are assembled, pre-processor commands and assembler directives control the process.

Software developers writing assembly programs should be familiar with the following topics:

- Assembler Keywords and Symbols
- Assembler Expressions
- Assembler and Pre-Processor Operators
- Numeric Formats
- Comment Conventions
- Assembler Directives

5.1.1 Assembler Keywords and Symbols

The assembler supports a set of pre-defined keywords that includes register names, assembly instructions, and assembler directives. Each along with their definition and usage can be found in BlueCore3-Multimedia Kalimba DSP User Guide (bc3-ug-001Pa).

The set of keywords can be extended with symbols that declare sections, variables, constants, and address labels.

When defining symbols in assembly source code, it is recommended that the following conventions are followed:

- Define symbols that are unique within the file in which they are declared.
- Begin symbols with alphabetic characters;

Note:

Symbols beginning with a digit (0-9) are interpreted as a number.

Alphabetic characters (A-Z and a-z), digits (0-9) and the special character '_' (underscore) are all acceptable in names. Symbols are case-sensitive, so `input_addr` and `INPUT_ADDR` define unique variables.

- An underscore '_' as the first two characters of a symbol may have special meaning. Several identifiers are predefined by the pre-processor: '___DATE___' is translated to a string containing the date of assembly
- A dollar '\$' as the first character indicates a global symbol (see 5.2.4).
- Do not use a reserved keyword to define a symbol.
- Use colon (:) to identify address label symbols. Label symbols may appear at the beginning of an instruction line or stand alone on the preceding line.

The following unassociated lines of code demonstrate examples of symbol usage:

```
.BLOCK/DM1 DM1_vars;
  .VAR count;           // count is a 24-bit variable in DM1
  .VAR tempdata;       // tempdata is a 24-bit variable in DM1
.ENDBLOCK;

subroutine_1:          // subroutine_1 is a local label

.MODULE $M.main;      // $M.main is a section in program memory

  $main:               // $main is a global label

.ENDMODULE;

.MODULE $timer;

  .VAR maximum;        // variable available locally as maximum or
                       // globally as $timer.maximum

.ENDMODULE;
```

5.2 Formats and Conventions

5.2.1 Numeric Formats

The kalasm2 assembler supports binary, decimal, hexadecimal, scientific and fractional numeric formats (bases) within expressions and assembly instructions.

Table 5.1 describes the conventions of notation the assembler uses to distinguish between numeric formats. Any symbol beginning with a number is processed as a number; it is an error to declare a label or variable whose first character is a number.

Convention	Description
<i>0xnumber / 0Xnumber</i>	A '0x' or '0X' prefix indicates a hexadecimal number
<i>0bnumber / 0Bnumber</i>	A '0b' or '0B' prefix indicates a binary number
<i>number</i>	No prefix indicates a decimal number
<i>Number.number</i>	A '.' indicates a fractional number as opposed to an integer
<i>numberEnumber / numberEnumber</i>	An 'e' or 'E' indicates a scientific number

Table 5.1 Numeric Formats

5.2.2 Fractional Type Support in Q Format

A fractional constant uses the floating-point representation when a decimal point is specified. Fractionals are represented as signed values in the Q1.23 format (see section 5), which means the values must be greater than or equal to -1.0 and less than 1.0. Any variables initialised with values larger than this will be saturated and the assembler will produce a warning.

Example:

```
// legal fracts
.VAR/DM1 myFracts[4] = 0.5, -0.5e-4, -0.25e-3, 0.875;

// illegal fracts
.VAR/DM1 OutOfRangeFract = 1.5;
```

The conversion formula used is:

$$\text{FracValue} = \text{round}(\text{DoubleValue} \times 2^{23})$$

Equation 5.1 Fractional Type Conversion Formula

5.2.3 Comment Conventions

The assembler supports C and C++-style comments in assembly code. Table 5.2 describes the kalasm2 comment formats.

Convention	Description
<i>/* comment */</i>	A "/* */" string encloses a multiple-line or partial line comment.
<i>// comment</i>	A pair of slashes "//" denotes a single-line comment.
Note: the assembler will allow nested comments.	

Table 5.2 Comment Conventions

5.2.4 Global Symbols

A symbol may be declared as global by beginning its name with a dollar sign '\$'.

As with local variables, they must be declared from within a module but may be accessed from any module simply by referring to their name including the '\$'.

Global constants are different to other entities, as they can be declared outside of a module. Also they can be defined multiple times, so long as they are defined as the same value each time.

Example:

```
.VAR $CosLUT[100] = "cosine.dat";  
i0 = &$CosLUT; r0 = M[$CosLUT];
```

Note:

Global variables may be referenced before they are declared.

Alternatively Local symbols can be accessed globally by preceding their name with the name of the module that they are declared in followed by a dot ".".

For example:

```
.MODULE $Decoder;  
  // ..  
  
  .VAR/DM1 state;  
  
  reset_decoder:  
  
  // ..  
  
.ENDMODULE;  
  
.MODULE $M.another_module;  
  // ..  
  
  // access a local variable from module $Decoder  
  r0 = M[$Decoder.state];  
  
  // use a local address label from module $Decoder  
  call $Decoder.reset_decoder;  
  
  // ..  
  
.ENDMODULE;
```

5.3 Special Assembler Operators

5.3.1 LENGTH Operator

The “length of” operator can be used to obtain the length of a declared variable.

`LENGTH(symbol)` - Length of symbol in words

The following example demonstrates how the LENGTH operator is used to load L (length) and I (index) registers when setting up circular buffers:

```
.define n 10

MODULE $M.program;           // main code section
.CODESEGMENT PM;
.DATASEGMENT DM;
$program:

.VAR/DMCIRC real_data[n];    // n = number of input samples

I5 = &real_data;             // buffer's base address
L5 = LENGTH(real_data);      // buffer's length

M4 = 1;                       // post-modify I5 by 1
r10 = LENGTH(real_data);     // set loop counter to n
DO loop1;
    r0 = M[I5,M4];           // get next sample
    ...                       // do something with it
loop1:
...

.ENDMODULE;
```

This code fragment initialises I5 (index) and L5 to the base address and length of the circular buffer `real_data`. The buffer length value contained in L5 determines when addressing wraps around to the top of the buffer.

5.3.2 BITREVERSE Operator

The BITREVERSE operator aids in processing the output of algorithms such as the fast fourier transform (FFT).

When the bit reverse flag is set, bit reversed addressing is activated on Address Generator 1. This causes the output of Address Generator 1 (linked to index registers I0-3) to be bit reversed before its value is driven onto the address bus.

As a result, for correct operation, the index registers should be loaded with the bit reversed value of the starting address of the buffer; this is achieved using the BITREVERSE operator.

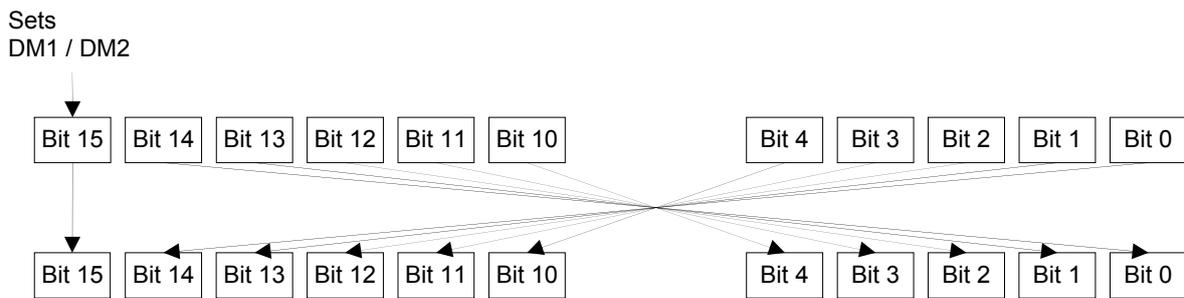
Example of use:

```
I3 = BITREVERSE(&$Inputreal);
```

The operation is performed by the assembler when the files are being assembled, it is not available during code execution. Consequently, the following code usage is illegal:

```
// Illegal code!
I3 = BITREVERSE(r1);
```

The bit reverse operation performed by this command is intended to be used to generate bit reversed addresses; it therefore must not change the MSB as this determines which memory bank the address applies to. Bit reversed addresses follow this model:



5.4 Pre-Processor Commands

This section provides reference information about the kalasm2 pre-processor commands, including syntax and usage examples. Pre-processor 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

A detailed description of each command follows Table 5.3.

Command/Operator	Description
<code>.include</code>	Includes the contents of a file
<code>.define</code>	Defines a macro
<code>defined</code>	Checks if a symbol has been defined
<code>.undef</code>	Removes macro definition
<code>.if</code>	Begins an <code>.if/.endif</code> pair
<code>.elif</code>	Sub-divides an <code>.if/.endif</code> pair
<code>.else</code>	Identifies alternative instructions within a <code>.if/.endif</code> pair
<code>.endif</code>	Ends an <code>.if/.endif</code> pair
<code>.ifdef</code>	Begins an <code>.ifdef/.endif</code> pair and tests if macro is defined
<code>.elifdef</code>	Sub-divides an <code>.ifdef/.endif</code> pair
<code>.ifndef</code>	Begins an <code>.ifndef/.endif</code> pair and tests if macro is defined
<code>.elifndef</code>	Sub-divides an <code>.ifndef/.endif</code> pair
<code>.message</code>	Displays a user defined message
<code>.warning</code>	Reports a warning message
<code>.error</code>	Reports an error message
<code>.line</code>	Allows the user to set the file name for errors and messages
<code>.file</code>	Allows the user to set the file name for errors and messages

Table 5.3 Pre-Processor Commands

5.4.1 .include

The `.include` command directs the pre-processor to insert the text from a file at the command location. The pre-processor searches for the file in the order:

1. The current directory,
2. The directories you specify using the `-I` command,
3. The standard list of system directories.

Syntax:

```
.include <fileName>           //include a system file
.include "fileName"          //include a user file

.include macroFileNameExpansion

// Include a file named through macro expansion.
// This command directs the pre-processor to expand the
// macro. The pre-processor processes the expanded text,
// which must match either <fileName> or "fileName ".
```

Example:

```
.ifndef $USE_FFT
    // if USE_FFT has been defined include fft.h
    .include "math/fft.h"
.endif
```

5.4.2 .define

The `.define` command has two functions:

- Defining symbolic constants
- Defining macros.

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

Note:

Defining this type of macro has the same effect as using the Find/Replace feature of a text editor, although it does not replace literals in double quotation marks ("").

When you define a macro in your source code, the pre-processor replaces all subsequent occurrences of the macro reference with its definition.

For multi-statement macro definitions, terminate all the statements or all but the last statement with a semicolon (;).

Note:

It is a 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 termination semicolon.

Arguments can be added to the macro definition. Simply follow the name of the macro by brackets containing a comma separated list of arguments.

Syntax:

```
.define macroSymbol replacementText
```

Where:

- *macroSymbol* - Macro identifying symbol.
- *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 = r1
```

5.4.3 `defined`

This command is slightly different from the other pre-processor commands; it does not begin with a period (.) and is used to check if an item has been defined already.

It is usually used with the `.if` command to perform a similar operation to the `.ifdef` command. The advantage of this command is the ability to test multiple items:

Check if `$BUFFER_SIZE` is defined

```
.if defined ($BUFFER_SIZE)
...
.endif
```

This is the same as

```
.ifdef $BUFFER_SIZE
...
.endif
```

However to check if `$BUFFER_SIZE_MAX` and `$BUFFER_SIZE_MIN` are defined

```
.if defined ($BUFFER_SIZE_MAX) && defined ($BUFFER_SIZE_MIN)
...
.endif
```

5.4.4 `.undef`

The `.undef` command directs the pre-processor to undefine a macro.

Syntax:

```
.undef macroSymbol
```

Where:

- `macroSymbol` - macro created with the `.define` command.

Example:

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

5.4.5 .if

The `.if` command begins a `.if/.endif` pair. Statements inside a `.if/.endif` pair can include other pre-processor commands and conditional expressions.

The pre-processor processes instructions inside the `.if/.endif` pair only when the condition that follows the `.if` evaluates as TRUE.

The number of `.if` commands must equal the number of `.endif` commands.

Syntax:

```
.if condition
```

Where:

- `condition` - Expression to evaluate as TRUE (non-zero) or FALSE (zero). The conditional expression can include the `defined ()` operator, as in:

```
.if defined (SBC_DECODER)
```

which has the same meaning as:

```
.ifdef SBC_DECODER.
```

The advantage of the first form is that it can include multiple conditional operands:

```
.if defined (SBC_DECODER) || defined (SBC_ENCODER).
```

Example:

```
.if x!=100 //test for TRUE condition
...
// The pre-processor executes instructions
// after .if only when x != 100.
.endif
```

5.4.6 .elif

The `.elif` command (else if) is used within an `.if/.endif` pair.

The `.elif` includes an alternative condition to test when the initial `.if` condition evaluates as FALSE.

The pre-processor tests each `.elif` condition inside the `.if/.endif` pair and processes instructions that follow the first true `.elif`.

Note:

You can have an unlimited number of `.elif` commands inside one `.if/.end` pair.

The use of this combination parallels the use of the "switch case" command available in many other languages.

Syntax:

```
.elif condition
```

Where:

- `condition` - Expression to evaluate as TRUE (non-zero) or FALSE (zero).

Example:

```
.if X == 1
    // The pre-processor will execute these
    // instructions when X == 1.
    ...
.elif X == 2
    // The pre-processor executes instructions
    // following .elif when x != 1 and x == 2.
    ...
.else
    // The pre-processor will execute these
    // commands when X != 1 and X != 2.
    ...
.endif
```

5.4.7 .else

The `.else` command is used within a `.if/.endif` pair and adds an alternative instruction to the `.if/.endif` pair.

Note:

You can use only one `.else` command inside an `.if/.endif` pair.

The pre-processor executes instructions that follow `.else` after all the preceding conditions are evaluated as FALSE (zero).

If no `.else` text is specified, and all preceding `.if` and `.elif` conditions are FALSE, the pre-processor does not process any instructions inside the `.if/.endif` pair.

Syntax:

```
.else
```

Example:

```
.if X == 1
    // The pre-processor will execute these
    // instructions when X == 1.
    ...
.elif X == 2
    // The pre-processor executes instructions
    // following .elif when x != 1 and x == 2.
    ...
.else
    // The pre-processor will execute these
    // commands when X != 1 and X != 2.
    ...
.endif
```

5.4.8 .endif

The `.endif` command is required to terminate `.if/.endif`, `.ifdef/.endif`, and `.ifndef/.endif` pairs.

The number of `.endif` commands must match the number of `.if` commands.

Syntax:

```
.endif
```

Example:

```
.if condition
    // The pre-processor executes instructions after .if
    // when condition evaluates as TRUE.
    ...
.endif

// These instructions are evaluated whether the condition
// evaluates to TRUE or FALSE.

...
```

5.4.9 .ifdef

The `.ifdef` (if defined) command begins an `.ifdef / .endif` pair and commands the pre-processor to test whether a macro is defined. The pre-processor considers a macro defined if it has a non-zero value.

The number of `.ifdef` commands must match the number of `.endif` commands.

Syntax:

```
.ifdef macroSymbol
```

Where:

- `macroSymbol` - Macro created with the `.define` command. The conditional expression can include the `defined()` operator, as in:

```
.if defined (SBC_DECODER)
```

which has the same meaning as:

```
.ifdef SBC_DECODER.
```

The advantage of the first form is that it can include multiple conditional operands as in:

```
.if defined (SBC_DECODER) || defined (SBC_ENCODER).
```

Example:

```
.ifdef SBC_DECODER
  // Tests if SBC_DECODER has been defined
  ...
.endif
```

5.4.10 .elifdef

The `.elifdef` command (else if defined) is used within an `.ifdef / .endif` pair.

The `.elifdef` includes an alternative condition to test when the initial `.ifdef` condition evaluates as FALSE.

The pre-processor tests each `.elifdef` condition inside the `.ifdef / .endif` pair and processes instructions that follow the first true `.elifdef`.

Note:

You can have an unlimited number of `.elifdef` commands inside one `.ifdef / .end` pair.

The use of this combination parallels the use of the "switch case" command available in many other languages.

Syntax:

```
.elifdef condition
```

Where:

- `condition` - Expression to evaluate as TRUE (non-zero) or FALSE (zero).

Example:

```
.ifdef $CODEC_EXT_RIGHT
    // The pre-processor will execute these instructions
    // if $CODEC_EXT_RIGHT is defined.
    ...

.elifdef $CODEC_EXT_LEFT
    // The pre-processor executes instructions following
    // .elifdef if $CODEC_EXT_LEFT is defined.
    ...

.else
    // The pre-processor will execute these commands if
    // neither $CODEC_EXT_RIGHT or $CODEC_EXT_LEFT are defined.
    ...

.endif
```

5.4.11 .ifndef

The `.ifndef` command (if not defined) begins an `.ifndef / .endif` pair and directs the pre-processor to test for an undefined macro. The pre-processor considers a macro undefined if it has no defined value or has a value defined as zero.

Note:

If a macro is not assigned a value when it is declared the default value (True / 1) is applied.

The number of `.ifndef` commands must equal the number of `.endif` commands.

Syntax:

```
.ifndef macroSymbol
```

Where:

- `macroSymbol` - Macro created with the `.define` command.

Example:

```
.ifndef SBC_DECODER
    // tests for SBC_DECODER code and defines it if it is
    // not already defined.
    .define SBC_DECODER
.endif
```

5.4.12 .elifndef

This command operates exactly as `.elifdef`, however instructions are performed if the item in question is not defined.

Syntax:

```
.elifndef condition
```

Where:

- `condition` - Expression to evaluate as TRUE (non-zero) or FALSE (zero).

Example:

```
.ifndef $CODEC_EXT_RIGHT
    // The pre-processor will execute these instructions
    // if $CODEC_EXT_RIGHT is NOT defined.
    ...
.elifndef $CODEC_EXT_LEFT
    // The pre-processor executes instructions following
    // .elifdef if $CODEC_EXT_LEFT is NOT defined.
    ...
.else
    // The pre-processor will execute these commands if neither
    // $CODEC_EXT_RIGHT nor $CODEC_EXT_LEFT are NOT defined.
    ...
.endif
```

5.4.13 .message

The `.message` command is the first of three commands that allow the user to send a message to the screen at assembly. When the assembler reaches this point in the code, the message is sent and the assembler continues.

Syntax:

```
.message messageText
```

Where:

- `messageText` - User-defined text. To break a long `messageText` without changing its meaning, place the backslash character (`\`) at the end of each line except the last.

Example:

```
.message \  
MyMessage:\  
This point in the code has been reached
```

Output:

```
MESSAGE: fileName lineNo: MyMessage:This point in the code has been  
reached
```

5.4.14 .warning

The `.warning` command is the second command available to send a message. Its response is stronger than the message command, as the name suggests this generates a warning on the screen. The pre processor uses the text following the `.warning` command as the warning message, and then as in, the previous case continues to assemble the file.

Syntax:

```
.warning messageText
```

Where:

- `messageText` - user-defined text. To break a long `messageText` without changing its meaning, place the backslash character (`\`) at the end of each line except the last.

Example:

```
.ifndef SBC_DECODER  
  .warning\  
  MyWarning:\  
  Expecting a SBC_DECODER.\  
  Check the Linker Description File!  
.endif
```

Output:

```
WARNING: fileName lineNo: MyWarning: Expecting a SBC_DECODER. Check  
the Linker Description File!
```

5.4.15 .error

The `.error` command is the final command in the set of three. Its action is stronger than the warning command as it halts any further assembler action. The pre-processor uses the text following the error command as the error message.

Syntax:

```
.error messageText
```

Where:

- `messageText` - User-defined text. To break a long `messageText` without changing its meaning, place the backslash character (`\`) at the end of each line except the last.

Example:

```
.ifndef SBC_DECODER
.error \
MyError:\
Expecting an SBC_DECODER.\
Check the Linker_Description File!
.endif
```

Output:

```
ERROR:   fileName lineNo: MyError: Expecting an SBC_DECODER. Check
the Linker Description File!
```

5.4.16 .line

If any of the above message commands are used the line number displayed in the output can be set using this command. If used this command will override the actual line number of the file.

Note:

This will only affect commands after it is called and not any messages before.

Syntax:

```
.LINE number
```

Where:

- `number` - User-defined line number.

Example:

```
.FILE "thisFile"
.LINE 10

.ifndef SBC_DECODER
.error You have not defined SBC_DECODER.
.endif
```

Displays the following at assembly:

```
ERROR:   thisFile 12: You have not defined SBC_DECODER.
```

5.4.17 .file

If any of the above message commands are used the file name displayed in the output can be set using this command. If used this command will override the actual name of the file. This command can also be used with the .line command.

Note:

This will only affect commands after it is called and not any messages before.

Syntax:

```
.FILE "fileName"
```

Where:

- fileName - User-defined file name this needs to be included in quotations.

Example:

```
.FILE "thisFile"

.ifndef SBC_DECODER
    .error You have not defined SBC_DECODER.
.endif
```

Displays the following at assembly:

```
ERROR:   thisFile 04:  You have not defined SBC_DECODER.
```

5.5 Assembler Directives

Directives in an assembly source file control the assembly process. Unlike instructions, directives do not produce code that the DSP runs.

The following is a general syntax for assembler directives:

```
.directive [/qualifiers | arguments];
```

Each assembler directive starts with a period (.) and ends with a semicolon (;). Directives can where required include qualifiers and arguments.

A directive's qualifier immediately follows the directive and is separated by a slash (/); arguments follow qualifiers.

Whilst the assembler allows directives in upper and lower case, by convention Assembler directives are always uppercase to help distinguish them from other symbols in the source code.

Supported directives are shown in Table 5.4.

A description of each directive appears in the following sections.

Directive	Description
.BLOCK .ENDBLOCK	Allows variables to be grouped into a single contiguous area of memory.
.VAR	Allows declaration of variables
.CONST	Allows declaration of constants
.MODULE .ENDMODULE	Allows sectioning of assembler code. A program consists of one or more modules that may be manually or automatically allocated in memory
.CODESEGMENT	Defines where operation codes are to be stored in Kalimba
.DATASEGMENT	Defines where data words are to be stored in Kalimba

Table 5.4 Assembler Directives

5.5.1 .VAR, Declare a Variable

A single .VAR directive can declare and optionally initialise a variable. A variable may be a single data memory location or an array of contiguous locations.

The .VAR directive takes one of the following forms:

```
.VAR/type      varName;
.VAR/type      varName = initExpr;
.VAR/type      varName [length] = initExpr1 ,initExpr2 ,...;
.VAR/type      varName [length] = "fileName.dat";
```

Where:

- /type keyword maps a variable into the DSP data memory as shown in Table 5.5.

Note:

If no type is specified the variable is allocated to the default data segment, as specified by the .DATASEGMENT command, see 5.5.

- User-defined varName symbol identifies the variable.

Note:

It is recommended that the name does not begin with two underscores (__). To define a global variable prefix the name with a dollar (\$), but this is the only occasion where the symbol may be used in a variable name.

- The optional [length] parameter defines the length of the associated variable in words. If no length is given, the variable is created with length 1.
- The initExpr parameters set initial values for variables and buffer elements. They may be the address of other variables, numbers or an initialisation file and should be presented as a list of comma (,) separated items terminated with a semi-colon (;).
- The fileName.dat parameter indicates that the elements of a buffer are initialised from an external data file.

The name must be enclosed in quotations (").

This parameter is useful for loading filter coefficient data into a buffer. If the initialisation file is in the current project directory, only the filename needs to be given.

Otherwise, specify the directory and the name of the initialisation file, absolute or relative paths are acceptable.

Note:

The length must be specified explicitly for this option

- When formatting external files used to initialise variables it is useful to consider the data file as an extension of kalimba code ie separate items in lists with commas (,) and terminate the list with a semi-colon (;).

Note:

The terminator may be included in either the program or the external file, but not both.

To maintain consistency it is recommended that the last element in a data file is not terminated; rather the terminator is applied by the calling routine.

When declaring or initialising variables, if the number of initial values is less than the size of the variable the remaining locations are un-initialised. If the variable is smaller than the number of initial values, the assembler will produce an error message.

The following lines of code demonstrate some .VAR directives:

```
.VAR/DM1      var2;
.VAR/DM2      var3;
.VAR/DM       var4;
.VAR/DM1      lstate1 = 0x1000;
.VAR/DM2      lstate2 [3] = 0x2000,0x2001,var2;
.VAR/DMCIRC   Mstate = 0x1000;
.VAR          Ostate [3] = 0x2000,0x2001,var2;
.VAR          state1 = 0x1000;
.VAR          Xstate1 = 0x1000 ;
.VAR          state2 [3]= 0x2000,0x2001,var2;
.VAR          state3 [10] = file.dat;
.VAR          Xstate3 [10]= file.dat;
.VAR          Qstate = 0x1000 ;
.VAR          Rstate [10] = file.dat,
.VAR          Sstate2 [3];
.VAR          Tstate2 [3] = var2,var3,var4;
```

5.5.2 .CONST, Declare a Constant

The .CONST directive declares a constant.

The directive follows the following syntax for declaring a constant value:

```
.CONST NUM_FIR_COEFS 51;
.CONST NUM_SAMPLES 8000;
.VAR/DM1CIRC filter_coefs [NUM_FIR_COEFS] = fir_coefs.dat;
.VAR/DM1CIRC delayline [NUM_FIR_COEFS];
.VAR          inoutdata [NUM_SAMPLES] = inputdata.dat;
```

The .CONST directive can also be used to declare a constant address value with the following syntax:

```
.CONST $A_FIXED_ADDRESS          0xFFFFF00;
```

Unlike global variables, global constants may not be referenced before they have been declared. Global constants can however be declared outside of a module, and can be declared several times so long as they are declared to the same value.

5.5.3 .BLOCK & .ENDBLOCK, Declare a variable grouping

The `.BLOCK` directive marks the beginning of a logical section that mirrors an array of contiguous locations in data memory.

Statements between `.BLOCK` and the following `.ENDBLOCK` directive comprise the contents of the section.

Sections may only contain variable declarations.

The main uses of `.BLOCK` and `.ENDBLOCK` is:

- To group variables into contiguous locations so that look up tables may be reused efficiently
- Define private areas of memory that contain global variables
- Partially initialise an area of memory

The Linker Description File may be used to define how sections, and modules, are placed in the DSP program and data memory.

The directive follows the following syntax:

```
.BLOCK/type sectionName;
    < variable definitions with or without initialisation values >
.ENDBLOCK;

.BLOCK/type sectionName[length] = <initialisation file>;
    < variable definitions without initialisation values >
.ENDBLOCK;

.BLOCK/type sectionName = <initialisation values>;
    < variable definitions without initialisation values >
.ENDBLOCK;
```

Where:

- `/type` keyword maps a section into the DSP data memory.

The type defines the memory location (DM1 / DM2) of all of the variables declared within the section. However if the block is defined as circular only the first variable can be guaranteed to be circular.

One of the types from Table 5.5 may be specified for each `.BLOCK` directive.

Note: these values in this table are dependant on the definitions in the "default.asm" file, and is correct for the standard file.

- `blockName` a name to identify the section.

Type	Description
DM	Either of the data memories, the decision of whether to use DM1 or DM2 is left to the linker, this option may be omitted
DM1	Section is explicitly data memory 1
DM2	Section is explicitly data memory 2
DMCIRC	Allocates the section to a location consistent with the location of a Kalimba circular buffer
DM1CIRC	Allocates the section to a location consistent with the location of a Kalimba circular buffer in data memory 1
DM2CIRC	Allocates the section to a location consistent with the location of a Kalimba circular buffer in data memory 2

Table 5.5 Example Memory Type Section

Blocks may be defined as global variables as may the variables declared within them. Blocks may be declared to be circular buffers.

The following lines of code demonstrate some .BLOCK directives:

```
.BLOCK $CosLUT;
  .VAR CosLUT[256] = cosinelut.dat;
  .VAR $SinLUT[256] = sinelut.dat;
.ENDBLOCK;

.BLOCK Private1;
  .VAR UninitialisedVariable;
  .VAR $InitialisedVariable[6] = 0.11,0.11,0.22,0.22,0.11,0.11;
.ENDBLOCK;

.BLOCK/DMCIRC LUT1[256]=lut.dat;
  .VAR dummy[64];
  .VAR $LUTPtr1[128];
  .VAR $LUTPtr2[64];
.ENDBLOCK;
```

Important:

As their location within memory is important, circular buffers cannot be defined within each other, this means it is not possible to have a block declared as a circular buffer and then variables within it also defined as circular buffers. Obviously the first variable will be circular.

5.5.4 .MODULE & .ENDMODULE, Declare a Program Module

The basic unit of DSP assembly code is a module; every assembler file must begin with a `.MODULE` declaration and end with a `.ENDMODULE` declaration. Modules are separately assembled and then linked together.

The `.MODULE` directive takes one of the following forms.

```
.MODULE moduleName;
```

Where:

1. `moduleName` a name to identify the module.

Note:

A module name must begin with a '\$' since all modules are global, this allows routines to use variables and labels local to a module from outside that module.

In the early Matlab assembler, Kalasm, modules could be called by their `moduleName`, because the `moduleName` was also a label which located the start of the module. In `kalasm2` the `moduleName` no longer functions as a label, therefore a label has to be included at the start of every module to allow the module to be called.

A possible naming convention is therefore to prefix all module names with "M." and prefix this with a dollar (\$), then to include the module name as a label in the code.

For example, consider this simple section of code, which defines a constant and adds it to a register storing the result in a second:

```
00  .MODULE $M.addconst2register;
01  .CODESEGMENT PM;
02  .DATASEGMENT DM;
03  .CONST  CONST_TO_ADD_TO_REG 12;
04  $addconst2register:

05  r1 = CONST_TO_ADD_TO_REG;
06  r2 = r0 + r1;

07  .ENDMODULE;
```

Clearly, the constant is defined on line 03 and loaded into register `r1` on line 05.

This is then added to `r0` and stored in `r2` on 06.

The module is defined as `$M.addconst2register` however it cannot be called by another function using this name, hence the label `$addconst2register` on line 04.

As long as a global label is included at the start of the module the name of the module is irrelevant. However, if the global label is omitted the function cannot be called or jumped to.

For convenience it is desirable that the function should be sensibly named. The nomenclature below achieves this and helps to avoid confusion.

```
00  .MODULE $M.moduleName;

01  .CODESEGMENT PM;
02  .DATASEGMENT DM;

03  $moduleName:
```

5.5.5 .DATASEGMENT and .CODESEGMENT

These commands define the default area for code and data. The segment specified here is the location used when one is not explicitly provided. A declaration is required for each module; this provides a means to control the order in which the code is placed in program memory:

```
.MODULE $M.main;
  .CODESEGMENT PM;           // defaults all code to program memory
  .DATASEGMENT DM;          // defaults all data to data memory
main:

  .VAR    audio_left_in;    // uses the default DM segment
  .VAR/DM audio_left_out;   // uses DM, the default segment
  .VAR/DM1 audio_right_in;  // uses DM1 segment
  .VAR/DM2 audio_right_out; // uses DM2 segment

  ...

.ENDMODULE;

.MODULE $M.decode;
  .CODESEGMENT PM 1;        // defaults all code to location PM_1
  .DATASEGMENT DM1;        // defaults all data to bank 1
decode:

  .VAR    stream_no;        // both will be placed in the default
  .VAR    start_time;       // location data memory 1

  ...

.ENDMODULE;
```

The values you use as defaults must have already been defined in the file `default.asm`. In this file you specify the order the data is located in memory, which bank of memory it is placed in – PM, DM1 or DM2 – and for data whether the variable must be circular or not. See the file provided in your BlueLab installation as an example.

Note:

The allocation of code in program memory is not arbitrary. Two routines have specified locations, firstly the routine `$reset` must be at location 0, and the routine `$interrupt_handler` must be at location 2. This is achieved by defining two segments with link orders 1 and 2. All subsequent segments have a link order greater than 2. This means the linker will place these at the start of program memory and then continue with the remaining code there after. If you wish to modify the program memory segments, ensure these two segments remain the highest priority segments.

The location of data memory can be very important. For example when processing stereo data it is often useful to place one buffer in DM1 and the second buffer in DM2. This allows an element from each buffer to be read in one instruction without causing a hardware stall.

6 Accompanying Tools Provided with kalasm2

6.1 The Kalimba Packer Utility (kalpac2)

The Kalimba packer utility generates a KAP file from the Linker Output file (.KLO) generated by kalasm2.

When called Kalpac2 will generate a KAP file with the same name as the input .KLO file in a directory of the same name.

Syntax:

```
Kalpac2 fileName[.klo]
```

Where:

- `fileName.klo` – input .KLO file, which may be called with or without the .KLO extension attached.

This will produce a .KAP file:

```
\fileName\fileName.kap
```

Optionally, Kalpac2 may be called using the option

```
Kalpac2 fileName[.klo] directoryName
```

Where:

- `fileName.klo` – input .KLO file, which may be called with or without the .KLO extension attached.
- `directoryName` – the name of the directory the file will be produced in

This will produce a .KAP file:

```
\directoryName\fileName.kap
```

In order to program the DSP the assembled code must be available as a .KAP file.

When kalasm2 is run in the BlueLab xIDE application kalpac2 is called as part of the build script and the .kap file is produced automatically.

7 Some Background on DSPs & Number Representation

In general, DSP architecture needs to handle numbers with precision, whether the architecture is floating-point or fixed-point. A typical microcontroller (MCU) is typically involved with more control-orientated tasks that involve turning on and off outputs and reading inputs. Conversely a DSP tends to be involved with the manipulation of 'real world' signal values that come in from an analogue to digital converter (ADC) and output through a digital to analogue converter (DAC). This manipulation often involves a series of mathematical computations typically based on multiplies and addition/accumulates (MAC).

The BlueCore3-Multimedia contains the Kalimba DSP core that is based on a 24-bit fixed-point DSP architecture. The precision found in fixed-point DSP architectures is limited compared to floating-point devices, but fixed-point devices offer superior performance for cost and size. The overhead of extra real estate on the die for the floating-point functionality is significant and in the majority of applications, the precision available on fixed-point devices is more than acceptable.

A more detailed coverage of this area is available in the BlueCore3-Multimedia Kalimba DSP User Guide (bc3-ug-001Pa), where the number representation system for the following is defined:

- Binary integer representation
- Binary fractional representation
- Integer multiplication
- Fractional multiplication

7.1 Signed Fractional Numbers

As mentioned BlueCore3-Multimedia is a fixed point DSP, consequently a format is required to represent fractional numbers.

The format used is Q.23 Format, a variant of Q Format. This uses the most significant bit (MSB) to represent the sign of the number and to fix the radix point, with the following 23 bits setting the value:

Bit	23	22	21	20	19	18	...	1	0
Number	1	0	1	0	1	1	...	0	0
Bit Weights	-2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	...	2^{-22}	2^{-23}
Results	-1	0	0.25	0	0.0625	0.03125	...	0	0

Radix point

Example

$$\begin{aligned}
 1010\ 1100\ 0000\ 0000\ 0000\ 0000\ (\text{binary}) &= -2^0 + 2^{-2} + 2^{-4} + 2^{-5} \\
 &= -1 + 0.25 + 0.0625 + 0.03125 \\
 &= -0.65625
 \end{aligned}$$

Equation 7.1 Q.23 Number format

In Q Format the position of the radix point determines the range of numbers the format can represent. In this variant with one bit to the left of the radix point, the following range of numbers can be represented:

	Lower	Upper
Binary	0b 1111 1111 1111 1111 1111 1111	0b 0111 1111 1111 1111 1111 1111
Hexadecimal	0x FF FF FF	0x 7F FF FF
Integer	-8388608	8388607
Rational	-1	$+1 - 2^{-23}$
Decimal	-1.000...0	+0.99999988079071

7.2 Some Implications of Q Format Representations

As this is a limited precision device, some limitations have to be considered when using the Kalimba DSP:

7.2.1 Interpretation

The Arithmetic Logic Unit (ALU) in Kalimba is completely oblivious to any formatting used to represent values within Kalimba, it is the developer's responsibility to take into account the scaling applied to numbers.

	0x1	0x2
Integer	1	2
Fractional	2^{-23}	2^{-22}

Consider the following:

Decimal: $2 + 2^{-23} \approx 2$

Convert to Kalimba representation
 $2 \rightarrow 0x2$
 $2^{-23} \rightarrow 0x1$

Kalimba ALU: $0x2 + 0x1 = 0x3$

The ALU will always perform the same addition, however there are two options for multiplications, either fractional multiply where both numbers are assumed to be fractions or integer where numbers are treated as integers.

Loading values into registers is generally unambiguous; however there are two numbers which may cause problems. This is because they are valid numbers in both nomenclatures: +1 and -1. To overcome this, kalasm2 uses the following syntax:

<code>r1 = 1</code>	<code>→ 0x00 00 01</code>	<code>r1 = -1</code>	<code>→ 0xFF FF FF</code>
<code>r1 = 1.0</code>	<code>→ 0x7F FF FF</code>	<code>r1 = -1.0</code>	<code>→ 0x70 00 00</code>

7.2.2 Accuracy of 1

Kalimba cannot accurately represent 1.0 as a fractional value, examining the upper limit specified above it is clear this is less than one. The reason for this is obvious when stepping through the number gap $-1 \rightarrow +1$ with a step size of 2^{-23} there are actually $2^{24}+1$ values. Kalimba only has 224 values it can use, so the numbers $(1-2^{-23})$ and 1.0 are both stored as $0x7FF\ FF\ FF$, with the value of $(1-2^{-23})$ used in calculations, hence the following result:

$$r1 = 1.0; \quad r2 = -1.0; \quad r3 = r1 + r2; \quad r3 = -2^{-23} \neq 0$$

This is a minute error in the calculation and typically much larger errors generated elsewhere will dominate, however, it should still be noted.

7.2.3 Out of Range Numbers

The limit of fractional numbers is $-1.0 \rightarrow +1.0$, however not all values for all calculations lie within this range, for example how to perform the multiplication: 1.5×0.5 ?

The solution is to use scaling. Since the values which may be represented using fractional notation, must be in the range of ± 1 , larger numbers must be scaled to fit within this range.

This can be done by performing a calculation to rescale the result.

To do this the maximum/minimum attainable values for a particular variable must be determined and then a scaling calculation applied to ensure that any possible results will fall within the range of ± 1 .

Consider the case of summing a sine and cosine wave:

$$\sin(\omega x) + 1.7 \cdot \cos(\omega x) = 1.97 \cdot \sin(\omega x + \theta)$$

In the above example the results returned would be in the range of ± 1.97 . Since the result passed to the DSP must not exceed ± 1 some form of scaling would be needed.

In this case, halving the values before they are passed to the DSP may be an appropriate solution.

In some circumstances, it may then be necessary or desirable to rescale the DSP output to their original .values

8 Document References

Document	Reference
BlueCore3-Multimedia Kalimba DSP User Guide	Bc3-ug-001Pa, a, July 2003
xIDE Debugger User Guide	Bcore-ug-011Pa
BlueCore3-Multimedia Data Sheet	BC358239A-ds-001Pb

Document	Reference
BlueCore3-Multimedia Kalimba DSP User Guide	Bc3-ug-001Pa, a, July 2003
xIDE Debugger User Guide	Bcore-ug-011Pa
BlueCore3-Multimedia Data Sheet	BC358239A-ds-001Pb

Appendix A kalasm2 switches

```
C:\BlueLab\tools\bin>kalasm2 -h
KALASM Copyright (C) CSR 2004
Created NO VERSION CONTROL
Usage: KALASM <options and inputs files>
where input files may be:
    xxx.asm ASM file
    xxx.kobj object file
    xxx.klib library file
where options may be:
    -?                Display this message
    -h                Display this message
    --               Stop processing switches
    -c                Do not link
    -Fi<filename>    Include arguments from file
    -Fp[<filename>]  Define the name of the preprocessor output file
    -Fo<filename>    Define the name of the object file
    -Fg<filename>    Define the name of the group file
    -Fd<filename>    Define the name of the default file
    -Fl[<filename>]  Define the name of the listing file
    -Fe<filename>    Define the name of the linker output file
    -Fm[<filename>]  Define the name of the linker map file
    -I<path name>    Add path name to the include file search path
    -Ta<filename>    Process file as ASM file
    -To<filename>    Process file as object file
    -Tl<filename>    Process file as library file
    -Po              Run the preprocessor only
    -Pl              Preprocessor output file includes .line & .file
    -nc              Do not generate error context with error messages
    -nd              Do not generate debug information in the output file
    -ec<number>     Set maximum error count
    -W               Change warnings into error
    -D<name>[=<value>] Define a preprocessor symbol
    -&              Do not warn if an & is missed from an address
```

Terms and Definitions

BlueCore™	Group term for CSR's range of Bluetooth wireless technology chips
Bluetooth®	Set of technologies providing audio and data transfer over short-range radio connections
Bluetooth SIG	Bluetooth Special Interest Group
CSR	Cambridge Silicon Radio
DSP	Digital Signal Processor
Kalimba	A DSP core architecture designed by CSR
kalasm2	Kalimba assembler

Document History

Revision	Date	History
a	01 SEP 03	Original publication of this document. (CSR reference: bc3-ug-002Pa)
b	14 OCT 03	New section 4 added to explain accompanying tools
c	26 JAN 04	File names brought into line with BlueLab Multimedia e.g. .APP becomes .KAP
d	08 MAR 05	Important Note: Update of kalasm to Kalasm2 in line with BlueLab3 update

BlueLab™

Kalimba DSP Assembler

User Guide

bc3-ug-002Pd

May 2005

Unless otherwise stated, words and logos marked with ™ or ® are trademarks registered or owned by Cambridge Silicon Radio Limited or its affiliates. Bluetooth® and the Bluetooth logos are trademarks owned by Bluetooth SIG, Inc. and licensed to CSR. Other products, services and names used in this document may have been trademarked by their respective owners.

The publication of this information does not imply that any license is granted under any patent or other rights owned by Cambridge Silicon Radio Limited.

CSR reserves the right to make technical changes to its products as part of its development programme.

While every care has been taken to ensure the accuracy of the contents of this document, CSR cannot accept responsibility for any errors.

CSR's products are not authorised for use in life-support or safety-critical applications.