Programming Tool for the Cirrus Logic 32-Bit Audio DSPs

Cirrus Logic C-Compiler

User's Manual

AUG 2013
DS811UM15

http://www.cirrus.com

Contacting Cirrus Logic Support

For all product questions and inquiries contact a Cirrus Logic Sales Representative.

To find one near you, go to **http://www.cirrus.com**

Contents

Tables

Chapter 1
Introduction

1

1.1 Welcome to the Cirrus C-Compiler (CCC)

The Cirrus® C Compiler (CCC) compiles C source files to produce assembler code that is able to run on the Cirrus DSP platforms such as the CS4953xx, CS4970x4, CS485xx, CS47xxx, and the CS498xx multi-core DSPs.

The CCC is part of the Cirrus Logic Integrated Development Environment (CLIDE) that is available to Cirrus Logic customers to enable them to develop their own custom audio algorithms to run on Cirrus Logic DSPs. CLIDE receives input from DSP Composer™ and provides output to DSP Composer. The Cirrus Device Manager (CDM) must be running to use the CLIDE tool set. When the SDK for the Cirrus DSP used in the customer's design is launched, the CDM should also be automatically launched.

The *CLIDE User's Manual* describes the CLIDE graphical user's interface (GUI), which allows the user to access the following applications from CLIDE:

- *Cirrus Logic C-Compiler*—described in this manual
- *CASM*—Cirrus Logic Cross-assembler (CASM) described in the *32-bit DSP Assembly Programmer's Guide*
- *CLIDE* debugger—described in the *CLIDE User's Manual,* debugs both Assembly and C language source files, and replaces the Hydra debugger
- *Primitive Elements Wizard*—described in the *CLIDE User's Manual* and is an XML file Wizard used to create custom primitives that are debugged within CLIDE
- Simulator—described in the *CLIDE User's Manual*
- Source editor—described in the *CLIDE User's Manual*

1.2 Standard Conformance of CCC

CCC supports the C99 standard with Embedded C extension for fixed point types and memory zone qualifier, but without the following features:

- Floating point types (except for floating point constants in expressions that can be evaluated to a constant value in compile time) and related standard libraries
- File and console input/output and related standard libraries
- Variable number of function arguments and related standard library
- Several other standard libraries: time.h, signal.h, locale.h, and setjmp.h
- Wchar type
- Restricted pointers
- Variable length arrays
- The long long int type and library functions (the type key word is supported, but it's size is not 64 bits)
- Universal character names (\u and \U)
- Mixed declarations and code
- More flexible array initialization (for example, "int x[5] = {[2] = 2, [4] = 3};")
- Declaration in for loop (for example, "for (int x = 0...")
- Multiple appearance of inline attribute in function declaration
- Static array size in function parameter (for example, "int foo(int x[static 5])")
- Saturation on overflow for fixed point abs functions declared in stdfix.h (see Section 10.4)

Note: Bit fields are supported with the following restrictions:

- No single bit field can be longer than 32 bits.
- All bit fields are treated as unsigned integers. A “signed” attribute is ignored.

Chapter 2
CCC Command Line Options

2

2.1 Description of CCC Command Line Options

This chapter describes the options for using the command line in the CCC.

2.1.1 Command Line Format

The format of the compiler command line is as follows:

```
ccc2 -<options> <C source file>
```

When calling the CCC compiler without arguments, the command line options Help message is displayed.

2.1.1.1 Details of Command Line Options

The <options> field contains zero or more compiler control directives. Each directive starts with the character "-" and ends with a space. Table 2-1 contains the command line options that are available.

Table 2-1. Command Line Options

Command Line Option	Description
-c	Stop after assembling. Do not link. Compiler produces the assembler (.s) and the object (.o) file.
-cdl	Create file dependency list. Important for incremental build. The list will be created in a file with extension .cdf.
-Cl	Emits corresponding C line for every instruction block. This command option is only valid if the -g option is used.
-D<symbol>	Define a symbol.
-edi-lines	Emit corresponding C line numbers as comments in the .s file.
-emit-hints	Emit useful information about generated code, such as: advises how to change the C code in order to improve the generated code, information about stack usage for passing arguments, information about points in code where SRS is used, et cetera.
-esc	Emit .s file in the same folder where .c file is located, regardless of -o switch.
-falias-analysis[=N]	Use alias analysis information. N specifies how deep in the function call graph the analysis should go. Increasing the limit makes analysis more precise but also increases compilation time. Default value is 1. Value -1 means "as deep as possible".
-fcompact	Turn on compactor optimization, which utilizes instruction-level parallelism.
-fconst-pool[=M]	Load constants from memory. M can be X, Y, or P, to select which memory zone the pool is going to be. Default is P.
-fcse	Turn on common subexpression elimination optimization.
-femit-asm-struct	Emits an assembly .struct definition in the .s file for the struct types used in the C code.
-fif	Try several compilation strategies and choose the best result. Compilation lasts longer.
-finline-functions	Automatically find suitable functions to inline. Available only if -wpo is enabled.
-fno-inline	Do not inline functions. That is, ignore the inline function attribute.
-fnfs	Do not use stack. Statically allocate function memory instead. Valid only if there is no recursion in the program. Refer to Section 7.2.1 for additional information regarding static stack frame calling conventions.
-fno0init	Do not initialize to 0 static storage duration objects that are not explicitly initialized. Useful for reducing .uld size.
-fno-hw-loop-detection	Do not create hw loops.
-fprc	Propagate constants and evaluate constant expressions.
-funsafe-loop-optimizations	Assume number of loop iterations is never 0. If number of loop iterations is not constant, but can be calculated each time before the loop is entered, hardware loop can be used only if the number of iterations is never 0.

Table 2-1. Command Line Options (Continued)

Command Line Option	Description
-fwrapv	Specify signed integer overflow behavior as wrap-around, which makes code slightly larger. Otherwise, the overflow behavior is undefined.
-g	Write DWARF2 debug information into the object file and create debuggable code. (See Chapter 6 for more information on debug attributes.)
-h	Print command line help screen.
-ida	Ignore function debug_on and debug_off attributes. (See Chapter 6 for more information on debug attributes.)
-ira	Use interprocedural register allocation. Allocates only registers clobbered by a called function for each function call. Default setting assumes all clobberable registers are clobbered.
-I<include_path>	Pass the include path to the compiler.
-o<output file name>	Specify the name of the output files (file extension, if given, is ignored). Default output file name is the same as the input file name, or the last input file name, if multiple input files are provided.
-ruf	Report error for undeclared function calls.
-s	Silent option. Compile without providing any progress information.
-S	Stop before calling assembler and linker. Compiler output is just the assembler (.s) file.
-v	Display the version of the compiler.
-w	Suppress warnings
-wpo	Whole program optimization. Turn on when you pass all C source files of your program at once. Other switches become available.

2.1.1.2 System Environment Variables Used by the Compiler

The compiler makes use of the following system environment variables:

- The CCC_INCLUDE variable specifies the default include path.
- The CCC_LIB variable specifies the default runtime library path.
- The C-compiler's compile process is also affected by the CASMSPEC and CLINKSPEC environment variables. (See the *32-bit DSP Assembly Programmer's Guide* for more information about the CASMSPEC and CLINKSPEC environment variables.)

When called from command line, CCC calls casm and clink like this:

```
casm.exe FileName.s -oOutputFileName.O --debug --casmspec -s -c
clink.exe OutputFileName.O LibPath\crt0.o -lLibPath -oOutputFileName.img --
clinkspec
where LibPath is geathered from CCC_LIB environment variable.
```

The whole build process is configurable from CLIDE on Project Properties page.

2.1.1.3 Additional Information and Examples of Using Command Line Options

2.1.1.3.1 -fno0int

Example 1 With -fno0int

```
int big_buffer[10000];
int big_initialized_buffer[10000] = {0};
```

Example 2 Without -fno0init

```
_big_buffer:
.bsc (0x2710), 0x00000000
_big_initialized_buffer:
.bsc (0x2710), 0x00000000

With -fno0init:
_big_buffer:
.bss (0x2710)
_big_initialized_buffer:
.bsc (0x2710), 0x00000000
```

2.1.1.3.2 -fwrapv

Example 3 With/Without -fwrapv

With -fwrapv:

```
int is_negative(int x)
{
     int ret;

     if (x < 0)
     {
          ret = 1;
     }
     else
     {
          ret = 0;
     }

     return ret;
}
```

Without -fwrapv:

```
_is_negative:
     a0 & a0
     if (a >= 0) jmp (else_0)
     uhalfword(a0) = (0x1)
     jmp (endif_0)
else_0:
     a0 = 0
endif_0:
     ret
```

With -fwrapv:

```
_is_negative:
     a0 = a0h     #!!!
     a0 & a0
     if (a >= 0) jmp (else_0)
     uhalfword(a0) = (0x1)
     jmp (endif_0)
else_0:
     a0 = 0
endif_0:
     ret
```

Example 4 Function is_negative is from Example 3

```
int t = 0xFFFFFFFE;

void main()
{
     int y = is_negative(t + 2); //overflow occurs
     ...
}
```

Without -fwrapv, y will be 0. With -fwrapv, y will be 1.

Example 5

```
int shift_right_for_2(int x)
{
     return x >> 2;
}
```

Without -fwrapv:

```
_shift_right_for_2:
     a0 = a0 >> 1
     a0 = a0 >> 1
     a0l = (0x0)
     ret
```

With -fwrapv:

```
_shift_right_for_2:
     a0 = a0h#!!!
     a0 = a0 >> 1
     a0 = a0 >> 1
     a0l = (0x0)
     ret
```

Example 6 shift_right_for_2 is from Example 5

```
int t = 0xFFFFFFFF;

void main()
{
     t <<= 2; //overflow occurs
     int y = shift_right_for_2(t);
     ...
}
```

Without -fwrapv, y will be 0xFFFFFFFF. With -fwrapv, y will be 0x3FFFFFFF.

2.1.1.3.3 -fconst-pool

Example 7

```
int foo(int x)
{
     return x + 0x1234567;
}
```

Without -fconst-pool:

```
     .code_ovly
_foo:
     a1 = (0x123)
     lo16(a1) = (0x4567)
     a0 = a0 + a1
     ret
```

With -fconst-pool=X:

```
     .xdata_ovly
__extractedConst_0_1
     .dw  (0x1234567)

     .code_ovly
_foo:
     a1 = pmem[__extractedConst_0_1 + 0]
     a0 = a0 + a1
     ret
```

Example 8

```
int bar(int x)
{
     return x - 0x1234567;
}

int foo(int x)
{
     return x + 0x1234567;
}
```

Without -fconst-pool:

```
     .code_ovly
_bar:
     a1 = (0x123)
     lo16(a1) = (0x4567)
     a0 = a0 - a1
     ret

_foo:
     a1 = (0x123)
     lo16(a1) = (0x4567)
     a0 = a0 + a1
     ret
```

With -fconst-pool:

```
     .code_ovly
__extractedConst_0_2
     .dw  (0x1234567)

     .code_ovly
_bar:
     a1 = pmem[__extractedConst_0_2 + 0]
     a0 = a0 - a1
     ret

_foo:
     a1 = pmem[__extractedConst_0_2 + 0]
     a0 = a0 + a1
     ret
```

2.1.1.3.4 -emit-hints

There are currently four groups of info messages that the CCC reports when the -emit-hints switch is turned on:

1. Information about stack usage for function arguments.

Every function call where stack is being used for the passing of arguments or a return value is gong to be reported. See Section 9.2.1 for additional information.

2. Information about translation of loops into hardware loops.

If the compiler is not able to translate a certain loop in the source code into a hardware loop, there will be message containing the reason it was unable to translate. If you want to make the compiler translate that loop into a hardware loop, try to satisfy the condition stated in the message. See Section 9.2.2.1 for additional information.

3. Information about usage of compiler internal runtime functions in the generated code.

Cirrus DSP does not have direct hardware support for all C functionalities, so some of them are implemented by calling internal runtime functions. The code that causes the usage of those functions can usually be changed so that it does not require those functions. See Section 3.3.4 for additional information.

Section 3.3 explains that the following three types are best used in C programs for the Cirrus DSP: fract, long fract, and long accum. If you use just those fixed-point types, then only one of the internal functions in the following table can be present in the compiled code. All other internal functions are related to the usage of fixed-point types other than fract, long accum, and long fract.

Table 2-2. Internal Functions

int_to_fract int_to_laccum int_to_lfract uint_to_fract uint_to_laccum uint_to_lfract	An integer (signed or unsigned) variable is being converted to one of the fixed-point types. Very often, this conversion is a result of a coding mistake. It is useful only in a small number of cases. If bitwise conversion was intended, then it can be performed by intrinsic functions in the stdfix.h header. See Section 3.3.2 for more information.
laccum_to_int	A fixed-point variable is being converted to an integer type. Very often, this conversion is a result of a coding mistake. It is useful only in a small number of cases. If bitwise conversion was intended then it can be performed by intrinsic functions in the stdfix.h header. See Section 3.3.2 for more information.
div_sat_laccum	A fixed-point division is being performed. The function does full precision division. If you want faster or less precise division, call the appropriate function. Several different division functions are given in dsplib.h.
div udiv	Integer division (signed or unsigned) is being performed. The function does full precision division. If you want faster or less precise division, write and call the appropriate function.
lshift rshift ulshift urshift	The shifting of integer variables (signed or unsigned) for a variable number (that is, not constant) of places is being performed. Listed functions shift values one bit at a time for a desired number of times, but better results (speed-wise) can be obtained by multiplying the value with a constant picked from the look-up table. In the next version, CCC compiler will be able to do this automatically, but, at the moment, the only way to do it is manually in the source code.

Table 2-2. Internal Functions

lshift_laccum rshift_laccum	The shifting of fixed-point variable for a variable number (that is, not constant) of places is being performed. The situation is similar to integer shifting functions, but it has to be noted that some fixed-point types are more than 32 bits wide, and their shifting using the look-up table is not as simple.
mul_sat_laccum	This function is called when the multiplication of two fixed-point types is being performed, where at least one of the types is wider than 32 bits. Check if that was the intention or just a coding error and whether that precise multiplication is necessary. If it is necessary, it is important to note that this function performs full precision multiplication of two long _Accum types, but the same function is called for all other cases. Therefore, if lower precision types are multiplied (for example, _Fract * long _Accum) then this function introduces unnecessary overhead. For the next version of the CCC, it is planned to have separate functions for every possible multiplication case, but, at the moment, the only way to get the best efficiency in the described cases is to manually write assembly function for the desired multiplication (or embed the assembly code in asm statement).
mod umod	This performs a division reminder operation on integers (signed or unsigned). There is no hardware support for this operation, so try to avoid it if code speed is important. In cases where the right operand is a power of two, this operation can be performed using bit masking. In other cases, avoiding division reminder operation is not trivial.

4. Information about the use of an SRS (Shifter/Rounder/Saturator) unit.

This group of messages is useful if you want to control the SRS unit from C code, as seen in Section 3.3.2. That way, you can be sure that changes of SRS behavior which you introduce are going to affect only the intended lines of code. In the SRS for shifting example in Section 3.3.2, from line 4 to line 10, the behavior of SRS is altered. An info message would be emitted for line 7 ("*p = (long accum)(*p);"), but not for any other line in between the 4th and 10th line. That way, you can be sure that only line 7 would be affected by SRS changes, which was the intention in the example.

Chapter 3
Data Types

3

3.1 Overview

CCC fully supports all standard C data types, but support for the "long long" integer type and floating-point types is currently restricted. CCC extends the standard C data type set with fixed-point data types as described in ISO/IEC TR 18037.

The size of the long long type is 32 bits. A future version of CCC will implement long long as 64 bits.

Since Cirrus DSPs that CCC targets are fixed-point DSPs, floating-point arithmetic can only be supported through emulation. Because of that, support of floating-point data types in CCC is restricted to usage in constant expressions. In other words, floating-point arithmetic is supported if it can be performed in compile time.

For more information on the C programming language standard, see ISO/IEC JTC1 SC22 WG14 N102.

For extensions to the C programming language standard for embedded processors, see "Programming languages, their environments and system software interfaces — Extensions for the programming language C to support embedded processors." in ISO/IEC TR 18037.

3.2 Integer Data Types

Sizes of integer data types are given in Table 3-1:

Table 3-1. Integer Data Types

Type	Size in Bits
char	32
short	32
int	32
long	32
long long	32

The size of the pointers are 16 bits. Enums are implemented as a signed integer. The char type without an explicit sign qualifier is treated as unsigned.

3

3.3 Fixed-point Data Types

The compiler complies with the fixed-point types standard declared in the Embedded C Extension (ISO/IEC JTC1 SC22 WG14 N1021). In the document, two groups of fixed-point data types are added to the C language: the fract types and the accum types. The data value of a fract type (both signed and unsigned) has no integral part, hence values of a fract type are in range [-1.0, +1.0]. Accum types in CCC are defined to have 8 integral bits, and the values of accum type are in range [-256.0, +256]. Unsigned accum types have 9 integral bits, and its values are in range [0.0, 512.0]. A type's precision depends on the number of fractional bits it has. Table 3-2 describes the available fixed-point types along with their size and format in CCC.

Table 3-2. Fixed-point Data Types

Type Syntax			Size in Bits	Number Format
signed	short	_Fract	32	s.31
	—		32	s.31
	long		32	s.31
unsigned	short		32	0.31
	—		32	0.31
	long		64	0.63
signed	short	_Accum	40	s8.31
	—		40	s8.31
	long		72	s8.63
unsigned	short		40	9.31
	—		40	9.31
	long		72	9.63

The formal names of the new keywords start with an underscore followed by an uppercase letter to avoid possible name clashes in existing programs. However, in the header stdfix.h, these formal names are used to define the natural spellings as aliases. So, instead of "_Fract" and "_Accum", the user can use "fract" and "accum".

In the Cirrus DSP architecture, data registers are 32 bits long, a data register pair is 64 bits long, and accumulators are 72 bits long; all of which are signed. Long accum, fract, and long fract data types are best used in C programs for the Cirrus DSP processor. Other data types are supported, but operations with those types usually require more instructions and often involve calls to runtime functions (see Section 3.3.4).

3.3.1 Fixed-point Constants

In C language, the type of a constant is determined by the constant suffix. For example, suffix "f" defines float type constant, therefore constant "0.5f" is a float type constant, while "0.5" (without suffix) is a double type constant. Suffixes for fixed-point constants are shown in Table 3-3.

Note: The suffix is not case sensitive; the table only gives lowercase letters.

Table 3-3. Fixed-point Constants

Type Syntax			Suffix
signed	short	_Fract	hr
	—		r
	long		lr
unsigned	short		uhr
	—		ur
	long		ulr
signed	short	_Assum	hk
	—		k
	long		lk
unsigned	short		uhk
	—		uk
	long		ulk

A fixed-point constant evaluates to the closest representable value that is in the range of the indicated type. Exceptions are constants with the value of the type's upper bound range (such as 1.0r, 256.0lk, 512.0ulk, et cetera). These constants denote the maximal value for the type because the exact value is not representable (value ranges are left-closed and right-open intervals). A _Fract constant can also be expressed in hexadecimal format. This hexadecimal number is interpreted as a bit representation of the fixed-point value rather than the integer value. For this approach to work, the suffix of the fixed-point constant must be 'r'.

Note: The C standard specifies that constants without a suffix are considered to be an integer type if a decimal point is not present or double type if a decimal point is present. All other types are defined by a suffix. Fixed-point constants must always have a suffix. Otherwise, they are not fixed-point constants. For more information about usage of floating-point (both float and double) constants, see Section 3.4.

Here are examples of some fixed point constants:

```
0.8932r// _Fract constant
0.8932LR// long _Fract constant
12.6574k// _Accum constant
-150.778892LK// long _Accum constant
```

```
20.15r// _Fract constant, but outside the range. Evaluates to the maximal
value for the type. A warning will be issued.
-20.15r// _Fract constant, but outside the range. Evaluates to the minimal
value for the type. A warning will be issued.
1.0r// _Fract constant, but outside the range. Evaluates to the maximal value
for the type. There will be no warning, because it is regular case.

12k// _Accum constant, same as 12.0k

0x40000000r// _Fract constant, same as 0.5r
0x40000000// integer constant, same as 1073741824

0.3// double constant
```

3.3.2 Fixed-point Type Conversions

Conversions from a fixed-point to an integer type round toward zero. For example, if a _Fract value is converted to an integer value, the only possible results are -1 and 0.

Conversions from an integer to a fixed-point type are defined so that the integral part of the fixed-point type is the same as the integer value, while the fractional part is zero. If the integer value is outside of the fixed-point type range, the result is saturated. That is, it is set to the closest representable value. For example, if an integer value is converted to a _Fract value, the only possible results are -1.0r, 0.0r, and maximum _Fract value (1.0r).

Bitwise conversions between integer and fixed-point types are supported through several functions defined in <stdfix.h>. The bitwise fixed-point to integer conversion functions return an integer value equal to the fixed-point value of the argument multiplied by 2^F, where F is the number of fractional bits in the fixed-point type. The result type is an integer type big enough to hold all valid result values for the given fixed-point argument type. The bitwise integer to fixed-point conversion functions return a fixed-point value equal to the integer value of the argument divided by 2^F, where F is the number of fractional bits in the fixed-point result type of the function. Available functions for bitwise conversion are given in Table 3-4:

Table 3-4. Available Functions for Bitwise Conversion

int	bitshr(short _Fract);
unsigned int	bitsuhr(unsigned short _Fract);
int	bitsr(_Fract);
unsigned int	bitsur(unsigned _Fract);
short _Fract	hrbits(int);
unsigned short _Fract	uhrbits(unsigned int);
_Fract	rbits(int);
unsigned _Fract	urbits(unsigned int);

Since there are no integer types that are bigger than 32 bits, bitwise conversion of fixed-point types other than _Fract and short _Fract is not possible. However, three additional functions are provided for bitwise conversion of parts of the long _Accum type:

```
int bitsgetlo(long _Accum); // returns lowest 32 bits of long _Accum value
int bitsgethi(long _Accum); // returns higher 32 bits of long _Accum value
int bitsgetg(long _Accum); // returns highest 8 bits of long _Accum value
```

Conversions from a fixed-point to a floating-point type are not defined. Conversions from a floating-point to a fixed-point type are defined so that the resulting fixed-point value is the closest representable value of the indicated fixed-point type. Conversions between different fixed-point types are defined as follows:

1. If the source type has greater precision, then rounding is performed.
2. If the source type has a bigger value range, then saturation is performed if the source value is outside of the destination type range.

For the following conversions, the SRS (a part of Cirrus DSP architecture) is utilized for performing saturation and rounding:

Table 3-5. Conversions for Saturation and Rounding

Destination Type	Source Type
short _Fract, _Fract	long _Fract
short _Fract, _Fract	short _Accum
short _Fract, _Fract	_Accum
short _Fract, _Fract	long _Accum
long _Fract	long _Accum

The SRS is utilized only for the conversions noted in Table 3-5.

The value of the mr_r, mr_s, and mr_sr registers affect the behavior of the SRS unit. See Section 2.2.2.2 of the Assembly Language Programmer's Manual for details on how the settings of the mr_sr register affects SRS. The best way for a programmer to change the value of these registers is by calling the appropriate set function declared in the stdfix.h header file.

In addition to saturation and rounding, SRS can also perform shifting. If SRS is set to perform shifting, then shifting will be done implicitly when the type conversions that utilize SRS are performed.

The following examples illustrate the use of intrinsic functions for getting or setting the value of the mr_sr register:

```
#include <stdfix.h>

/* set mr to no shift, no round */
set_mr_sr(NO_SHIFT | NO_ROUND);

/* set mr to no shift, round toward zero */
set_mr_sr(NO_SHIFT | ROUND_TO_0);

/* set mr to shift left by 1, round toward zero */
set_mr_sr(SHIFT_LEFT_1 | ROUND_TO_0);

/* set mr to keep current shift mode, round by adding .5 and truncating */
set_mr_r(ADD_0_5_TRUNCATE);

/* set mr to shift right by 1, keep current round mode */
set_mr_s(SHIFT_RIGHT_1);
```

In crt0.o, the mr_sr register is set to NO_SHIFT and ADD_0_5_TRUNCATE before the call to main(). However, if crt0.o is not used, the programmer must take care of mr_sr register and set it to the appropriate value. The best practice is to define mr_sr value, which is valid for the whole program. Set it at the beginning. When there is a need to change the rounding and shifting behavior for some part of the code, just before the code set mr_sr to new value, reset it to the original value at the end of the code piece.

The following example shows one usage of SRS for shifting:

```
// Without utilization of SRS for shifting
void foo(fract* p)
{
int i;
for (i = 0; i < 10; i++)
{
*p = *p >> 1;
p += 1;
}
}
// With utilization of SRS for shifting
void foo(fract* p)
{
int i;
set_mr_sr(SHIFT_RIGHT_1 | ADD_0_5_TRUNCATE);
for (i = 0; i < 10; i++)
{
*p = (long accum)(*p); // in conversion from long accum to fract SRS will
shift right implicitly
p += 1;
}
set_mr_sr(NO_SHIFT | ADD_0_5_TRUNCATE);
}
```

3.3.3 Fixed-point Type Expressions

Operations that can be performed on fixed-point types are:

- Unary:
  - negation (-)
- Binary:
  - relational operations (<, <=, ==, !=, >=, >)
  - addition (+), subtraction (-), multiplication (*), and division (/)
  - shifting left (<<) and right (>>)

with appropriate assignment forms: +=, -=, *=, /=, >>=, and <<=. In other words, fixed-point types support all operations applicable on floating-point types, with addition of shifting.

When a fixed-point type operand is involved in an operation, the following rules apply:

1. If both operands are fixed-type and one of the operands is signed and the other is unsigned, the unsigned operand will be converted to its corresponding signed type. This is opposite from the usual arithmetic conversions for integer types where unsigned has the advantage.
2. No other conversions are performed on the source types. This is also different than the usual arithmetic conversion in C where both operands are converted to the common type (except in case of shifting).
3. The resulting type is the higher rank type of the two operands (except for relational operations where resulting type is integer). Type rank is given in Table 3-6. Fixed-point types have higher rank than integer types but lower rank than floating-point types.
4. The only exception to rule 3 is multiplication of _Fract type (or short _Fract type) operands. In this case, the resulting type is long _Accum in order to capture the full precision of the multiplication.

Note: Shift operation expects the right hand operand to be an integer type and to have a positive value. The resulting type of a shift operation is always the type of the left hand operand (which is for fixed-point types consistent with rule 3 above).

Table 3-6 shows the rank of the arithmetic data types, from highest to lowest:

Table 3-6. Arithmetic Data Types

Rank	Type
highest	long _Accum
lowest	_Accum
	short _Accum
	long _Fract
	_Fract
	short _Fract
	int

The following examples illustrate the main consequences of the above arithmetic conversions:

```
_Fract f = 0.25r;
int i = 3;
f = f * i;
// The type of "f * i" expression is _Fract, and the value in f will be 0.75r

unsigned _Fract u = 0.5ur;
_Fract s = 0.5r;
s = s * u;
// The type of "s * u" expression is (signed)_Fract, and the value in s will
be 0.25r
```

```
unsigned long _Fract u = 0.5ulr;
_Fract s = 0.5r;
long _Fract p = s * u;
// The type of "s * u" expression is (signed) long _Fract, and the value in s
will be 0.25lr


_Fract x = -1.0r;
_Fract y = -1.0r;
long _Accum z = x * y;
// The type of "x * y" expression is long _Accum, and the value in z will be
1.0lk


_Fract x = -1.0r;
_Fract y = -1.0r;
_Fract z = x * y;
// The type of "x * y" expression is long _Accum, and conversion from long
_Accum to _Fract will be performed when assigning to z, with 1.0r (which is
different than 1.0lk) as the resulting value, due to saturation.


fract x = 0.25r;
fract y = x << -1;
// Value in y is undefined, because right hand operand is not positive.


fract x = 0.25r;
fract y = x << 3;
// Overflow happens, and value in y is not defined.


fract x = 0.25r;
fract y = (long accum)x << 3;
// y will be 1.0r. (long accum)x << 3 is 2.0lk (no overflow now), which will
be, due to saturation, reduced to 1.0r when moving to y.


fract x = 0.25r;
fract y = (long accum)x << 15;
// y will again be undefined, because overflow now happens with long accum
type and all the bits are shifted out of accumulator.


fract x = 0.25r;
fract y = (sat fract)x << 15;
// y will be 1.0r. Saturated fract type will guarantee that saturation will
happen on overflow, however, this is done at the expense of additional
instruction cycles.
```

3.3.4 Costly Fixed-point Expressions

The DSP architecture is designed to efficiently calculate most expressions between fixed-point data types or between fixed-point and integer data types. However, some operations between some data types require extra calculations that must be performed via a call to a

subroutine, which can require many cycles. As of version 6.7.9, the C Compiler will emit indications into the output log about which lines of C code involve expressions that require function calls to implement.

Some general guidelines for operations which require this type of costly implementation are:

1. Some operations with fixed-point operands other than (short) _Fract, long _Accum, and long _Fract. Those are the following types: (short) _Accum, unsigned versions of all fixed-point types, and saturated versions of all fixed-point types.
2. Division. All division operations are implemented via a software function as the DSP hardware does not support division.
3. Division remainder operation (mod, %) (for the same reasons as division).
4. Conversions from integer to fixed-point type, and vice versa. See Section 3.3.2 to see what those conversions imply. Bitwise integer to/from fixed-point conversion, also described in Section 3.3.2, do not fall under this category and are performed very efficiently.

3.4 Floating Point Data Types

Floating-point types are not completely supported by CCC. Support of floating-point data types in CCC is restricted to usage of floating-point constants in constant expressions. The whole constant expression, however, must not be of floating-point type. Also, long double type is not supported in any form. For example, this is supported:

```
fract x = 1.8 / 3;
long accum y = 700.0 / 40;
// the value in x will be 0.6r and the value in y will be 17.5lk;
```

But this is not:

```
fract x;
fract y = 1.8 * x;
// Exression "1.8 * x" is of floating-point type. It must be calculated in
runtime, and that is not supported.
```

It is important to note that precision of float type is lower than precision of _Fract and _Accum types while double type has lower precision than long _Fract and long _Accum. For example:

```
fract x = 0.12341512364123; // double type constant
fract y = 0.12341512364123r; // _Fract type constant
fract z = 0.12341512364123f; // float type constant
// x and y will have the same value, but z will not, because precision of
float type is lower than fract.
// Value in x and y will be closer to the actual value of the constant.


long accum a = 0.123415123641231235132;
long accum b = 0.123415123641231235132lk;
// a and b will not have the same value. Value in b will be closer to the
actual value of the constant.
```

Chapter 4
Memory Zones

4

4.1 Overview of Cirrus DSP Architecture Memory Zones

All Cirrus DSPs have two data memory zones (X and Y) and one program memory zone (P). The CCC supports these three memory zones. In Cirrus DSP architecture, there is also a possibility to read and write 64-bit words in a single instruction by accessing the addresses of the X and Y data memory zones. This is the fourth memory zone that the compiler supports, the XY memory zone, also called "L" memory.

When declaring a variable in C code, you can specify the variable with a memory-zone qualifier in which the variable resides. Thus, a variable is defined by its name, its type and its memory zone. If a memory zone is not specified, the compiler assumes X memory zone

4.2 Memory-zone Qualifiers

The Cirrus-C Compiler supports four different memory zones as described in Table 4-1.

Table 4-1. Memory Zones and their Qualifiers

Memory Zones	Memory-zone Qualifiers
Memory zone X	__memX
Memory zone Y	__memY
Memory zone P	__memP
Memory zone XY	__memXY

4.3 Declaration of a Global Variable

A memory zone can be specified only for global variables, which means that variables declared inside a function cannot be allocated in a named address space unless they are also explicitly declared as *static* or *extern*.

When a memory-zone qualifier is not used in a declaration, a global variable is placed in the default memory zone, memory zone X, as shown in Example 4-1.

Example 4-1 Global variable in the default memory zone

C Code:

```
int global = 0;

int main(void)
{
  return global;
}
```

Assembly Code:

```
  .xdata_mpm
  .public _global
_global:
  .dw (0x0)
  .code_mpm

# Function : _main

_main
  a0 = xmem[_global]
__main_END
  ret

  .public _main
```

The only way to place a global variable in a zone other than the default memory zone is to use a memory-zone qualifier, as shown in Example 4-2.

Example 4-2 Using a memory-zone qualifier

C Code:

```
__memY int global = 0;

int main(void)
{
  return global;
}
```

Assembly Code:

```
.ydata_mpm
  .public _global
_global:
  .dw (0x0)
  .code_mpm

# Function : _main

_main
  a0 = ymem[_global]
__main_END
  ret

  .public _main
```

You can also declare a pointer to a variable that resides in a specific memory zone, as shown in Example 4-3.

Example 4-3 Declaring a pointer to a variable in a memory zone

C Code:

```
__memY int global = 0;
__memY int * pglobal = &global;

int main(void)
{
  return *pglobal;
}
```

Assembly Code:

```
   .ydata_mpm
   .public _global
_global:
   .dw (0x0)
   .xdata_mpm
   .public _pglobal
_pglobal:
   .dw _global
   .code_mpm

# Function : _main

_main
   xmem[i7] = i4
   i4 = xmem[_pglobal]
   nop
   a0 = ymem[i4]
__main_END
   i4 = xmem[i7]
   ret

   .public _main
```

In Example 4-3 the pointer variable is placed in the default memory zone (X), but a pointer variable can have a memory-zone qualifier of its own. Example 4-4 shows how a pointer variable that points to the value in the Y memory zone and resides in the Y memory zone is declared.

Example 4-4 Declaring a pointer to a variable that points to a value and resides in a memory zone

C Code:

```
__memY int global = 0;
__memY int * __memY pglobal = &global;

int main(void)
{
  return *pglobal;
}
```

Assembly Code:

```
   .ydata_mpm
   .public _global
_global:
   .dw (0x0)
   .ydata_mpm
   .public _pglobal
_pglobal:
   .dw _global
   .code_mpm

 # Function : _main

_main
   xmem[i7] = i4
   i4 = ymem[_pglobal]
   nop
   a0 = ymem[i4]
__main_END
   i4 = xmem[i7]
   ret

   .public _main
```

4.4 Declaration of a Local Variable

There is no way to specify a memory zone for a local variable. Usage of memory-zone qualifiers in a declaration of a local variable is not allowed. The exception to this rule is a case when a programmer specifies the memory zone to which a local pointer variable is pointing. But in this case, location of the local pointer variable is still handled in the same way as any other local variable. If you try to specify a memory zone for a local variable, the compiler ignores the memory qualifier, and puts the variable in an accumulator or on the stack.

The C program in Example 4-5 shows a declaration of two local pointers to integer data in Y memory zone. One pointer has an additional memory-zone qualifier that indicates that a programmer wants to place the pointer in the Y memory zone. However, both pointers end up as index registers because they do not reside in any zone, but the example pointers are still pointing to the memory zone Y.

Example 4-5 Two local pointers to integer data in Y zone

C Code:

```
__memY int data1 = 1;
__memY int data2 = 2;

int main (void)
{
  __memY int * localP1 = &data1;
  __memY int * __memY localP2 = &data2;
  return *localP1 + *localP2;
}
```

Assembly Code:

```
   .ydata_mpm
   .public _data1
_data1:
   .dw (0x1)
   .ydata_mpm
   .public _data2
_data2:
   .dw (0x2)
   .code_mpm

 # Function : _main
_main
   xmem[i7] = i5;i7 += 1
   i5 = (0) + (_data1)
   xmem[i7] = i4
   b0 = ymem[i5]
   i4 = (0) + (_data2)
   a0 = ymem[i4]
   a0 = a0 + b0
__main_END
   i4 = xmem[i7];i7 -= 1
   i5 = xmem[i7]
   ret

   .public _main
```

4.5 Memory-zone Typecast Operation

After it is declared, a variable cannot change its memory zone using a typecast operation. However, a redirection of a pointer variable is allowed.

Example 4-6 shows the usage of a pointer that is declared to point to long fract data in the XY memory zone to read from the XY zone, but also to write to the X zone and the Y zone. Notice that only one index register is in use, thus saving resources.

Example 4-6 Pointer declared to read from XY zone and write to X zone and Y zone

C Code:

```
#include<stdfix.h>

__memXY long fract whole = 0.27192006lr;
__memX fract high;
__memY fract low;

void main(void)
{
   long accum accumulator;
   __memXY long fract *pointer;
   pointer = &whole;
   accumulator = *pointer;
   whole = accumulator;
   high = *(__memX fract*)pointer;
   low = *(__memY fract*)pointer;
}
```

Assembly Code:

```
   .ydata_mpm
   .public _low
_low .bsc (0x1), 0x00000000
   .data_mpm
   .public _whole
_whole:
    .dd (0x22ce46ca69c61751)
   .xdata_mpm
   .public _high
_high .bsc (0x1), 0x00000000
   .code_mpm

 # Function : _main
_main
   i0 = (0) + (_whole)
   a0 = xymem[i0]
   a0 = long(a0)
   xmem[_whole] = a0h
   ymem[_whole+0] = a0l
```

```
a0 = xmem[i0]
xmem[_high] = a0
b0 = ymem[i0]
ymem[_low] = b0
__main_END
ret
.public _main
```

Chapter 5
Use of Circular Buffers

5

5.1 Overview

The Cirrus 32-bit DSP architecture uses modulo address arithmetic. The CCC takes advantage of this capability by supporting the use of circular buffers in the C code.

To use circular buffers, the file circbuff.h must be included in the C source file. This file defines all the values of the modulo addresses that are supported by the Cirrus 32-bit DSP, and a definition of a macro for use with circular buffers in C code.

5.2 Circular Buffer Modulo Addressing

Modulo addressing can be used to implement circular buffers. The size of a circular buffer must be a power of 2, ranging from 4 to 32768. Table 5-1 contains the supported modulo values.

Table 5-1. Supported Modulo Addressing Values

Symbol	Addressing Mode
MOD_LIN	Linear Addressing
MOD_4	Modulo 4
MOD_8	Modulo 8
MOD_16	Modulo 16
MOD_32	Modulo 32
MOD_64	Modulo 64
MOD_128	Modulo 128
MOD_256	Modulo 256
MOD_512	Modulo 512
MOD_1024	Modulo 1024
MOD_2048	Modulo 2048
MOD_4096	Modulo 4096
MOD_8192	Modulo 8192
MOD_16384	Modulo 16384
MOD_32768	Modulo 32768
MOD_BITREV	Reverse Binary Addressing

Example 5-1 and Example 5-2 show examples of modulo addressing.

Example 5-1 Example 1 of modulo addressing

```
#include <circbuff.h>

int *foo(int *p)
{
   return CIRC_INC(p, MOD_16 + 2);
}
   .code_mpm

   .extern ___circ_inc
 # Function : _foo
_foo
   nm0 = (12290)
   nop
   i0 += n
   AnyReg(a0, i0)
   nm0 = (0)
__foo_END
   ret

   .public _foo
```

Example 5-2 Example 2 of modulo addressing

```
#include <circbuff.h>

void main(_Fract *p3) {
  int i;
  _Fract y0 = 0;
  for (i=0;i<16;i++) {
    *p3 = y0; p3 = CIRC_INC(p3, MOD_64 + 4);
  }
}
   .code_mpm

   .extern ___circ_inc

 # Function : _main
_main
   x0 = (0)
   nm1 = (20484)
   do (16), Boundary_0
Label_3
Boundary_0:
   xmem[i1] = x0;i1 += n
```

```
   nop
   nm1 = (0)
__main_END
   ret

   .public _main
```

To use modulo addressing, circular buffers must be placed in memory such that their base address is a multiple of their size.

Circular buffers can be declared in the C code or Assembler code. If the circular buffer is declared in the C code, it has to use the attribute align directive for it. As shown in Example 5-3, array a is placed at an address that is a multiple of 16 and the address of array b is a multiple of 32:

Example 5-3 Using a circular buffer

```
#include <stdfix.h>
#include <circbuff.h>

__memX fract __attribute__((__aligned__(16))) a[16];
__memY fract __attribute__((__aligned__(32))) b[16];

void foo(__memX fract *p2, __memY fract *p3) {
  int i;
  fract y0 ;
  for (i=0;i<16;i++) {
    y0 = *p3; p3 = CIRC_INC(p3, MOD_16 + 4);
    *p2++ = y0;
  }
}

int main()
{
foo(a, b);
}
   .xdata_mpm align 16
   .public _a
   _a .bsc (0x10), 0x00000000
   .ydata_mpm align 32
   .public _b
_b .bsc (0x10), 0x00000000
   .code_mpm

   .extern ___circ_inc

 # Function : _foo
_foo
   AnyReg(i2, i0)
```

5

```
   nm0 = (12292)
   do (16), Boundary_0
Label_3
   AnyReg(i0, i1)
   x0 = ymem[i1]
   i0 += n
   AnyReg(i1, i0)
Boundary_0:
   xmem[i2] = x0;i2 += 1

   nm0 = (0)
__foo_END
   ret

   .public _foo

 # Function : _main
_main
   i0 = (0) + (_a)
   i1 = (0) + (_b)
Label_1073741823
   call _foo
Label_4
__main_END
   ret

   .public _main
```

Chapter 6
Function Attributes

6

6.1 Function Attributes

Function attributes inform CCC how that function should be compiled. Table 6-1 provides a list of all available function attributes.

Table 6-1. List of Function Attributes

Attribute Syntax	Description
__attribute__((debug_on))	Emits debug information for the function (ignore –g switch). If the -ida command line switch is present, this function is ignored.
__attribute__((debug_off))	Does not emit debug information for the function (ignore –g switch). If the -ida command line switch is present, this function is ignored.
__attribute__((fg_call))	This function represents the entry point for module foreground thread
__attribute__((bg_call))	This function represents the entry point for module background thread
__attribute__((fg_primitive_call))	This function represents the entry point for DSP Composer primitive foreground thread
__attribute__((bg_primitive_call))	This function represents the entry point for DSP Composer primitive background thread

Chapter 7
Generated Code

7

7.1 Overview

This chapter describes aspects of generated code in the Cirrus DSP architecture:

- CCC calling conventions
- Preserving hardware registers during function calls

7.2 CCC Calling Conventions

In the CCC, function arguments are passed in the following manner:

- Arguments are numbered from left to right.
- When passing pointers, index registers are used
- When passing all other basic types (int, fract, accum...), accumulators are used
- Data registers are never used for passed arguments
- Variables of the structure type are passed via the stack.
- Index register i7 is used as a stack pointer. The stack is assumed to be in X memory, and grows upward (toward higher addresses).
- Return values, if any, are in the a0 accumulator.
- For pointer arguments, the following index registers are used, in the following exact order:

  i0, i1, i4, i5

  The first pointer argument is passed through i0, second through i1, and so on.
- If there are more than four pointers in the function argument list, the remainder are passed via the stack.
- For the basic type arguments, the following accumulators are used in this exact order: a0, a1, b0, b1.
  - The first basic type argument is passed through a0, second through a1, etc.
  - Arguments are numbered from left to right.
  - If there are more than four basic type arguments in the function argument list, the remainder are passed via the stack.
- Parameters are placed on the stack in a reversed order (the first parameter needed to be

7

passed getting the highest location on the stack), but read from within the function based on the distance from their location on the stack and the value of the stack pointer at the moment of entry.

- The following examples represent the stack as it is seen from within the called function:

Example 7-1 Example 1 calling conventions

```
foo(int p1, int p2, int p3, int p4, int p5, int p6)
```

Arguments will be passed as shown in Figure 7-1.

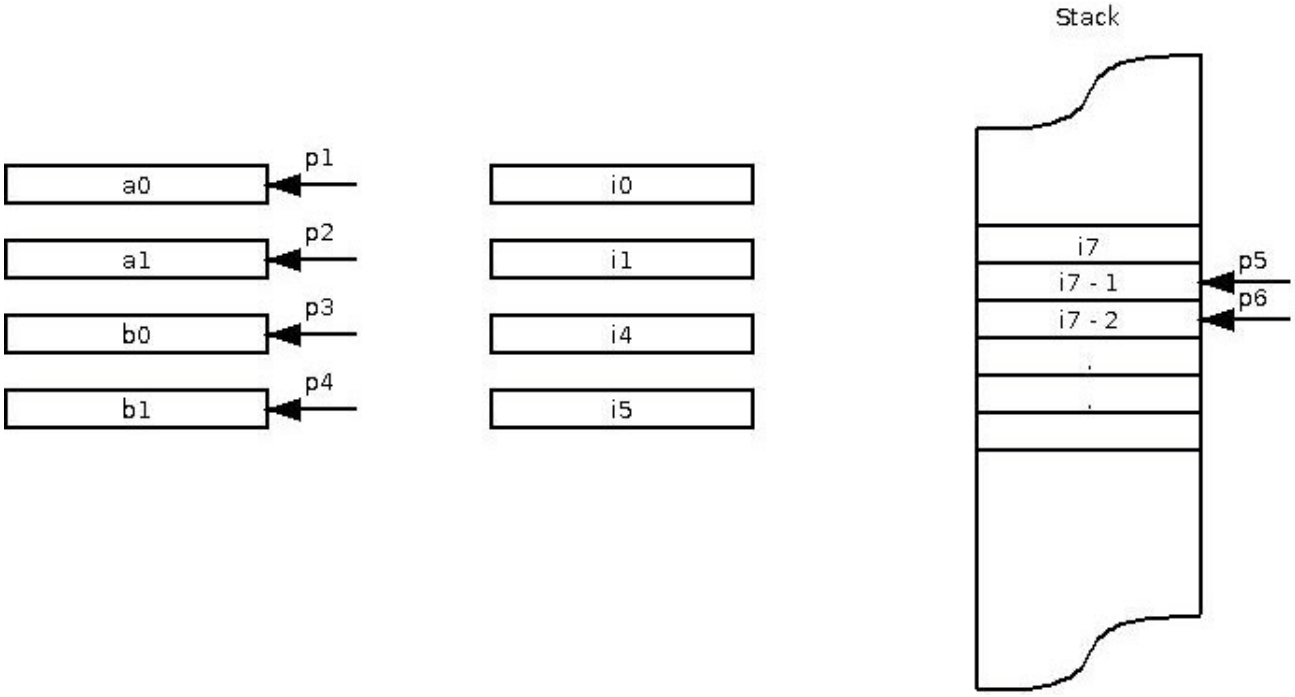

Figure 7-1. Passing of Arguments When Using Accum Data Types

Example 7-2 Example 2 calling conventions

```
foo(int*p1, int*p2, int*p3, int*p4, int*p5, int*p6)
```

Arguments will be passed as shown in Figure 7-2.

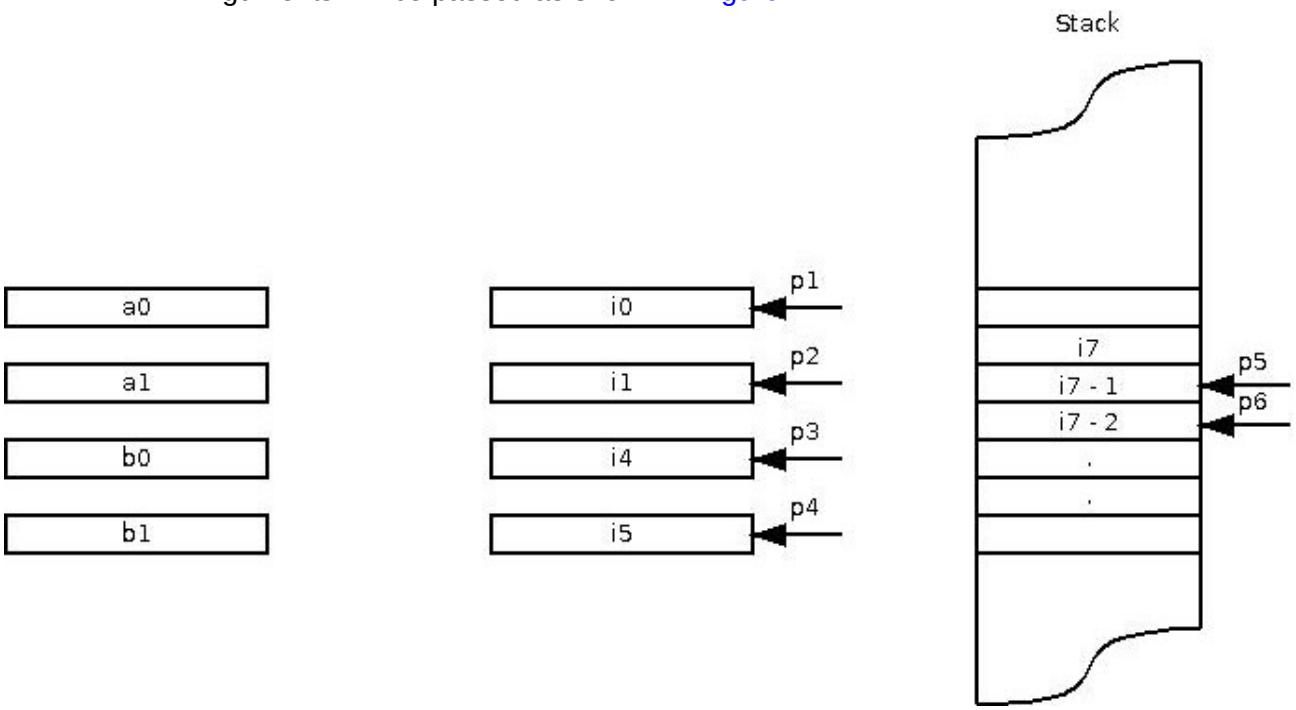

Figure 7-2. Passing of Arguments When Using Ptr Data Types

Example 7-3 Example 3 calling conventions

```
typedef struct ExampleStruct
{
     int m1;
     int*m2;
     long accum m3;
}
foo(ExampleStruct p1, ExampleStruct p2)
```

Arguments will be passed as shown in Figure 7-3.

7

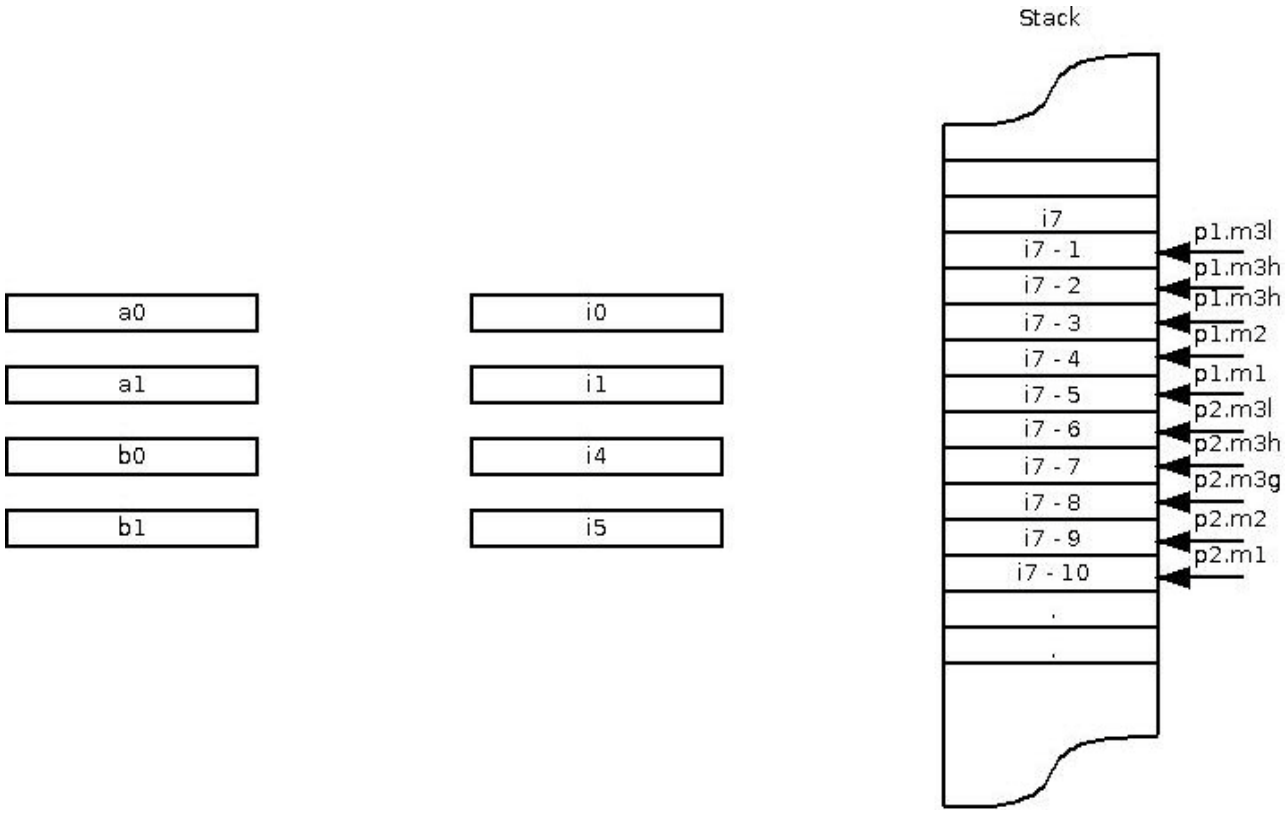

Figure 7-3. Passing of Arguments When Using Struct Data Types

Values of long fract or long accum types display an apparently different behavior when it comes to storage on the stack. The reason for this is the difference in the size of these types compared to the other basic types (64 and 72 bits as opposed to 32 bits, these sizes translate to 3, 2, and 1 memory locations, respectively).

The fact that the stack pointer is first moved for the number of locations required to store the value after which parts of the variable are stored on the stack one by one toward the previous value of the stack pointer causes the need for more steps to be taken in case of the 2 mentioned types. This behavior is not visible for other basic types since the whole process is finished in one step. This information is diagrammed in Figure 7-4.

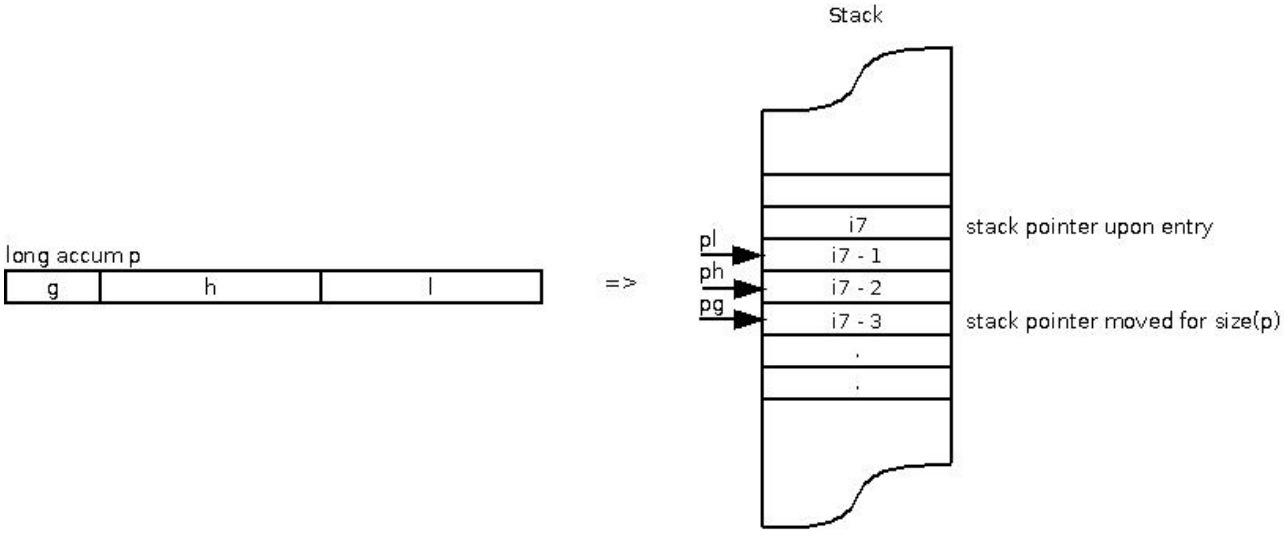

Figure 7-4. Storing Values of Long Accum Type on the Stack

Example 7-4 Example 4 calling conventions

```
foo(long accum p1, long accum p2, long accum p3, long accum p4, long accum p5,
long accum p6)
```

Arguments will be passed as shown in Figure 7-5.

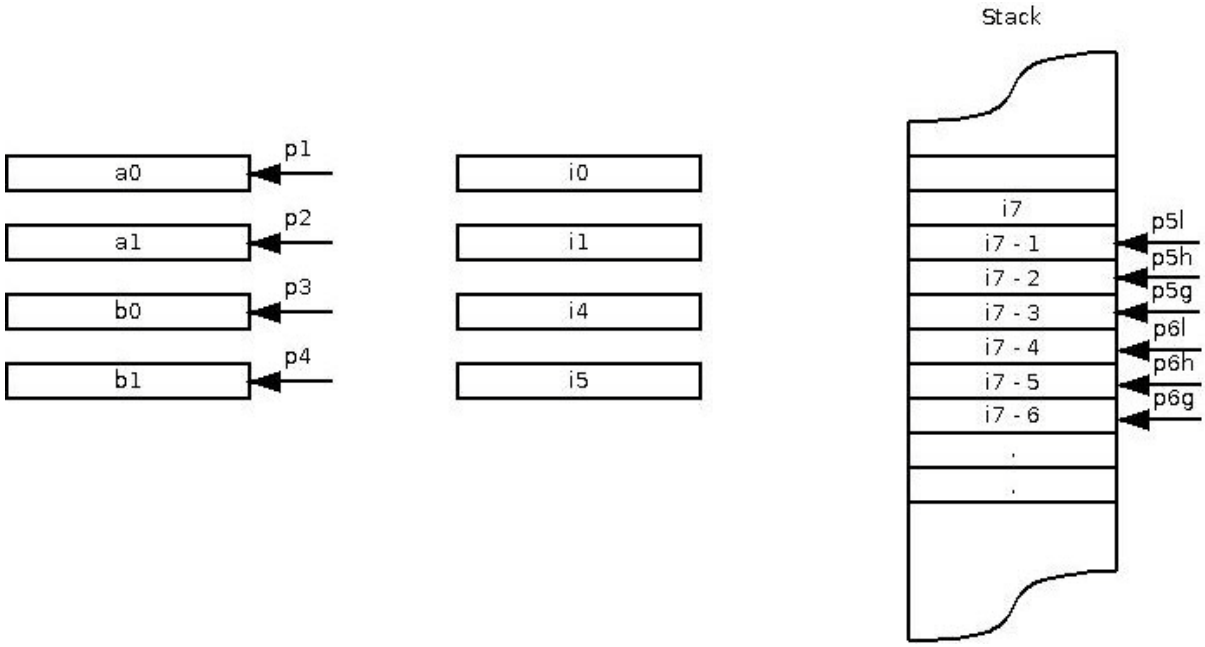

Figure 7-5. Passing of Arguments When Using Long Accum Data Types

7

Example 7-5 Example 5 calling conventions

```
typedef struct ExampleStruct
{
     int m1;
     int*m2;
     long accum m3;
}
foo(int p1, int*p2, Example Struct p3, long accum p4, int p5, int*p6, int p7,
int*p8, int p9, long accum p10, int*p11, int*p12)
```

Arguments will be passed as shown in Figure 7-6.

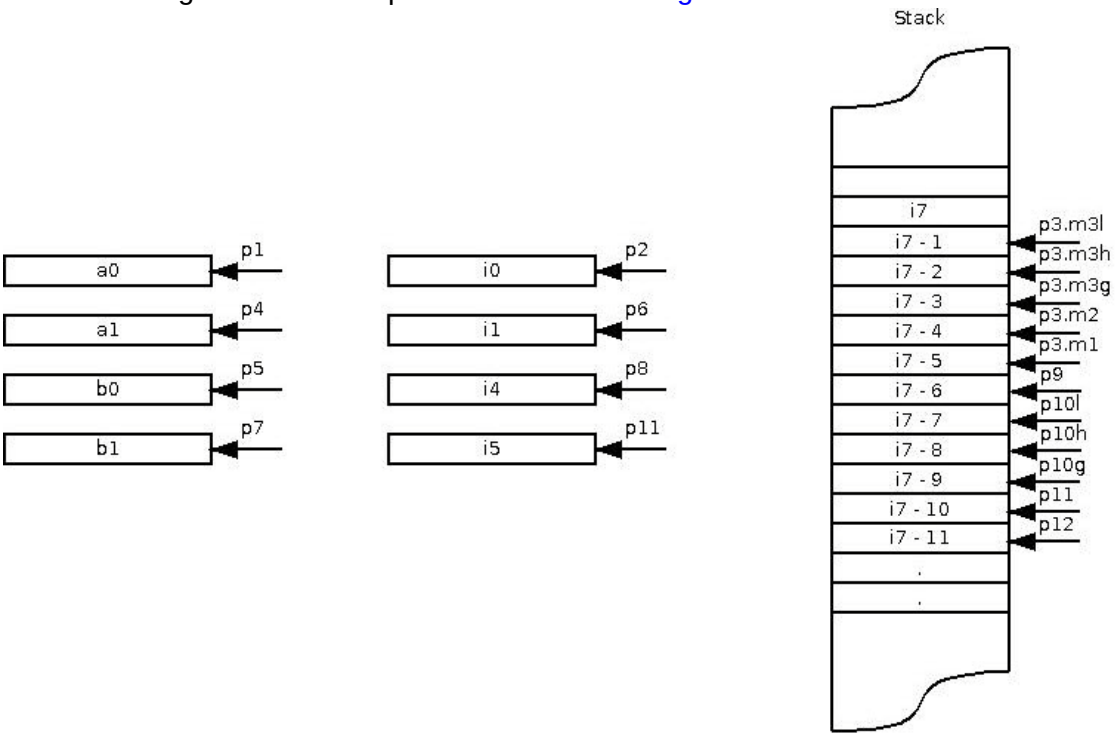

Figure 7-6. Passing of Arguments When Using Mixed Data Types

Some function attributes can change calling convention significantly. Attributes like *fg_call* and *fg_primitive_call,* force compiler to use user defined array named *__C_STACK_FG* as a stack. *__C_STACK_FG* array is expected to be defined by the user in the same memory zone in which stack is considered to be (x memory zone by default). Similar to this b*g_call* and b*g_primitive_call,* force compiler to use user defined array named __C_STACK_BG as a stack. In addition, function attributes *fg_primitive_call* and b*g_primitive_call* force compiler to use Composer primitive calling convention for calling functions with those attributes.

Normally when calling primitives, Composer is using index registers i7, i6 and i5 for pointer argument passing. Compiler in other hand is using i0, i1 and i4. When *fg_primitive_call or* b*g_primitive_call attributes* are used, compiler realizes that Composer primitive calling convention is used and that it should emit instructions which would redirect pointer arguments to the expected resources. Example 7-6 shows generated instructions for pointer arguments redirection:

Example 7-6 Generated instructions if fg_primitive_call and bg_primitive_call function arguments are used

```
void __attribute__((fg_primitive_call))  main()
{
}
      .extern ___C_STACK_FG


.code_ovly

_main
      i0 = i7 + (0)
      i7 = (___C_STACK_FG)
      i1 = i6 + (0)
i4 = i5 + (0)
__main_END
      ret

.public _main
```

7.2.1 Static Stack Frame Calling Convention

When the -fnfs option is on, no stack would be used. For every function argument that needs to be passed by stack according to calling convention, memory will be statically allocated. Assembly symbols for those memory locations are formed like this:

__funcname_param_no, where 'funcname' is function name, and 'no' is the index of the parameter numbering them from left to right, while the first index is 0.

Example 7-7

```
foo(int p1, int p2, int p3, int p4, int p5, int p6);
```

For p5, the __foo_param_4 symbol will have its address, while for p6, it will be __foo_param_5.

7.3 Hardware Register Preservation During Function Calls

The CCC allows both traditional C functions and also C-callable assembly functions. Each type of function has special considerations for preserving hardware register contents.

7.3.1 The CCC Translates C Functions into Assembly Functions

The CCC uses the registers i0, i1, i4, i5, a0, a1, b0, b1, i7 (as the stack pointer) and the stack to pass the C function's arguments into the translated assembly function. See Section 7.2, "CCC Calling Conventions" on page 7-1 for the details of argument passing.

In addition to unused parameter registers, the registers a2, a3, b2, b3, x2, x3, y2, y3, i2, i3, and i6 are considered nonvolatile and must be saved and restored by a function that uses them.

The registers x0, x1, y0 and y1 are considered volatile and must be considered destroyed on function calls.

7.3.2 User-defined C-callable Assembly Functions

It is often useful to call an assembly function from the C code. If a user defines one of these assembly functions, the user **must** include instructions within that assembly function to save and restore any of the following registers that are modified by it: a2, a3, b2, b3, x2, x3, y2, y3, i2, i3, i6 and i7. This is critical because other functions assume values that are preserved in these registers.

Example 7-8 shows how to define and use a C-callable assembly function:

Example 7-8 C-callable assembly function

Your C source file that uses a C-callable assembly function:

```
#include <stdfix.h>

 /* declare the function; it could be defined in a separate file that is
linked in at compile time */
extern accum example_assembly_function(accum a, accum b);
       int main()
{
          accum  a1_var, a0_var, b0_var;

          a0_var = example_assembly_function(a1_var, b0_var);
}
```

The assembly function for this example should look like this:

```
_example_assembly_function:

      # argument a is in a0

      # argument b is in a1
      ...
      # return value should be in a0
      ret
Note the '_' at the beginning of the label.
```

7.3.3 Register Clobber Information Pragma

The register clobber information pragma may be used when an assembly language function is called from C. This pragma provides the C compiler with information about registers clobbered by the assembly language function. The information is useful only when interprocedural register allocation is enabled (-ira command line flag). This enables the compiler to organise register usage of the caller function in a way which renders the writer of the assembly function free from the work of preserving register values. Based on the information, the compiler can generate the store and load instructions, before and after the call to the assembly function, required to preserve data which could not be moved to non-clobber registers. The register clobber information pragma only effects C function prototypes that are located one line after it.

```
Syntax: #pragma ASM_FUNC_CLOBBERED_REGS(<string of register names>)
```

The string of register names consists of one or more of the following register names, separated by commas:

a0,a1,a2,a3,b0,b1,b2,b3,x0,x1,x2,x3,y0,y1,y2,y3,i0,i1,i2,i3,i4,i5,i6

An example of a valid pragma looks like the following:

```
#pragma ASM_FUNC_CLOBBERED_REGS("a0,A2,b1,X3,i6")
```

Malformed register name list will still be parsed and will not cause errors under most circumstances. Unacceptable names and indexes are skipped, and the next register is parsed. Also note that i7 is reserved for the stack pointer and will not be assigned to the function used registers and is reported as a warning.

The following are examples of pragma use:

Example 7-9 Pragma Added to C Source

```
#pragma ASM_FUNC_CLOBBERED_REGS("a0")
void foo();
```

Relevant part of resulting code:

```
_main:
a1 = xmem[_x + 0]
call (_foo)
a0 =+ a1
ret
```

Pragma indicates that foo will use only a0, therefore the return value is placed inside it after the function call.

```
#pragma ASM_FUNC_CLOBBERED_REGS("a1")
void foo();
```

Relevant part of resulting code:

```
_main:
a0 = xmem[_x + 0]
call (_foo)
```

```
        ret
```

Pragma indicates that foo will use only a1, therefore the return value can be placed inside immediately, without risk of it being clobbered by the function call.

Chapter 8
Assembler Instructions with C-expression Operands

8

8.1 ASM Statement Overview

An ASM statement enables a programmer to specify the operands of assembler instructions using C expressions. A typical use can be found in an example of calling an assembly function from the C program or making some specific, small part of the code more efficient. In other words, an ASM statement provides the means for mixing assembler instructions with C statements. Example 8-1 uses the ASM statement to produce a "real multiply" instruction in the output assembler code. See Section 5.1.3 of the *32-bit DSP Assembly Programmer's Guide* for more information.

Example 8-1 Use ASM statement to produce a "real multiply" instruction

```
#include <stdfix.h>

accum foo(fract a, fract b) {
accum tmp;
asm("%0 = %1 * (unsigned)%2" : "=a"(tmp) : "x"(a), "y"(b));
return tmp;
}
.code_mpm

Compiler returns:

 # Function : _foo
_foo
AnyReg(x0, a0)
AnyReg(y0, a1)
Label_1610612734
#ASM_BLOCK_BEGIN
a0 = x0 * (unsigned)y0
#ASM_BLOCK_END
Label_1610612735
__foo_END
ret

.public _foo
```

8

8.2 Implementation of Extended ASM Support

The implementation of the extended ASM support in the CCC follows GNU GCC ASM extension definition, with the exception of advanced operand reference within the assembler code (using %[name] construction, available in GCC as of version 3.1).

The syntax of an extended ASM statement is as follows:

```
asm (
asm string template
[:constraint(output C operand)[,constraint(output C operand)...]]
[:constraint(input C operand)[,constraint(input C operand)...]]
[:clobber[,clobber]]
);
```

The first colon separates the ASM string template from the first output operand. The second colon separates the last output operand from the first input, if any. If there are no output operands, but there are input operands, the programmer must place two consecutive colons surrounding the place where the output operands would normally be listed. Commas separate the operands within each group.

An ASM statement consists of the following parts discussed in the next three sections:

- ASM string template
- Output operand list followed by input operand list
- Clobber list

8.2.1 ASM String Template

An ASM string template is a string containing the assembly instructions with %<digit> placeholder where the operand should be. Operands are numbered from left to right starting from zero as shown in Example 8-2.

Example 8-2 Numbering of operands

```
 asm("AnyReg(%0,%4)" : "=U"(out1), "=U"(out2) : "U"(in1), "U"(in2),
"U"(in3));
```

The first output operand (out1), and the third input operand (in3) are referenced within an ASM string template. The compiler simply puts real register names in the place of the %digit in the ASM string template and then prints the string to the output assembler file. If the ASM string template contains invalid assembler instructions, the assembler will report the corresponding error. This means that it is up to the programmer to check the validity of the ASM string template.

Because the character '%' is a reserved character, to produce it one time in the compiler-generated assembler code requires "%%" to be specified in the ASM string template.

Multiple instructions in the ASM string template should be separated by "\n\t", that is with new-line and tab indicators. In Example 8-3, both of these statements yield the same output code:

Example 8-3 Separating multiple instructions

```
asm("AnyReg(%0,%2)\n\tAnyReg(%1,%3)" : "=U"(out1), "=U"(out2) : "U"(in1),
    "U"(in2));
```

or

```
asm("AnyReg(%0,%2)
    AnyReg(%1,%3)" : "=U"(out1), "=U"(out2) : "U"(in1), "U"(in2));
```

8.2.2 Output Operand List Followed by Input Operand List

Each operand is described by an operand-constraint string followed by the C expression in parentheses. The total number of operands is currently limited to 10.

8.2.2.1 C Expressions

A C expression represents the interface between the assembler operands (which are registers) and C operands. A C expression can represent a destination of a value found in an assembler register after the execution of the ASM statement block (in case of the output operand), initialization expression of an assembler register (in case of the input operand) or both (in case of the input-output operand).

8.2.2.2 Operand-constraint Strings

An operand-constraint consists of the modifier constraint and the simple constraint.

8.2.2.2.1 Simple Constraints

Simple constraints declare the kinds of register that can be used as operands. Simple constraints available for Cirrus DSP architecture include the following:

- 'a' accumulators (`"a0", "a1", "a2", "a3"`)
- 'b' accumulators (`"b0", "b1", "b2", "b3"`)
- 'x'—one of the 4 X data registers (`"x0", "x1", "x2", "x3"`)
- 'y'—one of the 4 Y data registers (`"y0", "y1", "y2", "y3"`)
- 'i'—one of the 7 index registers (i7 is used by the compiler while i8 - i11 are reserved for operating system) (`"i0", "i1", "i2", "i3", "i4", "i5", "i6"`)
- 'U'—any register: accumulator, data or index register; compiler will take the first available

All input operands are assigned to different registers. The same goes for output operands. But an output operand can be located in the same register as an input operand, on the assumption that the inputs are consumed before the outputs are produced. The input-output operands are always assigned to unique registers.

A programmer should declare the operands in a way that there are not too many of them demanding different registers of the same kind. For example, there cannot be more than four input operands with the 'x' constraint, which means that the following code in Example 8-4 produces an error, since there are only 4 'x' data registers available:

8

Example 8-4 Error due to too many 'x' input operands

```
asm("%0 = %1*%2
     %0 += %3*%4
     %0 += %5*%6" : "=a"(out) : "x"(in1), "x"(in2), "x"(in3), "x"(in4),
"x"(in5), "x"(in6));
```

8.2.2.2.2 Modifier Constraints

Modifier constraints define operand kind. The modifiers supported by CCC are the following:

- "" (no modifier)—input operand. Indicates that the operand is a read-only. Modifier can be omitted only for the operands in the input operand list. The value of the C expression of the input operand will be its initial value.
- "="—output operands. Indicates that the operand is a write-only. This modifier can be used only for the operands in the output operand list. A C expression of the output operand must be the lvalue. After execution of the ASM statement block, the value of the output operand is stored in the C expression.
- "+"—input-output operands. Provides a way to declare a read-write operand. This modifier can be used instead of a "=" modifier. C expression of the input-output operand will initiate the operand and the same expression is also be the destination of the operand's final value.
- "&"—early clobber operand. Assuming that the inputs are consumed before the outputs can be a false assumption if the assembler code actually consists of more than one instruction. In this case, the "&" modifier can be used to specify an output operand, which should not be in the same register as any input operand or to specify an input operand, which should not be in the same register as any output operand.

  Suppose that in this small example:

  ```
  asm("AnyReg(%0,%2)/n/tAnyReg(%1,%3)" : "=U"(out1), "=U"(out2) : "U"(in1),
  "U"(in2));
  ```

  out1 is made to take the value of "in1" and out2 the value of "in2." The operands 0 and 3 can be assigned to the same register because one is input and the other is an output operand. Then, the generated assembler code would look like this:

  ```
  AnyReg(a0,a1)
  AnyReg(a1,a0)
  ```

Thus, after execution of the ASM statement block, the value of operand 3 (in2) will not be in operand 1 (out2). Write the ASM statement as follows:

```
asm("AnyReg(%0,%2)/n/tAnyReg(%1,%3)" : "=&U"(out1), "=U"(out2) : "U"(in1),
"U"(in2));
```

or

```
asm("AnyReg(%0,%2)/n/tAnyReg(%1,%3)" : "=U"(out1), "=U"(out2) : "U"(in1),
"&U"(in2));
```

would solve the problem and the output assembler code looks like this:

```
AnyReg(a0,a1)
```

```
AnyReg(a1,a2)
```

which produces the desired result.

Note: Writing "&" does not obviate the need to write "=".

Another way of declaring input-output operands is to write a digit as a constraint for an input operand. The digit used must be a reference to an already declared output operand. This enables the programmer to logically split an input-output operand into two separate operands, one input operand and one output operand. This approach allows the initialization and destination expression of an operand to be different. Here are two examples that illustrate the difference between the two ways of declaring input-output operands. See Example 8-5.

Example 8-5 Example 1 declaring input-output operands

```
int foo ()
{
int to = 2;
asm("%0 = %0 + (5)" : "+i"(to));

return t0;
}
```

The value of the variable in Example 8-5, after execution of the ASM statement, is 7.

But, in Example 8-6,

Example 8-6 Example 2 declaring input-output operands

```
int to = 2;
int ti = 3;
asm("%0 = %0 + (5)" : "=i"(to) : "0"(ti));
```

The value of the variable ti, which is 2 in this example, is used for initialization of the %0 operand. The value in the to variable is 8.

8.2.3 Clobber List

A clobber list is a list of registers that are implicitly used or changed by assembler instructions used in the ASM statement. In Example 8-7, the compiler does not know that registers a0 and a1 are actually used, or as in this case, changed.

Example 8-7 No clobber list

```
asm("AnyReg(a0,%0)\n\tAnyReg(a1,%1)" : : "a"(1), "a"(2));
```

The compiler's assembler output looks like this:

```
AnyReg(a1,a0)
AnyReg(a0,a1)
```

Thus, the value 1 ends up in both registers, instead of value 1 in a0 and value 2 in a1. Any value that can have resided in register a0 or register a1 before the ASM statement execution would be corrupted, and all the variables that were assigned to those registers would also be corrupted.

With usage of the clobber list, the above example would look like this:

Example 8-8 Using a clobber list

```
asm("AnyReg(a0,%0)\n\tAnyReg(a1,%1)" : : "a"(1), "a"(2) : "a0", "a1");
```

The compiler determines that registers a0 and a1 are implicitly used and it would maintain the consistency of all variables. Additionally, it does not assign registers a0 and a1 to any of the ASM statement operands.

Registers that can be stated in a clobber list are:

```
"a0", "a1", "a2", "a3", "b0", "b1", "b2", "b3" (all accumulators)
"x0", "x1", "x2", "x3", "y0", "y1", "y2", "y3" (all data registers)
"i0", "i1", "i2", "i3", "i4", "i5", "i6" (first seven index registers)
```

It is not advisable for the programmer to use assembler instructions that change the other five index registers (i7-i11). Because the five index registers are never assigned to any operand, there is no need to put them in the clobber list.

A typical use of the ASM statement is to create a C-callable assembly function. Example 8-9 shows how _asmfunc (an assembly function) can be called from func() in C. Function _asmfunc expects parameters in a0 (count), i0 (dest) and x0 (src), while it returns a value in a0 and changes i3 and y2 during its execution.

Example 8-9 _asmfunc called from C func()

```
int func (int* src, int* dest, int count)
{
int result;

asm("
\tx0=%1; a0=+%3
\ti0=%2
\tcall _asmfunc
\t%0=+a0"
: "=a"(result)
: "y"(src), "y"(dest), "a"(count)
: "a0", "i0", "i3", "x0", "y2");

return result;
}

int main()
{
int from[100], to[100];
return func(from, to, 100);
```

Chapter 9
Optimizing C Code Targeted for Cirrus Logic DSPs

9

9.1 Overview

This chapter describes steps to develop C code for optimal performance when running on the Cirrus Logic DSP.

Section 9.2 describes coding guidelines that a developer should use to create a new C language-based application optimized for the Cirrus Logic DSP platform.

Section 9.3 describes a methodology for taking an existing C-language application and turning it into an executable that runs on a Cirrus Logic DSP.

9.2 C-Language Coding Guidelines

Follow the coding guidelines described in this section in order to generate an executable file targeted for Cirrus Logic DSPs. The coding guidelines are based on the requirements of the CCC and the Cirrus Logic DSP hardware platform. The coding guidelines cover the following topics:

- General coding guidelines (Section 9.2.1)
- CCC-specific and Cirrus Logic DSP coding guidelines (Section 9.2.2)

9.2.1 General Coding Guidelines

Follow these general coding guidelines in order to develop application code that the CCC can compile for optimal performance on the Cirrus Logic DSP platform:

1. Reduce the number of arguments that are passed to functions.

   Passing arguments always requires additional instructions at the place where the function is called (to push arguments on the stack or store in registers) and at the beginning of the function body (to pop arguments off the stack or read from registers). In the case of a defined symbol or a global variable, the value of the global variable is used directly in place where it is needed.

   Passing arguments is especially costly when passed via stack. To know in which cases this is happening, see the description of the calling convention in Section 7.2. Also, when option -emit-hints is turned on, compiler reports all function calls where stack is being used for passing arguments. Try changing the code so that there are no reports of stack usage for argument passing. Alternatively, consider the option -fnfs.

Example 9-1 Convert Function Parameters to Global Variables or Defined Symbols

```
int foo (int window_size, int sample_rate, int number_of_channels)

Optimized example:
#define window_size 512
int sample_rate 32000;
#define number_of_channels 7
int foo ();
```

2. Eliminate instances where structures are passed as function arguments, either by value or by pointer.

   When a structure is passed by value, make the structure global, and access the structure's fields directly within the function body, rather than passing it in the parameter list.

   In Example 9-1, some function parameters are converted to global variables or defined symbols. In Example 9-2, structures are made global. In Example 9-3, structure pointer passing is eliminated.

   See Section 7.2 for a description of how arguments passed in code are compiled by the Cirrus Logic C-Compiler.

Example 9-2 Make Structures Global

```
int foo (t_lvl_mod_cfg S)
{
     ...
     x = S.scale_factor;
     ...
}
Optimized example:
 t_lvl_mod_cfg g_S;
int foo ()
{
     ...
     x = g_S.scale_factor;
     ...
}
```

Accessing a structure field through a pointer is slightly less efficient than using direct access. In the case when a pointer to a structure is passed to a function, check to see if the function is always called with a pointer to that structure, or if the function can take different pointers to different structures. If the same pointer is always passed, then the pointer passing can be removed, and the structure accessed directly in the body of the function.

Example 9-3 Eliminate Structure Pointer Passing

```
int foo (t_lvl_mod_cfg* pS)
```

```
        {
           ...
           x = pS->scale_factor;
           ...
        }
        void main ()
        {
           ...
           return foo (&g_S);
        }
```

Optimized example:

```
        int foo ()
        {
           ...
           x = g_S.scale_factor;
           ...
        }
```

When the passing of a structure as argument is a must, keep in mind that passing a structure pointer is more efficient than passing a structure by value. Passing by value uses local stack for that argument while field access is equally inefficient. When option -emit-hints is on, compiler reports function calls where stack is being used for passing structures by value. Change those functions so that structure pointer is passed or consider using -fnfs option.

9.2.2 CCC and Cirrus Logic DSP-specific Optimizations

There are five types of optimizations for the C-Compiler and Cirrus Logic DSPs:

- Create loops that can be implemented as hardware loops (Section 9.2.2.1).
- Change the method of accessing arrays (Section 9.2.2.2).
- Increase the efficiency of comparison code (Section 9.2.2.3).
- Copy whole structures or parts of structure that are contiguous in memory (Section 9.2.2.4).
- Avoid using large switch statements or if-else constructions (Section 9.2.2.5).

9.2.2.1 Create Loops That Can Be Implemented as Hardware Loops

The CCC will translate to hardware loops only those loops for which it can calculate the number of iterations in compile-time. The loops that have a compile-time determinable number of iterations we will call "fixed-iterations loops". Of all possible fixed-iterations loops written in C, the CCC can only calculate the number of iterations for some cases. General guidelines for writing loops that the CCC can translate into hardware loops are:

- The loop boundaries must not change during the loop execution.
- The loop index must be updated only by addition or subtraction of a constant value.
- The loop index must not be updated in a conditional statement.

If you write a fixed-iterations loop which is not being translated to a hardware loop, turn on -emit-hints option, and see what the compiler reports for that loop. You should get precise reasons why the compiler cannot handle that particular loop as a hardware loop.

There are two groups of fixed-iterations loops. One group contains loops that have a constant number of iterations. Another group contains loops that have a number of iterations that varies between different evocations of the loop but is fixed once the loop starts execution (Example 9-8). The translating of the second group of loops to hardware loops is possible only when the -funsafe-loop-optimization option is on. That option lets the compiler assume that the number of iterations in such loop is never going to be 0. That assumption enables the compiler to safely translate those loops into hardware loops.

Here are several examples of loops that the CCC can successfully translate into hardware loops:

Example 9-4 Example 1 of loop that can be implemented as hardware loop by CCC

```
int foo ()
{
   int i, sum = 0;
   for (i = 0 ; i < 10 ; i++)
      sum += array1[i] + array2[i];
   return sum;
}
```

Example 9-5 Example 2 of loop that can be implemented as hardware loop by CCC

```
int foo ()
{
   int i, sum = 0;
   for (i = -5 ; i < 5 ; i+=2)
      sum += array1[i] + array2[i];
   return sum;
}
```

Example 9-6 Example 3 of loop that can be implemented as hardware loop by CCC

```
#define win_size 512
int foo ()
```

```
{
   int i, sum = 0;
   for (i = 0 ; i < win_size ; i++)
      sum += array1[i] + array2[i];
   return sum;
}
```

Example 9-7 Example 4 of loop that can be implemented as hardware loop by CCC

```
int foo ()
{
   int i, sum = 0;
   for (i = 0 ; i < 1024 ; i *= 2)
      sum += array1[i] + array2[i];
   return sum;
}
```

Example 9-8 Example 5 of loop that can be implemented as hardware loop by CCC

```
int foo (int n)
{
int i, sum = 0;
for (i = 0 ; i < n ; i *= 2)
sum += array1[i] + array2[i];
return sum;
}
```

This only works if the -funsafe-loop-optimizations option is on.

9.2.2.2 Change the Method of Accessing Arrays

Access arrays using pointers instead of square bracket indexing.

Square bracket indexing is undesirable because it creates the index as a regular integer. Every time the array is accessed the value of the index is added to array's base address to obtain the address of a particular element in the array.

By using pointers to access the array, as shown in Example 9-9, the Cirrus Logic DSP's AGU is utilized along with index register update instructions to optimize array access.

Example 9-9 Use Pointers to Access Arrays

```
int foo ()
{
   int i, sum = 0;
   for (i = 0 ; i < 10 ; i++)
      sum += array1[i] + array2[i];
   return sum;
}
```

Optimized example:

```
int foo ()
{
   int i, sum = 0;
   int* p_array1 = array1;
   int* p_array2 = array2;
   for (i = 0 ; i < 10 ; i++)
      sum += *p_array1++ + *p_array2++;
   return sum;
}
```

There are cases when compiler is smart enough to recognize square bracket indexing and change it to pointer addressing at compile time. For this to work, the conditions described in Table 9-1 have to be met:

Table 9-1. Conditions for Translating Square Bracket Indexing into Pointer Addressing

Conditions	Unsuccessful Addressing Translation
Condition 1: Array square bracket indexing must be inside a loop.	Example 9-11
Condition 2: The loop must be such that the compiler can translate it to a hardware loop.	Example 9-12
Condition 3: The loop iteration variable must be used for array indexing.	Example 9-13
Condition 4: Array must be one-dimensional.	Example 9-14

Example 9-10 provides an example of square bracket indexing where it is not necessary to change to pointer addressing, while still producing optimal code:

Example 9-10 Use Pointers to Access Arrays

```
int foo ()
{
int i;
int sum = 0;
for (i = 0; i < 10 ; i++)
   sum += array1[i] + array2[i];
return sum;
}
```

Example 9-11 to Example 9-14 show situations where it is necessary to use a pointer instead of a square bracket indexing is needed:

Example 9-11 Condition 1 (square bracket indexing must be inside a loop) not satisfied (Error noted in Red)

```
int foo ()
{
int i;
int sum;
sum += array1[i] + array2[i]; <square bracket indexing is not in a loop>
return sum;
}
```

Example 9-12 Condition 2 (loop must be such that the compiler can translate it to a hardware loop) is not satisfied (Error noted in Red)

```
int foo ()
{
int i = 0;
while (0 == array[i]); <loop cannot be translated to hardware loop>
i += 1;
return array[i];
}
```

Example 9-13 Condition 3 (loop iteration variable must be used for array indexing) is not satisfied (Error noted in Red)

```
int foo ()
{
int i, sum = 0, j = 0;
for (i = 0 ; i < 10 ; i++)
sum += array[j++]; <i must be used for indexing>
return sum;
}
```

Example 9-14 Condition 4 (array must be one-dimensional) is not satisfied (Error noted in Red)

```
int foo ()
{
```

```
int i;
int sum = 0;
for (i = 0; i < 10 ; i++)
sum += array1[i][0]; <array is not one dimensional>
return sum;
}
```

9.2.2.3 Increase the Efficiency of Comparison Code

When comparing two variables and subtracting them, there could be an overflow. The CCC handles overflows by preparing the values (sign extending and dividing by 2) prior to subtraction. Example 9-15 illustrates this problem.

Example 9-15 Comparing variables

```
#include <stdfix.h>

int foo (long accum a , long accum b)
{
if (a >= b)
return 1;
else
return 0;
}
.code_mpm

 # Function : _foo
_foo
a0 = a0 >> 1
a1 = a1 >> 1
Label_1073741826
a0 = a0 - a1
if(a>=0) jmp Label_0
uhalfword(a0) = (0)
jmp (__foo_END)
Label_0
uhalfword(a0) = (1)
__foo_END
ret

.public _foo
```

In Example 9-15 there are two instructions that are used only to prepare the values in the accumulators for subtraction. In cases when it is certain that there will be no overflow, the code in Example 9-16 can be used and the two instructions can be eliminated.

Example 9-16 Shorter assembler output code

```
#include <stdfix.h>
```

```
int foo (long accum a , long accum b)
{
if (a - b >= 0)
return 1;
else
return 0;
}
.code_mpm

 # Function : _foo
_foo
Label_1073741824
a0 = a0 - a1
if(a>=0) jmp Label_0
uhalfword(a0) = (0)
jmp (__foo_END)
Label_0
uhalfword(a0) = (1)
__foo_END
ret

.public _foo
```

9.2.2.4 Copy Whole Structures That Are Contiguous In Memory

Copy whole structures or parts of structure that are contiguous in memory.

Example 9-17 Copying structures, excluding fields that maintain their values

```
struct S {
  int a, b, c, d, e, f, g;
};
typedef struct S tS;
...
tS X, Y;
...
X.a = Y.a;
X.b = Y.b;
X.c = 1;
X.d = Y.d;
X.e = Y.e;
X.f = 0;
X.g = Y.g;
...
Optimized code:
...
X = Y;
```

```
X.c = 1;   //Excluding c,f to maintain their values
X.f = 0;
...
```

Example 9-18 Rearranging fields in a structure

```
struct S {
  int a, b, c, d, e, f, g;
};
typedef struct S tS;
...
tS X, Y;
...
X.a = Y.a;
X.b = Y.b;
X.d = Y.d;
X.e = Y.e;
X.g = Y.g;
...
//Difference here is that fields c and f are not set, but need to maintain
//their previous value instead)
Optimized code:
struct S {
  int a, b, d, e, g, c, f; // Note reordering of the fields
};
typedef struct S tS;
...
tS X, Y;
...
int i;
int* tmpX = &(X.a);
int* tmpY = &(Y.a);
for (i = 0 ; i < 5 ; i++)
  *tmpX++ = *tmpY++;
...
```

In Example 9-17 and Example 9-18, a hardware loop with two instructions is used. In the optimized code, copying any number of fields is translated into five instructions:

- Two instructions for pointer setting
- One for a do instruction
- Two for a loop body.

With the unoptimized code, copying a single field produces two instructions each. For example, copying five fields, field by field, produces ten instructions.

Notice that the coding examples shown in Example 9-17 and Example 9-18 are optimized for code size only, where code size = number of instructions. Code optimized for size uses less program memory, but the actual code run is 3 instructions longer.

9.2.2.5 Avoid Using Large Switch Statements or If-else Constructions

Try to simplify conditional logic whenever possible. Avoid large switch statements or if-else constructions when writing code to take advantage of the Cirrus Logic DSP's calculation ability. When programming in Assembly language, the necessity of simplifying conditional logic is obvious. But, when writing C Language applications, the programmer can focus on complying with standard C Language coding conventions, rather than the impact of the code's implementation on the operation of the DSP.

Example 9-19 Changing a switch statement into an array

```
enum OutputConfig { OUT_MONO, OUT_STEREO, OUT_4CH, OUT_3_1CH, OUT_5_1CH,
OUT_5CH, OUT_7CH };
...
switch (output_type)
{
case OUT_MONO:
  value = 0x7a;
  break;
case OUT_STEREO:
  value = 0x82;
  break;
case OUT_4CH:
  value = 0xEB;
  break;
case OUT_3_1CH:
  value = 0x11;
  break;
...
}
Optimized code:
enum OutputConfig { OUT_MONO, OUT_STEREO, OUT_4CH, OUT_3_1CH, OUT_5_1CH,
OUT_5CH, OUT_7CH };
int OCArray [7] = {0x7a, 0x82, 0xEB, 0x11 ... };
...
value = OCArray[output_type];
```

9.3 Optimizing an Existing Application to Run on Cirrus DSPs

This section describes a methodology for implementing an existing C programming language application on one of the Cirrus Logic DSPs. The implementation process starts with an application written in C that is submitted by or obtained from an application developer. The final goal is to compile the developer's optimized C code and turn it into an executable file running on the Cirrus Logic DSP.

This methodology comprises a series of incremental porting steps, Phase 1 through Phase 5:

- Phase 1: Developer Submits C-Code
- Phase 2: Make general code optimizations such as reducing the number of function

arguments

- Phase 3: Convert data types to data types supported by the Cirrus Logic DSP Architecture and optimize Developer's C-Code for 32-bit precision and fixed-point arithmetic functionality
- Phase 4: Apply Target and Assembly compiler-specific modifications to the code
- Phase 5: Create and test downloadable code by doing the following:
  - Integrate code into the Cirrus Logis OS framework environment
  - Further utilization and optimization activities
  - Verify implementation

9.3.1 Phase 1: Developer Submits Initial C Code Application

A developer creates an application in the C programming language. This code is considered the "golden" code and is used to validate Phase 2 to Phase 5 code.

9.3.2 Phase 2: Make General Code Optimizations

Make the optimizations described in Section 9.2.

9.3.3 Phase 3: Convert Data Types to Data Types Supported by the Cirrus Logic DSP Architecture

Applications implemented in C programming language often support several types of precision, and both floating-point, and fixed-point arithmetic. Cirrus Logic DSP architecture requires that all arithmetic be 32-bit precision and fixed-point. Section 9.3.3.1 describes how to make the needed conversions. After the data-type conversions are made in the C code, the resulting code can be validated against the golden output and serves as the code that is optimized in Phase 3.

9.3.3.1 Convert C-Code Floating-point to Fixed-point Arithmetic

In the original C code, fixed-point arithmetic is often emulated in some way, for example, by using special kind of functions and macros on integer or floating-point types. Compiling the emulation of arithmetics yields highly inefficient executable code, so Phase 1 code must be modified to use fixed-point types specified by the Embedded C Extensions.[1] The C Extensions fixed-point types are not supported by Microsoft Visual Studio. Code using the C Extensions can be difficult to run and debug. This issue can be solved by relying on C++ classes to emulate Embedded C specific fixed-point data types.

See Chapter 3, "Data Types" for a description of the fixed-point types that the CCC supports. Of the supported types, two types are most efficient:

- *fract* - a type which represents a 32-bit value (memory location width and data register), and
- *long accum* - representing a 72-bit value (accumulator)

9.3.4 Phase 4: Apply Target and Assembly Compiler-specific Modifications to the Code

Phase 3 code is the code that can be fully translated into the assembly file. Phase 3 code needs to be modified by applying compiler and target-specific code, in most cases directly in the assembler.

For example, CCC compiled code can differ from Microsoft Visual Studio compiled code in type conversion from integer to fixed-point or vice versa.

Consider this piece of code:

```
fract f = FRACT_NUM(0.34999999962747097015380859375);
int result;
result = f;
```

In Microsoft Visual Studio, the compiled result is 0x2CCCCCCC.

When compiled by the CCC, the result is 0.

1. ISO/IEC TR 18037:2004, "C -- Extensions to Support Embedded Processors" can be obtained at http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=30822

To achieve the same behavior in the CCC as in the case of Microsoft Visual Studio,

```
fract f = FRACT_NUM(0.34999999962747097015380859375);
int result;
#ifndef __CCC
// Microsoft Visual Studio branch
result = f;
#else
// CCC Branch branch
asm("%0 = %1" : "=a"(result) : "a"(f));
#endif
```

Note: To ensure maximal mathematical accuracy for a line of C code that encapsulates more than one type of mathematical operation and data type, read Section 3.6.1.

9.3.5 Phase 5: Create and Test Downloadable Code

Create downloadable code by doing the following:

- Integrate code into the Cirrus Logic framework environment, for example, by using the CLIDE buildable project template.
- Use DSP Composer to download the .uld and test.
- Use CLIDE to debug.

For more detailed information about creating downloadable code, refer to the *32-bit DSP Assembly Programmer's Guide*.

Chapter 10

Known Issues in CCC

In the 6.5 release of CCC, the following issues are known to exist.

10.1 Sequence Point Before Function Call

Consider the following code:

```
int a = 5;
int bar(int x)
{
if (5 == x && 6 == a)
    return 1;

return 0;
}
int main()
{
    return bar(a++);
}
```

The return value of function **main** should be 1. Side-effect should be executed after parameter preparation but **before** the actual function call.

CCC will calculate side-effects **after** the function call. Therefore, **main** will return a value of 0. The value in **a** will still be incremented, but **after** the function call.

10.2 Externalized Inline Functions

Functions that are declared with **inline** attribute, while also being externalized, might cause problems when whole program optimization (-wpo switch) is performed. The program may not compile.

10.3 Problems with More Flexible Array Initialization from C99

More flexible array initialization syntax example:

```
int x[5] = {[2] = 2, [4] = 3};
```

This results in array **x** being initialized with 0, 0, 2, 0, 3. More flexible array initialization that is introduced in C99 is not supported by CCC, and the appropriate error message will be reported if that kind of initialization is encountered. However, in some cases the error will not be reported, and the compilation will be completed, but it will have the incorrect initialization.

Problem cases are characterized by using flexible array initialization for structure members that are arrays.

10.4 Overflow when Using Abs Function for Fixed Point Types

According to language standards, abs functions for fixed point types (**absr**, **abslk**, **abshr**...) are supposed to saturate on overflow. Implementation of those functions in CCC will not saturate in case of overflow. Functions are implemented in a highly efficient manner, and the saturation on overflow will reduce that efficiency. This is actually not a bug, but a choice of implementation.

The only case of overflow is if input is the lowest negative value of a type. If it is important to have saturation on overflow, check if value of the input is the lowest negative value before the actual call of the abs function. A future release of CCC will add versions of abs functions that do perform saturation on overflow.

10.5 Saturated Fixed Point Type Unary Negation Does Not Saturate

For saturated fixed point type (declared with _Sat), unary negation should saturate. It will not saturate in CCC. Use "0 – x" construction if you want saturation to occur.

10.6 Multiplication of Unsigned Saturated Short Accum Type with Signed Integer Produces Wrong Code

Avoid such expressions.

Revision History

Revision	Date	Changes
UM1	October, 2008	Initial Release.
UM2	July, 2009	Add -Cl command option to Table 2-1 and an explanation of that option in Section 2.1.1.3. Updated Section 8.2.2.1.
UM3	February, 2010	Updated introductory material in Chapter 1. Updated Table 2-1, "Command Line Options," on page 2. Updated Section 3.3. Added Section and Section 3.5. Added new Chapter 6, "Function Attributes". The chapter numbers for the chapters that follow the new Chapter 7 were incremented.
UM4	September, 2010	Updated chapter numbering, re-added Chapter 7, "Generated Code."
UM5	November, 2010	Reworked Table of Contents, provided titles for examples.
UM6	January, 2011	Changed "Users Manual" to "User's Manual". Revised Table 3-7, "Bitwise Conversion from _Fract to int," on page 7.
UM7	April, 2011	Expanded Section 7.2, added Example 7-6. Expanded Section 3.3 and Example 3-2.
UM8	August, 2011	Added Section 3.3.1.
UM9	January, 2012	Moved "Rounding and Shifting" information to Section 3.6. Added Section 3.6.1. Added note to Section 9.3.4.
UM10	April, 2012	Updated signed and unsigned short fract and short accum values in Table 3-3. Updated examples in Chapter 4, "Memory Zones" to differentiate between C code and assembly code.
UM11	September, 2012	Added Section 1.2. Updated descriptions in Table 2-1. Updated text and examples in Section 2.1.1.2 and Section 2.1.1.3. Updated examples in Section 3.6. Added Chapter 10.
UM12	December, 2012	Updated Section 1.2 regarding bit field support.
UM13	March, 2013	Added five examples to Section 7.2. Added Section 7.3.3. Changed "usage pragma" to "clobber information pragma" throughout document.
UM14	June, 2013	Added -ira command line option to Table 2-1. Re-write of Chapter 3.
UM15	August, 2013	Updated command line format in Section 2.1.1. Added -emit-hints, -femit-asm-struct, and -fno-inline to Table 2-1. Added Section 2.1.1.3.4. Added example to Section 3.3.2. Added Section 7.2.1. Added Section Appendix A.

Appendix A

Support of Standard Libraries

Appendix A-1. Support of Standard Libraries

	Support	Note
Standard C		
assert.h	PARTIAL	On false assert, no info about the file and line are emitted. The program exits with a non-zero return value.
complex.h	NONE	
ctype.h	NONE	
errno.h	NONE	
fenv.h	NONE	
float.h	NONE	
inttypes.h	NONE	
iso656.h	FULL	
limits.h	FULL	The limit of long long type are currently the same as the limits for int. Refer to Section 1.2.
locale.h	NONE	
math.h	NONE	
setjmp.h	NONE	
signal.h	NONE	
stdarg.h	NONE	
stdbool.h	FULL	
stddef.h	FULL	
stdint.h	FULL	Exception for long long type and parts related to signal.h and wchar.h.
stdio.h	PARTIAL	Only the putchar function is supported. It works in a single core simulator.
stdlib.h	PARTIAL	Only abort, exit, abs, and labs function are supported. Itoa is also declared in stdlib.h, although it is not standard C function.
string.h	PARTIAL	Only memcpy, memset, strcpy, strcmp, and strlen functions are supported. There are also additional non-C standard versions of those functions for different memory zones.
tgmath.h	NONE	
time.h	NONE	

Ap

Appendix A-1. Support of Standard Libraries

	Support	Note
wchar.h	NONE	
wctype.h	NONE	
Embedded C Extension		
stdfix.h	PARTIAL	The following groups of functions are not supported (chapters numbers are from ISO/IEC TR 18037 document specifying Embedded C extension): • The fixed-point arithmetic operation support functions from chapter 7.18a.6.1 • The fixed-point rounding functions from chapter 7.18a.6.3 • The fixed-point countls functions from chapter 7.18a.6.4 • Type-generic fixed-point functions from chapter 7.18a.6.7
iohw.h	NONE	

The following is a list of all functions implemented as part of standard libraries.

- Functions in stdio.h

```
int putchar(int c);
```

- Functions in stdlib.h

```
void abort(void);
void exit(int status);
int abs(int j);
long int labs(long int j);
```

- Functions in stdlio.h

```
void* memcpy(void* dest, const void* src, size_t n);
__memX void* memcpy_x2x(__memX void* dest, const __memX void* src, size_t n);
__memY void* memcpy_x2y(__memY void* dest, const __memX void* src, size_t n);
__memY void* memcpy_y2y(__memY void* dest, const __memY void* src, size_t n);
__memX void* memcpy_y2x(__memX void* dest, const __memY void* src, size_t n);
__memXY void* memcpy_xy2xy(__memXY void* dest, const __memXY void* src,
size_t n);
char* strcpy(char* dest, const char* src);
__memX char* strcpy_x2x(__memX char* dest, const __memX char* src);
__memY char* strcpy_y2y(__memY char* dest, const __memY char* src);
__memY char* strcpy_x2y(__memY char* dest, const __memX char* src);
__memX char* strcpy_y2x(__memX char* dest, const __memY char* src);
int strcmp(const char* cs, const char* ct);
void* memset(void* d, int c, size_t n);
__memX void* memset_x(__memX void* d, int c, size_t n);
__memY void* memset_y(__memY void* d, int c, size_t n);
size_t strlen(const char* s);
size_t strlen_x(const __memX char* s);
size_t strlen_y(const __memY char* s);
```

- Functions in stdfix.h
  - Absolute value functions for fixed-point types:

```
short fract abshr(short fract f);
fract absr(fract f);
long fract abslr(long fract f);
short accum abshk(short accum f);
accum absk(accum f);
long accum abslk(long accum f);
```

  - Bitwise conversion from a fixed-point type to an integer type (definitions of return types are given in stdfix.h):

```
int_hr_t bitshr(short fract f);
int_r_t bitsr(fract f);
uint_uhr_t bitsuhr(unsigned short fract f);
uint_ur_t bitsur(unsigned fract f);
```

  - Bitwise conversion from an integer type to a fixed-point type (definitions of argument types are given in stdfix.h):

```
short fract hrbits(int_hr_t n);
fract rbits(int_r_t n);
unsigned short fract uhrbits(uint_uhr_t n);
unsigned fract urbits(uint_ur_t n);
```

  - Functions for setting and getting mr_sr register (refer to Section 3.3.2):

```
void set_mr_sr(int x);
void set_mr_r(int x);
void set_mr_s(int x);
int get_mr_sr();
int get_mr_r();
int get_mr_s();
```

  - Functions for getting parts of 72-bit wide long _Accum type variable as 32-bit wide _Fract type value:

```
_Fract rgetlo(long _Accum a);
_Fract rgethi(long _Accum a);
_Fract rgetg(long _Accum a);
```

  - Functions for getting parts of 72-bit wide long _Accum type variable as integer type value:

```
int bitsgetlo(long _Accum a);
int bitsgethi(long _Accum a);
int bitsgetg(long _Accum a);
```


Ap