CSR Kalimba C User's Guide

Preston Gurd
Archelon Inc.
460 Forestlawn Road
Waterloo, Ontario, Canada
N2K 2J6

Table of Contents

Revision History

Revision of May 1, 2009

The section on Binding Variables to Registers now correctly refers to the AG2 length registers as `L4` and `L5`, instead of incorrectly referring to them as `L2` and `L3`.

Revision of June 1, 2007

This is the first version of this document.

Introduction

This document is a User's Guide to the C compiler for the CSR Kalimba 24 bit DSP. This document assumes that you know C and that you know the Kalimba instruction set.

For full information about the processor, please see CSR's *Kalimba DSP User Guide*.

The Archelon Kalimba C compiler generates code for the subset of the architecture that generally provides all of the functions of a general purpose processor. It does not generate any code that uses any conditional prefixes, nor does it generate code which uses the special functions of the Address Generator. However, the compiler provides excellent support for in-line assembly code, with the ability to reference C variables in inline assembly code.

The C preprocessor

The preprocessor completely conforms to the ANSI/ISO Standard 9899-1990, so it should do everything which you expect it to do.

For quick reference, the standard directives are

```
#define      - define a one line macro
#include     - include a file
#if          - test if an expression is non-zero
#ifdef       - test if a symbol is defined
#ifndef      - test if a symbol is undefined
#else        - start alternate part of a #if
#endif       - close out a  #if/#ifdef/#ifndef
```

The preprocessor is automatically invoked on any C file.

Archelon's C preprocessor supports some additional directives, which are intended to make it more useful as a preprocessor for assembler code. The extra directives are

```
#macro       - start a multi-line macro
#endm        - end a multi-line macro

#set         - set a preprocessor symbol to a value

#rept        - start a repeat block
#endr        - end a repeat block

#error       - display the arguments on the error output and terminate execution
#warning     - display the arguments on the error output and continue
```

The multi-line macro facility works much like #define, except that it can span multiple lines instead of being expanding inside one line. This makes it very useful for writing macros for assembler code. Instead of writing

```
#define name( arg1, arg2 ) some text
```

you can write

```
#macro name( arg1, arg2 )
macro line 1
macro line 2
#endm
```

Inside any kind of macro, you can use the # operator, which, when followed by a macro argument name, turns the actual argument into a string. Also, you can use the ## operator to do concatenation (also known as "token pasting");

The `#set` directive has the form

```
#set name expression
```

The value of name will be the number, which results from evaluating the arithmetic *`expression`*. This differs from `#define`, which defines its argument name to be a string. You can use `#set` to do things like generate unique label numbers during multi-line macro expansions.

The repeat block allows you to expand a sequence of lines a number of times.
For instance

```
#rept 3
line 1
line 2
line 3
#endr
```

will expand to

```
line 1
line 2
line 3
line 1
line 2
line 3
```

The preprocessor provides the following predefined symbols:

`__LINE__`	- current line number
`__FILE__`	- current source file
`__DATE__`	- date
`__TIME__`	- time
`__STDC__`	- always set to 1
`__ARCHELON__`	- always set to 1
`__kalimba__`	- always set to 1

Kalimba C

The C compiler for the CSR Kalimba generally conforms to the ANSI/ISO Standard 9899-1990.

Deviations from the C Standard

Floating point is not supported.

The Kalimba C compiler deviates from the C standard in the it treats global static variables the same way as global variables – i.e. their scope is not restricted to the current file and they can be accessed by code in other files. However, local static variables (static variables declared within a function body) are handled as the C standard requires

Enhancements to C

The Archelon Kalimba C compiler supports C++ style comments, which begin with `//` and extend to the end of the current line, and also allows you intermix variable declarations with executable statements, instead of requiring that all variable declarations appear at the beginning of a compound statement. These features are part of the 1999 C standard.

The Archelon Kalimba C compiler also supports a non-standard `loop` statement, in order to allow you an easy way to have your C code use the Kalimba's hardware loop features. The basic syntax for the `loop` statement is

```
loop ( N ) S
```

where *N* is an expression and *S* is a statement. This construct causes the statement *S* to be repeated *N* times. If *N* is zero, then *S* is skipped.

The Archelon Kalimba C compiler supports fractional integral types, using the keyword `_frac` instead of `signed` or `unsigned`. See below for more details about fractional types.

The Archelon Kalimba C compiler supports non-standard additional syntax for binding register variables to be within a specified register set or to be in a specific register or register pair.

The Archelon Kalimba C compiler allows you to write inline assembly code which contains symbolic references to variables declared in C.

Types

Since the Kalimba is a 24 bit word-addressable machine, the sizes of the basic types are as follows:

type	width in bits	format
`char`	24 bit	two's complement
`short`	24 bit	two's complement
`int`	24 bit	two's complement
pointer	24 bit	two's complement
`long`	48 bits	two's complement
`float`	48 bits	NOT IMPLEMENTED
`double`	48 bits	NOT IMPLEMENTED

Signed-ness of `char`

The default signed-ness of the `char` type is implementation dependent. In the Kalimba C compiler, the `char` type is *signed* by default (i.e. it is signed unless you write un`signed char`).

Bit Fields

Bit fields are unsigned by default. If you want to make a bit field be signed, then you must include the `signed` keyword in the declaration. Bit locations are assigned from right to left.

Alignment of Data

Since there are no type alignment restrictions in the Kalimba, all types are aligned mod 1.

Adjacency of data

In the Kalimba C compiler, there is no guarantee that a pair of adjacent global or static declarations, neither of which have initial values, will be emitted in the order in which they were written. So writing something like

```
int x;
char buffer[256]; /* buffer and x may be disjoint */
```

will not guarantee that `buffer` will follow `x`. If you need `buffer` to immediately follow `x`, then you must provide initial values for both, as in

```
int x = 0;
char buffer[256] = { 0 }; /* buffer will immediately follow x */
```

which will cause the compiler to emit the assembly file definitions in the order in which they were written.

Fixed Point Types

The char, int, and short types may be `signed` (the default), `unsigned`, or `_frac`. The `_frac` type is a signed fractional (or fixed point) type. The leftmost bit of a `_frac` is the sign bit. The remaining bits represent a fraction in the range from 0 to (nearly) 1. So a `_frac` type can represent values in the range from -1 to (nearly) 1 inclusive. It is sort of like a poor man's floating point type.

The long types are stored as two words, in big-endian order (the msb word first, then the lsb word).

You can write `_frac` constants by using a floating point constant in the range from -1.0 to 1.0, suffixed by `` `r' `` or `` `R' ``. For instance

```
0.125r
0.99R
```

For convenience, "1.0r" is taken to mean the largest possible _frac constant (0x7fffffff on the adg387).

Cast operations between _frac and non_frac integers simply pass the value without change, so you can write things like

```
_frac int a;

a = 0x1e;
```

If you otherwise mix `_frac` and non-`_frac` operands in an expression, then the operation will be done as a `_frac` operation.

In general, the rules for `_frac` types are similar to the rules for floating point types. This means that remainder operation is not defined. Although shifting is not defined for floating point operands, the compiler will allow you to shift fractional types, without having to go to the bother of a lot of type casting back and forth, because it is often convenient to be able to do so.

The long fractional multiply returns the most significant 48 bits of the 96 bit result of a long multiplication, after it has been left shifted by one to eliminate the double sign bit.

The short fractional multiply returns the most significant 24 bits of the 48 bit result of a short multiplication, after it has been left shifted by one to eliminate the double sign bit..

Fractional divide will saturate the result. If the result would be greater than or equal to 1.0, then fractional division returns the largest positive fractional number (1.0r). If the result would be less then -1.0, then fractional division will return -1.0r. Otherwise, the fractional result will a number in the range from -1.0 to 1.0r (nearly one).

Placement of Data in Memory

The data memory map CSR Kalimba processor includes two distinct data memory areas, and also has the ability to use flash memory for read-only data. For certain applications using circular buffers, there is also a need to be able to position arrays such that you can use them as circular buffers. All data memories share a common address space.

The Archelon Kalimba C compiler addresses these needs by using memory type keywords and a command line option to allow you to direct the compiler as to how you want your data placed in memory.

The memory type keywords are `_DMEM1`, `_DMEM2`, `_DMEM3`, `_DMEM4`, and `_DMEM5`. You can only use these keywords on global or static variables. If you do not use any of these keywords, then `_DMEM1` is the default.

To use any of these keywords, just write one them at the beginning of a variable declaration. For instance,

```
int x;   /* uses default _DMEM1 */
_DMEM2 int y;
```

Variables in the default memory go into the first data memory, called `DM1` by the assembler. Variables in `_DMEM2` go into the second data memory, called `DM2` by KALASM2. Variables in `_DMEM3`, `_DMEM4`, or `_DMEM5` are allocated with an alignment suitable for a circular buffer, as follows:

Keyword	Assembler declaration	Memory used
`_DMEM3`	`.VAR/DMCIRC`	`DM1 or DM2`
`_DMEM4`	`.VAR/DM1CIRC`	`DM1`
`_DMEM5`	`.VAR/DM2CIRC`	`DM2`

For instance, to declare a 128 word circular buffer in DM2, you would write

```
_DMEM4 int my_buffer[128];
```

The compiler can optionally also put certain kinds of data in flash memory. If you use the MCC command line option

```
-Vuse_flash=1
```

Then the compiler will put all switch statement jump tables, all string constants (unless string constants have been made writable by using the `+wrstr` command line option), and any data declared using the C `const` keyword in flash memory, using the assembler syntax

```
.VAR/FLASHDATA ...
```

However, this will not apply to variables declared with circular buffer alignment using `_DMEM3`, `_DMEM4`, or `_DMEM5`, since the assembler does not provide a way to properly align the data for circular buffer use in flash memory.

Note that the `-Vuse_flash=1` is off by default, because the assembler's default `groups.asm` file does not contain an entry for `FLASHDATA`. Before using this option, you must modify your `groups.asm` file to define the `FLASHDATA` segment.

Placement of Code in memory

By default, all C code is placed in code RAM.

To put code into flash memory, I suggest you use inline assembler code outside of function bodies. Although I am not at all sure of what the exact syntax should be, it might be something like this:

```
void
ram_func(void)
{
}

/$
.ENDMODULE;
.MODULE $M.filename_flash;
.CODESEGMENT PM_FLASH;
.DATASEGMENT DM;
$/

void
flash_func(void)
{
}
```

Inline functions

If your function declaration includes the keyword `__inline__`, then the compiler will expand the function inline at each point that it is called.

Even though you may write your code so that every invocation of an inline function can be expanded inline, it may also be necessary for the compiler to produce a copy of the function which can be called directly, in case a function in some other file calls it or in case the function gets called indirectly.

If your inline function will only be used in a single file and if you will use it only after you declare it, you should use `static`. If the storage class of the inline function is `static`, then the compiler will not produce a callable expansion of the function unless your code also takes the address of the function.

If you want to put your inline function in a header file, you should use `extern`. If the storage class of the inline function is `extern`, then the compiler will not produce a callable expansion of the function at all; if you need one, then you should provide a separate declaration without the `extern` keyword.

In the absence of either a `static` or `extern` keyword, the compiler must always also generate a callable copy of the function. If you have an inline function declared `extern` in a header file and if the function may be called indirectly, you must provide, in a separate file, another copy of the same code, but without the `extern` keyword.

Interrupt functions

The Kalimba C compiler allows you to declare functions which can be entered from the interrupt vector by using the keyword `_INTERRUPT` in the function declaration. For instance, here is a simple example of an interrupt function:

```
volatile int got_interrupt;

_INTERRUPT void func( void )
{
      got_interrupt = 1;
}
```

On entering/exiting an interrupt function the compiler will save/restore the entire accumulator register, plus r0 and `r1`, along with any other registers used within the body of the function. The compiler saves/restores the accumulator register by pushing/popping to/from the System Stack, rather then by storing the in the local stack frame.

To connect an interrupt vector to an interrupt function, you will need write assembler code to separately initialize the interrupt vector with the address of the function, using the function name prefixed by `$` (i.e. "$func" in the above example).

#Pragma directives

You use `#pragma` directives to give the compiler further information on how to compile a program. Although every C compiler has its own set of `#pragma` directives, the ability to compile code containing a different compiler's pragmas is supported by the rule that, if a compiler does not recognize a compiler directive, then it ignores the directive.

In this section, we will discuss those `#pragma` directives which the KALIMBA C compiler supports and which you are likely to want to use. In general, and unless otherwise noted, you should only use an KALIMBA C compiler pragma between functions.

Global Optimizer pragmas

The global optimizer provides an additional, higher level of optimization compared to the default. However, because of the aggressive nature of the various optimizations, it is not possible to generate symbolic debug tables for functions, which have been compiled with the global optimizer turned on. Although you can turn on the optimizer for all files, by setting "`+O`" in the C command line in the Miscellaneous tab of the Project Options dialog box of the IDE, you will probably want to use the finer control over optimization, which is afforded by these pragmas.

If you want to turn *on* the global optimizer, then type

```
#pragma +O
```

If you want to turn *off* the global optimizer, then type

```
#pragma -O
```

Because you cannot turn the global optimizer on or off inside a function, these pragmas will only take effect starting at the beginning of the next function. Each remains in effect until the compiler sees a different optimizer pragma or reaches the end of the current source file.

Inline Assembler pragmas

By default, the compiler assumes that inline assembly code does not contain any call instructions. This is necessary, since the compiler does not know what is going on inside a block of inline assembly code. By assuming that there are

no calls, the compiler can be more optimistic about the number of registers it needs to save/restore on entry/exit, making for code that is smaller and faster overall.

In the event that you wish to use a call instruction in a block of inline assembly code, then you should precede the function with the following pragma:

```
#pragma inline_asm_uses_call true
```

After the function body, you should restore the initial default by writing

```
#pragma inline_asm_uses_call false
```

For example, you might write something like this:

```
#pragma inline_asm_uses_call true

void f( void )
{
      /$
      call ...
       $/
}

#pragma inline_asm_uses_call false
```

Binding variables to registers

If you write an assembly code wrapper function, you will need to be able to associate a variable with a particular register set or a particular register. Archelon C provides a way for you to do this.

To begin with, the following table shows the internal register set names used by the compiler, and how each register in each set maps to a Kalimba register name.

Register Set	Register Number	Assembler Name
bankI	0	r0
	1	r1
	2	r2
	3	r3
	4	r4
	5	r5
	6	r6
	7	r7
	8	r8
	9	r9
	10	r10
	11	Null
	12	rMac0
	13	rLink
	14	rFlags
	15	rIntLink
AG1	0	I0
	1	I1

	2	I2
	3	I3
AG2	0	I4
	1	I5
	2	I6
	3	I7
M	0	M0
	1	M1
	2	M2
	3	M3
L	0	L0
	1	L1
	4	L4
	5	L5

To bind a C variable to a particular register set, you just need to use the register keyword and provide the register set name, prefixed by underbar. For instance,

```
register _AG2 int *p;
```

This will associate a register, in the AG2 set with the pointer variable p. The compiler will determine the actual register number chosen within the AG2 set.

To bind a C variable to a particular register or register pair, you must use a slightly different syntax:

```
register int *p @ <register_set> [ <register_number> ] ;
```

For instance,

```
register int *p @ AG1[2];
register int q @ M[0];
register long l @ bankI[2];
```

For the declaration of the long variable above, the compiler will allocate two registers – r2 and r3.

Inline Assembly Code

The Kalimba C compiler makes it very easy to drop into assembly code inside a C function.

To put assembly code inside your C function, simply write /$ and then start writing assembly code. At the end of the assembly code, write $/ to revert back to C code. As you might expect, you can put as many lines of assembly code as you wish between the /$ and the $/.

When writing inline code, you can access any of the C variables currently in scope by entering

```
@name
```

where *name* is the name of the variable. The string @*name* will be replaced by a value depending on its storage class as follows:

Storage Class	*Value*
extern	$*name*
global	$*name*

local static	*file_Vn* (a generated name)
auto	a positive offset from the stack pointer
argument	frame size + argument offset
register	the register name

If the variable is declared using the `register` keyword, and if the type of the variable is large enough to require it to be placed in two or more registers, then you can optionally select which register to which you wish to refer by suffixing the name with a dot following by a single digit. For instance, suppose you declare a `long` variable called `x`. Since a `long` is 48 bits, `x` will be in two registers. Suppose the compiler puts `x` into `r4` and `r5`. Then, in inline code, `@x` or `@x.0` will be replaced by `r4` (which will contain the most significant bits), while `@x.1` will be replaced by `r5` (which will contain the least significant bits).

When referring to variables in in-line assembly code, you must use the name in the correct context. If the storage class is `register`, then you just use `@variable`. If the storage class is `auto` (i.e. local to the function and either marked `auto` or else not `register` and not `static`), then you must use "`M[r9 + @variable]`". If the storage class is global or static, then you must use "`M[Null + @variable]`". For instance,

```
int x;

void test( register int y )
{
            int z;

      /$
            ; load a static or global into r0
            r0 = M[Null + @x];

      ; load a register variable into r0
            r0 = @y;

      ; load a stack variable into r0
            r0 = M[r9 + @z];
      $/
}
```

If you need to put an actual `@` in your code , you must precede the `@` character with a `\` (backslash) character to prevent the compiler from thinking you want to refer to a variable.

You can also get at structure offsets by using the construct

```
@tag.member_name[.member_name]*
```

where *tag* is the "structure tag" as defined in C and *member_name* is the name of one of the members of that structure. While the member name has struct or union type, you can append an additional member name. The whole thing is replaced by a constant which is the offset from the start of the original structure to the specified member.

For instance, if you have declared the structure

```
struct mystruct {
      int a, b;
      struct {
            int x;
            int y;
      } c;
}
```

then you can obtain the constant offset to member `y` in in-line assembly code by writing

```
@mystruct.c.y
```

You can also obtain the size of a C object in in-line assembly code by writing

```
@sizeof( expression )
```

where *`expression`* is any C expression which is a legal argument to the C `sizeof` built-in function.

In order to refer to a variable in in-line code, you and the compiler must agree on where the variable is located. For variables declared outside of functions, this is not a problem, because the variable is always in memory unless you declared it as a global register variable, in which case it is in a register. For local variables, the compiler takes special actions to make sure that it puts the variable where you expect to find it. If you did not declare the variable with a `register` keyword, then the compiler will always make sure to put the variable in memory.

If you did use the register keyword, then the compiler will bind the variable to a ``hard' register', if there is an unallocated register available. (A “hard” register is an actual machine register; in other circumstances, the compiler allocates “pseudo” registers, which are bound to “hard” registers by a later register allocation pass.) Otherwise, it will issue a fatal error message telling you that it could not bind the variable to a register. Because the compiler uses hard registers, each register variable declared in a function and referred to in in-line code reduces the number of registers available. If you declare too many register variables, you may leave too few registers, with the result that the register allocator will generate an error message complaining that it does not have enough "colorable" registers.

When the compiler sees inline assembly code, it also makes worst case assumptions about the state of common sub-expressions (by marking all current common sub-expressions as if they were invalidated by the inline code) and of the state of the registers (by assuming all non-colorable registers are modified by the in-line code).

It also assumes that your inline code does *not* contain any call instructions. If it does, then you should use the inline_asm_uses_call pragma.

C wrappers for assembly coded functions

If you want to write a C callable function in assembly code, it is often most convenient to code it as a C function whose body is a block of in-line code. The benefit is that, if you declare all variables to be used in C, then the compiler will look after saving/restoring register on entry/exit and will look after assigning register numbers. Such a function would have a structure like this:

```
int
f( register int x, register int y )
{
      register int a, b, c;
      /$
      ....assembly code using @x, @y, @a, @b, @c
       $/
      return c;
}
```

In this case, only the declarations and return statement are in C; everything else is in assembler. Note that the code uses the `register` keyword in declarations in order to force the compiler to put them into registers, so that they can be used as registers in the in-line code.

Normally, when you declare a local variable in a function, the C compiler feels free to decide whether or not it will be kept in a register. However, if a variable if referenced in inline assembly code (/$ ... $/), the C compiler will only put

the variable in a register if you use the `register` keyword. Otherwise, unless it is declared as `static`, it will be stored in the local stack frame. This allows you and the compiler to agree in advance as to how to access a C variable when you access it symbolically in your inline assembly code.

If, in inline assembly code, you refer to a register directly (by using the name of the register), then you must save/restore it on entry/exit to/from the inline assembly code. The best way to avoid such concerns is to always declare your registers as C variables and then refer to them using `@`*`name`* in the inline assembly code. If you want to use a scratch register (`r0-r1`) which is not being used to hold an incoming function argument, then you should bind the C variable to the scratch register directly. For instance, to bind an `int` variable `q` to register `r5` and to bind the `long` variable `p` to the register pair `r6/r7`, you would write

```
register int q @ bankI[5];
register long p @ bankI[6];
```

Please note also that the compiler's internal name for the register set, which contains the general purpose registers, is "`bankI`".

In general, if the wrapper function does not contain any function calls, then arguments passed in registers will be used as is. Otherwise, they will be copied to other, non-scratch registers. When in doubt, look at the assembly code output file generated by the compiler, as it will contain compiler generated comments indicating which register it is using for each register variable.

Debugging

The front end of the compiler assumes there are an infinite number of registers in the machine, so that it can put any local, non-static variable (which has not had its address taken) into a register. Following code generation, the register allocator makes the number of registers used by the front end fit into the number of registers available. Where necessary, it generates spills and unspills to save and restore a register to it can be used for some other purpose.

In order to generate the best possible code, the register allocator assigns a register to each individual live range of a variable. It does not necessarily use the same register for every live range. This means that (a) a local variable may be in different registers at different times and that (b) you can only view a local variable at a point where it has an active live range. If you want to have a local variable, which you can always easily see in the debugger, then you can do that by declaring it as `static` (as long as the function will not be called recursively or re-entered in some other way).

If you want to have some idea as to what variables are being assigned to what registers, you can use the `+reginfo` command line option. This option annotates the assembly file, generated by the compiler, with additional comments showing what variables are in what registers.

Optimization

By default, the compiler attempts to optimize your code as much as it can, without losing the ability to allow you to do C source level debugging. In doing so, it is able to do many of the commonly useful optimizations, including constant folding, common sub-expression elimination, some reduction in strength operations (e.g. replacing certain multiplies by shifts), to name just a few.

The compiler also provides an optional, higher level of optimization, which does a variety of additional code improvements, including reduction in strength, constant propagation, loop induction variable elimination, global common sub-expression elimination, aliasing analysis, redundant load elimination, loop invariant removal, and dead code elimination.

The principal benefit of using the global optimizer is that it will generally make your code run faster, especially if it contains loops, which use array subscripting.

The principal disadvantages of using the global optimizer are that (1) it may make your program larger (because the optimizer may generate loop setup code to make loops run faster) and that (2) it will be harder to debug a globally optimized function because the aggressive nature of the optimizations may reorganize your code to such an extent that it is hard to see how the original source code has been mapped to assembler code.

It may not be a good idea to just blindly build or re-build your entire application with the global optimizer. For one thing, although we are justifiably proud of our global optimizer and although we have done our very best to test it thoroughly, it is still a relatively new body of code, which may contain as yet undiscovered bugs, so we recommend a certain amount of caution in using it. Also, the optimizer sometimes does not do well, when applied to a function, which has been hand-optimized in C (e.g. if you have already rewritten array references in loops to use pointers, instead of subscripts).

You can selectively control which functions are built with the global optimizer turned on and which functions are not, by using the global optimizer pragmas, which are described above. Or, you can turn on the global optimizer, on a per file basis, by using the "`+O`" command line option.

C Runtime Environment

For the purposes of the C compiler, the Kalimba has a set of 10 general purpose registers, called `r` registers.

Registers `r0-r1` are global scratch registers (caller-saved), which can be used by anyone at anytime. If you call a C function from assembler, then you must save any of these registers, which are in use before the call, and then restore them afterward.

Registers `r3` to `r8` are local registers (callee-saved), if you write an assembler function which is called from C, then you must save on entry and restore on exit any of these registers which you use in your function.

Register `r9` is reserved to for use as the C runtime stack pointer.

Register `r10` is reserved for use by the hardware do loop, and is used by the compiler when you write a structure assignment or when you write a (non-standard) `loop` statement in C. This register is treated as callee-saved, as well, so that it you use `r10` in assembly code, called from C, then you must save `r10` on entry and restore it on exit.
.
The stack pointer always points to the last used location on the stack, so it can safely be used by interrupt service functions. The stack grows from high addresses towards low addresses. *There is no check for stack overflow*. This means that you have to guard against having the stack grow into the space occupied by global variables, by being careful about how you write your code. You should be very careful to not use up a lot of stack space by declaring large arrays and/or structures on the stack.

Also, the `rMac` register is treated as a global scratch register, which can be used by anyone at any time.

If you are writing assembler code, which is to be callable from C code, you must prefix the name of your assembler code entry point with an "$", because Kalimba C always prefixes its names with "$". For instance, the C function "`main`" is emitted with the name "`$main`" in assembler.

C Program Startup

C code starts up with the function `cstart`. The `cstart` function sets up the stack pointer, and then calls your `main` function. Should `main` end or return, `cstart` will then loop forever until the processor is reset.

Passing Arguments to C functions

There are two cases to consider: (1) functions with a fixed number of arguments and (2) functions with a variable number of arguments (denoted by the presence of ... in the function argument list).

In the case of a function with a fixed number of arguments, some of the arguments can be passed in registers, while any remaining arguments are passed on the stack. The rules for determining what arguments can be passed in registers are as follows. There are two registers available for arguments (r0-r1). The compiler goes through the argument list from left to right, checking each argument. If the argument's type is suitable for being put into a register and if there are enough argument registers left to hold the type, then the argument will be placed in the register selected, and the process repeats for the next argument. If the argument's type is not suitable for being put into a register or if there are not enough unallocated argument registers left to hold the argument, then the argument in question, and all subsequent arguments, are placed on the stack, being stored in order from right to left.

A type is suitable for being placed in an argument register if it is any basic type (i.e. it is neither a structure nor a union).

In the case of a function with a variable number of arguments, all arguments are pushed onto the stack from left to right, and none are passed in registers. It is very important that any call to a function with a variable number of arguments have a correct prototype for the called function in scope, so that the compiler will set up the call correctly. If there is no prototype in scope, the compiler will treat the call as having a fixed number of arguments, likely placing some arguments in registers instead of putting them on the stack where the called function expects them. The compiler is configured to warn you if you call a function which does not have a prototype in scope, by having the `+wp` command line option set by default.

If a function returns a structured type (a struct or union), then the compiler prepends an additional "target" argument in front of the first argument. The target argument contains the address at which to store the structured type being returned by the function. If the function does not have a variable number of arguments, then this first argument register will use up the first of the two registers available for argument passing.

Return values from C functions

A function which returns a `char` or `short` type will put its return value in the `r0` register.

A function which returns a `long` type will put its return value in the `r0 and r1` registers, with the most significant bits in `r0` and the least significant bits in `r1`.

A function, which returns a structured type, will copy the structure being returned to the address passed as the first "target" argument.

Entry to a C function

Upon entry to a C function, the compiler emits code to subtract from the stack pointer the amount of space needed for local variables (including the register save area) on the stack (if any), then saves all local registers used by storing them onto the stack. The amount of the space allocated by the subtraction is called the *frame size*.

If the function contains a call to a C function or a call to a C runtime library function, then the compiler will also emit code to save the `rLink` register, as well.

The first thing the compiler does upon entering a function is to create its local stack frame, which contains local variables, compiler temporary variables, and a register save area. It does this by subtracting a constant from the `r9` stack pointer.

Next it saves any of the registers which the calling C function expects to be preserved. These include `r2-r8` and `r10`. If the function will call other functions, then there will also be code to save the `rLink` register, which contains the function return address that was set by the call instruction. The register save area is at the end of the stack frame, because it uses space that was allocated by the compiler last.

Any function arguments, which were passed on the stack, are beyond the end of the stack frame. For instance, if the stack frame size is 8, then the first argument on the stack will be at `r9+8`, the second argument on the stack will be at `r9+9`, and so on.

The compiler does not normally save `r0-r1` and assumes that they are free to be used at any time. Of course, it also knows if `r0` and `r1` contain arguments, then it will not use those registers unless it has either finished using the arguments or else moved the arguments elsewhere. However, it will save `r0` and `r1` if the function is an interrupt function.

If you declare any variables in C, which are bound to any of the address generation unit registers (`I0-I7`, `M0-M3`, `L0-L3`) and which you then use in inline assembly code, the C compiler will save these registers on entry. It will save them by pushing them onto the System Stack, instead of storing them in the local stack frame, because it happens to be faster to use the System Stack.

Accessing Arguments passed on the stack

The formula to access a function argument, which has been passed in the stack, is

```
SP + frame_size + offset
```

Where `SP` is the stack pointer register (`r9`), `frame_size` is the amount subtracted from the stack pointer on entry, and `offset` is the offset to the argument, prior to the subtraction. For instance, if you have the function

```
void f( int a, int b, int c, int d )
{
      /* function body code */
}
```

then, on entry and before the new stack frame is allocated by subtraction from the stack pointer, `a` is in register `r0`, `b` is in register `r1`, `c` is at offset 0, and `d` is at offset 1. In general, offsets to arguments not in registers are computed by considering the arguments in order from left to right.

Exit from a C function

On exit from a C function, the compiler emits code to undo what it did on entry. First, it restores the registers it saved by loading them from the register save area in the stack frame. If it did a subtract from the stack pointer on entry, then it does the corresponding add using the same constant value.

If you declare any variables in C, which are bound to any of the address generation unit registers (`I0-I7`, `M0-M3`, `L0-L3`) and which you then use in inline assembly code, the C compiler will also restore these registers on exit. It will restore them by popping them from the System Stack, instead of loading them from the local stack frame, because it happens to be faster to use the System Stack.

If the function contains a call to a C function or a call to a C runtime library function, then the compiler will also emit code to restore the rLink register, as well.

Finally, the compiler generates a return instruction, to return to the caller.

Function cleanup

When your function returns, the code that called it must clean up the stack, if it had pushed any arguments on the stack for the call. To do so, it adds to the stack pointer the total size of the arguments pushed. This leaves the stack pointer in the state that it was just prior to generating code for the call.

The include file "kalimba.h"

The "kalimba.h" include file defines some variables and functions, which are very useful for programming the CSR Kalimba.

All function names, defined in this file, are prefixed by at least one underbar, in order to avoid conflicts with names in user C code, as per the recommendation in the C Standard.

Memory Mapped Registers

The include file contains definitions which allow you to access memory mapped registers as if they were global variables.

The defined names are those documented in Section B.5 of CSR's *Kalimba DSP User Guide*.

Builtin (aka intrinsic) functions

```
int _push( int );
```
- this function pushes its argument onto the System Stack and returns the pushed value as its result.

```
int _pop( void )
```
- this function pops the 24 bit word on the top of the System Stack and returns that value as its result.

Using the Tools

In this section, we will briefly describe how to compile, assemble, and link CSR Kalimba C and assembly code, which you do by using commands typed into a "command prompt" window.

To begin with, you will be using, directly or indirectly, the following commands:

`MCPP`	- the C preprocessor
`MCC`	- the C compiler
`KALASM2`	- the CSR Kalimba Assembler/Linker

In the examples below, "<ctools_dir>" is the path, with drive letter if necessary, to the directory in which you installed the Archelon CSR Kalimba C tools, and "<kalasm_dir>" is the path, again including a drive letter if necessary, to the directory in which you installed CSR's KALASM2.

Before trying to run MCC, you must first set up a couple of environment variables, which will allow MCC to find, load, and run the code generator for the CSR Kalimba processor. To do so, you must set the environment variables MCSYS, MCDIR, and MCINCLUDE as follows:

```
set MCSYS=kalimba
set MCDIR=<ctools_dir>\mcdir
```

```
set MCINCLUDE=<ctools_dir>\include
```

The mcdir directory contains the kalimba.cif file, which contains the code generator. The include directory contains the Standard C include files for the Kalimba.

To use these commands, you must put the directory in which each resides in your shell's PATH environment variable. You can do this by entering

```
set path="<ctools_dir>\bin;%path%"
set path="<kalasm_dir>\bin;%path%"
```

All but one of these commands will print a brief summary of its arguments on the standard output, when you type the command name with no arguments. The exception to this is MCPP. If it is invoked with no arguments, it assumes that it will be reading from the standard input and writing to the standard output. To get it to display its arguments, you must invoke it with an incorrect argument, as in "MCPP +ha". MCC outputs more than one screen full of text. To see it one screen full at a time, pipe it through more by typing "mcc | more".

The maximum command line length allowed for the MCC and MCPP commands is 512 bytes. If and when that is not enough, the MCC and MCPP commands support an "x=filename" command line option which allows command line options to be read from the file "filename". Normally, you need not worry about invoking MCPP, since MCC invokes it automatically by default.

Compiling

To compile a C program, so as to create an object file for linking, you would usually type (at least)

```
mcc file.c
```

When you enter the above command line, the C compiler will pass the file file.c through the MCPP preprocessor, compile the resulting output file, and place the generated assembly code into file.asm.

The MCC C compiler automatically turns on the command line options

```
+wp +c
```

The +wp option causes the compiler to issue a warning message whenever it sees a call to a function, which does not have a function prototype in scope. The +c means to include the original C source code as comments in the assembly output.

If you wish to compile your entire source file with the optional global optimizer turned on, then you can do this by using the "+O" command line option. If you need finer control over which functions get compiler using the global optimizer, then you should instead use the global optimizer pragmas.

Assembling

To process an assembly language file into an object file, you will usually type

```
kalasm2 -c -F<kalasm_dir>\groups.asm -F<kalasm_dir>\default.asm file.asm
```

where "<kalasm_dir>" is the path to the directory containing KALASM2 and its support files "groups.asm" and "default.asm". The "-c" option tells KALASM2 to generate only generate an object file, omitting the linking step. The above command will create an object code file, called "file.kobj" and a listing file, called "file.lst".

Linking

To link a collection of object files into an application ready for loading into the Kalimba, you just need to invoke KALASM2, with all the desired object files as arguments. For more information about this, please see the CSR KALASM2 documentation.

Index