PSoC™ Designer:
C Language Compiler

User Guide
Revision 1.09

CMS10004A
Last Revised: May 30, 2001
Cypress MicroSystems, Inc.

Copyright Information

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 M8C but need a *quick* jump-start (Now we only have a minute and-a-half left.)

Overview 35 seconds

You have the M8C, PSoC™ Designer, and the C compiler… This guide provides:

- enabling and accessing procedures
- instructions for using the C compiler within PSoC Designer parameters
- references for the internal workings of the compiler.

Basics 30 seconds

After generating your device configuration, click the Application Editor icon in the toolbar to access the C compiler and 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 File >> New to add .c files to your project.

Quick Reference 15 seconds

Click a hyperlink to reference key material:

- Notation Standards
- Accessing/Enabling the Compiler
- Compiler Files
- Basics
- Functions
- Processing Directives (#'s)
- Librarian
- Status Window Messages

Bottom Line 10 seconds

The PSoC™ Designer C Compiler is an "extra" tool you can use to customize the functionality you desire into the M8C microprocessor.

 Time's up… Now get to work.

Documentation Conventions

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

Convention	Usage

Times New Roman Size 10-12	Displays input:
	`//-------------------------------------------------------------`
	`// Sample Code`
	`// Burn some cycles`
	`//-------------------------------------------------------------`
	
	`void main()`
	`{`
	`  char cOuter, cInner;`
	`  for(cOuter=0x20; cOuter>0; cOuter--)`
	`  {`
	`  for(cInner=0x7F; cInner>0; cInner--)`
	`  {`
	`  }`
	`  }`
	`  }`

`Courier Size 12`	Displays file locations: `c:\Program Files\Cypress MicroSystems\PSoC Designer\tools`
Italics	Displays file names: *projectname.rom*
[**Ctrl**] [**C**]	Displays keyboard commands: [**Enter**]
File >> Open	Displays menu paths: Edit >> Cut

Notation Standards

Following, are notation standards used throughout the PSoC Designer suite of product documentation.

Virtual Registers: Virtual registers _r0, _r1, _r2, _r3, _r4, _r5, _r6, _r7, _r8, _r9, _r10, _r11, _rX, _rY occupy 14 bytes of RAM. These locations are used for temporary data when using the C language compiler. Currently, these virtual memory register locations are allocated even if the source for the M8C in PSoC Designer is written only in assembly language.

Table of Contents

Section 1. Introduction

1.1. Purpose

The purpose of the *PSoC Designer: C Language Compiler User Guide* is reference for using a C language compiler within the parameters of PSoC Designer.

The PSoC Designer C Compiler compiles each .c source file to an M8C 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 M8C design functionality.

For comprehensive details on system use and assembly language, see:

- *PSoC Designer: Integrated Development Environment User Guide*
- *PSoC Designer: Assembly Language User Guide*

Together, these three user guides complete the PSoC Designer documentation suite.

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. Linker	Discusses C compiler linker options deployed within PSoC Designer.
Section 7. Librarian	Discusses C compiler library functions used within PSoC Designer.
Section 8. Command Line Overview	Overviews C compiler command line features that can be used strictly within the constraints of PSoC Designer.
Section 9. Tool References	Directs users to available reference resources.

1.3. Product Upgrades

Cypress MicroSystems provides scheduled upgrades and version enhancements, as requested by customers, for PSoC Designer *free of charge*. Compiler upgrades are included in your PSoC Designer Support Contract.

You can order PSoC Designer and Compiler upgrades from your distributor on CD-ROM or, better yet, download them directly from the Cypress MicroSystems web site at http://www.cypressmicro.com/.

Also provided at the web site are critical updates to system documentation. To stay current with system functionality you can find documentation updates under the Support hyperlink, again, at http://www.cypressmicro.com/.

Check the Cypress MicroSystems web site frequently for both product and documentation updates. As the M8C and PSoC Designer evolve, you can be sure that new features and enhancements will be added.

1.4. Support

Support for the C Language Compiler is bundled into a PSoC Designer Support Contract. For details, see the *PSoC Designer: Integrated Development Environment User Guide*.

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 Tools >> Customize menu option.
2. Select Compiler tab.
3. 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).

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 877.751.6100.

2.2. Accessing the Compiler

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

Application Editor

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

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.

2.3. Menu Options

The PSoC Designer Application Editor toolbar is shown below:

Following, is a description of the menu options available for use with the compiler:

Icon	Option	Menu	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
	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
	Repeat Replace			Repeats last replace
	Set Editor Options			Set options for editor
	Undo	Edit >> Undo	[**Ctrl**] [**Z**]	Undo last action
	Redo	Edit >> Redo	[**Ctrl**] [**Y**]	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 two primary code libraries used by PSoC Designer. The runtime library (named *libcm8c.a*) in the PSoC Designer application …`\tools` directory (e.g., …`\Program Files\Cypress Microsystems\PSoC Designer\tools`) contains many functions typically used in 'C' programming.

The second library contains an archive of the library source code added to a PSoC Designer project. Device Editor automatically adds the library source code to your project during the device configuration generation process (based on selected User Modules). However, other library objects can be 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 section 6 in *PSoC Designer: Integrated Development Environment User Guide*.

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.

4.1. Types

PSoC Designer C Compiler supports the following standard data types:

All types support the signed and unsigned type modifiers.

Type	Bytes	Description	Range
char	1	A single byte of memory that defines characters	unsigned 0…255 signed -128…127
Int	2	Used to define integer numbers	unsigned 0…65535 signed -32768…32767
short	2	Standard type specifying 2-byte integers	unsigned 0…65535 signed -32768…32767
long	4	Standard type specifying the largest integer entity	unsigned 0…4294967295 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	2	Used to define a list of aliases that represent integers.	0…65535

All floating-point operations are supported in the PSoC Designer C Compiler.

floats and doubles are in IEEE standard 32-bit format with 7-bit exponent and 23-bit mantissa.

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 parenthesis where appropriate to prevent ambiguity.

P.	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
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 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

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

4.7.2. pragma Directives

#pragma	Description
#pragma ioport LED:0x04; char LED;	Defines a variable that occupies a region in I/O space. This variable can then be used in I/O reads and writes. The #pragma ioport must precede a variable declaration defining the variable type used in the pragma
#pragma fastcall GetChar	Provides an optimized mechanism for argument passing. This #pragma is used only for assembly functions called from "C."

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 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 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:

All strings are null terminated strings.

Function	Prototype	Description	Header
itoa	void itoa(char *string, int value, int radix)	Converts the integer value into a string representation of the specified radix.	*stdlib.h*
strcpy	char *strcpy(char *s1, char *s2) char *cstrcpy(char *s1, const char *s2)	Copies "s2" into "s1." Returns s1.	*string.h*
strcmp	int strcmp(char *s1, char *s2) int cstrcmp(char *s1, const char *s2)	Compares two strings. Returns: 0 = Strings are equal >0 = First different element in "s1" is greater than the corresponding element in "s2," else <0.	*string.h*
strlen	size_t strlen(char *s) size_t cstrlen(const char *s)	Returns the length of "s." The terminating null is not counted.	*string.h*
strcat	char *strcat(char *s1, const char *s2) char *cstrcat(char *s1, const char *s2)	Concatenates "s2" onto "s1." Returns s1.	*string.h*

You can also view these functions at a command prompt window by typing:

```
…:\Program Files\Cypress MicroSystems\PSoC Designer\tools>
ilibw –t libcm8c.a
```

5.2. Interfacing C and Assembly

To optimize argument passing and return value activities between the PSoC Designer C Compiler and Assembler, employ 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:

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

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 section **6.5 Inline Assembly**.

6.2. Addressing Absolute Memory Locations

Your program may need to address absolute memory locations. Use inline assembly or a separate assembler file to declare data that are located in specific memory addresses, and then follow the **Assembly Interface and Calling Conventions** as described ahead.

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 M8C 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 parenthesis is 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. Complier 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.) An example for inline assembly is:

```
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 asm keywords.

'C' variables can be referenced within the assembly string. See the following example as valid:

```
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 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 asm statements that are outside of a function. You may ignore these warnings.

6.6. IO Registers

IO registers are specified using the following #pragma:

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

6.7. Interrupts

All interrupt-level functions must be written in assembly language. Interrupt 'C' functions are not supported.

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:

- **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.

6.9.1. 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]
        …
        [virtual registers]
[low memory]
```

It is up to you, the programmer, to ensure that the stack does not leak into the variable section. Otherwise, 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 a M8C 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 *dst, const char *src), 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]
        [local variables and other compiler generated temporaries]
X:      [old X]
        [return address]
        [incoming arguments]
        …
[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 into program memory. Effectively, the type for a literal string is const char * and you must ensure that function parameters take the appropriate argument type.

6.14. Virtual Registers

Virtual registers _r0, _r1, _r2, _r3, _r4, _r5, _r6, _r7, _r8, _r9, _r10, _r11, _rX, _rY occupy 14 bytes of RAM and are used because the M8C only has a single 8-bit accumulator. These locations are for temporary data storage when using the compiler. The virtual registers are allocated on the bottom of data memory. Currently, the virtual memory register locations are allocated even if the source for the M8C is written in assembly language.

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.

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 located in the PSoC Designer working directory of `c:\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 *ilibw.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. To do this, open the file in the IDE and invoke the File >> Compile File To Object command.

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 work 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 crtm8c.o    ; delete
```

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 a M8C 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:

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

9.3.1. 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

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

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

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.
-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 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:<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. For example: -llpm8c "liblpm8c.a" using full printf -lfm8c "libfpm8c.a" using floating point printf

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.

Preprocessor

Note that these errors and warnings are associated with C Compiler errors and 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
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

(Preprocessor cont.)

Unknown preprocessor control line
Unterminated #if/#ifdef/#ifndef
Unterminated string or char const

Preprocessor Command Line Errors

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

C Compiler

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
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

(C Compiler cont.)

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>
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

(C Compiler cont.)

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

Assembler

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

Assembler Command Line Errors

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

Linker

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