



PSoC™ Designer:  
**C Language Compiler**

User Guide

Revision 1.17 (Cypress Revision \*B)  
Spec.# 38-12001

Last Revised: December 5, 2003  
Cypress MicroSystems, Inc.



CYPRESS MICROSYSTEMS

Cypress Microsystems, Inc.  
2700 162nd St. SW, Building D  
Lynnwood, WA 98037  
Phone: 800.669.0557  
Fax: 425.787.4641

<http://www.cypress.com/> [http://www.cypress.com/aboutus/sales\\_locations.cfm](http://www.cypress.com/aboutus/sales_locations.cfm) [support@cypressmicro.com](mailto:support@cypressmicro.com)

Copyright © 2002-2003 Cypress Microsystems, Inc. All rights reserved.  
PSoC™ (Programmable System-on-Chip) is a trademark of Cypress Microsystems, Inc.

Copyright © 1999-2000 iMAGEcraft Creations Inc. All rights reserved.  
All Microsoft products referenced herein are either trademarks or registered trademarks of Microsoft Corp.  
All Intel products referenced herein are either trademarks or registered trademarks of Intel Corporation.  
© Copyright 1994-2002 Motorola, Inc. All Rights Reserved.

The information contained herein is subject to change without notice.

## Table of Contents

<b>List of Tables .....</b>	<b>v</b>
<b>Two-Minute Overview .....</b>	<b>1</b>
<b>Documentation Conventions .....</b>	<b>2</b>
<b>Section 1. Introduction .....</b>	<b>3</b>
1.1 What is the PSoC Designer C Compiler? .....	3
1.2 Section Overview .....	4
1.3 Product Upgrades .....	4
1.4 Support .....	5
<b>Section 2. Accessing the Compiler .....</b>	<b>7</b>
2.1 Enabling the Compiler .....	7
2.2 Accessing the Compiler .....	7
2.3 Menu Options .....	8
<b>Section 3. Compiler Files .....</b>	<b>11</b>
3.1 Startup File .....	11
3.2 Library Descriptions .....	11
<b>Section 4. Compiler Basics .....</b>	<b>13</b>
4.1 Types .....	13
4.2 Operators .....	14
4.3 Expressions .....	16
4.4 Statements .....	16
4.5 Pointers .....	17
4.6 Re-entrancy .....	17
4.7 Processing Directives (#'s) .....	17
4.7.1 Preprocessor Directives .....	18
4.7.2 pragma Directives .....	18
<b>Section 5. Functions .....</b>	<b>21</b>
5.1 Library Functions .....	21
5.1.1 String Functions .....	21
5.1.2 Mathematical Functions .....	24
5.1.3 API Software Library Functions .....	26
5.2 Interfacing C and Assembly .....	26
<b>Section 6. Additional Considerations .....</b>	<b>29</b>
6.1 Accessing M8C Features .....	29
6.2 Addressing Absolute Memory Locations .....	29
6.3 Assembly Interface and Calling Conventions .....	30
6.4 Bit Twiddling .....	30
6.5 Inline Assembly .....	31
6.6 Interrupts .....	31
6.7 IO Registers .....	32

6.8 Long Jump/Call .....	32
6.9 Memory Areas .....	33
6.9.1 Flash Memory Areas .....	33
6.9.2 Data Memory .....	33
6.10 Program and Data Memory Usage .....	33
6.10.1 Program Memory .....	33
6.10.2 Data Memory .....	34
6.11 Program Memory as Related to Constant Data .....	34
6.12 Stack Architecture and Frame Layout .....	35
6.13 Strings .....	35
6.14 Virtual Registers .....	36
6.15 Convention for Restoring Internal Registers .....	36
<b>Section 7. Linker .....</b>	<b>37</b>
7.1 Linker Operations .....	37
7.1.1 Customized Linker Actions .....	38
<b>Section 8. Librarian .....</b>	<b>39</b>
8.1 Librarian .....	39
8.1.1 Compiling a File into a Library Module .....	39
8.1.2 Listing the Contents of a Library .....	42
8.1.3 Adding or Replacing a Module .....	42
8.1.4 Deleting a Module .....	42
<b>Section 9. Command Line Compiler Overview .....</b>	<b>43</b>
9.1 Compilation Process .....	43
9.2 Driver .....	43
9.3 Compiler Arguments .....	44
9.3.1 Arguments Affecting the Driver .....	45
9.3.2 Preprocessor Arguments .....	45
9.3.3 Compiler Arguments .....	45
9.3.4 Linker Arguments .....	45
<b>Section 10. Code Compression .....</b>	<b>47</b>
10.1 Theory of Operation .....	47
10.2 Code Compressor Process .....	47
10.2.1 'C' and Assembly Code .....	47
10.2.2 Where are the "Program Execution" Bytes? .....	48
10.2.3 What Can the PSoC Debugger Expect? .....	48
10.3 PSoC Designer Integration of the Code Compressor .....	48
10.3.1 boot.asm .....	48
10.3.2 Text Area Requirement for Code Compressor .....	48
10.4 Code Compressor and the AREA Directive .....	49
10.5 Build Messages .....	50
10.6 Up against the Wall? .....	50
10.7 Additional Things to Consider When Using Code Compression .....	51
<b>Appendix A. Status Window Messages .....</b>	<b>53</b>
1.1 Preprocessor .....	53
1.2 Preprocessor Command Line Errors .....	55
1.3 C Compiler .....	55
1.4 Assembler .....	59
1.5 Assembler Command Line Errors .....	61
1.6 Linker .....	61

**Index ..... 63**



## List of Tables

Table 1: Documentation Conventions .....	2
Table 2: Compiler Menu Options .....	8
Table 3: Supported Data Types .....	13
Table 4: Supported Operators .....	15
Table 5: Preprocessor Directives .....	18
Table 6: pragma Directives .....	18
Table 7: String Functions .....	22
Table 8: Mathematical Functions .....	24
Table 9: API Software Library Functions .....	26
Table 10: #pragma Fastcall Conventions for Argument Passing .....	26
Table 11: #pragma Fastcall Conventions for Return Value .....	27
Table 12: Compiler Argument Prefixes .....	44
Table 13: Arguments Affecting the Driver .....	45
Table 14: Preprocessor Arguments .....	45
Table 15: Compiler Arguments .....	45
Table 16: Linker Arguments .....	45
Table A.1: Preprocessor Errors/Warnings .....	53
Table A.2: Preprocessor Command Line Errors/Warnings .....	55
Table A.3: C Compiler Errors/Warnings .....	55
Table A.4: Assembler Errors/Warnings .....	59
Table A.5: Assembler Command Line Errors/Warnings .....	61
Table A.6: Linker Errors/Warnings .....	61






---

## Two-Minute Overview

This two-minute overview of *PSoC™ Designer: C Language Compiler User Guide* was purposefully placed up front for you advanced engineers who are ready to write source for the device but need a *quick* jump-start. (Now we only have a minute and-a-half left.)

<b>Overview</b>	35 seconds	You have the device, PSoC Designer, and the C compiler... This guide provides: <ul style="list-style-type: none"><li>▪ enabling and accessing procedures,</li><li>▪ instructions for using the C compiler within PSoC Designer parameters,</li><li>▪ references for the internal workings of the compiler.</li></ul>
<b>Basics</b>	30 seconds	After generating your device configuration, click the Application Editor icon  in the toolbar to access the pre-configured source files.  The source tree of project files appears in the left frame. The folders can be expanded to reveal the files. Double-click individual files to open and edit them in the main window. Click <u>F</u> ile >> <u>N</u> ew to add .c files to your project.
<b>Quick Reference</b>	25 seconds	Click a hyperlink to reference key material: <ul style="list-style-type: none"><li>▪ <a href="#">Accessing the Compiler</a></li><li>▪ <a href="#">Compiler Files</a></li><li>▪ <a href="#">Compiler Basics</a></li><li>▪ <a href="#">Functions</a></li><li>▪ <a href="#">Processing Directives (#'s)</a></li><li>▪ <a href="#">Librarian</a></li></ul>
<b>Bottom Line</b>	10 seconds	The PSoC Designer C Compiler is an “extra” tool you can use to customize the functionality you desire into the PSoC device.



Time's up...

## Documentation Conventions

Following, are easily identifiable conventions used throughout the PSoC Designer suite of product documentation.

**Table 1: Documentation Conventions**

Convention	Usage
Courier Size 10-12	Displays input and output: <pre>//----- // Sample Code // Burn some cycles //-----  void main() { char cOuter, cInner; for(cOuter=0x20; cOuter&gt;0; cOuter--) { for(cInner=0x7F; cInner&gt;0; cInner--) { } } }</pre>
Courier Size 12	Displays file locations: C:\Program Files\Cypress MicroSystems\PSoC Designer\tools
<i>Italics</i>	Displays file names: <i>projectname.rom</i>
<b>[bracketed, bold]</b>	Displays keyboard commands: <b>[Enter]</b> or <b>[Ctrl] [C]</b>
File >> Open	Displays menu paths: Edit >> Cut

## Section 1. Introduction

### 1.1 What is the PSoC Designer C Compiler?

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 builder/linker combines them together to form an executable file. This .rom 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 see:

- *PSoC Designer: Integrated Development Environment User Guide*
- *PSoC Designer: Assembly Language User Guide*
- *PSoC Designer: ICE Connection Troubleshooting Guide*
- *CY8C25122, CY8C26233, CY8C26443, CY8C26643 Device Data Sheet for Silicon Revision D*
- *CY8C27143, CY8C27243, CY8C27443, CY8C27543, CY8C27643 PSoC Mixed-Signal Array Data Sheet*
- *CY8C24123, CY8C24223, CY8C24423 PSoC Mixed-Signal Array Data Sheet*
- *CY8C22113, CY8C22213 PSoC Mixed-Signal Array Data Sheet*
- *In-System Serial Programming (ISSP) CY3207ISSP User Guide*

Together, these documents complete the PSoC Designer documentation suite.

Additional recommended reading includes:

- *C Programming Language*, Second Edition, Brian W. Kernighan and Dennis Ritchie, Pearson Education March 1989
- *C: A Reference Manual*, Fifth Edition, Samuel P. Harbison and Guy L. Steele, Pearson Education February 2002

## 1.2 Section Overview

Section 1. Introduction	Describes the purpose of this guide, overviews each section, and gives product upgrade and support information.
Section 2. Accessing the Compiler	Describes enabling and accessing the compiler and its options.
Section 3. Compiler Files	Discusses and lists startup and C library options within PSoC Designer.
Section 4. Compiler Basics	Lists C compiler types, operators, expressions, statements, and pointers that are compatible within PSoC Designer parameters.
Section 5. Functions	Lists C compiler functions that are compatible within PSoC Designer parameters.
Section 6. Additional Considerations	Lists additional compiler options you can use to leverage the functionality of your code or program.
Section 7. Linker	Discusses C compiler linker options deployed within PSoC Designer.
Section 8. Librarian	Discusses C compiler library functions used within PSoC Designer.
Section 9. Command Line Compiler Overview	Overviews C compiler command line features that can be used strictly within the constraints of PSoC Designer.
Section 10. Code Compression	Details code compression features, benefits and guidelines.

## 1.3 Product Upgrades

Cypress MicroSystems 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, better yet, 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 PSoC >> More Resources at <http://www.cypress.com/>.

Check the web site frequently for both product and documentation updates. As the device families and PSoC Designer evolve, you can be sure that new technology, features and enhancements will be added.

## 1.4 Support

Support for PSoC Designer and its C Compiler is *free* and available online. Resources include Seminars, Discussion Forums, Application Notes, PSoC Consultants, TightLink Email/Knowledge Base, Tele-Training, and Support Technicians.

Application Hotline: 425.787.4814



## Section 2. Accessing the Compiler

**In this section you will learn** how to enable and access the compiler and its options.

### 2.1 Enabling the Compiler

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

1. Access **T**ools >> **O**ptions >> 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. Scroll to read the License Agreement, then click a check to accept the License Agreement and hit **OK**.

To view the version details for the iMAGEcraft Compiler, click **V**ersion. When finished, click the “x” in the upper-right to close.

If, for some reason, you have not received a *key code* or are uncertain of how to proceed, contact a Cypress MicroSystems Support Technician at 425.787.4814.

### 2.2 Accessing the Compiler

All features of the compiler are available and accessible in the Application Editor subsystem of PSoC Designer.

To access the Application Editor subsystem, click the Application Editor icon . This icon can be found in the subsystem toolbar .

Such features include adding and modifying .c project files, both of which are described ahead in brief, and in the *PSoC Designer: Integrated Development Environment User Guide* in detail.

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

## 2.3 Menu Options






The following menu options are available in PSoC Designer for writing and editing assembly language and C compiler files:

**Table 2: Compiler Menu Options**

Icon	Option	Menu	Shortcut	Feature
	Compile/ Assemble	<u>B</u> uild >> <u>C</u> ompile/Assemble	[ <b>Ctrl</b> ] [ <b>F7</b> ]	Compiles/assembles the most prominent open, active file (.c or .asm)
	Build	<u>B</u> uild >> <u>B</u> uild	[ <b>F7</b> ]	Builds entire project and links applicable files
	New File	<u>F</u> ile >> <u>N</u> ew	[ <b>Ctrl</b> ] [ <b>N</b> ]	Adds a new file to the project
	Open File	<u>F</u> ile >> <u>O</u> pen	[ <b>Ctrl</b> ] [ <b>O</b> ]	Opens an existing file in the project
	Indent			Indents specified text
	Outdent			Outdents specified text
	Comment			Comments selected text
	Uncomment			Uncomments selected text
	Toggle Book- mark			Toggles the bookmark: Sets/removes user-defined bookmarks used to navigate source files
	Clear Book- mark			Clears all user-defined bookmarks
	Next Book- mark			Goes to next bookmark
	Previous Bookmark			Goes to previous bookmark
	Find Text	<u>E</u> dit >> <u>F</u> ind	[ <b>Ctrl</b> ] [ <b>F</b> ]	Find specified text
	Replace Text	<u>E</u> dit >> <u>R</u> eplace	[ <b>Ctrl</b> ] [ <b>H</b> ]	Replace specified text



**Table 2: Compiler Menu Options, continued**

Icon	Option	Menu	Shortcut	Feature
	Find in Files	<u>E</u> dit >> F <u>in</u> d in Files		Find specified text in specified file(s)
	Repeat Replace			Repeats last replace
	Set Editor Options			Set options for editor
	Undo	<u>E</u> dit >> <u>U</u> ndo	[ <b>Ctrl</b> ] [ <b>Z</b> ]	Undo last action
	Redo	<u>E</u> dit >> <u>R</u> edo	[ <b>Ctrl</b> ] [ <b>Y</b> ]	Redo last action



## Section 3. Compiler Files

**In this section you will learn** startup file procedures and can reference supported library files.

### 3.1 Startup File

PSoC Designer creates a startup file called *boot.asm*. Its primary functions within the parameters of PSoC Designer include initializing C variables, organizing interrupt tables, and calling `_main`. The underscore (`_main`) allows *boot.asm* to call a “main” in either C or assembly.

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 MicroSystems Technical Support Representative.

The *boot.asm* startup file also defines the reset vector. You do not need to modify the startup file to use other interrupts because PSoC Designer manages interrupts and vectors.

### 3.2 Library Descriptions

There are three primary code libraries used by PSoC Designer: *libcm8c.a*, *libpsoc.a*, and *cms.a*.

The *libcm8c.a* library resides in the PSoC Designer `...\tools` directory (`...\Program Files\Cypress Microsystems\PSoC Designer\tools`). This library contains many functions typically used in 'C' programming.

The *libpsoc.a* 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 “officially” 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): `\ / : * ? " < > | & + , ; = [ ] % $ ` ' .`

The *cms.a* library resides in the `...\tools` directory. This library 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.

## Section 4. Compiler Basics

**In this section you can reference** PSoC Designer C Compiler basics, which include types, operators, expressions, statements, and pointers.

With few exceptions, PSoC Designer C Compiler implements the ANSI C language. The one notable exception is that the standard requires that double floating point be at least 64 bits, but implementing full 64 bits is prohibitive on 8-bit microcontrollers. Therefore, PSoC Designer C Compiler treats the “double” data type the same as the “float” data type.

In terms of the compiler, the ONLY non-ANSI feature is that doubles are only 32-bit. ANSI “bar” proper would require that to be at least 64 bits. Otherwise, if it is in C89 or the ISO equivalent, it is supported by the compiler.

### 4.1 Types

PSoC Designer C Compiler supports the following standard data types:

All types support the signed and unsigned type modifiers.

**Table 3: Supported Data Types**

Type	Bytes	Description	Range
char	1	A single byte of memory that defines characters	<sup>1</sup> unsigned 0...255 signed -128...127
int	2	Used to define integer numbers	unsigned 0...65535 <sup>1</sup> signed -32768...32767
short	2	Standard type specifying 2-byte integers	unsigned 0...65535 <sup>1</sup> signed -32768...32767

**Table 3: Supported Data Types, continued**

Type	Bytes	Description	Range
long	4	Standard type specifying the largest integer entity	unsigned 0...4294967295 <sup>1</sup> signed - 2147483648...21474836 47
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

<sup>1</sup> Default, if not explicitly specified as signed or unsigned.

The following type definitions are included in *m8c.inc*. Express common conventions for additional data types.

```
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. Floating Point Intrinsic Functions:

```
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

Following is 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: 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	(decla- ration)	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
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	

**Table 4: Supported Operators, continued**

Pre.	Op.	Function	Group	Form	Description
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 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**, only the test runs after execution of 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.

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. See **Program Memory as Related to Constant Data** in section 6.

Pointers require 2 bytes of memory storage to account for the size of both the data and program memory.

## 4.6 Re-entrancy

Currently, there are no pure re-entrant library functions. It is possible, however, 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 (#'s)

PSoC Designer C Compiler supports the following preprocessors and pragmas:

## 4.7.1 Preprocessor Directives

**Table 5: Preprocessor Directives**

Preprocessor	Description
<code>#define</code>	Define a preprocessor constant or macro
<code>#else</code>	Executed if <code>#if</code> , <code>#ifdef</code> , or <code>#ifndef</code> fails
<code>#endif</code>	Close <code>#if</code> , <code>#ifdef</code> , or <code>#ifndef</code>
<code>#if</code>	Branch based on an expression
<code>#ifdef</code>	Branch if preprocessor constant has been defined
<code>#ifndef</code>	Branch if a preprocessor constant has <i>not</i> been defined
<code>#include</code>	Insert a source file
<code>#line</code>	Specify the number of the next source line
<code>#undef</code>	Remove a preprocessor constant

## 4.7.2 pragma Directives

**Table 6: pragma Directives**

#pragma	Description
<code>#pragma ioport LED:0x04;</code> <code>char LED;</code>	Defines a variable that occupies a region in I/O space. This variable can then be used in I/O reads and writes. The <code>#pragma ioport</code> must precede a variable declaration defining the variable type used in the pragma.
<code>#pragma fastcall GetChar</code>	Provides an optimized mechanism for argument passing. This <code>#pragma</code> is used only for assembly functions called from "C."
<code>#pragma</code> <code>abs_address:&lt;address&gt;</code>	Allows you to locate 'C' code/Flash data at a specific address such as <code>#pragma abs_address:0x500</code> . The <code>#pragma end_abs_address</code> (described below) should be used to terminate the block of code/Flash data. Note that "data" includes both ROM and RAM.

**Table 6:       pragma Directives, continued**

#pragma	Description
#pragma end_abs_address	Terminates a block of code/Flash data that was located with the abs_address pragma. This allows the code that follows the end_abs_address pragma to be located from the last relocatable point. Note that “data” includes both ROM and RAM.
#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.
#pragma interrupt_handler <func1> [ ,<func2> ]*	For interrupt handlers. Virtual registers are saved only if they are used, unless the handler calls another function. In that case, all Virtual registers are saved.
#pragma nomac #pragma usemac	These two pragmas override the command line -nomac argument, or <u>P</u> roject >> <u>S</u> ettings, Compiler tab, Enable MAC option. See Section 3, Compiler Project Settings in the <i>PSoC Designer: Integrated Development Environment User Guide</i> . The pragmas should be specified outside of a function definition.



## Section 5. Functions

**In this section you can reference** compiler functions supported within PSoC Designer.

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 of 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 below. Note that two versions of these functions are provided. The 'c' prefix indicates that the source string `s2` is located in Flash, as designated by the *const* qualifier. PSoC Designer supports the following library functions:

#### 5.1.1 String Functions

The following functions can be found in the PSoC Designer installation directory at `...:\Program Files\Cypress Microsystems\PSoC Designer\tools\include\string.h` and `stdlib.h`

All strings are null terminated strings.

**Table 7: String Functions**

Function	Prototype	Description	Header
abs	int abs(int);		stdlib.h
atof	double atof(CONST char *);		stdlib.h
atoi	int atoi(CONST char *);		stdlib.h
atol	long atol(CONST char *);		stdlib.h
itoa	void itoa(char *string, unsigned int value, int base);		stdlib.h
ltoa	void ltoa(char *string, unsigned long value, int base);		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 val- ues outside * of the range listed. Please contact us if you need a (much) larger * version that han- dles greater ranges. * Note that the prototype differs from the earlier version of this * function. */	stdlib.h
rand	int rand(void);		stdlib.h
srand	void srand(unsigned);		stdlib.h
strtol	long strtol(CONST char *, char **, int);		stdlib.h
strtoul	unsigned long strtoul(CONST char *, char **, int);		stdlib.h
	char *cstrncpy(char *, const char *cs, size_t);		string.h
cstrcat	char *cstrcat(char *, const char *);		string.h
cstrcmp	int cstrcmp(const char *cs, char *);		string.h
cstrcpy	char *cstrcpy(char *, const char *cs);		string.h
cstrlen	size_t cstrlen(const char *cs);		string.h
memchr	void *memchr(void *, int, size_t);		string.h
memcmp	int memcmp(void *, void *, size_t);		string.h

**Table 7: String Functions, continued**

Function	Prototype	Description	Header
memcpy	void *memcpy(void *, void *, size_t);		string.h
memmove	void *memmove(void *, void *, size_t);		string.h
memset	void *memset(void *, int, size_t);		string.h
strcat	char *strcat(char *, CONST char *);		string.h
strcmp	int strcmp(CONST char *, CONST char *);		string.h
strcoll	int strcoll(CONST char *, CONST char *);		string.h
strcpy	char *strcpy(char *, CONST char *);		string.h
strcspn	size_t strcspn(CONST char *, CONST char *);		string.h
strlen	size_t strlen(CONST char *);		string.h
strncat	char *strncat(char *, CONST char *, size_t);		string.h
strncmp	int strncmp(CONST char *, CONST char *, size_t);		string.h
strncpy	char *strncpy(char *, CONST char *, size_t);		string.h
strpbrk	char *strpbrk(CONST char *, CONST char *);		string.h
strrchr	char *strrchr(CONST char *, int);		string.h
strspn	size_t strspn(CONST char *, CONST char *);		string.h
strstr	char *strstr(CONST char *, CONST char *);		string.h

You can view the list of string functions at a command prompt window by typing: `...:\Program Files\Cypress Microsystems\PSoC Designer\tools> ilibw -t libcm8c.a`

## 5.1.2 Mathematical Functions

The following functions can be found in the PSoC Designer installation directory at ...:\Program Files\Cypress Microsystems\PSoC Designer\tools\include\math.h.

**Table 8: Mathematical Functions**

Function	Description
float fabs(float x);	fabs 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 *epr);	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 $0.5 \leq \text{abs}(m) < 1$ .
float tanh(float x);	The function returns the hyperbolic tangent of x.
float sin(float x);	Return the sine of x.
float atan(float x);	The function returns the angle whose tangent is x, in the range $[-\pi/2, +\pi/2]$ radians.
float atan2(float y, float x);	The function returns the angle whose tangent is y/x, in the full angular range $[-\pi, +\pi]$ radians.
float asin(float x);	The function 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);	The fmod function 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);	The function returns the square root of x, $x^{(1/2)}$ .
float cos(float x);	The function 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);	ldexp calculates the value that it takes and returns float rather than double values. ldexp returns the calculated value.



**Table 8: Mathematical Functions, continued**

Function	Description
float modf(float y, float *i);	modf 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);	floor finds the nearest integer less than or equal to x. floor returns the integer result as a double.
float ceil(float y);	ceil finds the nearest integer greater than or equal to x. ceil returns the integer result as a double.
float fround(float d);	Produce 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);	The function 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);	acos 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.
float exp(float x);	exp 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 <sup>x</sup> .
float log(float x);	Return 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);	pow calculates x raised to the exp1.0nt y. On success, pow returns the value calculated.
float sinh(float x);	sinh computes the hyperbolic sine of the argument x. The function returns the hyperbolic sine of x.
float cosh(float x);	cosh computes the hyperbolic cosine of the argument x. The function returns the hyperbolic cosine of x.

### 5.1.3 API Software Library Functions

**Table 9: API Software Library Functions**

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

The header and include files can be found at: ...:\Program Files\Cypress Microsystems\PSoC Designer\tools\include.

## 5.2 Interfacing C and Assembly

To optimize argument passing and return values from a C function to an assembler function, use the `#pragma fastcall`.

The fastcall convention was devised to create an efficient argument/return value mechanism between 'C' and assembly language functions.

Fastcall is only used by 'C' functions calling assembly written functions. Functions written in 'C' cannot utilize the fastcall convention.

The following table reflects the set of `#pragma fastcall` conventions used for *argument passing* register assignments:

**Table 10: #pragma Fastcall Conventions for Argument Passing**

Argument Type	Register	Argument Register
char	A	
char, char	A, X	First char in A and second in X
int	X, A	MSB in X and LSB in A

**Table 10: #pragma Fastcall Conventions for Argument Passing , continued**

Argument Type	Register	Argument Register
Pointer	A, X	MSB in A and LSB in X
char, ...	A, X	First argument passed in A. Successive arguments are pointed to by X, where X is set up as a pointer to the remaining arguments. Typically, these arguments are stored on the stack
Int,...	X	X is set up as a pointer that points to the contiguous block of memory that stores the arguments. Typically, the arguments are stored on the stack.
All the others	X	Same as above

Arguments that are pushed on the stack are pushed from right to left.

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).

The following table reflects the set of #pragma fastcall conventions used for *return value* register assignments:

**Table 11: #pragma Fastcall Conventions for Return Value**

Return Type	Return Register	Comment
char	A	
int	X, A	
long	__r0..__r3	Delivered in the virtual registers
pointer	A, X	



## Section 6. Additional Considerations

**In this section you will learn** additional compiler options to leverage the functionality of your code/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.

The PSoC Designer C Compiler accepts the extension: inline assembly: `asm ("mov A, 5");` see [Inline Assembly](#).

### 6.2 Addressing Absolute Memory Locations

If your program needs to address Absolute Memory Locations there is one (verified) option:

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 as follows:

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

where `MyData` references memory location `0x200`.

## 6.3 Assembly Interface and Calling Conventions

Standard to PSoC Designer C Compiler and Assembler, an underscore 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. In 'C' however, the function would still be referenced as "foo()". The underscore is also applied to variable names.

## 6.4 Bit Twiddling

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

**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 1
```

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 compiler programs. See [Section 4. Compiler Basics](#) 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 can be referenced within the assembly string, as in the following example:

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

C variables have an implicit underscore at the beginning that needs to be used when using C variables from assembly.

Inline assembly may be used inside or outside a C function. The compiler indents each line of the inline assembly for readability. The PSoC Designer 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. You may get a warning on assembly statements that are outside of a function. You may ignore these warnings.

If you are referencing registers inline, be sure to include reference to `m8c.h`.

## 6.6 Interrupts

Interrupt handlers can be written in C. In order to employ interrupt handlers in C, you must first inform the compiler that the function is an interrupt handler. You do this by using 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.

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.

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
```

## 6.7 IO Registers

IO registers are specified using the following `#pragma`:

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

## 6.8 Long Jump/Call

The assembler/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:

### 6.9.1 Flash Memory Areas

- **interrupt vectors**: This area contains the interrupt vectors.
- **func\_lit**: Function table area. Each word in this area contains the address of a function entry.
- **lit**: This area contains integer and floating-point constants.
- **idata**: The initial values for the global data are stored in this area.
- **text**: This area contains program code.

### 6.9.2 Data Memory

- **data**: This is the data area containing 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**: This is the data area containing "uninitialized" C global variables. Per ANSI C definition, these variables will get initialized to zero at startup time.

The job of the linker is to collect areas of the same types from all the input object files and concatenate them together in the output file. For further information, see [Section 7. Linker](#).

## 6.10 Program and Data Memory Usage

### 6.10.1 Program Memory

The program memory, which is read only, 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 you, the programmer, to ensure that the stack does not exceed the high memory limit of 0xFF, or unexpected results will occur.

## 6.11 Program Memory as Related to Constant Data

The M8C 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;
```

*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. So, this 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 `strcpy` is `strcpy(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 undefined behaviors may result.**

## 6.12 Stack Architecture and Frame Layout

The stack must reside in page 0 and grows towards high memory. Most local variables and function parameters are allocated on the stack. A typical function stack frame looks as follows:

```

[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. Note that because the M8C limits the stack access to the first page only, 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 due to a total RAM space of 256 bytes.

## 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 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/boot.asm` can be modified to omit memory allocation for `_r4` to `_rZ` (10 bytes) by setting the equate `"C_LANGUAGE_SUPPORT"` to zero (0).

## 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 MicroSystems 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.

## Section 7. Linker

**In this section 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 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. In some detail, the linking steps involve:

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 request (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.
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.

For additional information about Linker, and specifying Linker settings, refer to the *PSoC Designer: Integrated Development Environment User Guide (Project Settings)*.

## 7.1.1 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 [Section 9. Command Line Compiler Overview](#)).

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. Using a custom AREA and the *custom.lkp* file allows you to set a specific starting address for this AREA.

For example, if you wish to create code in a separate code AREA that should be located in the upper 2k of the Flash, you could use this feature. For the sake of this example, let's call the custom code AREA 'BootLoader'. 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 Iools >> 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).

## Section 8. Librarian

**In this section 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 `\Program Files\Cypress MicroSystems\PSoC Designer\tools\libcm8c.a`.

There are times when you need to modify or create libraries. A command line tool called *libw.exe* is provided for this purpose. Note that a library file must have the `.a` extension. For further reference, see [Section 7. Linker](#).

#### 8.1.1 Compiling a File into a Library Module

Each library module is simply an object file. Therefore, to create a library module, you need to compile a source file into an object file. There are several ways that you can create a library.

One method is to create a brand new project. Add all the necessary source files that you wish to be added to your custom library, to this project. You then add a project-specific MAKE file action to add those project files to a custom library.

Let's take a closer look at this method, using an example. A blank project is created for any type of part, since we are only interested in using 'C' and/or assembly, the Application Editor, and the Debugger. 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, we use this to our advantage.

The following shows information from example `local.dep`:

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

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

obj/%.o : %.asm project.mk
ifeq ($(ECHO_COMMANDS),novice)
    echo $(call correct_path,$<)
endif

$(ASMCMD) $(INCLUDEFLAGS) $(DEFAULTASMFLAGS) $(ASM-
```



```

FLAGS) -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*. Let's look closely at this addition.

```

$(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 we see 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`). As you remember, 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/

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\PSoC-ToolsLib.a $(library_file)`).

Another goal 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 we will not redefine 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 **T**ools >> **O**ptions >> **B**uilder tab and click a check at “Use verbose build messages.”

Use the **P**roject >> **S**ettings, **L**inker 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 Module

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, so to create a new library, just give `ilibw` a new library file name.

### 8.1.4 Deleting a 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 crt8c.o;
```

## Section 9. Command Line Compiler Overview

**In this section you will learn** supported compiler command line options. This section covers the uses of the C compiler outside of PSoC Designer and contains information that is not required when using the compiler within PSoC Designer.

### 9.1 Compilation Process

Underneath the user friendly 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 to find out "what's under the hood."

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 from you within the IDE. However, it can be important to have an understanding of what happens "under the hood." Examine the following:

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 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.

.c files are C compiler source files and .asm files are assembly source files, respectively. The design philosophy for the IDE is to make it as easy to use as possible. The command line compiler, though, 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 the command line length limit on Windows® 95/98/NT..., 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 as 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. As you can see, 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.

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. For example:

**Table 12: Compiler Argument Prefixes**

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

### 9.3.1 Arguments Affecting the Driver

**Table 13: 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.

### 9.3.2 Preprocessor Arguments

**Table 14: 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.

### 9.3.3 Compiler Arguments

**Table 15: 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.

### 9.3.4 Linker Arguments

**Table 16: Linker Arguments**

Argument	Action
-L<dir>	Specify the library directory. Only one library directory (the last specified) will be used.
-O	Not currently implemented, no effect.
-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.

**Table 16: Linker Arguments, continued**

Argument	Action
-W	Turn on relocation wrapping. Note that you need to use the -WI prefix because the driver does not know of this option directly (i.e., -WI-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 address. 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:<addresses>	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 <i>libcm8c.a</i> . This can be used to change the behavior of a function in <i>libcm8c.a</i> since <i>libcm8c.a</i> is always linked in last. The "libname" is the library file name without the "lib" prefix and without the ".a" suffix. For example: -llpm8c "liblpm8c.a" using full printf -lfm8c "libfpm8c.a" using floating point printf

## Section 10. Code Compression

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

### 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.

In the Project Settings dialog box you can enable the PSoC Designer Code Compressor. To access the dialog box, click **Project >> Settings, Compiler tab**. The Code Compressor can be enabled or disabled for the open project using the check box adjacent to the “Enable Code Compression” field.

### 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 were 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 Where are the “Program Execution” Bytes?

The Code Compressor process, spawned from the linker, makes an assumption that “program execution” bytes are “tagged” by the “AREA” they reside in. This assumption adds a plethora 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 were free to create “data” tables in areas that the Code Compressor now expects only code. This is a project-compatibility issue discussed later.

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 of 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 What Can the PSoC Debugger Expect?

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 PSoC Designer Integration of the Code Compressor

### 10.3.1 boot.asm

*boot.asm* 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*.

Note that it is not a requirement for the *boot.asm* file 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 a “AREA text” before them to switch the area.

### 10.3.2 Text Area Requirement for Code Compressor

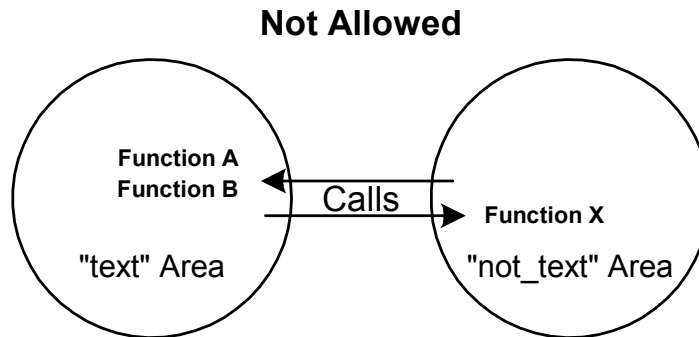
The text area should be the last (e.g., highest memory addresses) relocatable code area if your expectation is to reduce the entire program image. You cannot shrink the



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 not allowed or potentially problematic. Code areas created with the AREA directive, using a name other than “text,” are not compressed or “fixed up” (following compression). Therefore, if Function A in the “text” Area calls Function X in the “non\_text” Area, then Function X calls Function B where there would be “the potential” that the location of Function B changed. The call or jump generated in the code for Function X would go to the wrong location.

It is allowable for Function A to call a function in a “non\_text” Area and simply return.

For example, if Function A in the “text” Area calls Function X in the “non\_text” Area, then Function X calls to Function B could be invalid. 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 fixed up.

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 be not used directly otherwise.

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. These messages are listed and described below.

Messages for code compression appear following the Linker step of compilation/build.

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.

## 10.6 Up against the Wall?

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.

## 10.7 Additional Things to Consider When Using 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” `NOF` 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” `NOF` 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. Status Window Messages

Following is a complete list of preprocessor, preprocessor command line, compiler, compiler command line, assembler, assembler command line, and linker errors and warnings.

### 1.1 Preprocessor

Note that these errors and warnings are associated with C compiler errors and warnings.

**Table A.1: Preprocessor Errors/Warnings**

Error/Warning
# 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

**Table A.1: Preprocessor Errors/Warnings, continued**

Error/Warning
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
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

## 1.2 Preprocessor Command Line Errors

**Table A.2: Preprocessor Command Line Errors/Warnings**

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

## 1.3 C Compiler

**Table A.3: C Compiler Errors/Warnings**

Error/Warning
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

**Table A.3: C Compiler Errors/Warnings, continued**

Error/Warning
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
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>



**Table A.3: C Compiler Errors/Warnings, continued**

Error/Warning
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>
overflow in value for enumeration constant
redeclaration of <name> previously declared at <line>

**Table A.3: C Compiler Errors/Warnings, continued**

Error/Warning
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

## 1.4 Assembler

**Table A.4: Assembler Errors/Warnings**

Error/Warning
'[' 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

**Table A.4: Assembler Errors/Warnings, continued**

<b>Error/Warning</b>
target too far for instruction
too many include files
too many nested .if
undefined mnemonic <word>
undefined symbol
unknown operator
unmatched .else
unmatched .endif

## 1.5 Assembler Command Line Errors

**Table A.5: Assembler Command Line Errors/Warnings**

Error/Warning
cannot create output file %s\n
too many include paths

## 1.6 Linker

**Table A.6: Linker Errors/Warnings**

Error/Warning
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

### Symbols

#pragma Fastcall Conventions for Argument Passing [26](#)

#pragma Fastcall Conventions for Return Value [27](#)  
\* [19](#)

pragma Directives #pragma interrupt\_handler [19](#)

### A

Absolute Memory Locations [29](#)

Accessing M8C Features [29](#)

Accessing the Compiler [7](#)

API Software Library Functions [26](#)

Assembly Interface and Calling Conventions [30](#)

### B

Bit Twiddling [30](#)

### C

char LED [18](#)

Character Type Functions [21](#)

Code Compression

    Additional Things to Consider [51](#)

    Build Messages [50](#)

    Code Compressor and the AREA Directive [49](#)

    Code Compressor Process [47](#)

    PSoC Designer Integration of the Code Compressor [48](#)

    Theory of Operation [47](#)

    Up against the Wall? [50](#)

Compilation Process [43](#)

Compiler Arguments [44](#)

    Arguments Affecting the Driver [45](#)

    Compiler Argument Prefixes [44](#)

    Compiler Arguments [45](#)

    Linker Arguments [45](#)

    Preprocessor Arguments [45](#)

Convention for Restoring Internal Registers [36](#)

### D

Driver [43](#)

### E

Enabling the Compiler [7](#)

Errors/Warnings

    Assembler [59](#)

    Assembler Command Line Errors [61](#)

    C Compiler [55](#)

    Linker [61](#)

    Preprocessor [53](#)

    Preprocessor Command Line Errors [55](#)

Expressions, Supported [16](#)

### F

File Name Conventions [8](#), [12](#)

Files

    Library Descriptions [11](#)

    Startup File [11](#)

### I

Inline Assembly [31](#)

Interfacing C and Assembly (Fastcall) [26](#)

Interrupts [31](#)

IO Registers [32](#)

### L

Librarian [39](#)

    Adding or Replacing a Module [42](#)

    Compiling a File into a Library Module [39](#)

    Deleting a Module [42](#)

    Listing the Contents of a Library [42](#)

Library Functions [21](#)

Linker Operations [37](#)

    Customized Linker Actions [38](#)

Long Jump/Call [32](#)

### M

Mathematical Functions [24](#)

Memory Areas [33](#)

    Data Memory [33](#)

    Flash Memory Areas [33](#)

Menu Options [8](#)

## O

Operators, Supported Operators 14

## P

Pointers 17

pragma Directives 18

- #pragma abs\_address 18

- #pragma end\_abs\_address 19

- #pragma fastcall GetChar 18

- #pragma ioport LED 0x04 18

- #pragma nomac 19

- #pragma text 19

- #pragma usemac 19

Processing Directives (#'s) 17

- pragma Directives 18

- Preprocessor Directives 18

Product Upgrades 4

Program and Data Memory Usage 33

- Data Memory 34

- Program Memory 33

- Program Memory as Related to Constant Data 34

Purpose 3

## R

Re-entrancy 17

Reference Materials

- ANSI C Programming 4

- PSoC Designer 3

## S

Section 1. Introduction 3

Section 2. Accessing the Compiler 7

Section 3. Compiler Files 11

Section 4. Compiler Basics 13

Section 5. Functions 21

Section 6. Additional Considerations 29, 37

Section 7. Linker 37

Section 8. Librarian 39

Section 9. Command Line Compiler Overview 43

Section 10. Code Compression 47

Section Overview 4

Stack Architecture and Frame Layout 35

Statements, Supported 16

String Functions 21

Strings 35

Support 5

## T

Types, Supported Data Types 13

## V

Virtual Registers 36





## Document Revision History

<b>Document Title:</b> PSoC Designer: C Language Compiler User Guide <b>Document Number:</b> 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 CMS customers.
*A			UWE	Added "Convention for Restoring Internal Registers."
*B			HMT	--Added code-compression details. --Options using <i>custom.lkp</i> . --Reworked "Compiling a File into a Library Module." --Added typedef and fixed Inline Assembly example. --Added ftoa, updated address/links.