C Language Compiler User Guide

Document # 38-12001 Rev. *E

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone (USA): 800.858.1810
Phone (Intnl): 408.943.2600
http://www.cypress.com

Contents

1. **Introduction** 9

1.1 Purpose 9
1.2 Section Overviews 10
1.3 Support 10
1.3.1 Technical Support 10
1.3.2 Product Upgrades 10
1.4 Documentation Conventions 11
1.4.1 Acronyms 11

2. **Accessing the Compiler** 13

2.1 Enabling the Compiler 13
2.2 Accessing the Compiler 13
2.3 Menu and Toolbar Options 14

3. **Compiler Files** 15

3.1 Startup File 15
3.2 Library Descriptions 15

4. **Compiler Basics** 17

4.1 Data Types 17
4.2 Operators 18
4.3 Expressions 19
4.4 Statements 20
4.5 Pointers 20
4.6 Re-Entrancy 20
4.7 Processing Directives 21
4.7.1 Preprocessor Directives 21
4.7.2 Pragma Directives 21

5. **Functions** 23

5.1 Library Functions 23
5.1.1 String Functions 23
5.1.2 Mathematical Functions 28
5.1.3 API Software Library Functions 29
5.2 Interfacing C and Assembly 30

6. **Additional Considerations** 31

6.1 Accessing M8C Features 31
6.2 Addressing Absolute Memory Locations 31
6.3 Assembly Interface and Calling Conventions 31

6.4 Bit Toggling....32
6.5 Inline Assembly....32
6.6 Interrupts....33
6.7 IO Registers....33
6.8 Long Jump/Call....33
6.9 Memory Areas....34
6.9.1 Flash Memory Areas....34
6.9.2 Data Memory Areas....34
6.10 Program and Data Memory Usage....34
6.10.1 Program Memory....34
6.10.2 Data Memory....34
6.11 Program Memory as Related to Constant Data....35
6.12 Stack Architecture and Frame Layout....36
6.13 Strings....36
6.14 Virtual Registers....36
6.15 Convention for Restoring Internal Registers....36

7. Linker 37

7.1 Linker Operations....37
7.2 Linking Process....37
7.2.1 Customized Linker Actions....37

8. Librarian 39

8.1 Librarian....39
8.1.1 Compiling a File into a Library Module....39
8.1.2 Listing the Contents of a Library....41
8.1.3 Adding or Replacing a Library Module....41
8.1.4 Deleting a Library Module....41

9. Command Line Overview 43

9.1 Compilation Process....43
9.2 Compiler Driver....43
9.3 Compiler Arguments....44
9.3.1 Compiler Argument Prefixes....44
9.3.2 Arguments Affecting the Driver....44
9.3.3 Preprocessor Arguments....44
9.3.4 Compiler Arguments....45
9.3.5 Linker Arguments....45

10. Code Compression 47

10.1 Theory of Operation....47
10.2 Code Compressor Process....47
10.2.1 C and Assembly Code....47
10.2.2 Program Execution Bytes....47
10.2.3 Impact to Debugger....48
10.3 Integration of the Code Compressor....48
10.3.1 boot.asm file....48
10.3.2 Text Area Requirement....48
10.4 Code Compressor and the AREA Directive....48
10.5 Build Messages....49
10.6 Considerations for Code Compression....50

Appendix A. Errors and Warnings Messages 51

A.1 Preprocessor 51
A.2 Preprocessor Command Line 52
A.3 C Compiler 53
A.4 Assembler 56
A.5 Assembler Command Line 57
A.6 Linker 57

List of Tables

Table 1-1. Overview of the C Language Compiler User Guide Sections 10
Table 1-2. Documentation Conventions 11
Table 1-3. Acronyms 11
Table 2-1. Compiler Menu and Toolbar Options 14
Table 4-1. Supported Data Types 17
Table 4-2. Supported Operators 18
Table 4-3. Preprocessor Directives 21
Table 4-4. Pragma Directives 21
Table 5-1. String Functions 24
Table 5-2. Mathematical Functions 28
Table 5-3. API Software Library Functions 29
Table 5-4. Pragma Fastcall16 Conventions for Argument Passing 30
Table 5-5. Pragma Fastcall16 Conventions for Return Value 30
Table 9-1. Compiler Argument Prefixes 44
Table 9-2. Arguments Affecting the Driver 44
Table 9-3. Preprocessor Arguments 44
Table 9-4. Compiler Arguments 45
Table 9-5. Linker Arguments 45
Table A-1. Preprocessor Errors and Warnings 51
Table A-2. Preprocessor Command Line Errors 52
Table A-3. C Compiler Errors and Warnings 53
Table A-4. Assembler Errors and Warnings 56
Table A-5. Assembler Command Line Errors 57
Table A-6. Linker Errors and Warnings 57

1. Introduction

1.1 Purpose

The PSoC Designer C Compiler compiles each .c source file to a PSoC device assembly file. The PSoC Designer Assembler then translates each assembly file (either those produced by the compiler or those that have been added) into a relocatable object file, .o. After all the files have been translated into object files, the linker combines them together to form an executable file. This .hex file is then downloaded to the emulator where it is debugged to perfect design functionality.

For comprehensive details on hardware, system use, and assembly language, refer to the following documents. Together, these documents comprise the PSoC Designer documentation suite.

- PSoC Designer PSoC Programmer User Guide
- PSoC Designer C Language Compiler User Guide
- PSoC Designer Assembly Language User Guide
- PSoC Designer ICE Connection and Troubleshooting Guide
- PSoC Designer USB Adapter Installation Guide
- PSoC Technical Reference Manual
- Device-specific PSoC Mixed-Signal Array Data Sheet

Additional recommended reading includes:

- *C Programming Language*, Second Edition, Brian W. Kernighan and Dennis Ritchie, Prentice Hall, March 1988.
- *C: A Reference Manual*, Fifth Edition, Samuel P. Harbison and Guy L. Steele, Prentice Hall, February 2002.

1.2 Section Overviews

Table 1-1. Overview of the C Language Compiler User Guide Sections

Section	Description
Introduction (on page 9)	Describes the purpose of this guide, presents an overview of each section, supplies support information, and describes the documentation conventions.
Accessing the Compiler (on page 13)	Describes enabling and accessing the compiler, and supplies menu and toolbar options.
Compiler Files (on page 15)	Discusses and lists startup and C library files within PSoC Designer.
Compiler Basics (on page 17)	Supplies C Compiler data types, operators, expressions, statements, pointers, re-entrancy, and processing directives.
Functions (on page 23)	Lists C Compiler library functions and describes how to interface between the C and assembly languages.
Additional Considerations (on page 31)	Lists additional compiler options to leverage the functionality of your code or program.
Linker (on page 37)	Discusses C Compiler linker options deployed within PSoC Designer.
Librarian (on page 39)	Discusses C Compiler library functions used within PSoC Designer.
Command Line Overview (on page 43)	Overviews supported compiler command line options for users who want to use the compiler outside PSoC Designer.
Code Compression (on page 47)	Details the theory of operation, the process and integration of code compression, and other features and guidelines of the code compressor.

1.3 Support

Free support for PSoC Designer and its C Compiler is available online at http://www.cypress.com/. Resources include Training Seminars, Discussion Forums, Application Notes, PSoC Consultants, TightLink Technical Support Email/Knowledge Base, and Application Support Technicians.

Before utilitzing the Cypress support services, know the version of PSoC Designer installed on your system. To quickly determine the version, build, or service pack of your current installation of PSoC Designer, click **Help > About PSoC Designer**.

1.3.1 Technical Support

Enter a support request in the TightLink Technical Support System with a guaranteed response time of four hours at http://www.cypress.com/support/login.cfm or www.cypress.com and click on Technical and Support KnowledgeBase at the bottom of the page. You can also view and participate in discussion threads about a wide variety of PSoC device topics on the Cypress support forums.

1.3.2 Product Upgrades

Cypress provides scheduled upgrades and version enhancements for PSoC Designer free of charge. Compiler upgrades are included in your PSoC Designer C Compiler license agreement. You can order PSoC Designer and Compiler upgrades from your distributor on CD-ROM or download them directly from http://www.cypress.com/. Also provided at the web sites are critical updates to system documentation. To stay current with system functionality you can find documentation updates under Design Resources.

1.4 Documentation Conventions

The following are easily identifiable conventions used throughout this user guide.

Table 1-2. Documentation Conventions

Convention	Usage
`Courier New Size 12`	Displays file locations and source code: `C:\ …cd\icc\`, user entered text.
Italics	Displays file names and reference documentation: *sourcefile.hex*
[**bracketed, bold**]	Displays keyboard commands in procedures: [**Enter**] or [**Ctrl**] [**C**]
File > New Project	Represents menu paths: File > New Project > Clone
Bold	Displays commands, menu paths and selections, and icon names in procedures: Click the **Debugger** icon, and then click **Next**.
Text in gray boxes	Displays cautions or functionality unique to PSoC Designer or the PSoC device.

1.4.1 Acronyms

The following are acronyms used throughout this user guide.

Table 1-3. Acronyms

Acronym	Description
ADC	analog-to-digital converter
API	application programming interface
C	(refers to C programming language)
DAC	digital-to-analog converter
DRC	design rule checker
EPP	enhanced parallel port
FPMP	Flash program memory protection
grep	global regular expression print
ICE	in-circuit emulator
IDE	integrated development environment
IO	input/output
ISR	interrupt service routine
MCU	microcontroller unit
MHz	megahertz
OHCI	open host controller interface
PWM	pulse width modulator
RAM	random access memory
ROM	read only memory
SSC	system supervisory call
UART	universal asynchronous receiver transmitter
UHCI	universal host controller interface
USB	universal serial bus

2. Accessing the Compiler

In this chapter you will learn how to enable and access the compiler, and its menu and toolbar options.

2.1 Enabling the Compiler

Enabling the compiler is done within PSoC Designer. To accomplish this, execute the following procedure.

1. Access **Tools > Options > Compiler** tab.
2. Enter your key code. You have this key code if you purchased the C Language Compiler License when you received PSoC Designer (by download, mail, or through a distributor).
3. At the License Agreement screen, scroll or use [**Page Down**] to view the terms of the license agreement. Click **Yes** to accept the agreement.

   To view the version details for the ImageCraft C Compiler, click **Version**. When finished, click the **OK** button

   To remove an expired license and enter a new key code, uncheck the I Accept box. You will be asked to confirm the removal. Click **Yes**, then enter the new code.

   If, for some reason, you have not received a key code or are uncertain of how to proceed, contact a Cypress Support Technician at license@cypressmicro.com.

2.2 Accessing the Compiler

All features of the compiler are available and accessible in the Application Editor subsystem of PSoC Designer by clicking the Application Editor icon. This icon can be found in the subsystem toolbar.

Features of the compiler include adding and modifying .c project files. These are described in thus user guide in brief and in the *PSoC Designer Integrated Development Environment User Guide* in detail.

2.3 Menu and Toolbar Options

Table 2-1 lists the menu and toolbar options that are available in PSoC Designer for writing and editing assembly language and C Compiler files.

Table 2-1. Compiler Menu and Toolbar Options

Icon	Option	Menu Path	Shortcut	Feature
	Compile/ Assemble	Build > Compile/ Assemble	[**Ctrl**] [**F7**]	Compiles/assembles the most prominent open, active file (.c or .asm)
	Build	Build > Build	[**F7**]	Builds entire project and links applicable files
	Execute Program			Switches into Debugging subsystem, connects, downloads file, runs... all from one click
	New File	File > New	[**Ctrl**] [**N**]	Adds a new file to the project
	Open File	File > Open	[**Ctrl**] [**O**]	Opens an existing file in the project
	Indent			Indents specified text
	Outdent			Outdents specified text
	Comment			Comments selected text
	Uncomment			Uncomments selected text
	Toggle Bookmark			Toggles the bookmark: Sets/removes user-defined bookmarks used to navigate source files
	Clear Bookmark			Clears all user-defined bookmarks
	Next Bookmark			Goes to next bookmark
	Previous Bookmark			Goes to previous bookmark
	Find Text	Edit > Find	[**Ctrl**] [**F**]	Find specified text
	Replace Text	Edit > Replace	[**Ctrl**] [**H**]	Replace specified text
	Find in Files	Edit > Find in Files		Find specified text in specified file(s)
	Repeat Find			Repeats last find
	Set Editor Options			Set options for editor
	Undo	Edit > Undo	[**Ctrl**] [**Z**]	Undo last action
	Redo	Edit > Redo	[**Ctrl**] [**Y**]	Redo last action

3. Compiler Files

In this chapter you will learn startup file procedures and how to reference supported library files.

3.1 Startup File

PSoC Designer creates a startup file called *boot.asm*. This file is generated from the *boot.tpl* file whenever a Generate Source occurs. It loads the initial device configuration and initializes global C variables. It also contains the interrupt table vector. At the end of *boot.asm*, there is a `ljmp` to main.asm. The underscore (`_main`) allows *boot.asm* to call the C or assembly main routine.

Many functions within PSoC Designer are built upon specifications in this file. Therefore, it is highly recommended that you do not modify the startup file. If you have a need, first consult your Cypress Technical Support Representative.

The *boot.asm* startup file also defines the reset vector. Normally, you do not need to modify the startup file to use other interrupts because PSoC Designer manages all interrupt vectors. If you need to add a `ljmp` to a custom interrupt handler, the *boot.tpl* file can be modified.

3.2 Library Descriptions

There are four primary code libraries used by PSoC Designer: *libcm8c.a* (SMM and LMM)*, libpsoc.a,* and *cms.a.*

libcm8c.a – This library resides in the PSoC Designer …`\tools` directory (…`\Program Files\Cypress\PSoC Designer\tools`). This library contains many functions typically used in C programming including SMM and LMM. SMM supports paging with the small memory model and LMM supports paging with the large memory model.

libpsoc.a – This library resides in the project `\lib` directory and contains user module functions. Device Editor automatically adds the source code for your user modules to the library during the generate application process. However, other library objects can be manually added to this library.

To add existing object files, copy your source file to the project …`\lib` directory, then add it to the project in PSoC Designer. For details on adding existing files to your project, see *PSoC Designer Integrated Development Environment User Guide*.

Avoid use of the following characters in path and file names (they are problematic):
\ / : * ? " < > | & + , ; = [ ] % $ ` '.

cms.a – This library resides in the …`\tools` directory. It contains convenient functions that do not involve user modules. For example, the functions to read and write flash reside here (Flash Block Programming). C prototypes for using these functions are given in the include file (*flashblock.h*) stored in the …`\tools \include` directory.

4. Compiler Basics

In this chapter you can reference PSoC Designer C Compiler data types, operators, expressions, statements, pointers, re-entrancy, and processing directives.

With one exception, the PSoC Designer C Compiler is a "conforming freestanding implementation" of the ANSI X3.159-1989 C Standard (C89), or the equivalent ISO/IEC 9899:1990 C Standard. The non-Standard exception is that floating-point doubles are only 32 bits. Doing 64-bits doubles would be prohibitive on an 8-bit microcontroller.

4.1 Data Types

Table 4-1 lists the supported PSoC Designer C Compiler standard data types. All types support the signed and unsigned type modifiers.

Table 4-1. Supported Data Types

Type	Bytes	Description	Range
char	1	A single byte of memory that defines characters	[a] unsigned 0…255 signed -128…127
int	2	Used to define integer numbers	unsigned 0…65535 [1] signed -32768…32767
short	2	Standard type specifying 2-byte integers	unsigned 0…65535 [1] signed -32768…32767
long	4	Standard type specifying the largest integer entity	unsigned 0…4294967295 [1] signed -2147483648…2147483647
float	4	Single precision floating point number in IEEE format	1.175e-38…3.40e+38
double	4	Single precision floating point number in IEEE format	1.175e-38…3.40e+38
enum	1 if enum < 256 2 if enum > 256	Used to define a list of aliases that represent integers.	0…65535

a. Default, if not explicitly specified as signed or unsigned.

The following type definitions are included in the *m8c.h* file:

typedef unsigned char BOOL;
typedef unsigned char BYTE;
typedef signed char CHAR;
typedef unsigned int WORD;
typedef signed int INT;
typedef unsigned long DWORD;
typedef signed long LONG;

The following floating-point operations are supported in the PSoC Designer C Compiler:

compare (= =)	add (+)
multiply (*)	subtract (-)
divide(/)	casting (long to float)

Floats and doubles are in IEEE 754 standard 32-bit format with 8-bit exponent and 23-bit mantissa with one sign bit.

4.2 Operators

Table 4-2 displays a list of the most common operators supported within the PSoC Designer C Compiler. Operators with a higher precedence are applied first. Operators of the same precedence are applied right to left. Use parentheses where appropriate to prevent ambiguity.

Table 4-2. Supported Operators

Pre.	Op.	Function	Group	Form	Description
1	++	Postincrement		a ++	
1	--	Postdecrement		a --	
1	[ ]	Subscript		a[b]	
1	( )	Function Call		a(b)	
1	.	Select Member		a.b	
1	->	Point at Member		a->b	
2	sizeof	Sizeof		sizeof a	
2	++	Preincrement		++ a	
2	--	Predecrement		-- a	
2	&	Address of		&a	
2	*	Indirection		*a	
2	+	Plus		+a	
2	-	Minus		-a	
2	~	Bitwise NOT	Unary	~ a	1's complement of a
2	!	Logical NOT		!a	
2	(declaration)	Type Cast			(declaration)a
3	*	Multiplication	Binary	a * b	a multiplied by b
3	/	Division	Binary	a / b	a divided by b
3	%	Modulus	Binary	a % b	Remainder of a divided by b

Table 4-2. Supported Operators *(continued)*

Pre.	Op.	Function	Group	Form	Description
4	+	Addition	Binary	a + b	a plus b
4	-	Subtraction	Binary	a - b	a minus b
5	<<	Left Shift	Binary	a << b	Value of a shifted b bits left
5	>>	Right Shift	Binary	a >> b	Value of a shifted b bits right
6	<	Less		a < b	a less than b
6	<=	Less or Equal		a <= b	a less than or equal to b
6	>	Greater		a > b	a greater than b
6	>=	Greater or Equal		a >= b	a greater than or equal to b
7	==	Equals		a == b	
7	!=	Not Equals		a != b	
8	&	Bitwise AND	Bitwise	a & b	Bitwise AND of a and b
9	^	Bitwise Exclusive OR	Bitwise	a ^ b	Bitwise Exclusive OR of a and b
10	|	Bitwise Inclusive OR	Bitwise	a | b	Bitwise OR of a and b
11	&&	Logical AND		a && b	
12	||	Logical OR		a || b	
13	? :	Conditional		c?a:b	
14	=	Assignment		a = b	
14	*=	Multiply Assign		a *= b	
14	/=	Divide Assign		a /= b	
14	%=	Remainder Assign		a %= b	
14	+=	Add Assign		a += b	
14	-=	Subtract Assign		a -= b	
14	<<=	Left Shift Assign		a <<= b	
14	>>=	Right Shift Assign		a >>= b	
14	&=	Bitwise AND Assign		a &= b	
14	^=	Bitwise Exclusive OR Assign		a ^= b	
14	|=	Bitwise Inclusive OR Assign		a |= b	
15	,	Comma		a , b	

4.3 Expressions

PSoC Designer supports standard C language expressions.

4.4 Statements

PSoC Designer C Compiler supports the following standard statements:

- **if else** – Decides on an action based on **if** being true.
- **switch** – Compares a single variable to several possible constants. If the variable matches one of the constants, a jump is made.
- **while** – Repeats (iterative loop) a statement until the expression proves false.
- **do** – Same as **while**, except the test runs after execution of a statement, not before.
- **for** – Executes a controlled loop.
- **goto** – Transfers execution to a label.
- **continue** – Used in a loop to skip the rest of the statement.
- **break** – Used with a **switch** or in a loop to terminate the **switch** or loop.
- **return**– Terminates the current function.
- **struct** – Used to group common variables together.
- **typedef** – Declares a type.

4.5 Pointers

A pointer is a variable that contains an address that points to data. It can point to any data type (i.e., int, float, char, etc.). A generic (or unknown) pointer type is declared as void and can be freely cast between other pointer types. Function pointers are also supported. Note that pointers require two bytes of memory storage to account for the size of both the data and program memory.

Due to the nature of the Harvard architecture of the M8C microprocessor, a data pointer may point to data located in either data or program memory. To discern which data is to be accessed, the `const` qualifier is used to signify that a data item is located in program memory. See Program Memory as Related to Constant Data on page 35.

4.6 Re-Entrancy

Currently, there are no pure re-entrant library functions. However, it is possible to create a re-entrant condition that will compile and build successfully. Due to the constraints that a small stack presents, re-entrant code is not recommended.

4.7 Processing Directives

PSoC Designer C Compiler supports the following preprocessors and pragmas directives:

4.7.1 Preprocessor Directives

Table 4-3. Preprocessor Directives

Preprocessor	Description
#define	Define a preprocessor constant or macro.
#else	Executed if #if, #ifdef, or #ifndef fails.
#endif	Close #if, #ifdef, or #ifndef.
#if (include or exclude code)	Based on an expression.
#ifdef (include or exclude code)	A preprocessor constant has been defined.
#ifndef (include or exclude code)	A preprocessor constant has not been defined.
#include	Include a source file. < > are used to specify the PSoC Designer Include folder. “ “ are used to specify the Project folder.
#line	Specify the number of the next source line.
#undef	Remove a preprocessor constant.

4.7.2 Pragma Directives

Table 4-4. Pragma Directives

#pragma	Description
#pragma ioport LED:0x04; char LED;	Defines a variable that occupies a region in IO space (register). This variable can then be used in IO reads and writes. The #pragma ioport must precede a variable declaration defining the variable type used in the pragma.
#pragma fastcall GetChar	Fastcall has been replaced by fastcall16 (see below).
#pragma fastcall16 GetChar	Provides an optimized mechanism for argument passing. This pragma is used only for assembly functions called from C.
#pragma abs_address:<address>	Allows you to locate C code data at a specific address such as #pragma abs_address:0x500. The #pragma end_abs_address (described below) should be used to terminate the block of code data. Note that data includes both ROM and RAM.
#pragma end_abs_address	Terminates the previous #pragma abs_address: <address> pragma.
#pragma text:<name>	Change the name of the text area. Make modifications to *Custom.LKP* in the project directory to place the new area in the code space.

Table 4-4. Pragma Directives *(continued)*

#pragma	Description
#pragma interrupt_handler <func1> [,<func2>]*	For interrupt handlers written in C. Virtual registers are saved only if they are used, unless the handler calls another function. In that case, all Virtual registers are saved. This interrupt handler changes the `ret` to `reti` at the end of the function. The function can be used as an interrupt handler by adding a `ljmp _name` at the interrupt vector in *boot.tpl*. It cannot be used in regular C code because the `reti` expects the flags to be pushed on the stack. In the large memory model, the Page Pointer registers (CUR_PP, IDX_PP, MVW_PP, and MVR_PP) are saved and restored in addition to the Virtual registers for a #pragma interrupt_handler.
#pragma nomac #pragma usemac	These two pragmas override the command line nomac argument, or **Project > Settings > Compiler** tab, Enable MAC option. Refer to the compiler project settings in the *PSoC Designer Integrated Development Environment User Guide*. The pragmas should be specified outside of a function definition. Note that if compiler MAC is enabled (**Project > Settings > Compiler** tab, Enable MAC is checked by default), the compiler will use the MAC in ISRs, intermittently corrupting the foreground computations that use the MAC. It is the programmer's responsibility to use pragma nomac at the beginning of each ISR function written in C.

5. Functions

In this chapter you can reference compiler library functions supported within PSoC Designer and learn how to interface between the C and assembly languages.

PSoC Designer C Compiler functions use arguments and always return a value. All C programs must have a function called `main()`. Each function must be self-contained in that you may not define a function within another function or extend the definition or a function across more than one file.

It is important to note that the compiler generates inline code whenever possible. However, for some C constructs, the compiler generates calls to low level routines. These routines are prefixed with two underscores and should not be called directly by the user.

5.1 Library Functions

Use `#include <associated-header.h>` for each function described in this section.

5.1.1 String Functions

All strings are null terminated strings. The prototypes for the all the string functions can be found in the two include files *string.h* and *stdlib.h* located in …`\PSoC Designer\tools\include.`

You can view the list of all library functions, including all the string functions, at a command prompt window with working directory …`\PSoC Designer\tools` by issuing the command `“ilibw –t lib\SMM\libcm8c.a”.`

In the include file *const.h* located in `...\PSoC Designer\tools\include`, CONST is defined to be the empty string. Therefore, it has no effect on the declarations in which it appears, unlike lower-case `const` that specifies the data is allocated in Flash rather than RAM. When a function prototype uses CONST to describe an argument, it means that the function will not modify the argument. This is a promise by the programmer that implemented the function, not something that is enforced by the C Compiler.

Some of the normal prototypes in *string.h* have an additional version prefixed with ‘c’, e.g. cstrlen. This prefix indicates that one of the parameters is located in Flash, as designated by the `const` qualifier.

The following C programming language web sites were used in preparation of the material presented in this section.

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclib/html/_vclibraries_home.asp

http://www.gnu.org/software/libc/libc.html

Table 5-1. String Functions

Function	Prototype and Description	Header
abs	int abs(int); Returns the absolute value of number.	stdlib.h
atof	double atof(CONST char *); Converts a string to double. Returns the double value produced by interpreting the input characters as a number. The return value is 0.0 if the input cannot be converted to a value of that type. The return value is undefined in case of overflow.	stdlib.h
atoi	int atoi(CONST char *); Converts a string to integer. Returns the int value produced by interpreting the input characters as a number. The return value is 0 if the input cannot be converted to a value of that type. The return value is undefined in case of overflow.	stdlib.h
atol	long atol(CONST char *); Converts a string to long integer. Returns the long value produced by interpreting the input characters as a number. The return value is 0L if the input cannot be converted to a value of that type. The return value is undefined in case of overflow.	stdlib.h
char *itoa	char *itoa (char *string, int value, int base); Converts an integer to a string. This function converts the digits of the given value argument to a null-terminated character string. The base must be in the range 2 - 36. If the base equals 10 and the given value is negative, the string is preceded by a '-'. Returns a pointer to the string.	stdlib.h
char *ltoa	char *ltoa (char *string, long value, int base); Converts a long integer to a string. This function converts the digits of the given long value argument to a null-terminated character string. The base must be in the range 2 - 36. If the base equals 10 and the given value is negative, the string is preceded by a '-'. Returns a pointer to the string.	stdlib.h
char *utoa	char *utoa(char *string, unsigned int value, int base); Converts an unsigned integer to a string. This function converts the digits of the given value argument to a null-terminated character string. The base must be in the range 2 - 36. Returns a pointer to the string.	stdlib.h
char *ultoa	char *ultoa(char *string, unsigned long value, int base); Converts an unsigned long integer to a string. This function converts the digits of the given value argument to a null-terminated character string. The base must be in the range 2 - 36. Returns a pointer to the string.	stdlib.h
ftoa	char *ftoa(float f, int *status); /* ftoa function */ #define _FTOA_TOO_LARGE -2 /* |input| > 2147483520 */ #define _FTOA_TOO_SMALL -1 /* |input| < 0.0000001 */ /* ftoa returns static buffer of ~15 chars. If the input is out of * range, *status is set to either of the above #define, and 0 is * returned. Otherwise, *status is set to 0 and the char buffer is * returned. * This version of the ftoa is fast but cannot handle values outside * of the range listed. Please contact us if you need a (much) larger * version that handles greater ranges. * Note that the prototype differs from the earlier version of this * function. */	stdlib.h

Table 5-1. String Functions *(continued)*

Function	Prototype and Description	Header
rand	int rand(void); Generates a pseudorandom number. The rand function returns a pseudorandom integer in the range 0 to RAND_MAX. Returns a pseudorandom number.	stdlib.h
srand	void srand(unsigned); Sets a random starting point. The srand function sets the starting point for generating a series of pseudorandom integers. To reinitialize the generator, use 1 as the seed argument. Any other value for seed sets the generator to a random starting point. rand retrieves the pseudorandom numbers that are generated. Calling rand before any call to srand generates the same sequence as calling srand with seed passed as 1.	stdlib.h
strtol	long strtol(CONST char *, char **, int); Converts strings to a long-integer value. The strtol function converts string1 to a long. strtol stops reading the string string1 at the first character it cannot recognize as part of a number. This may be the terminating null character, or it may be the first numeric character greater than or equal to base. String 2 is the pointer to the character that stops scan.	stdlib.h
strtoul	unsigned long strtoul(CONST char *, char **, int); Convert strings to an unsigned long-integer value. The strtoul function converts string1 to an unsigned long. strtol stops reading the string string1 at the first character it cannot recognize as part of a number. This may be the terminating null character, or it may be the first numeric character greater than or equal to base. String 2 is the pointer to the character that stops scan.	stdlib.h
cstrcat	char *cstrcat(char *dest, const char *src); The function appends a copy of the string pointed to by src (including the terminating null character) to the end of the string pointed to by dest. The initial character of src overwrites the null character at the end of dest. The function returns the value of dest.	string.h
cstrcmp	int cstrcmp(const char *s1, char *s2); The function compares the string pointed to by s1 to the string pointed to by s2. The function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2.	string.h
cstrcpy	char *cstrcpy(char *dest, const char *src); The function copies the string pointed to by src (including the terminating null character) into the array pointed to by dest. The function returns the value of dest.	string.h
cstrncpy	char *cstrncpy(char *dest, const char *src, size_t n); The function copies not more than n characters (characters that follow a null character are not copied) from the string pointed to by src to the string pointed to by dest. The function returns the value of dest.	string.h
cstrlen	size_t cstrlen(const char *s); The function returns the number of characters in s preceding the terminating null character.	string.h
memchr	void *memchr(CONST void *ptr, int c, size_t n); The function locates the first occurrence of c (converted to an unsigned char) in the initial n characters (each interpreted as unsigned char) of the object pointed to by ptr. The function returns a pointer to the located character, or a null pointer if the character does not occur in the object.	string.h

Table 5-1. String Functions *(continued)*

Function	Prototype and Description	Header
memcmp	int memcmp(CONST void *ptr1, CONST void *ptr2, size_t n); The function compares the first n characters of the object pointed to by ptr1 to the first n characters of the object pointed to by ptr2. The function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by ptr1 is greater than, equal to, or less than the object pointed to by ptr2.	string.h
memcpy	void *memcpy(void *dest, CONST void *src, size_t n); The function copies n characters from the object pointed to by src into the object pointed to by dest. If copying takes place between objects that overlap, the behavior is undefined. The function returns the value of dest.	string.h
memmove	void *memmove(void *dest, CONST void *src, size_t n); The function copies n characters from the object pointed to by src into the object pointed to by dest. The function works correctly for overlapping objects. The function returns the value of dest.	string.h
memset	void *memset(void *ptr, int c, size_t n); The function copies the value of c (converted to an unsigned char) into each of the first n characters of the object pointed to by ptr. The function returns the value of ptr.	string.h
strcat	char *strcat(char *dest, CONST char *src); The function appends a copy of the string pointed to by src (including the terminating null character) to the end of the string pointed to by dest. The initial character of src overwrites the null character at the end of dest. If copying takes place between objects that overlap, the behavior is undefined. The function returns the value of dest.	string.h
strchr	char *strchr(CONST char *s, int c); The function locates the first occurrence of c (converted to a char) in the string pointed to by s. The terminating null character is considered to be part of the string. The function returns a pointer to the located character, or a null pointer if the character does not occur in the string.	string.h
strcmp	int strcmp(CONST char *s1, CONST char *s2); The function compares the string pointed to by s1 to the string pointed to by s2. The function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2.	string.h
strcoll	int strcoll(CONST char *s1, CONST char *s2); The function compares the string pointed to by s1 to the string pointed to by s2 using the collating convention of the current locale. The function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2.	
strcpy	char *strcpy(char *dest, CONST char *src); The function copies the string pointed to by src (including the terminating null character) into the array pointed to by dest. The function returns the value of dest.	string.h
strcspn	size_t strcspn(CONST char *s1, CONST char *s2); The function computes the length of the maximum initial segment of the string pointed to by s1 which consists entirely of characters *not* from the string pointed to by s2. The function returns the length of the segment.	string.h

Table 5-1. String Functions *(continued)*

Function	Prototype and Description	Header
strlen	size_t strlen(CONST char *s); The function returns the number of characters in s preceding the terminating null character.	string.h
strncat	char *strncat(char *dest, CONST char *src, size_t n); The function appends not more than n characters (a null character and characters that follow it are not appended) from the string pointed to by src to the end of the string pointed to by dest. The initial character of src overwrites the null character at the end of dest. A terminating null character is always appended to the result. The function returns the value of dest.	string.h
strncmp	int strncmp(CONST char *s1, CONST char *s2, size_t n); The function compares not more than n characters (characters that follow a null character are not compared) from the string pointed to by s1 to the string pointed to by s2. The function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2.	string.h
strncpy	char *strncpy(char *dest, CONST char *src, size_t n); The function copies not more than n characters (characters that follow a null character are not copied) from the string pointed to by src to the string pointed to by dest. The function returns the value of dest.	string.h
strpbrk	char *strpbrk(CONST char *s1, CONST char *s2); The function locates the first occurrence in the string pointed to by s1 of any character from the string pointed to by s2. The function returns a pointer to the character, or a null pointer if no character from s2 occurs in s1.	string.h
strrchr	char *strrchr(CONST char *s, int c); The function locates the last occurrence of c (converted to a char) in the string pointed to by s. The terminating null character is considered to be part of the string. The function returns a pointer to the located character, or a null pointer if the character does not occur in the string.	string.h
strspn	size_t strspn(CONST char *s1, CONST char *s2); The function computes the length of the maximum initial segment of the string pointed to by s1 which consists entirely of characters from the string pointed to by s2. The function returns the length of the segment.	string.h
strstr	char *strstr(CONST char *s1, CONST char *s2); The function locates the first occurrence in the string pointed to by s1 of the sequence of characters (excluding the terminating null character) in the string pointed to by s2. The function returns a pointer to the located string, or a null pointer if the string is not found. If s2 points to a string with zero length, the function returns s1.	string.h

5.1.2 Mathematical Functions

Prototypes for the mathematical functions can be found in the include file *math.h* located in `... \PSoC Designer\tools\include`.

Table 5-2. Mathematical Functions

Function	Description
float fabs(float x);	Calculates the absolute value (magnitude) of the argument x, by direct manipulation of the bit representation of x. Return the absolute value of the floating point number x.
float frexp(float x, int *eptr);	All non zero, normal numbers can be described as m * 2**p. frexp represents the double val as a mantissa m and a power of two p. The resulting mantissa will always be greater than or equal to 0.5, and less than 1.0 (as long as val is nonzero). The power of two will be stored in *exp. Return the mantissa and exponent of x as the pair (m, e). m is a float and e is an integer such that x == m * 2**e. If x is zero, returns (0.0, 0), otherwise 0.5 <= abs(m) < 1.
float tanh(float x);	Returns the hyperbolic tangent of x.
float sin(float x);	Returns the sine of x.
float atan(float x);	Returns the angle whose tangent is x, in the range [-pi/2, +pi/2] radians.
float atan2(float y, float x);	Returns the angle whose tangent is y/x, in the full angular range [-pi, +pi] radians.
float asin(float x);	Returns the angle whose sine is x, in the range [-pi/2, +pi/2] radians.
float exp10(float x);	Returns 10 raised to the specified real number.
float log10(float x);	log10 returns the base 10 logarithm of x. It is implemented as log(x) / log(10).
float fmod(float y, float z);	Computes the floating-point remainder of x/y (x modulo y). The fmod function returns the value for the largest integer i such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y.
float sqrt(float x);	Returns the square root of x, x^(1/2).
float cos(float x);	Returns the cosine of x for x in radians. If x is large the value returned might not be meaningful, but the function reports no error.
float ldexp(float d, int n);	Calculates the value that it takes and returns float rather than double values. ldexp returns the calculated value.
float modf(float y, float *i);	Splits the double val apart into an integer part and a fractional part, returning the fractional part and storing the integer. The fractional part is returned. Each result has the same sign as the supplied argument val.
float floor(float y);	Finds the nearest integer less than or equal to x. floor returns the integer result as a double.
float ceil(float y);	Finds the nearest integer greater than or equal to x. ceil returns the integer result as a double.
float fround(float d);	Produces a quotient that has been rounded to the nearest mathematical integer; if the mathematical quotient is exactly halfway between two integers, (that is, it has the form integer+1/2), then the quotient has been rounded to the even (divisible by two) integer.
float tan(float x);	Returns the tangent of x for x in radians. If x is large the value returned might not be meaningful, but the function reports no error.
float acos(float x);	Computes the inverse cosine (arc cosine) of the input value. Arguments to acos must be in the range -1 to 1. The function returns the angle whose cosine is x, in the range [0, pi] radians.

Table 5-2. Mathematical Functions *(continued)*

Function	Description
float exp(float x);	Calculates the exponential of x, that is, the base of the natural system of logarithms, approximately 2.71828). The function returns the exponential of x, e^x.
float log(float x);	Returns the natural logarithm of x, that is, its logarithm base e (where e is the base of the natural system of logarithms, 2.71828...). The function returns the natural logarithm of x.
float pow(float x,float y);	Calculates x raised to the exp1.0nt y. On success, pow returns the value calculated.
float sinh(float x);	Computes the hyperbolic sine of the argument x. The function returns the hyperbolic sine of x.
float cosh(float x);	Computes the hyperbolic cosine of the argument x. The function returns the hyperbolic cosine of x.

5.1.3 API Software Library Functions

The header and include files can be found at: …`\PSoC Designer\tools\include`.

Table 5-3. API Software Library Functions

Function	Prototype	Description	Header
bFlashWriteBlock	BYTE bFlashWriteBlock (FLASH_WRITE_STRUCT *) See *flashblock* header file for definition of structure.	Writes data to the Flash Program Memory.	*flashblock.h, flash-block.inc* (for assembly language)
FlashReadBlock	void FlashReadBlock (FLASH_READ_STRUCT *) See *flashblock* header file for definition of structure.	Reads data from the Flash Program Memory into RAM.	*flashblock.h, flash-block.inc* (for assembly language)

5.2 Interfacing C and Assembly

The C Compiler fastcall convention was devised to create an efficient function parameter and return value passing mechanism between C and assembly language functions. It cannot be used between two C functions. By using the #pragma fastcall16, certain types will be passed in registers. See Table 5-4 below for details about when it will happen.

The #pragma fastcall16 has replaced #pragma fastcall and use of #pragma fastcall is highly discouraged. Fastcall16 works equally well as its legacy, Fastcall, for all memory models and eliminates dependencies on implementation details of the C Compiler. One key difference between the disciplines is that 16 bits are reserved for all pointers in the Fastcall16 specification.

In the tables that follow, the reference of returned structures reside in the A and X registers. If passed by value, a structure is always passed through the stack, and not in registers. Passing a structure by reference (i.e., passing the address of a structure) is the same as passing the address of any data item, that is, a pointer (which is 2 bytes).

Table 5-4 reflects the set of #pragma fastcall16 conventions used for argument passing register assignments. Arguments that are pushed on the stack are pushed from right to left.

Table 5-4. Pragma Fastcall16 Conventions for Argument Passing

Argument Type	Register	Argument Register
Single Byte	A	The argument is passed in A.
Two Single Bytes	A, X	The first argument is passed in A, the second in X.
Double Byte	X, A	The MSB is passed in X, the LSB in A.
Pointer	A, X	The MSB is passed in A, the LSB in X.
All Others	None	Arguments are stored on the stack in standard byte order and in reverse order or appearance. In other words, the MSB of the last actual parameter is pushed first and the LSB of the first actual parameter is pushed last.

Table 5-5 reflects the set of #pragma fastcall16 conventions used for return value register assignments.

Table 5-5. Pragma Fastcall16 Conventions for Return Value

Return Type	Return Register	Comment
Single Byte	A	The argument is returned in A.
Double Byte	X, A	The MSB is passed in X, the LSB in A.
Pointer	A, X	The MSB is passed in A, the LSB in X.
All Others	None	Use a pass-by-reference parameter or global variable instead of returning arguments longer than 16 bits.

6. Additional Considerations

In this chapter you will learn additional compiler options to leverage the functionality of your code or program.

6.1 Accessing M8C Features

The strength of the compiler is that while it is a high-level language, it allows you to access low-level features of the target device. Even in cases where the target features are not available in the compiler, usually inline assembly and preprocessor macros can be used to access these features transparently (refer to Inline Assembly on page 32).

6.2 Addressing Absolute Memory Locations

There are two options for addressing absolute memory locations:

1. Use the #pragma abs_address. For example, to address an array in Flash memory:

   ```
   #pragma abs_address: 0x2000
   const char abMyStringData [100]={0};
   #pragma end_abs_address
   ```

2. Optionally, an absolute memory address in data memory can be declared using the #define directive:

   ```
   #define MyData (*(char*) 0x200)
   ```

   where `MyData` references memory location 0x200.

6.3 Assembly Interface and Calling Conventions

Standard to the PSoC Designer C Compiler and Assembler is an underscore which is implicitly added to C function and variable names. This should be applied when declaring and referencing functions and variables between C and assembly source. For example, the C function defined with a prototype such as "`void foo();`" would be referenced as `_foo` in assembly. However In C, the function would still be referenced as `foo()`. The underscore is also applied to variable names. Refer to Interfacing C and Assembly on page 30 for #pragma fastcall routines.

6.4 Bit Toggling

A common task in programming a microcontroller is to turn bits on and off in registers. Fortunately, standard C is well suited to bit toggling without resorting to assembly instructions or other non-standard C constructs. PSoC Designer supports the following bitwise operators:

a | b bitwise or	The expression is denoted by "a" is bitwise or'ed with the expression denoted by "b." This is used to turn on certain bits, especially when used in the assignment form |=. For example: `PORTA |= 0x80; // turn on bit 7 (msb)`
a & b bitwise and	This operator is useful for checking if certain bits are set. For example: `if ((PORTA & 0x81) == 0)// check bit 7 and bit 0` Note that the parentheses are needed around the expression of an & operator because it has lower precedence than the == operator. This is a source of many programming bugs in C programs. See Compiler Basics on page 17 for the table of supported operators and precedence.
a ^ b bitwise exclusive or	This operator is useful for complementing a bit. For example, in the following case, bit 7 is flipped: `PORTA ^= 0x80;// flip bit 7`
~a bitwise complement	This operator performs a ones-complement on the expression. It is especially useful when combined with the bitwise and operator to turn off certain bits. For example: `PORTA &= ~0x80;// turn off bit 7`

6.5 Inline Assembly

Besides writing assembly functions in assembly files, inline assembly allows you to write assembly code within your C file. (Of course, you may use assembly source files as part of your project as well.) The syntax for inline assembly is:

```
asm ("<string>");
```

For example:

```
asm ("mov A,5");
```

Multiple assembly statements can be separated by the `newline` character `\n`. String concatenations can be used to specify multiple statements without using additional assembly keywords. For example:

```
asm(".LITERAL \n"
"S:: db 40h \n"
".ENDLITERAL \n");
```

C variables have an implicit underscore at the beginning that needs to be used when using C variables from assembly. C variables can be referenced within the assembly string. For example:

```
asm (“mov A,_cCounter”);
```

Inline assembly may be used inside or outside a C function. The compiler indents each line of the inline assembly for readability. The assembler allows labels to be placed anywhere (not just at the first character of the lines in your file) so you may create assembly labels in your inline assembly code. If you are referencing registers inline, be sure to include reference to the *m8c.h* file. You may get a warning on assembly statements that are outside of a function. If so, you may ignore these warnings.

The compiler does not account for inline assembly in its generated code. Inline assembly may modify the behavior of generated C Compiler code.

6.6 Interrupts

Interrupt handlers can be written in C. In order to employ them, you must first inform the compiler that the function is an interrupt handler. To do this, use the following pragma (in the file where you define the function, before the function definition):

```
#pragma interrupt_handler <name> *
```

For an interrupt function, the compiler generates the `reti` instruction instead of the `ret` instruction, then saves and restores all registers used in the function. For example:

```
#pragma interrupt_handler       timer_handler
...
void timer_handler()
          {
          ...
          }
```

You may place multiple names in a single interrupt_handler pragma, separated by spaces. For example:

```
#pragma interrupt_handler timer_ovf sci_ovf
```

To associate the interrupt handler with an interrupt, add `ljmp _name` at the interrupt vector in the *boot.tpl* file.

Virtual registers are saved only if they are used by the routine. If your interrupt handler calls another function, then the compiler saves and restores all virtual registers, since it does not know which virtual register the called function uses. In the large memory model, the Page Pointer registers (CUR_PP, IDX_PP, MVW_PP, and MVR_PP) are saved and restored in addition to Virtual registers.

If the compiler MAC is enabled (**Project > Settings > Compiler** tab, Enable MAC is checked by default), the compiler will use the MAC in ISRs, intermittently corrupting the foreground computations that use the MAC. It is the programmer’s responsibility to use #pragma nomac at the beginning of each ISR function written in C.

6.7 IO Registers

IO registers are specified using the following #pragma:

```
#pragma ioport LED:0x04;                // ioport is at IO space 0x04
char LED;....                           LED must be declared in global scope
LED = 1;
```

6.8 Long Jump/Call

The assembler and linker will turn a `JMP` or `CALL` instruction into the long form `LJMP` and `LCALL` if needed. This applies if the target is in a different linker area or if it is defined in another file.

6.9 Memory Areas

The compiler generates code and data into different areas. (See the complete list of Assembler Directives in the *PSoC Designer Assembly Language User Guide*). The areas used by the compiler, ordered here by increasing memory address, are Flash memory areas and data memory areas.

6.9.1 Flash Memory Areas

- **top** – Contains the interrupt vectors and *boot.asm* code.
- **func_lit** – Contains the address of a function entry for each word (function table area).
- **lit** – Contains integer and floating-point constants.
- **idata** – Stores the initial values for the global data.
- **text** – Contains program code.
- **psoc_config** – Contains configuration load and unload routines.
- **usermodules** – Contains user module API routines.

6.9.2 Data Memory Areas

- **data** – Contains the data area housing global and static variables, and strings. The initial values of the global variables are stored in the "idata" area and copied to the data area at startup time.
- **bss** – Contains the data area housing uninitialized C global variables. Per ANSI C definition, these variables will get initialized to zero at startup time.
- **virtual registers** – Contains temporary variables used by the C Compiler.
- **internal RAM** – Contains page of RAM used by interrupts.

The linker will collect areas of the same types from all the input object files and combine them in the output file. For further information, see Linker on page 37.

6.10 Program and Data Memory Usage

6.10.1 Program Memory

The program memory, which is non volatile, is used for storing program code, constant tables, initial values, and strings for global variables. The compiler generates a memory image in the form of an output file of hexadecimal values in ASCII text (a .rom file).

6.10.2 Data Memory

The data memory is used for storing variables and the stack frames. In general, they do not appear in the output file but are used when the program is running. A program uses data memory as follows:

```
[high memory]
        [stack frames]
        [global variables]
        [initialized globals]
        [virtual registers]
[low memory]
```

It is up to the programmer to ensure that the stack does not exceed the high memory limit of 0xFF (0x7FF in the large memory model), otherwise unexpected results can occur (such as the stack wrapping around the lowest address).

6.11 Program Memory as Related to Constant Data

The M8C microprocessor is a Harvard architecture machine, separating program memory from data memory. There are several advantages to such a design. For example, the separate address space allows the device to access more total memory than a conventional architecture.

Due to the nature of the Harvard architecture of the M8C, a data pointer may point to data located in either data or program memory. To discern which data is to be accessed, the `const` qualifier is used to signify that a data item is located in program memory. Note that for a pointer declaration, the `const` qualifier may appear in different places, depending on whether it is qualifying the pointer variable itself or the items that it points to. For example:

```
const int table[] = { 1, 2, 3 };
const char *ptr1;
char * const ptr2;
const char * const ptr3;
```

In the example above, `table` is a table allocated in the program memory. `ptr1` is an item in the data memory that points to data in the program memory. `ptr2` is an item in the program memory that points to data in the data memory. Finally, `ptr3` is an item in the program memory that points to data in the program memory. In most cases, items such as `table` and `ptr1` are probably the most typical. The compiler generates the INDEX instruction to access the program memory for read-only data.

Note that the C Compiler does not require `const` data to be put in the read-only memory, and in a conventional architecture, this would not matter except for access rights. Therefore, the use of the `const` qualifier is unconventional, but within the allowable parameters of the compiler. However, this does introduce conflicts with some of the standard C function definitions.

For example, the standard prototype for `cstrcpy` is `cstrcpy(char *, const char *cs);` with the `const` qualifier of the second argument signifying that the function does not modify the argument. However, under the M8C, the `const` qualifier would indicate that the second argument points to the program memory. For example, variables defined outside of a function body or variables that have the static storage class, have file storage class.

If you declare local variables with the `const` qualifier, they will not be put into Flash and your program will not compile.

6.12 Stack Architecture and Frame Layout

The stack must reside on the last page of data memory and grows towards high memory. Most local variables (non-static) and function parameters are allocated on the stack. A typical function stack frame would be:

```
[high address]
        [returned values]
    X:  [local variables and other compiler generated temporaries]
        [return address]
        [incoming arguments]
        [old X]
    [low address]
```

Register X is used as the frame pointer and for accessing all stacked items. Because the M8C limits the stack access to one page, no more than 256 bytes can be allocated on the stack, even if the device supports more than 256 bytes of RAM. Less RAM is available to the stack if the total RAM space is 256 bytes for the target device.

6.13 Strings

The compiler allocates all literal strings in program memory. Effectively, the type for declaring a literal string is `const char` and the type for referencing it is `const char*`. You must ensure that function parameters take the appropriate argument type.

6.14 Virtual Registers

Virtual registers are used for temporary data storage when running the compiler. Locations _r0, _r1, _r2, _r3, _r4, _r5, _r6, _r7, _r8, _r9, _r10, _r11, _rX, _rY, and _rZ are available. Only those that are required by the project are actually used. This extra register space is necessary because the M8C only has a single 8-bit accumulator. The Virtual registers are allocated on the low end of data memory.

If your PSoC Designer project is written exclusively in assembly language, the *boot.tpl* and *boot.asm* files can be modified by setting the equate `C_LANGUAGE_SUPPORT` to zero (0). This will save time and Flash space in the boot code.

6.15 Convention for Restoring Internal Registers

When calling PSoC user module APIs and library functions, it is the caller's responsibility to preserve the A and X registers. This means that if the current context of the code has a value in the X and/or A register that must be maintained after the API call, then the caller must save (`push` on the stack) and then restore (`pop` off the stack) them after the call has returned.

Even though some of the APIs do preserve the X and A register, Cypress reserves the right to modify the API in future releases in such a manner as to modify the contents of the X and A registers. Therefore, it is very important to observe the convention when calling from assembly. The C Compiler observes this convention as well.

7. Linker

In this chapter you will learn how the linker operates within PSoC Designer.

7.1 Linker Operations

The main purpose of the linker is to combine multiple object files into a single output file suitable to be downloaded to the In-Circuit Emulator (ICE) for debugging the code and programming the device. Linking takes place in PSoC Designer when a project build is executed. The linker can also take input from a library which is basically a file containing multiple object files. In producing the output file, the linker resolves any references between the input files.

7.2 Linking Process

In some detail, the steps involved in the linking process as are follows. For additional information about Linker and specifying Linker settings, refer to the *PSoC Designer Integrated Development Environment User Guide (Project Settings)*.

1. Making the startup file (*boot.asm*) the first file to be linked. The startup file initializes the execution environment for the C program to run.
2. Appending any libraries that you explicitly requested (or in most cases, as are requested by the IDE) to the list of files to be linked. Library modules that are directly or indirectly referenced will be linked. All user-specified object files (e.g., your program files) are linked. Note that the *libpsoc.a* library contains the user module API and *PSoCConfig.asm* routines.
3. Scanning the object files to find unresolved references. The linker marks the object file (possibly in the library) that satisfies the references and adds it to its list of unresolved references. It repeats the process until there are no outstanding unresolved references.
4. Combining all marked object files into an output file and generating map and listing files as needed.

7.2.1 Customized Linker Actions

It is possible to customize the actions of the Linker when a PSoC Designer build does not provide the user interface to support these actions.

A file called *custom.lkp* can be created in the root folder of the project, which can contain Linker commands (see Command Line Overview on page 43). Note that the file name must be *custom.lkp*. Be aware that in some cases, creating a text file and renaming it will still preserve the .txt file extension (e.g., *custom.lkp.txt*). If this occurs, your custom commands will not be used. The make file process reads the contents of *custom.lkp* and amends those commands to the Linker action.

A typical use for employing the *custom.lkp* capability would be to define a custom relocatable code AREA. This allows you to set a specific starting address for this AREA. For example, to create code in a separate code AREA called “Bootloader” that should be located in the upper 2k of the Flash, you could use this feature. If you were developing code in C for the BootLoader AREA you would use the following pragma in your C source file:

```
#pragma text:BootLoader         // switch the code below from
                                // AREA text to BootLoader
                                // ... Add your Code ...
#pragma text:text               // switch back to the text AREA
```

If you were developing code in assembly you would use the AREA directive as follows:

```
AREA BootLoader(rom,rel)
; ... Add your Code ...
AREA text ; reset the code AREA
```

Now that you have code that should be located in the BootLoader AREA, you can add your custom Linker commands to *custom.lkp*. For this example, you would enter the following line in the *custom.lkp* file:

```
-bBootLoader:0x3800.0x3FFF
```

You can verify that your custom Linker settings were used by checking the 'Use verbose build messages' field in the Builder tab under the **Tools > Options** menu. You can build the project then view the Linker settings in the Build tab of the Output Status window (or check the location of the BootLoader AREA in the .mp file).

In the large memory model, RAM areas can be fixed to a certain page using `-B`. For example, `-Bpage3ram:3` puts the area page3ram on page 3.

8. Librarian

In this chapter you will learn the librarian functions of PSoC Designer.

8.1 Librarian

A library is a collection of object files in a special form that the linker understands. When your program references a library's component object file directly or indirectly, the linker pulls out the library code and links it to your program. The library that contains supported C functions is usually located in the PSoC Designer installation directory at `...\PSoC Designer\tools\libs\SMM` (or `LMM\...)\libcm8c.a.` (SMM or LMM for small memory model or large memory model paging support.)

There are times when you need to modify or create libraries. A command line tool called *ilibw.exe* is provided for this purpose. Note that a library file must have the .a extension. For more information, refer to the Linker on page 37.

8.1.1 Compiling a File into a Library Module

Each library module is simply an object file. To create a library module, create a new project. Add all the necessary source files that you want added to your custom library to this project. You then add a project-specific MAKE file action to create the custom library.

As an example, create a blank project for any type of part, since interest is in using C and/or assembly, the Application Editor, and the Debugger for this example. The goal for creating a custom library is to centralize a set of common functions that can be shared between projects. These common functions, or primitives, have deterministic inputs and outputs. Another goal for creating this custom library is to be able to debug the primitives using a sequence of test instructions (e.g., a regression test) in a source file that should not be included in the library. No user modules are involved in this example.

PSoC Designer automatically generates a certain amount of code for each new project. In this example, use the generated `_main` source file to hold regression tests but do not add this file to the custom library. Also, do not add the generated *boot.asm* source file to the library. Essentially, all the files under the Source Files branch of the project view source tree go into a custom library, except *main.asm* (or *main.c*) and *boot.asm*.

Create a file called *local.dep* in the root folder of the project. The *local.dep* file is included by the master *Makefile* (found in the …`\PSoC Designer\tools` folder). The following shows how the *Makefile* includes *local.dep* (found at the bottom of *Makefile*):

```
#this include is the dependencies
-include project.dep

#if you don't like project.dep use your own!!!
-include local.dep
```

The nice thing about having *local.dep* included at the end of the master *Makefile* is that the rules used in the *Makefile* can be redefined (see the Help > Documentation `\Supporting Documents\make.pdf` for detailed information). In this example, it is used as an advantage. The following code shows information from example *local.dep*:

```
# ----- Cut/Paste to your local.dep File -----
define Add_To_MyCustomLib
$(CRLF)

$(LIBCMD) -a PSoCToolsLib.a $(library_file)
endef

obj/%.o : %.asm project.mk
ifeq ($(ECHO_COMMANDS),novice)
      echo $(call correct_path,$<)
endif
      $(ASMCMD) $(INCLUDEFLAGS) $(DEFAULTASMFLAGS)
      $(ASMFLAGS) -o $@ $(call correct_path,$<)
      $(foreach library_file, $(filter-out obj/main.o,
      $@), $(Add_To_MyCustomLib))

obj/%.o : %.c project.mk
ifeq ($(ECHO_COMMANDS),novice)
      echo $(call correct_path,$<)
endif
      $(CCMD) $(CFLAGS) $(CDEFINES) $(INCLUDEFLAGS)
      $(DEFAULTCFLAGS) -o $@ $(call correct_path,$<)
      $(foreach library_file, $(filter-out obj/main.o,
      $@), $(Add_To_MyCustomLib))
# ------ End Cut -----
```

The rules (e.g., `obj/%.o : %.asm project.mk` and `obj/%.o : %.c project.mk`) in the *local.dep* file shown above are the same rules found in the master *Makefile* with one addition each. The addition in the redefined rules is to add each object (target) to a library called *PSoCToolsLib.a*:

```
$(foreach library_file, $(filter-out obj/main.o,
$@), $(Add_To_MyCustomLib))
```

The MAKE keyword `foreach` causes one piece of text (the first argument) to be used repeatedly, each time with a different substitution performed on it. The substitution list comes from the second `foreach` argument.

In this second argument, there is another MAKE keyword/function called `filter-out`. The `filter-out` function removes `obj/main.o` from the list of all targets being built (e.g., `obj/%.o`). This was one of the goals for this example.

You can filter out additional files by adding those files to the first argument of `filter-out` such as `$(filter-out obj/main.o obj/excludeme.o, $@)`. The MAKE symbol combination `$@` is a shortcut syntax that refers to the list of all the targets (e.g., `obj/%.o`).

The third argument in the `foreach` function is expanded into a sequence of commands, for each substitution, to update or add the object file to the library. This *local.dep* example is prepared to handle both C and assembly source files and put them in the library, *PSoCToolsLib.a*. The library is created and updated in the project root folder in this example. However, you can provide a full path to another folder (e.g., `$(LIBCMD) -a c:\temp\PSoCToolsLib.a $(library_file)`).

Another goal for this example was to not include the *boot.asm* file in the library. This is easy given that the master *Makefile* contains a separate rule for the *boot.asm* source file, which will not be redefined in *local.dep*.

You can cut and paste this example and place it in a *local.dep* file in the root folder of any project. To view messages in the Build tab of the Output Status window regarding the behavior of your custom process, go to Tools > Options > Builder tab and click a check at “Use verbose build messages.“

Use the Project > Settings > Linker tab fields to add the library modules/library path if you want other PSoC Designer projects to link in your custom library.

8.1.2 Listing the Contents of a Library

On a command prompt window, change the directory to where the library is and give the command `ilibw -t <library>`. For example:

```
ilibw -t libcm8c.a
```

8.1.3 Adding or Replacing a Library Module

To add or replace a library module, execute the following procedure.

1. Compile the source file into an object module.
2. Copy the library into the working directory.
3. Use the command `ilibw -a <library> <module>` to add or replace a module.

   `ilibw` creates the library file if it does not exist. To create a new library, just give `ilibw` a new library file name.

8.1.4 Deleting a Library Module

The command switch `-d` deletes a module from the library. For example, the following deletes *crtm8c.o* from the *libcm8c.a* library:

```
ilibw -d libcm8c.a crtm8c.o;
```

9. Command Line Overview

In this chapter you will learn supported compiler command line options for users who want to use the compiler outside PSoC Designer. PSoC Designer normally sets all options for you. Use the information presented in this chapter to alter certain aspects of compiler behavior inside PSoC Designer using the *local.mk* file.

9.1 Compilation Process

Underneath the integrated development environment (IDE) is a set of command line compiler programs. While you do not need to understand this section to use the compiler, it is good for those who want supplemental information.

Given a list of files in a project, the compiler's job is to transform the source files into an executable file in some output format. Normally, the compilation process is hidden within the IDE. However, it can be important to have an understanding of what happens:

1. The compiler compiles each C source file to an assembly file.
2. The assembler translates each assembly file (either from the compiler or assembly files) into a relocatable object file.
3. Once all files have been translated into object files, the linker combines them to form an executable file. In addition, a map file, a listing file, and debug information files are also output.

9.2 Compiler Driver

The compiler driver handles all the details previously mentioned. It takes the list of files and compiles them into an executable file (which is the default) or to some intermediate stage (e.g., into object files). It is the compiler driver that invokes the compiler, assembler, and linker as needed.

The compiler driver examines each input file and acts on it based on its extension and the command line arguments given.

The *.c* files are C Compiler source files and the *.asm* files are assembly source files. The design philosophy for the IDE is to make it as easy to use as possible. The command line compiler is extremely flexible. You control its behavior by passing command line arguments to it. If you want to interface the compiler with PSoC Designer, note the following:

- Error messages referring to the source files begin with "!E file(line):.."
- To bypass any command line length limit imposed by the operating system, you may put command line arguments in a file, and pass it to the compiler as @file or @-file. If you pass it as @-file, the compiler will delete *file* after it is run.

9.3 Compiler Arguments

This section documents the options that are used by the IDE in case you want to drive the compiler using your own editor/IDE such as Codewright. All arguments are passed to the driver and the driver in turn applies the appropriate arguments to different compilation passes. The general format of a command is:

```
iccm8c [ command line arguments ] <file1> <file2> ... [
<lib1> ... ]
```

where `iccm8c` is the name of the compiler driver. You can invoke the driver with multiple files and the driver will perform the operations on all of the files. By default, the driver then links all the object files together to create the output file.

9.3.1 Compiler Argument Prefixes

For most of the common options, the driver knows which arguments are destined for which compiler pass. You can also specify which pass an argument applies to by using a `-W<c>` prefix. Table 9-1 presents examples of compiler argument prefixes.

Table 9-1. Compiler Argument Prefixes

Prefix	Description
-Wp	Preprocessor (e.g., -Wp-e)
-Wf	Compiler proper (e.g., -Wf-atmega)
-Wa	Assembler
-Wl (Letter el.)	Linker

9.3.2 Arguments Affecting the Driver

Table 9-2. Arguments Affecting the Driver

Argument	Action
-c	Compile the file to the object file level only (does not invoke the linker).
-o <name>	Name the output file. By default, the output file name is the same as the input file name, or the same as the first input file if you supply a list of files.
-v	Verbose mode. Print out each compiler pass as it is being executed.
-I	Include the specified path.

9.3.3 Preprocessor Arguments

Table 9-3. Preprocessor Arguments

Argument	Action
-D<name>[=value]	Define a macro.
-U<name>	Undefine a macro.
-e	Accept C++ comments.
-I<dir> (Capital i.)	Specify the location(s) to look for header files. Multiple -I flags can be supplied.

9.3.4 Compiler Arguments

Table 9-4. Compiler Arguments

Argument	Action
-l (Letter el.)	Generate a listing file.
-A -A (Two A's.)	Turn on strict ANSI checking. Single -A turns on some ANSI checking.
-g	Generate debug information.
-Osize	Optimize for size.

9.3.5 Linker Arguments

Table 9-5. Linker Arguments

Argument	Action
-L<dir>	Specify the library directory. Only one library directory (the last specified) will be used.
-O	Invoke code compressor.
-m	Generate a map file.
-g	Generate debug information.
-u<crt>	Use <crt> instead of the default startup file. If the file is just a name without path information, then it must be located in the library directory.
-W	Turn on relocation wrapping. Note that you need to use the -Wl prefix because the driver does not know of this option directly (i.e., -Wl-W).
-fihx_coff	Output format is both COFF and Intel® HEX.
-fcoff	Output format is COFF.
-fintelhex	Output format is Intel HEX.
-fmots19	Output format is Motorola S19.
-bfunc_lit:<address ranges>	Assign the address ranges for the area named func_lit. The format is <start address>[.<end address>] where addresses are word addresses. Memory that is not used by this area will be consumed by the areas to follow.
-bdata:<address ranges>	Assign the address ranges for the area or section named data, which is the data memory.
-dram_end:<address>	Define the end of the data area. The startup file uses this argument to initialize the value of the hardware stack.
-l<lib name>	Link in the specific library files in addition to the default *libcm8c.a*. This can be used to change the behavior of a function in *libcm8c.a* since *libcm8c.a* is always linked in last. The "libname" is the library file name without the "lib" prefix and without the ".a" suffix.
-B<name>:<page>	Put the area <name> on page <page>.

10. Code Compression

In this chapter you will learn how, why, and when to enable the PSoC Designer Code Compressor.

The Code Compressor will take into account that it may have to start with code that is larger than the available memory. It assumes that the ROM is 20-25% larger and then attempts to pack the code into the proper ROM maximum size.

10.1 Theory of Operation

The PSoC Designer Code Compressor replaces duplicate code blocks with a call to a single instance of the code. It also optimizes long calls or jumps (`LCALL` or `LJMP`) to relative offset calls or jumps (`CALL` or `JMP`).

Code compression occurs (if enabled) after linking the entire code image. The Code Compressor uses the binary image of the program as its input for finding duplicate code blocks. Therefore, it works on source code written in C or assembly or both. The Code Compressor utilizes other components produced during linking and the program map is used to take into account the various code and data areas.

To enable the PSoC Designer Code Compressor, click **Project > Settings > Compiler** tab. Code Compressor options are enabled or disabled for the open project by checking one, none, or both Compression Technologies: Condensation (duplicate code) or Sublimation (unused user module API elimination).

10.2 Code Compressor Process

The Code Compressor process is invoked as a linker switch. The compression theory involves consolidating similar program execution bytes into one copy and using a call where they are needed. Since this process deals with program execution bytes, some assumptions must be made clear.

10.2.1 C and Assembly Code

The Code Compressor cannot differentiate between code created from assembly or C source files. The process comes from the linker which only sees source objects in relocatable assembly form (i.e., it only sees images of bytes in the memory map and dis-assembles the program bytes to discover the instructions).

10.2.2 Program Execution Bytes

The Code Compressor process, created from the linker, makes an assumption that program execution bytes are tagged by the “AREA” they reside in. This assumption adds an abundance of usability issues. There is a rigid set of AREAs that the Code Compressor process expects program execution bytes to be in. PSoC project developers are free to create data tables in areas that the Code Compressor now expects only code. This is a project-compatibility issue discussed later in Section 10.4 on page 48.

Because the Code Compressor only sees bytes, it needs to know which portion of the memory image has valid instructions. It does this easily if the compiler and you adopt the simple convention that only instructions go into the default text area. The Code Compressor can handle other instruction areas, but it needs to know about them.

Since the Code Compressor expects a certain correlation between areas and code it can compress, any user-defined code areas will not be compressed.

10.2.3 Impact to Debugger

The Code Compressor will adjust the debug information file as swaps of code sequences with calls are made. It is expected that there should be very little impact on the debugger. The swaps of code sequences with calls are analogous to C math, which inserts math library calls.

10.3 Integration of the Code Compressor

10.3.1 *boot.asm* file

The *boot.asm* file is held within an area called "TOP." This contains the interrupt vector table (IVT) as well as C initialization, the sleep interrupt handler, and other initial setup functions. To effectively use the Code Compressor and reduce the special handling required by it to coordinate a special case area (TOP), it is required that you delineate the TOP and text areas within *boot.asm*.

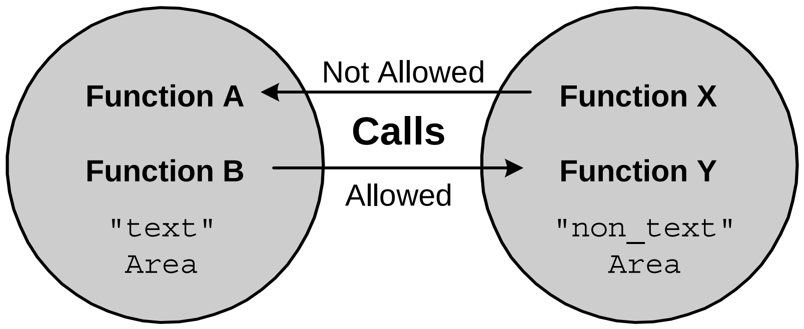
It is not a requirement for the *boot.asm* file to be split into multiple files. *boot.asm* just needs to use different AREAs for the different things (i.e., TOP for IVT). The startup code and the sleep timer may reside in *boot.asm*, as long as you use an "AREA text" before them to switch the area.

10.3.2 Text Area Requirement

The text area should be the last (e.g., highest memory addresses) relocatable code area if your expectation is to reduce the entire program image. You cannot shrink the whole program image if an absolute-code area is defined above the text area. However, you can still use the Code Compressor to shrink the "text" Area.

10.4 Code Compressor and the AREA Directive

The Code Compressor looks for duplicate code within the 'text" Area. The text Area is the default area in which all C code is placed.



The above diagram shows a scenario that is problematic. Code areas created with the AREA directive, using a name other than "text," are not compressed or fixed up following compression. If Function Y calls Function B, there is the potential that the location of Function B will be changed by the Code Compressor. The call or jump generated in the code for Function Y will go to the wrong location.

It is allowable for Function A to call a function in a “non_text” Area. The location for Function B can change because it is in the text Area. Calls and jumps are fixed up in the text Area only. Following code compression, the call location to Function B from Function X in the non-text Area will not be compressed.

All normal user code that is to be compressed must be in the default text Area. If you create code in other area (for example, in a bootloader), then it must not call any functions in the text Area. However, it is acceptable for a function in the text Area to call functions in other areas. The exception is the TOP area where the interrupt vectors and the startup code can call functions in the text Area. Addresses within the text Area must not be used directly.

If you reference any text area function by address, then it must be done indirectly. Its address must be put in a word in the area "func_lit." At runtime, you must de-reference the content of this word to get the correct address of the function. Note that if you are using C to call a function indirectly, the compiler will take care of all these details for you. The information is useful if you are writing assembly code.

10.5 Build Messages

When the Code Compressor is enabled, text messages will be displayed in the Build tab of the Output Status Window that describes the results of employing code compression. Messages for code compression appear following the Linker step of compilation/build. These messages are listed and described below.

1. `4054 bytes before Code Compression, 3774 after. 6% reduction`

   This is an example of code compression taking place. The values shown reflect the ‘text’ area bytes before and after code compression. This should not be confused with the entire program image.
2. `Program too small for worthwhile code compression`

   This message is shown when the Code Compressor has determined that no code savings could be accomplished; it is as though the Code Compressor option was turned off.

3.
   ```
   !X Cannot recover from assertion: new_target at internal source file
   ..\optm8c.c(180)
   Please report to "Cypress MicroSystems" support@cypressmicro.com
   ```

   This message informs the user that there was a fundamental mis-use of the Code Compressor. This is typically a result of placing a data table in the ‘text’ area.
4. `No worthwhile duplicate found`

   This message is possible with condensation code compression.
5. `No dead symbol found`

   This message is possible with sublimation code compression.

10.6 Considerations for Code Compression

1. Timing loops based on instruction cycles may change if those timing instructions are optimized.
2. Jump tables can change size. If the `JACC` instruction is used to access fixed offset boundaries in a table and the table includes entries with `LJMP` and/or `LCALL`, these can be optimized to relative jumps and/or calls.
3. ROM tables, in general, should be placed in the “lit” area. The Code Compressor expects code only to be in the ”text” area.
4. The Code Compression is turned off when an “effective suspend Code Compression” `NOP` instruction is seen. This instruction is `OR F,0` (or Suspend_CodeCompressor). Code compression resumes when a `RET` or `RETI` is encountered or another “effective resume Code Compression” `NOP` instruction (or Resume_CodeCompressor) is seen – `ADD SP,0`. This is useful when you wish to guard an instruction based cycle-delay routine.

Appendix A. Errors and Warnings Messages

This appendix supplies a complete list of preprocessor, preprocessor command line, compiler, assembler, assembler command line, and linker errors and warnings displayed in the PSoC Designer Status window.

A.1 Preprocessor

Table A-1. Preprocessor Errors and Warnings

Errors or Warnings
not followed by macro parameter
occurs at border of replacement
#defined token can't be redefined
#defined token is not a name
#elif after #else
#elif with no #if
#else after #else
#else with no #if
#endif with no #if
#if too deeply nested
#line specifies number out of range
Bad ?: in #if/endif
Bad syntax for control line
Bad token r produced by ## operator
Character constant taken as not signed
Could not find include file
Disagreement in number of macro arguments
Duplicate macro argument
EOF in macro arglist
EOF in string or char constant
EOF inside comment
Empty character constant
Illegal operator * or & in #if/#elsif
Incorrect syntax for `defined'
Macro redefinition
Multibyte character constant undefined
Sorry, too many macro arguments

Table A-1. Preprocessor Errors and Warnings *(continued)*

Errors or Warnings
String in #if/#elsif
Stringified macro arg is too long
Syntax error in #else
Syntax error in #endif
Syntax error in #if/#elsif
Syntax error in #if/#endif
Syntax error in #ifdef/#ifndef
Syntax error in #include
Syntax error in #line
Syntax error in #undef
Syntax error in macro parameters
Undefined expression value
Unknown preprocessor control line
Unterminated #if/#ifdef/#ifndef
Unterminated string or char const

A.2 Preprocessor Command Line

Table A-2. Preprocessor Command Line Errors

Errors
Can't open input file
Can't open output file
Illegal -D or -U argument
Too many -I directives

A.3 C Compiler

Table A-3. C Compiler Errors and Warnings

Errors or Warnings
Expecting <character>
Literal too long
IO port <name> cannot be redeclared as local variable
IO port <name> cannot be redeclared as parameter
IO port variable <name> cannot have initializer
<n> is a preprocessing number but an invalid %s constant
<n> is an illegal array size
<n> is an illegal bit-field size
<type> is an illegal bit-field type
<type> is an illegal field type
`sizeof' applied to a bit field
Addressable object required
asm string too long
Assignment to const identifier
Assignment to const location
Cannot initialize undefined
Case label must be a constant integer expression
Cast from <type> to <type> is illegal in constant expressions
Cast from <type> to <type> is illegal
Conflicting argument declarations for function <name>
Declared parameter <name> is missing
Duplicate case label <n>
Duplicate declaration for <name> previously declared at <line>
Duplicate field name <name> in <structure>
Empty declaration
Expecting an enumerator identifier
Expecting an identifier
Extra default label
Extraneous identifier <id>
Extraneous old-style parameter list
Extraneous return value
Field name expected
Field name missing
Found <id> expected a function
Ill-formed hexadecimal escape sequence
Illegal break statement
Illegal case label
Illegal character <c>
Illegal continue statement

Table A-3. C Compiler Errors and Warnings *(continued)*

Errors or Warnings
Illegal default label
Illegal expression
Illegal formal parameter types
Illegal initialization for <id>
Illegal initialization for parameter <id>
Illegal initialization of `extern <name>'
Illegal return type <type>
Illegal statement termination
Illegal type <type> in switch expression
Illegal type `array of <name>'
Illegal use of incomplete type
Illegal use of type name <name>
Initializer must be constant
Insufficient number of arguments to <function>
Integer expression must be constant
Interrupt handler <name> cannot have arguments
Invalid field declarations
Invalid floating constant
Invalid hexadecimal constant
Invalid initialization type; found <type> expected <type>
Invalid octal constant
Invalid operand of unary &; <id> is declared register
Invalid storage class <storage class> for <id>
Invalid type argument <type> to `sizeof'
Invalid type specification
Invalid use of `typedef'
Left operand of -> has incompatible type
Left operand of . has incompatible type
Lvalue required
Missing <c>
Missing tag
Missing array size
Missing identifier
Missing label in goto
Missing name for parameter to function <name>
Missing parameter type
Missing string constant in asm
Missing { in initialization of <name>
Operand of unary <operator> has illegal type
Operands of <operator> have illegal types <type> and <type>

Table A-3. C Compiler Errors and Warnings *(continued)*

Errors or Warnings
Overflow in value for enumeration constant
Redeclaration of <name> previously declared at <line>
Redeclaration of <name>
Redefinition of <name> previously defined at <line>
Redefinition of label <name> previously defined at <line>
Size of <type> exceeds <n> bytes
Size of `array of <type>' exceeds <n> bytes
Syntax error; found
Too many arguments to <function>
Too many errors
Too many initializers
Too many variable references in asm string
Type error in argument <name> to <function>; <type> is illegal
Type error in argument <name> to <function>; found <type> expected <type>
Type error
Unclosed comment
Undeclared identifier <name>
Undefined label
Undefined size for <name>
Undefined size for field <name>
Undefined size for parameter <name>
Undefined static <name>
Unknown #pragma
Unknown size for type <type>
Unrecognized declaration
Unrecognized statement

A.4 Assembler

Table A-4. Assembler Errors and Warnings

Errors or Warnings
'[' addressing mode must end with ']'
) expected
.if/.else/.endif mismatched
<character> expected
EOF encountered before end of macro definition
No preceding global symbol
Absolute expression expected
Badly formed argument, (without a matching)
Branch out of range
Cannot add two relocatable items
Cannot perform subtract relocation
Cannot subtract two relocatable items
Cannot use .org in relocatable area
Character expected
Comma expected
equ statement must have a label
Identifier expected, but got character <c>
Illegal addressing mode
Illegal operand
Input expected
Label must start with an alphabet, '.' or '_'
Letter expected but got <c>
Macro <name> already entered
Macro definition cannot be nested
Maximum <#> macro arguments exceeded
Missing macro argument number
Multiple definitions <name>
No such mnemonic <name>
Relocation error
Target too far for instruction
Too many include files
Too many nested .if
Undefined mnemonic <word>
Undefined symbol
Unknown operator
Unmatched .else
Unmatched .endif

A.5 Assembler Command Line

Table A-5. Assembler Command Line Errors

Errors
Cannot create output file %s\n
Too many include paths

A.6 Linker

Table A-6. Linker Errors and Warnings

Errors or Warnings
Address <address> already contains a value
Can't find address for symbol <symbol>
Can't open file <file>
Can't open temporary file <file>
Cannot open library file <file>
Cannot write to <file>
Definition of builtin symbol <symbol> ignored
Ill-formed line <%s> in the listing file
Multiple define <name>
No space left in section <area>
Redefinition of symbol <symbol>
Undefined symbol <name>
Unknown output format <format>

Index

A

absolute memory locations 31
accessing
  M8C features 31
  the compiler 13
acronyms 11
API software library functions 29
AREA directive 48
arguments, compiler 44
assembler
  command line errors 57
  errors and warnings 56
assembly interface and calling conventions 31

B

bit toggling 32
boot.asm file 15, 48
boot.tpl file 15
build messages for code compression 49

C

C compiler errors and warnings 53
calling conventions and assembly interface 31
character type functions 23
cms.a file 15
code compression
  AREA directive 48
  build messages 49
  considerations 50
  integration of code compressor 48
  process 47
  text area requirement 48
  theory of operation 47
compilation process 43
compiler
  accessing 13
  basics 17
  driver 43
  enabling 13
  files 15
  interrupts 33
  linker operations 37
  processing directives 21
  startup 15
compiler arguments
  compiler 45
  driver 44
  linker 45
  prefixes 44
  preprocessor 44
constant data 35
custom.lkp file 37

D

data memory
  areas 34
  usage 34
data types supported 17
documentation
  acronyms 11
  conventions 11
  overview 10
  purpose 9
  reference materials 9
  suite 9
driver 43

E

enabling the compiler 13
error messages 51
expressions, supported 19

F

fastcall function 30
files
  library descriptions 15
  name conventions 15
files for startup 15
Flash memory areas 34
frame layout 36
functions
  API software library 29
  interfacing C and assembly 30
  library 23
  mathematical 28
  string 23

H
help 10

I
inline assembly 32
integration of code compressor 48
interfacing C and assembly (fastcall) 30
interrupts 33
IO registers 33

L
libcm8c.a file 15
libpsoc.a file 15
librarian
- adding or replacing a library module 41
- compiling a file into a library module 39
- deleting a library module 41
- listing the contents of a library 41

library
- descriptions 15
- functions 23

linker
- customized actions 37
- errors and warnings 57
- process 37

long jump/call 33

M
M8C features, accessing 31
mathematical functions 28
memory areas
- data 34
- Flash 34

memory locations, absolute 31
memory usage, program and data 34
menu options 14

N
name conventions for files 15

O
operators, supported 18
options
- additional considerations 31
- command line 44

overview of document 10

P
page pointer registers 22
pointers 20
pragma directives 21
- #pragma abs_address 21
- #pragma end_abs_address 21
- #pragma fastcall GetChar 21
- #pragma fastcall16 argument passing 30
- #pragma fastcall16 GetChar 21
- #pragma fastcall16 return value 30
- #pragma interrupt_handler 22
- #pragma ioport 21
- #pragma nomac 22
- #pragma text 21
- #pragma usemac 22

preprocessor
- command line errors 52
- directives 21
- errors and warnings 51

process of code compression 47
processing directives
- pragma directives 21
- preprocessor directives 21

program memory usage 34
- as related to constant data 35

purpose of document 9

R
re-entrancy 20
reference materials 9
restoring internal registers 36

S
stack architecture 36
startup files 15
statements, supported 20
string functions 23
strings 36
suite of documentation 9
support 10

T
technical support systems 10
toolbar options 14

U
upgrades 10

V
virtual registers 36

Revision History

Document Revision History

Document Title: PSoC Designer C Language Compiler User Guide **Document Number**: **38-12001**				
Revision	**ECN #**	**Issue Date**	**Origin of Change**	**Description of Change**
**	115167	4/23/2002	Submit to CY Document Control. Updates.	New document to CY Document Control (Revision **). Revision 1.15 for Cypress customers.
*A			UWE	Added “Convention for Restoring Internal Registers.”
*B			HMT	--Added code-compression details. --Options using *custom.lkp*. --Reworked “Compiling a File into a Library Module.” --Added typedef and fixed Inline Assembly example. --Added ftoa, updated address/links.
*C			HMT	Updated everything, including fastcall16. Added references to LMM.
*D			HMT	String Function behavior changes.
*E		9/13/2005	ARI	Implememted Cypress formats and new template. Split single-body file into several chapters. Updated and added new material.
Distribution: External/Public **Posting**: None				