MPLAB™ C17 User's Guide

NOTES:

MPLAB-C17 USER'S GUIDE

Table of Contents

Chapter 1. About MPLAB-C17

Chapter 2. Getting Started with MPLAB-C17

Chapter 3. MPLAB-C17 Fundamentals

Table of Contents

Table of Contents

Appendix A. Porting Code from MPLAB-C to MPLAB-C17

Appendix B. ASCII Character Set

Table of Contents

Chapter 1. About MPLAB-C17

Introduction

This chapter describes the MPLAB-C17 ANSI-based C Compiler and suggests recommended reading.

Highlights

This chapter covers the following topics:

- **ANSI Compatibility**
- **System Requirements**
- **About this Guide**
- **Recommended Reading**
- **Warranty Registration**
- **Customer Support**

ANSI Compatibility

MPLAB-C17 is a free-standing ANSI C implementation except where specifically noted elsewhere in this User's Guide. The compiler deviates from the ANSI standard only where the standard and efficient PICmicro MCU support conflict.

System Requirements

MPLAB-C17 requires:

- PC compatible machine: 386 or higher.
- MS-DOS/PC-DOS version 5.0 or greater or Windows 95 or Windows NT

Since MPLAB-C17 is integrated with the MPLAB Integrated Development Environment, it is recommended that you install the current version of MPLAB software (MPLAB.EXE) on a host computer having the additional minimum configuration:

- VGA required. Super VGA recommended
- Microsoft® Windows® version 3.1 or greater operating in 386 enhanced mode
- 4 MB of Memory, 16 MB Recommended
- 8 MB of Hard Disk Space, 20 MB Recommended
- Mouse or other pointing device

About this Guide

This document describes how to use MPLAB-C17 to write C code for PICmicro microcontroller applications. For a detailed discussion about basic MPLAB functions, refer to the MPLAB User’s Guide, Document Number DS51025.

The User’s Guide layout is as follows:

MPLAB-C17 Preview - describes the benefits of using MPLAB-C17 to write C code for PICmicro microcontroller applications.

Chapter 1: About MPLAB-C17 - describes MPLAB-C17 ANSI-based C Compiler and suggests recommended reading.

Chapter 2: Getting Started with MPLAB-C17 - discusses how to use MPLAB-C17 with the MPLAB IDE and as a stand-alone compiler.

Chapter 3: MPLAB-C17 Fundamentals - describes the MPLAB-C17 programming language including functions, statements, operators, variables, and other elements.

Chapter 4: MPLAB-C17 and PICmicro Programming -

Chapter 5: Using MPLAB-C17 with Other Tools - describes how to use MPLAB-C17 with Microchip development tools.

Chapter 6: Mixing C with Assembly Language Modules - provides guidelines to using C with MPASM assembly language modules.

Chapter 7: ANSI Implementation Issues - details MPLAB-C17 specific parameters described as implementation defined in the ANSI standard.

Chapter 8: Libraries - includes Hardware Peripheral, Software Peripheral and General Software libraries.

Appendix A: Migrating from MPLAB-C to MPLAB-C17 - provides guidelines for migrating from MPLAB-C to MPLAB-C17.

Appendix B: ASCII Character Set - contains the ASCII character set.

Appendix C: Detailed MPLAB-C17 Examples - gives examples of actual working source code with comments included.

Appendix D: PIC17CXXX Instruction Set - gives the instruction set for the PIC17CXXX device family.

Appendix E: On-Line Support - Information on Microchip’s electronic support services.

Appendix F: References - gives references that may be helpful in programming with MPLAB-C17.

Worldwide Sales and Service - gives the address, telephone and fax number for Microchip Technology Inc. sales and service locations throughout the world.

Conventions Used in this Guide

This User's Guide follows these documentation conventions:

Table 1: Documentation Conventions

<table>
<tr><th>Character</th><th>Represents</th></tr>
<tr><td>Angle Brackets (< >)</td><td>Delimiters for special keys or values: <TAB>, <ESC>, <symbol> etc.</td></tr>
<tr><td>Pipe Character (|)</td><td>Choice of mutually exclusive arguments; an OR selection</td></tr>
<tr><td>Square Brackets ([ ])</td><td>Optional argument (unless specified otherwise)</td></tr>
<tr><td>Courier Font</td><td>User entered code or sample code</td></tr>
<tr><td>Underlined, Italics Text with Right Arrow ></td><td>Defines a menu selection from the menu bar: *File > Save*</td></tr>
<tr><td>0xnnn</td><td>0xnnn represents a hexadecimal number where n is a hexadecimal digit</td></tr>
<tr><td>In-text **Bold** Characters</td><td>Designates a button such as **OK**</td></tr>
</table>

Recommended Reading

README.MCC For the latest information on using MPLAB-C17, read the README.MCC file (an ASCII text file) included with the MPLAB-C17 software. README.MCC contains update information that may not be included in the *MPLAB-C17 User's Guide.*

PICmicro Microcontroller Data Book Contains comprehensive data sheets for Microchip PICmicro microcontroller devices available at print time. *Document Number DS00158, Microchip Technology Inc., Chandler, AZ.*

Embedded Control Handbook Contains a wealth of information about microcontroller applications. Document Number DS00092, Microchip Technology Inc., Chandler, AZ. The application notes described in this User's Guide are also available from the Microchip Internet Home Page. See *Appendix E: On Line Support, for more information.*

MPLAB User's Guide Comprehensive guide that describes installation and features of Microchip's MPLAB Integrated Development Environment, as well as the editor and simulator functions in the MPLAB environment. *Document Number DS30421, Microchip Technology Inc., Chandler AZ.*

MPASM User's Guide with MPLINK & MPLIB Describes how to use Microchip Universal PICmicro Microcontroller Assembler (MPASM), the Linker and Librarian (MPLINK & MPLIB). *Document Number DS33014, Microchip Technology Inc., Chandler, AZ.*

Midrange Architectural and Peripheral Module Reference

PIC17C4X Data Sheet *Document Number DS30412, Microchip Technology Inc., Chandler, AZ.*

PIC17C75X Data Sheet *Document Number DS30264, Microchip Technology Inc., Chandler, AZ.*

All of the above documents are available from your local sales office or your Microchip Field Application Engineer (FAE).

This User's Guide assumes that you are familiar with Microsoft Windows 3.x software systems. Many excellent references exist for this software program, and should be consulted for general operation of Windows.

Warranty Registration

Sending in your Warranty Registration Card ensures that you receive new product updates and notification of interim software releases that may become available.

Customer Support

Microchip endeavors to provide the best service and responsiveness possible to its customers. Technical support questions should first be directed to your distributor and representative, local sales office, Field Application Engineer (FAE), or Corporate Applications Engineer (CAE).

The Microchip Internet Home Page can provide you with technical information, application notes, and promotional news on Microchip products and technology. The Microchip Web address is http://www.microchip.com.

Chapter 2. Getting Started with MPLAB-C17

Introduction

This chapter discusses how to use MPLAB-C17 as a stand-alone compiler or as a fully integrated tool in the MPLAB Integrated Development Environment.

Highlights

Getting Started with MPLAB-C17 includes:

- **Installing MPLAB-C17**
- **MPLAB-C17 Project**
- **MPLAB-C17 Command Line Interface**
- **Using Multiple Files in a Project**
- Making projects in the MPLAB Integrated Development Environment

Installing MPLAB-C17

Windows Environment

To install MPLAB-C17, enter Windows, run the file MCCxxx.EXE on the CD-ROM, and follow the prompts. The install program creates a directory tree with five subdirectories BIN, H, LIB, SRC and examples. Note that MPLAB-C17 will create an environment variable, MCC_INCLUDE in your AUTOEXEC.BAT file. The MCC_INCLUDE environment variable specifies the directories to search for included files. For more information, refer to the #include directive. The install program will also add the compiler BIN directory to your PATH so you can run the compiler from any other directory.

DOS Environment

To install MPLAB-C17 in a DOS environment, run the MCCxxx.EXE file on the CD-ROM, and follow the prompts.

Command Line Interface

MPLAB-C17 can be invoked directly from the command line, independent of the MPLAB IED. The command line interface of MPLAB-C17 is as follows:

`MCC17 [options]` *`filename`*

where

`filename`	is the name of the file being compiled, and
`options`	is zero or more command line options.

For example, if the file TEST.C exists in the current directory, it can be compiled with the following command:

```
MCC17 -P=17C756 TEST.C
```

When no command line parameters are specified, or with '-?' or '-h', a help screen is displayed describing the command line usage and options.

Options to MPLABC-17 can be specified with either '/' or '-'.

Table 2:

Option	Default	Description
?,H	N/A	Help screen
I*path*	N/A	Add the semi-colon delimited path, path, to the search path for include files.
FO=*filename*	N/A	Use *filename* as the name of the output object file
FE=*filename*	N/A	Use *filename* as the name of the output error file
O	N/A	Optimize for smallest code Equivalent to: -Or -Oc -Op
Oc[+|-]	Enabled	With this optimization on, the compiler will intelligently determine the level of stack support to include for each function.
Or[+|-]	Enabled	With this optimization on, the compiler will run an optimization pass to remove extraneous bank select and **MOVLW** instructions.
Ol[+|-]	Enabled	When this optimization is on, the default storage class for local variables and function parameters is `'static'`.
Op[+|-]	Disable	When this optimization in on, `far` pointers to RAM are assumed to not point to SFRs. This simplifies setting the bank for access.
M{s|m|c|l}	S	Select the memory model s:small model (near ram, near rom) m:medium model (near ram, far rom) c:compact model (far ram, near rom) l: large model (far ram, far rom)
P=*processor*	17C44	Select to compile for the PIC17CXX processor
Dmacro[=text]	N/A	Define a macro. Equivalent to placing the following at the head of the file: #define macro text

Table 2:

Option	Default	Description
W{1|2|3}	2	Set compiler message level. 1 display errors only 2 display errors and warnings 3 display errors, warnings, and messages
NW*n*	N/A	Suppress message n, where n is the message number. Error messages cannot be suppressed.
Q	N/A	Suppress the sign-on banner

Creating Your First MPLAB-C17 Project

> **Example Files**
> There are a number of examples in the folder MCC\EXAMPLES. Execution of the batch file should compile each example after MPLAB-C17 is set up. You can use these files as “cookbooks” to begin development of your application.

This section demonstrates how to compile and link a few small projects. It starts with a simple project with only one C source file. For the purpose of this discussion it is assumed the compiler is installed on your C: drive in a directory called MCC. Therefore the following will apply:

Include directory: `C:\MCC\H`

Library directory: `C:\MCC\LIB`

Executable directory: `C:\MCC\BIN`

The include directory is where the compiler stores all its system header files. The MCC_INCLUDE environment variable should point to that directory (from the DOS command prompt, type "set" to check this). The library directory is where the libraries and startup code files reside. The executable directory is where the compiler programs are located.

The following is a very simple program that adds two numbers.

1. Type the following program and save it as EX1.C in a directory called (for example) C:\PROJ0.

```
#include <P17C756.H>
unsigned char Add(unsigned char a, unsigned char b);
char x, y, z;
void main()
{
  x = 2;
  y = 5;
  z = Add(x,y);
}
unsigned char Add(unsigned char a, unsigned char b)
{ return a+b; }
```


The first line of the program includes the header file P17C756.H which provides definitions for all special function registers on that part. For more information on header files see the section "MPLAB-C17 Specifics" in chapter 4.

2. Compile the program by typing the following command:

   ```
   mcc ex1.c /P=17c756
   ```

   This tells the compiler to compile the program for the PIC17C756. The compiler generates two files by default. EX1.O is the object file that the linker will use to generate (among other files) the executable (.HEX) file to program your PICmicro. The second file is EX1.ERR which is the error file containing any error messages and/or warnings that the compiler generates during compilation. These messages are also displayed on the screen. The EX1 program will produce a warning since the function main() was called without a prototype. To suppress the warning add the /NW1200 switch on the command line.

3. The C object file must be linked with the compiler startup code to work MPLINK. When using MPLINK, use the linker script for the desired target processor. Copy the linker script from the MPLAB directory into your project directory and customize as needed. Copy the script as follows:

   ```
   copy c:\mplab\17c756.lkr
   ```

   Now the linker script is in the current directory.

4. The startup code is described in detail in the section "MPLAB-C17 Specifics" in chapter 4. Link the startup code file, COS17.0, with the project. Link the processor definition file `P17C756.O` to reference any special function registers and `idata17.0`, which is required for initialized data. Here is the linker command to produce the executable:

   ```
   mplink -K . c0s17.o idata17.o p17c756.o ex1.o -L
     c:\mcc\lib -m exl.map -o exl.out 17c756.lkr
   ```

   (Although shown on two lines here, this should be on one line when executed.) The first option tells the linker that the linker script is in the current directory. The object files to be linked together are `c0s17.o`, `idata17.o`, `p17c756.o`, and `ex1.o`. The library directory where the startup object files are located is specified after the `-L` directive. A map file called '`ex1.map`' is generated with the `-m` directive. The `-o` directive tells the linker to generate an executable called `ex1.cof`. The linker script to use for this link session is `17c756.lkr`.

   The linker produces the files `ex1.out`, `ex1.cod`, and `ex1.hex`. The .COD file is required by MPLAB for source-level debugging. The .HEX file is used by device programmers such as PRO MATE and PICSTART Plus to program a PICmicro MCU device.

Using Multiple Files in a Project

Move the Add() function into a file called Add.C to demonstrate the use of multiple files in a project.

```
/* EX1.C */
#include <P17C756.H>

unsigned char Add(unsigned char a, unsigned char b);

char x, y, z;

void main()
{
   x = 2;
   y = 5;
   z = Add(x,y);
}
```

```
/* ADD.C */
#include <p17c756.h>

unsigned char Add(unsigned char a, unsigned char b)
{ return a+b; }
```

To compile these two files, the command lines would be:

```
mcc ex1.c /P=17c756
mcc add.c /P=17c756
```

Then link the resulting object files with the startup code as follows:

```
mplink -K . c0s17.o idata17.o p17c756.o ex1.o add.o -L
   c:\mcc\lib -m ex1.map -o ex1.out 17c756.lkr
```

(This should be entered on one line.) This will produce the same files as before.

Making Projects in the MPLAB Integrated Development Environment

Introduction

The project manager in MPLAB v3.40 has been extended to support multiple files. Previously established projects from MPLAB v3.31 and earlier will be converted automatically by MPLAB v3.40 when they are opened. Converted projects cannot be re-opened from previous versions of MPLAB.

Read the on-line help with MPLAB for further information on making projects with MPASM or other compilers.

Highlights

In this tutorial you will learn these functions of MPLAB Projects:

- Making a Project with MPLAB-C17
- New Project
- Set Language Tool Options
- Add Node to Project
- Make Project
- Install Language Tool
- Project Window
- Summary of Setting Up Projects

Making a Project with MPLAB-C17

This tutorial will show you how to use MPLAB-C17 with projects in MPLAB to build applications.

Set Development Mode

Set *Options>Development Mode* to MPLAB-SIM simulator and select the 17C756 PICmicro for this example.

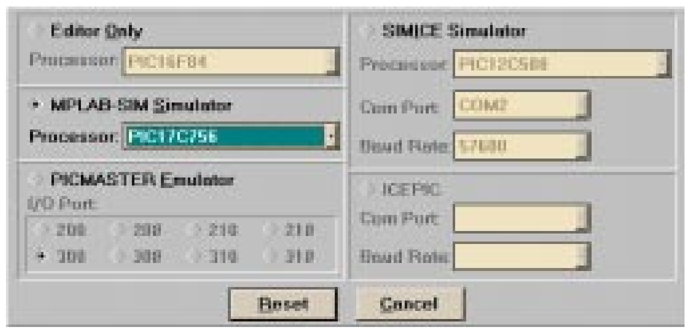


Figure 2.1:

Install MPLAB-17 Language Tool

Make certain that MPLAB-C17 is installed correctly in MPLAB. The “Install Language Tool” dialog should look like this:

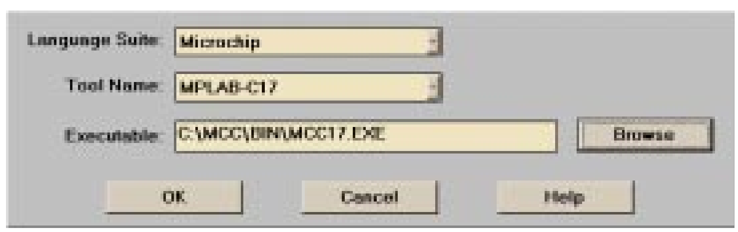


Figure 2.2:

If the executable is not shown in the window, use the Browse button to point to MCC17.EXE on your system.

New Project

Select *Project>New Project* and select a directory for a new project, then type in its name. Name it AD.PJT in the MCC\EXAMPLES\AD directory.

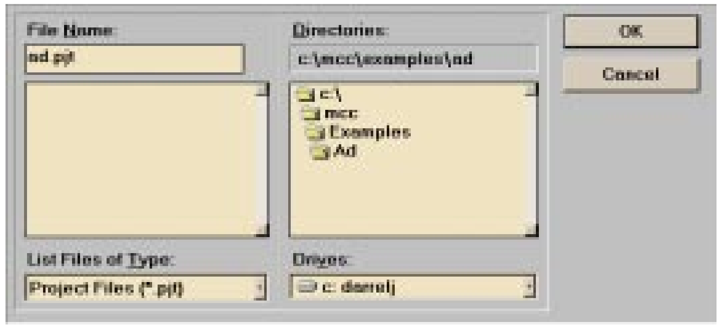
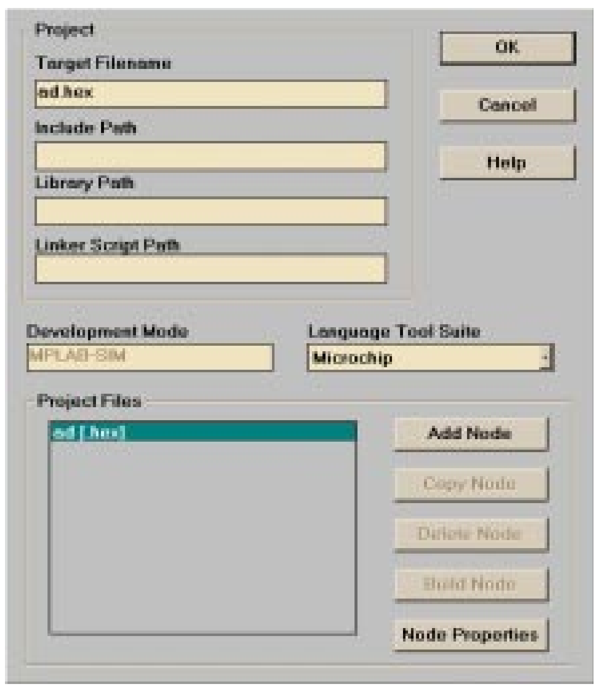


Figure 2.3:

After setting the project name, the Edit Project dialog will be shown.



Set Project Options

Select the name of the project in the “Project Files” dialog of the New Project Dialog and press “Node Properties.”

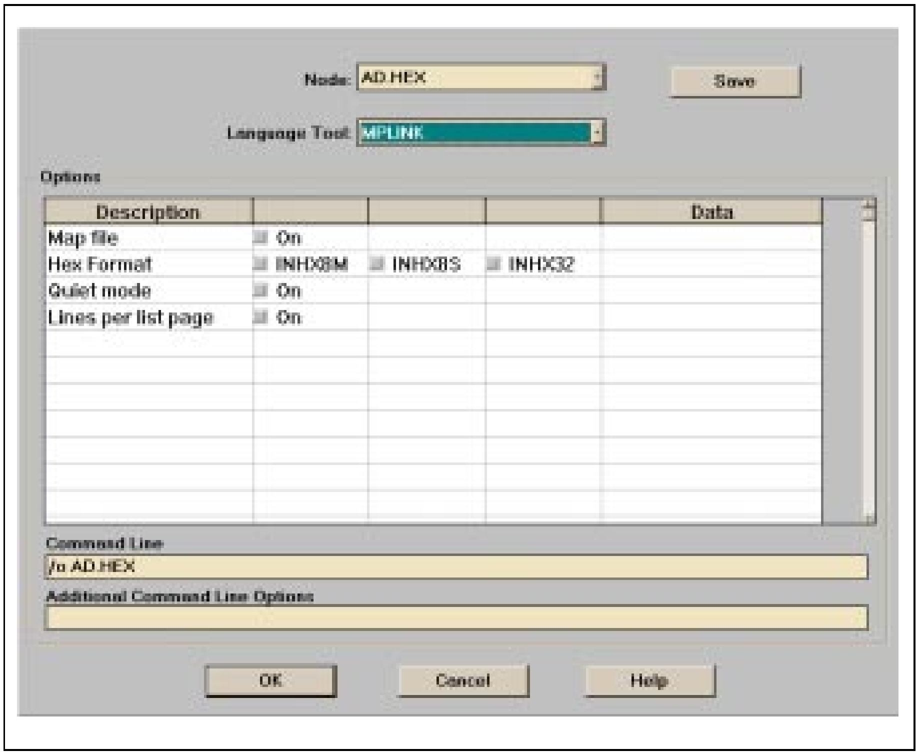


Figure 2.4:

Set the language tool to “MPLINK.”

Add First Source File

To determine which nodes to set up from this tutorial, look at AD.BAT. This is the batch file that will compile this example in DOS and is in the \MCC\EXAMPLES\AD directory. Use this data to add all required nodes. Here is a listing of the batch file:

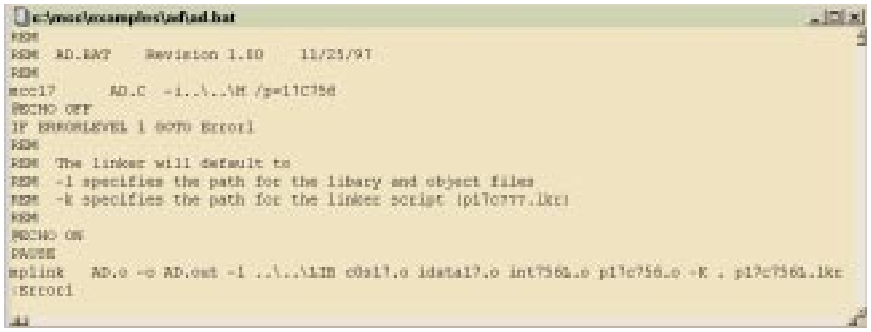


Figure 2.5:

The nodes required are AD.C, which must be compiled, and the following object files which need to be linked: C0S17.O, IDATA17.O, INT756L.O, P17C756.O and the linker script, P17C756L.LKR.

You can return to setting up the project from the *Project>Edit Project* menu selection.

Select "Add Node" from the Edit Project Dialog. Add the source file, AD.C from the \MCC\EXAMPLES\AD directory.

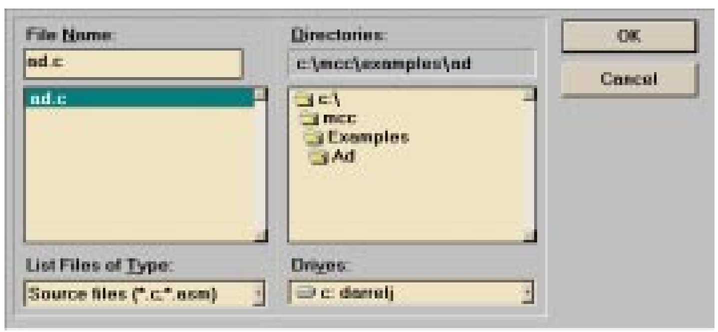
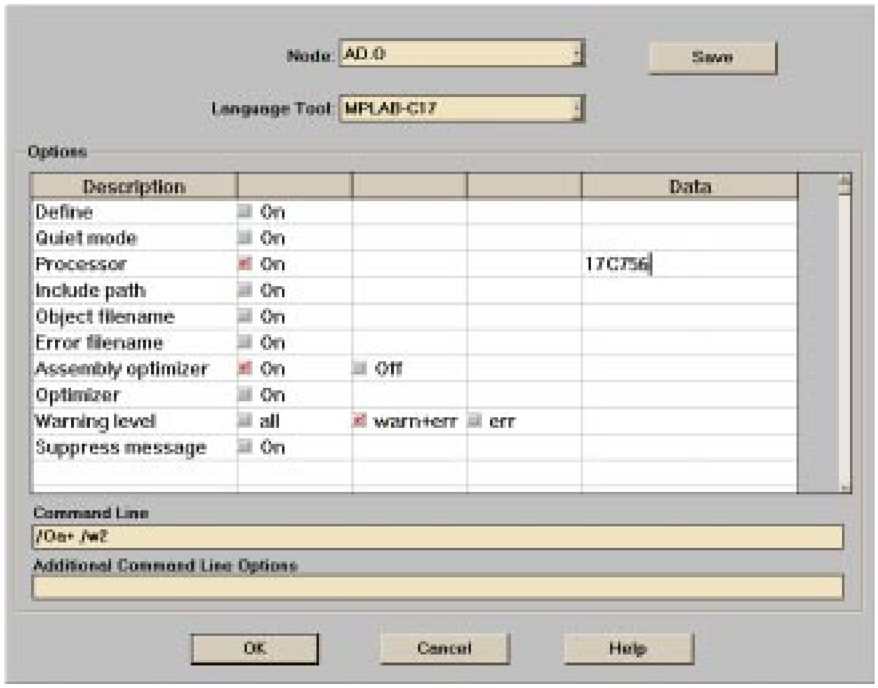


Figure 2.6:

When the file name is shown and selected in the Add Node dialog, press "Node Properties."

Set up this dialog this way:

- Set the "Language Tool" to MPLAB-C17.
- Check the "Processor" check box.
- Go to the "Data" column and enter "17C756."



Note: "Object filename" is set to "AD.O" automatically.

Adding Pre-Compiled Object Files

Use the "Add Node" button from the Edit Project dialog to add the precompiled object files from the MPLAB-C17 library in \MCC\LIB. Add C0S17.O as the first node. Options cannot be set on precompiled object files.

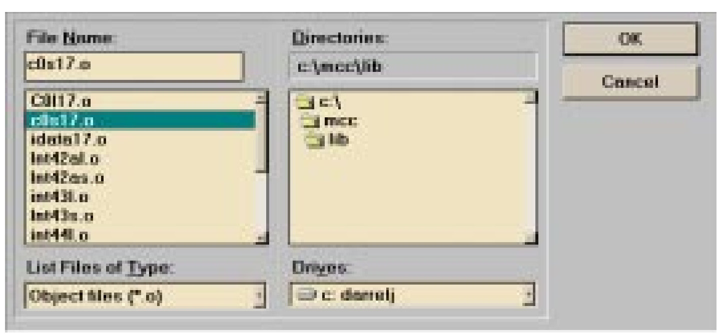


Figure 2.7:

Add the rest of the nodes that were listed in the batch file, IDATA17.O, INT756L.O and P17C756.O using the Add Node button from the Edit Project Dialog.

Select Linker Script

Select a linker script and add it as a node. Use the linker script in the MCC\EXAMPLES\AD directory. Options can not be set on a linker script.

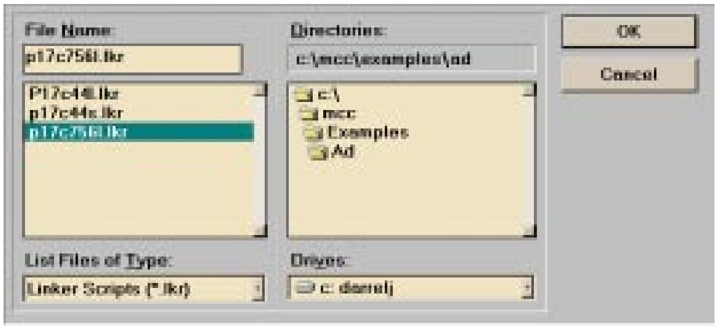


Figure 2.8:

Press **OK** on the New Project Dialog.

The Edit Project window should now look like this:

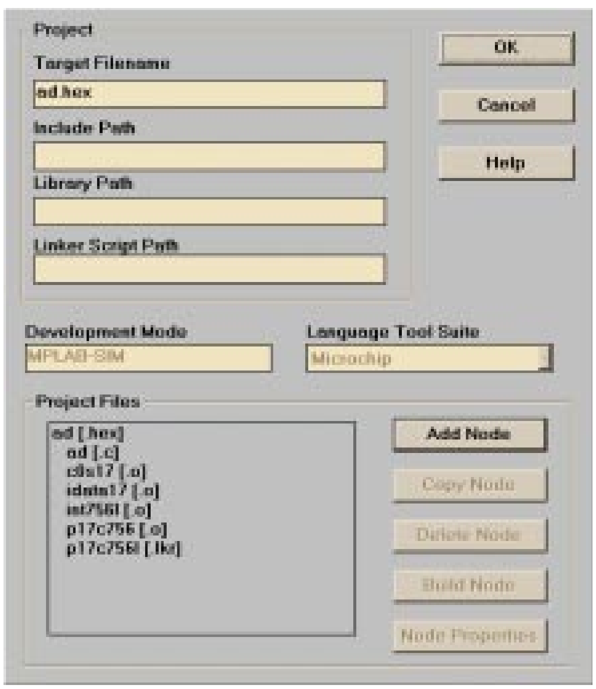


Figure 2.9:

Make Project

Select *Project>Make Project* from the menu to see the command lines sent to MPLAB-C17 and MPLINK to build the application. It should look like this:

Figure 2.10:

Troubleshooting

If this did not work, check these items:

Select *Project>Install Language Tool...* and check that MPLAB-C17 and MPLINK are pointed to the MCC17.EXE and MPLINK.EXE executables.

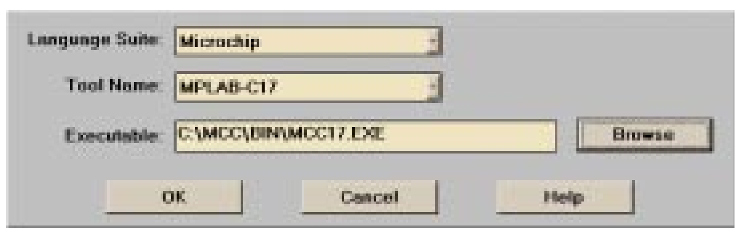


Figure 2.11:

Figure 2.12:

Project Window

Open the *Window>Project Window*. It should look like this:

Figure 2.13:

Summary of Setting Up Projects

Here is a quick list of the steps to set up a new project as described above:

- Create new project with *Project>NewProject*
- Set project Node Properties to MPLINK
- Add Source files, setting language tool to MPLAB-C17 or MPASM
- Set Processor in Node Properties of each source file
- For MPASM source files, set to generate object file
- Add Pre-Compiled Nodes (.O files and .LIB files)
- Add Linker Script

Chapter 3. MPLAB-C17 Fundamentals

Introduction

MPLAB-C17 Fundamentals describes the C programming language, including functions, statements, expressions and declarations.

Highlights

This chapter covers the following topics:

- **C Fundamentals**
- **Preprocessor Directives**
- **Variables**
- **Arrays and Strings**
- **Pointers**
- **Structures and Unions**
- **Functions**
- **Operators**
- **Program Control Statements**

C Fundamentals

This section is intended as a reference for programmers with a basic understanding of C programming. Various points are highlighted for users who are not experienced with programming microcontrollers in C, and deviations from ANSI C are described.

Programmers who are unfamiliar with the C language can refer to Appendix E for a list of C programming references.

This section discusses the following topics:

- Components of an MPLAB-C17 Program
- Comments
- C Keywords
- Constants

Components of an MPLAB-C17 Program

A C program is a collection of declarations, statements, comments, and preprocessor directives that typically do the following:

- Declare data structures

- Allocate data space
- Evaluate expressions
- Perform program control operations
- Control PICmicro MCU peripherals

The following is a shell for an MPLAB-C17 source file:

```
#include <P17CXX.h>
void main()
{
    /* User source code here */
}
```

The first line includes the processor definition file. This file defines processor-specific information such as special function registers. Any user-defined function prototypes should follow this line. Finally, the function main is defined, with the appropriate source code between the braces.

Comments

Description

P17CXX.H includes proper processor specific header file based on the processor selected on the command line.

Comments are used to document the meaning and operation of the source code. The compiler ignores all comments. A comment can be placed anywhere in a program where white space can occur. Comments can be many lines long and may also be used to temporarily remove a line of code. Comments cannot be nested.

Syntax

'/*' begins a comment, and '*/' terminates a comment.

'//' comments to the end of the line

Example

```
/* This is a block comment.
   It can have multiple lines
   between the comment delimiters.
*/
// This is a C++ style comment
```


C Keywords

Description

The ANSI C standard defines 32 keywords for use in the C language. Typically, C compilers add keywords that take advantage of the processor's architecture. The following table shows the ANSI C and the MPLAB-C17 keywords.

Additional MPLAB-C17 keywords are shown in bold.

_asm	double*	long*	struct
_endasm	else	**near**	switch
auto	enum	**ram**	typedef
break	extern	register	union
case	**far**	return	unsigned
char	float*	**rom**	void
const	for	short	volatile
continue	goto	signed	while
default	if	sizeof	
do	int	static	

*float, double, and long are not supported by MPLAB-C17

Constants

Description

A constant in C is any literal number, single character, or character string.

Syntax

Numeric Constants

By default, literal numbers are evaluated in decimal. Hexadecimal values can be specified by preceding the number by `0x`. Octal values can be specified by preceding the number by `0` (zero). Binary values can be specified by preceding the number by `0b`.

Character Constants

Character constants are denoted by a single character enclosed by single quotes. ANSI C escape sequences, as shown by the following table, are treated as a single character.

Table 3.1: ANSI C Escape Sequences

Escape Character	Description	Hex Value
\a	Bell (alert) character	07
\b	Backspace character	08
\f	Form feed character	0C
\n	New line character	0A
\r	Carriage return character	0D
\t	Horizontal tab character	09
\v	Vertical tab character	0B
\\	Backslash	5C
\?	Question mark character	3F
\’	Single quote (apostrophe)	27
\"	Double quote character	22
\0OO	Octal number (zero, Octal digit, Octal digit)	
\xHH	Hexadecimal number	

String Constants

String constants are denoted by zero or more characters (including ANSI C escape sequences) enclosed in double quotes. A string constant has an implied null (zero) value after the last character.

Example

Numeric Constants

```
         // Each of the following evaluates to a
         // decimal twelve
12       // Decimal
0x0C     // Hexadecimal
014      // Octal
0b1100   // Binary
```

Character Constants

```
’a’      // Lowercase ’a’
’\n’     // New Line
’\0’     // Zero or null character
```

String Constants

"Hello World"

Preprocessor Directives

Preprocessor directives give instructions on how to compile the source code. Preprocessor directives generally do not translate directly into executable code.

Preprocessor directives begin with the '#' character. This section discusses the following preprocessor directives:

- `#define`
- `#else`
- `#elif`
- `#endif`
- `#error`
- `#if`
- `#ifdef`
- `#ifndef`
- `#include`
- `#line`
- `#pragma`
- `#undef`

#define

Description

The #define directive defines string constants that are substituted into a source line before the source line is evaluated. These can improve source code readability and maintainability. Common uses are to define constants that are used in many places and provide short cuts to more complex expressions.

Syntax

define-directive:

```
    #define identifier pp-token-list new-line
    #define identifier lparen parameter-list ) pp-token-list
            new-line
    #define identifier lparen ) pp-token-list new-line
```

lparen:

```
    ([1]
```

[1] No whitespace may separate *lparen* and the macro name.

parameter-list:
    identifier
    parameter-list , identifier

Example

```
#define MAX_COUNT 100
#define VERSION "v1.0"
#define PERIMETER( x, y ) 2*x + 2*y
#define INCREMENTALL x++;\
               y++;  \
               z++;
```

#else

Description

Refer to #if, #ifdef, and #ifndef for a description of the #else directive.

#elif

Description

Refer to #if, #ifdef, and #ifndef for a description of the #elif directive.

#endif

Description

Refer to #if, #ifdef, and #ifndef for a description of the #endif directive.

#error

Description

The #error directive generates a user-defined error message at compile time. One use of #error is to detect cases where the source code generates constants that are out of range. No code is generated as a result of using this directive.

Syntax

error-directive:
    #error *pp-token-list new-line*

Example

```
#define MAX_COUNT 100
#define ELEMENT_SIZE 3
#if (MAX_COUNT * ELEMENT_SIZE) > 256
   #error "Data size too large."
#endif
```

#if

Description

The #if directive is useful for conditionally compiling code based on the evaluation of an expression. #if must be terminated by #endif. The #elif is used to test a new expression. The directive #else is also available to provide an alternative compilation. The defined() operator acts similarly to #ifdef when combined with #if.

Syntax

if-directive:

  #if *constant-expression new-line*

Example

```
#define MAX_COUNT 100
#define ELEMENT_SIZE 3
#if defined(MAX_COUNT) && defined(ELEMENT_SIZE)
#if (MAX_COUNT * ELEMENT_SIZE) > 256
   #error "Data size too large."
#else
   #define DATA_SIZE MAX_COUNT * ELEMENT_SIZE
#endif
#endif
```

#ifdef

Description

The #ifdef directive is similar to the #if directive, except that instead of evaluating an expression, it checks to see if the specified symbol has been defined. Like the #if directive, #ifdef must be terminated by #endif, and can optionally be used with #else.

Syntax

ifdef-directive:

#ifdef *identifier new-line*

Example

```
#ifdef DEBUG
   Count = MAX_COUNT;
#endif
```

#ifndef

Description

The #ifndef directive is similar to the #ifdef directive, except that it checks to see if the specified symbol has not been defined. Like the #if directive, #ifndef must be terminated by #endif, and can optionally be used with #else.

Syntax

ifndef-directive:

#ifndef *identifier new-line*

Example

```
#ifndef  DEBUG
#define  Debug(x)
#else
#define  Debug(x)  x
#endif
```

#include

Description

#include inserts the full text from another file at this point in the source code. The inserted file may contain any number of valid C statements.

Syntax

include-directive:

```
#include " filename " new-line
#include < filename > new-line
#include pp-token-list new-line
```

When <filename> is used, MPLAB-C17 looks for the file in the directory specified by the environment variable MCC_INCLUDE or in the command line parameter '/i'.

When "filename" is used, MPLAB-C17 looks for the file in the current directory and then in the directory specified by MCC_INCLUDE.

Example

```
#include <p17cxx.h>
#include "header.h"
```

#line

Description

The line directive causes the compiler to renumber the source text so that the following line has the specified line number.

Syntax

line-directive:

```
#line digit-sequence new-line
#line digit-sequence " filename " new-line
#line pp-token-list new-line
```

Example

```
#line 34            // This line is line 34
#line 55 "main.c"  // This line is line 55 of main.c
```

#pragma {code|udata|idata|romdata} [[name] [{{gpr | sfr} n} | {=address}]

Description

These directives change the section in which a type of data is allocated. Specifying an address for a new section will create an absolute section at that location and begin allocating data of the specified type into the new section. Issuing a section pragma without specifying a name for the section causes the compiler to revert to allocating data into the default section for that section type. Issuing a section pragma with a section name which is the same as a section name earlier in the source code file causes the compiler to resume allocation of the type of data into that section. Specifying an address twice for the same section name is an error. Specifying 'gpr | sfr nn' is equivalent to adding a '#pragma varlocate gpr | sfr n' for each variable contained inthe section.

Syntax

```
#pragma code mycode  // changes the allocation of code to a new
                     // section called 'mycode'

#pragma romdata      // changes the allocation of code to the
                     // default romdata section
```

#pragma nocontext

Description

For the next function defined after the #pragma nocontext directive, the compiler will not generate prologue or epilogue code to set up the stack frame or save and restore working register contents. Use this directive to optimize a function that has no return value, no arguments and no local variables.

Syntax

```
#pragma nocontext
```

#pragma nosaveregs

Description

For the next function defined after the #pragma nosaveregs directive, the compiler will not generate prologue or epilogue code to save and restore working register contents. Use this directive to optimize a function with no return value.

Syntax

```
#pragma nosaveregs
```

#pragma list

Description

The #pragma list directive turns on list file generation for all code following the directive.

Syntax

```
#pragma list
```


#pragma nolist

Description

The #pragma nolist directive turns off list file generation for all code following the directive.

Syntax

```
#pragma nolist
```

#undef

Description

The #undef directive undefines a string constant. After a string constant has been undefined, any reference to it generates an error unless the string constant is redefined.

Syntax

undef-directive:

    #undef *identifier new-line*

Example

```
#define MAX_COUNT 10
.
.
.
#undef MAX_COUNT
#define MAX_COUNT 20
```

#pragma varlocate {gpr | sfr} n

The `varlocate` pragma tells the compiler in which bank and in what address range (GPR or SFR) a variable will be located at link time, enabling the compiler to perform more efficient bank switching.

`varlocate` specifications are not enforced by the compiler at link time. The sections which contain the variables should be assigned explicitly in the linker script, or via absolute sections in the modules(s) where they are defined, into the correct bank.

Variables

This section examines how C uses variables to store data.

The topics discussed in this section are:

- Basic Data Types
- Variable Declaration
- Enumeration
- Typedef

Basic Data Types

Description

- void
- char
- int
- float - not supported in MPLAB-C17
- double - not supported in MPLAB-C17

The following modifiers are also allowed:

Table 3.2: Data Type Modifiers

Modifier	Applicable Data Type	Use
auto	any	Variable exists only during the execution of the block in which it was defined.
const	any	Declares data that will not be modified.
far	any	Declares paged/banked data
extern	any	Declares data that is allocated elsewhere
long	int	Not supported
near	any	Declares non-paged/non-banked data
register	any	No effect in MPLAB-C17
short	int	Declares an 16-bit integer.
signed	char, int, long*	Declares a signed variable.
static	any	Variable is retained unchanged between executions of the defining block.
unsigned	char, int, long*	Declares an unsigned variable.

The following table shows the size and range of common data types as implemented by MPLAB-C17.

Table 3.3: Data Type Ranges

Type	Bit Width	Range
void	N/A	none
char	8	-128 to 127
unsigned char	8	0 to 255
int	16	-32,768 to 32,767
unsigned int	16	0 to 65,535
short	16	-32,768 to 32,767
unsigned short	16	0 to 65,535
long*	32	-2,147,483,648 to 2,147,483,647
unsigned long*	32	0 to 4,294,967,295
float*	32	1.7549435E-38 to 6.80564693E+38
double*	32	1.7549435E-38 to 6.80564693E+38

* these types are not supported in MPLAB-C17

C represents all negative numbers in the two's complement format.

Integral data types are char, int, long of all sizes, and enumerations.

Variable Declaration

Description

A variable is a name for a specific memory location. In C, all variables must be declared before they are used. A variable's declaration defines the data type and the size of the variable.

Variables can be declared in two places: inside a function or outside all functions. The variables are called local and global, respectively.

Syntax

```
declaration:
      declaration-specifiers declarator-list ;

declarator-list:
      declarator
      declarator-list , declarator
```

declaration-specifiers:
  declaration-specifier
  declaration-specifiers declaration-specifier

declaration-specifier:
  type-name
  extern
  static
  ram
  rom
  const
  volatile
  near
  far

type-name:
  basic-type-name
  tag-type-name

basic-type-name:
  int
  short
  char
  unsigned
  long
  float
  double

tag-type-name:
  enumerated-type-name
  struct-or-union-type-name

Local variables (declared inside a function or a block of code) can only be used by statements within the block where they are declared. The value of a local variable cannot be accessed by functions or statements outside of the function. The most important thing to remember about local variables is that they are created upon entry into the block and destroyed when the block is exited. Local variables must be declared before executable statements.

Global variables can be used by all of the functions in the program. Global variables must be declared before any functions that use them. Most importantly, global variables are not destroyed until the execution of the program is complete.

Example

```
#include <p17cxx.h>
unsigned char GlobalCount;

void f2()
{
      unsigned char count;
      for(count=0;count<10;count++)
         GlobalCount++;
}
```

```
void f1()
{
       unsigned char count;
       for(count=0;count<10;count++)
       {
       unsigned char temp;
          f2();
          temp = count *2;
       }
}
void main(void)
{
       GlobalCount = 0;
       f1();
}
```

This program increments GlobalCount to 100. The operation of the program is not affected adversely by the variable named count located in both functions. The variable ’temp’ is allocated inside the for() loop and deallocated once the loop exits.

Storage Class (extern, static, volatile)

static/extern/volatile

’static’ and ’extern’ behave in the ANSI specified manner. ’static’ used with a local variable declaration inside of a block causes the variable to maintain its value between entrances to the block. ’static’ used for a global object (variable or function) declaration outside of all functions limits the scope of the object to the file containing the definition.

’extern’ does not allocate space for its object. The compiler assumes the definition appears in an external file. This external reference is resolved at link time.

A global object has external linkage by default.

Example

In file1.c:

```
static unsigned char a;
       unsigned char b;
void main(void)
{
       a = 1;
       b = 2;
       a = new_function();
       return a;
}
```

In file2.c:

```
extern int b;
int new_function(void)
{
     int c;
     c = b;/*   this will not produce an error, because
                b is extern by default in file1.c and
                declared extern in file2.c */
     return a;/*this will produce an undefined variable
                error because ’a’ is only valid within
                file1.c  */
}
```

Example

```
unsigned char hello()
{
      static unsigned char i = 0;
      i++;
      return i;
}
void main()
{
      unsigned char count;
      for( count = 0; count < 10; count++ )
      {
         unsigned char a;
         a = hello();
      }
}
/* For each call of the function hello, i will be
incremented. i is static and will maintain its value
between calls to hello.  hello is called 10 times, so i
will be ’10’ after the last call.  */
```

volatile

A `volatile` variable has a value that can be changed by something other than user code. A typical example is an input port or a timer register. These variables must be declared as ’volatile’ so the compiler makes no assumptions on their values while performing optimizations.

Example

```
unsigned char  x, y;
volatile unsigned char TMR0;
x = 0x55; //Compiler’s temporary registers contain 0x55
y = x;    //and those values are written to ’y’ since x is
             unchanged
```

```
TMR0 = 0x00;
y = TMR0; //The compiler must read TMR0 and cannot use the
         // 0x00 in its temporary variables since TMR0
            increments with execution.
```

Enumeration

Description

An Enumeration defines a list of named integer constants. The constants defined by an enumeration can be used in the place of any integral value. Enumerated types are implemented as signed int type in MPLAB-C17. This means that the enumerated values are between -32,768 to 32,767.

Syntax

enumerated-type-name:
 enum *identifier*
 enum *identifier* **{** *enumeration-list* **}**
 enum **{** *enumeration-list* **}**

enumeration-list:
 enumerated-value
 enumeration-list **,** *enumerated-value*

enumerated-value:
 identifier
 identifier **=** *constant-expression*

All enumeration identifiers (such as VALUE_1 in the example) must be unique across all defined enumerations.

Enumerated values can be specified for each enumerated member.

Example

```
enum tag_1 { VALUE_1, VALUE_2, VALUE_3 } enum_1;
/*    VALUE_1 is equal to 0 *
 *    VALUE_2 is equal to 1 *
 *    VALUE_3 is equal to 2 */

char char_1;
enum_1 = 42;    /* this will not produce an error */
char_1 = VALUE_3;/* this will assign char_1 value to 2 */
```

Example

```
enum tag_2 { VALUE_3, VALUE_4, VALUE_5 } enum_2;
/* this definition will cause an error because VALUE_3
already has a value of 2, and cannot also hold a value of 0 */
```

```
enum tag_3 { VALUE_6 =2, VALUE_7, VALUE_8=50, VALUE_9 }
enum_3;
/*  VALUE_6 is equal to 2  *
 *  VALUE_7 is equal to 3  *
 *  VALUE_8 is equal to 50 *
 *  VALUE_9 is equal to 51 */
enum color_type {red,green,yellow} color;
```

The entries in the enumeration list are assigned constant integer values, starting with zero for the first entry. Each entry is one greater than the previous one. Therefore, in the above example, red is 0, green is 1, and yellow is 2.

The default integer values assigned to the enumeration list can be overridden by specifying a value for a constant. The following example illustrates specifying a value for a constant.

```
enum color_type {red,green=9,yellow} color;
```

This statement assigns 0 to red, 9 to green, and 10 to yellow.

Once an enumeration is defined, the name can be used to create additional variables at other points in the program. For example, the variable mycolor can be created with the color_type enumeration by:

```
enum color_type mycolor;
```

Essentially, enumerations help to document code. Instead of assigning a value to a variable, use an enumeration to clarify the meaning of the value.

Using `typedef` to Create `Portable Programs`. When writing portable code, it is important that the data size be consistent. For example, suppose that 16-bit integers are required. Rather than declaring integers as int, declare them as a typedef name, such as myint. Near the top of the program, declare the typedef based on the target machine. When compiling with a tool that uses 16-bit integers, the typedef statement should read:

```
typedef int myint;
```

typedef

Description

The typedef statement creates a new name for an existing type. The new name can then be used to declare variables.

Syntax

The `'typedef'` keyword may be used anywhere the storage class specifiers `'extern'` and `'static'` may be used.

Example

```
typedef char string;
typedef unsigned int uint;
void main()
{
      string j[10];
      uint i;
      for(i=0;i<10;i++)
         j[i]=i;
}
```

When using a typedef statement, remember these two key points:

- A typedef does not deactivate the original name or type.
- Several typedef statements can be used to create many new names for the same original type.

The typedef typically has two purposes:

- Create portable programs
- Document source code

Functions

Functions are the basic building blocks of the C language. All executable statements must reside within a function.

The topics discussed in this section are:

- Function Declarations
- Function Prototyping
- Passing Arguments to Functions
- Returning Values from Functions

Function Declarations

Description

Functions must be declared before they are used. The compiler supports the modern ANSI form of function declarations.

Syntax

function-definition:
    function-declarator compound-statement

function-declarator:
    declaration-specifiers identifier **(** *parameter-list* **)**

parameter-list:
    parameter
    parameter parameter-list

parameter:
    type-specifier
    declarator

Example

```
unsigned char AddOne(unsigned char x)
{
      return(x + 1);
}
```

Function Prototyping

Description

A function prototype should be declared before the function is called. A function prototype declares the return type, name, and types of parameters for a function, but no other statements.

Syntax

function-prototype:
    function-declarator **;**

Example

```
unsigned char AddOne(unsigned char x);
```

Overhead of Passing Variables
MPLAB-C17 uses a software stack for passing variables into functions and for returning values from functions. This makes it possible to support quite complex functions and allows recursive functions, but there is some overhead in managing the software stack. You can choose to reduce code size by not passing on the stack,, using instead static variables. When compiling, the compiler will examine the function and only include the appropriate level of stack support code.

Passing Arguments to Functions

Description

A function argument is a value that is passed to the function when the function is called. C allows zero or more arguments to be passed to a function.

When a function is defined, formal parameters are declared between the parentheses that follow the function name.

Function parameters can have storage class ’auto’ or ’static’. ’auto’ parameters are placed on the software stack, enabling reentrancy, and

’static’ parameters are allocated globally, enabling direct access and, therefore, smaller code.

If the first parameter to a function is ’static’ and is 8 bits wide, the argument will be passed to the functin in `PRODL`. If it is ’static’ and 16-bits wide, the argument will be passed in `PROD`.

Example

The function below calculates the sum of two values that are passed to the function when it is called. When `sum()` is called, the value of each argument is copied into the corresponding parameter variable.

```
void sum( static unsigned char a, unsigned char b )
{
      int c;
      c = a+b;
}
void main()
{
      sum(1,10);
      sum(15,6);
      sum(100,25);
}
```

Functions pass arguments by value. Any changes made to the formal parameter do not affect the original value in the calling routine.

Returning Values from Functions

Description

A function in C can return a value to the calling routine by using the return statement. If the value being returned is 8-bits wide, it is returned in WREG. If it is 16-bits wide, it is retuned in the WREG/FSR1 pair. Otherwise, it is retuned on the software stack.

Syntax

return-statement:

return *expression* **;**

return ;

Example

```
unsigned char sum(unsigned char a, unsigned char b)
{
      return(a + b);
}
void main()
{
      unsigned char c;
      c = sum(1, 10);
      c = sum(15, 6);
      c = sum(100, 25);
}
```

When a return statement is encountered, the function returns immediately to the calling routine. Any statements after the return are not executed. The return value of a function is not required to be assigned to a variable or to be used in an expression; however, if it is not used, then the value is lost.

Operators

A C expression is a combination of operators and operands. For the most part, C expressions follow the rules of algebra.

This section discusses many different types of operators including:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Increment and Decrement Operators
- Conditional Operator
- Precedence of Operators
- Operator Differences

Arithmetic Operators

Description

The C language defines five arithmetic operators: addition, subtraction, multiplication, division, and modulus.

Syntax

```
arithmetic-expression:
      postfix-expression
      arithmetic-expression arithmetic-operator postfix-
      expression
```

```
arithmetic-operator:
      +  addition
      -  subtraction
      *  multiplication
      /  division
      %  modulus
```

The +, -, *, and / operators may be used with any basic data type.

The modulus operator, %, can only be used with integral data types.

Example

```
-b              //negative b
count - 163     //variable count minus 163
```

Relational Operators

Description

The relational operators in C compare two values and return '1' or '0' based on the comparison.

Syntax

```
relational-expression:
      arithmetic-expression
      relational-expression relational-operator arithmetic-
      expression
relational-operator:
      >  greater than
      >= greater than or equal to
      <  less than
      <= less than or equal to
      == equal to
      != not equal to
```

Example

```
count > 0
value <= MAX
input != BADVAL
```

Logical Operators

Description

The logical operators support the basic logical operations `AND`, `OR`, and `NOT`.

Syntax

```
logical-or-expression:
      logical-and-expression
      logical-or-expression || logical-and-expression
```

logical-and-expression:
    relational-expression
    logical-and-expression **||** *relational-expression*

logical-not-expression:
    ! *unary-expression*
    && *Logical AND*
    || *Logical OR*
    ! *Logical NOT*

Example

```
NotFound && (i <= MAX)
!(Value <= LIMIT)
((’a’ <= ch) && (ch <= ’z’)) || ((’A’ <= ch) && (ch <= ’Z’))
```

Bitwise Operators

Description

C contains six special operators which perform bit-by-bit operations on numbers. These bitwise operators can only be used on integral data types. The result of using any of these operators is a bitwise operation of the operands.

Syntax

bitwise-expression:
    postfix-expression
    bitwise-expression bitwise-operator postfix-expression

bitwise-not-expression:
    ~ *unary-expression*

bitwise-operator:
    & bitwise AND
    | bitwise OR
    ^ bitwise XOR
    ~ 1’s complement
    >> right shift
    << left shift

Example

```
Flags & MASK;   //Zero unwanted bits
Flags ^ 0x07;   //Flip bits 0, 1, and 2
Val << 2;       //Multiply Val by 4
```

Assignment Operators

Description

The most common operation in a program is to assign a value to a variable. In C, this is done by using the equals sign (=).

C also provides shortcuts for modifying a variable by performing an operation on itself. These shortcuts are special assignment operators.

Syntax

assignment-expression:
    unary-expression assignment-op expression

assignment-op:
    =
    +=
    -=
    ***=**
    /=
    %=
    |=
    ^=
    >>=
    <<=

Example

```
a += b + c;        //Same as a = a + b + c;
a *= b + c;        //Same as a = a * (b + c);
a *= (b + c);      //Same as a = a * (b + c);
r /= s;            //Same as r = r / s;
m *= 5;            //Same as m = m * 5;
Flags |= SETBITS;  //Set bits in Flags
Div2 >>= 1;        //Divide Div2 by 2
```

Increment and Decrement Operators

Description

C provides shortcuts for the common operation of incrementing or decrementing a variable. The increment and decrement operators are extremely flexible. They can be used in a statement by themselves, or they can be embedded within a statement with other operators. The position of the operator indicates whether the increment or decrement is to be performed before or after the evaluation of the statement in which it is embedded.

Syntax

pre-increment-expression:
    ++ *unary-expression*

pre-decrement-expression:
    -- *unary-expression*

post-increment-expression:
    postfix-expression **++**

post-decrement-expression:
    postfix-expression **--**

Example

```
void main()
{
      unsigned char a = 0, b, c;
      a++;            //same as ++a;
                      //a = 1
      b = 5 + a++;    //b = 6, a = 2
      c = 6 + --a;    //c = 7, a = 1
}
```

Conditional Operator

Description

The conditional operator is a shortcut for executing code based on the evaluation of an expression.

Syntax

conditional-expression:
    logical-OR-expression **?** *comma-expression* **:** *conditional-expression*

Example

```
c = (a>b) ? a : b;    //c is set to the larger of a and b
```

Precedence of Operators

Description

Precedence refers to the order in which operators are processed. The C language maintains a precedence for all operators. The following shows the precedence from highest to lowest. Operators at the same level are evaluated from left to right.

Highest
() [] -> .
! ~ ++ -- - (type cast) * & sizeof
* / %
+ -
<< >>
< <= > >=
== !=

<table>
<tr><td>&</td></tr>
<tr><td>^</td></tr>
<tr><td>|</td></tr>
<tr><td>&&</td></tr>
<tr><td>||</td></tr>
<tr><td>?</td></tr>
<tr><td>= += -= *= /=</td></tr>
<tr><td>,</td></tr>
<tr><td>Lowest</td></tr>
</table>

Example

<table>
<tr><th>Expression</th><th>Result</th><th>Note</th></tr>
<tr><td>10 - 2 * 5</td><td>0</td><td>*has higher precedence than +</td></tr>
<tr><td>(10 - 2) * 5</td><td>40</td><td></td></tr>
<tr><td>0x20 | 0x01 != 0x01</td><td>0x20</td><td>!= has higher precedence than |</td></tr>
<tr><td>(0x20 | 0x01) != 0x01</td><td>1</td><td></td></tr>
<tr><td>1 << 2 + 1</td><td>8</td><td>+ has higher precedence than <<</td></tr>
<tr><td>(1 << 2) + 1</td><td>5</td><td></td></tr>
</table>

Program Control Statements

This section describes the statements that C uses to control the flow of execution in a program, explains how relational and logical operators are used with these control statements, and covers how to execute loops.

Topics discussed in this section include:

- if Statement
- if-else Statements
- for Loop
- while Loop
- do-while Loop
- break Statement
- continue Statement
- switch Statement
- TRUE is any non-zero value
- FALSE is zero

if Statement

Description

The if statement is a conditional statement. The block of code associated with the if statement is executed based upon the outcome of a condition. If the condition evaluates to TRUE, the code is executed. Otherwise, the code is skipped.

Syntax

if-statement:
    if (*expression* **)** *statement*

Example

```
if(num > 0) Adjust(num);
if(count<0)
{
      count=0;
      EndFound = TRUE;
}
```

if-else Statements

Description

The if-else statement handles conditions where a program requires one set of instructions to be executed if a condition is TRUE and a different set of instructions if the condition is FALSE.

Syntax

if-else-statement:
    if (*expression* **)** *statement* **else** *statement*

Example

```
if(num < 0)
{
      num = 0;
      Valid = FALSE;
}
else
      Valid = TRUE;

if(num == 1)
      DoCase1();
else if(num == 2)
      DoCase2();
else if(num == 3)
      DoCase3();
else
      DoInvalid();
```


for Statement

Description

One of the three loop statements that C provides is the for loop. Use a for loop to repeat a statement or set of statements.

Syntax

for-statement:
      for (*expression* **;** *expression* **;** *expression* **)** *statement*

Example

```
unsigned char i;
for(i=0;i<10;i++)
      DoFunc();
for(num=100;num>0;num=num-1)
      { . . . }
for(count=0;count<50;count+=5)
      { . . . }
for(i=0; (i<MAX) && (Array[i]<>Target); i++); //Find Target
```

while Statement

Description

Another of the loops in C is the while loop. While an expression is TRUE, the while loop repeats a statement or block of code. The value of the expression is checked prior to each execution of the statement.

Syntax

while-statement:
      while (*expression* **)** *statement*

Example

```
X = GetValue()
while (1);//Loop Forever
{
      HandleValue(X);
      X = GetValue();
}
```

do-while Statement

Description

The final loop in C is the do loop. In the do loop, the statement is always executed before the expression is evaluated. Thus, the do statement always executes at least once.

Syntax

if-statement:
    do *statement* **while (** *expression* **) ;**

Example

```
do
{
      x = GetValue()
      HandleValue(x);
} while (x != 0);
```

switch Statement

Description

A switch statement is functionally equivalent to multiple if-else statements.

The switch statement has two limitations:

- The switch variable must be an 8-bit integral data type.
- The switch variable can only be compared against constant values.

Syntax

switch-statement:
    switch (*expression* **)** *statement*

case-statement:
    case *constant-expression* **:** *statement*

default-statement:
    default : *statement*

The use of the default label is good programming practice. It can catch out of range data that is not expected.

The switch variable is successively tested against a list of constants. When a match is found, execution continues at the labeled case staement. If no match is found, the statements associated with the default case are executed if a default label exists.

Example

```
switch(i)
{
      case 1:
         DoCase1();
         break;
      case 2:
         DoCase2();
         break;
      case 3:
         DoCase3();
         break;
      case 4:
```

```
        DoCase4();
        break;
    default:
        DoDefault();
        break;
}
x = 0;
switch(ch)
{
    case ’c’:          //Ignoring case, set x to:
    case ’C’: x++;     //   1 if ch is A
    case ’b’:          //   2 if ch is B
    case ’B’: x++;     //   3 if ch is C
    case ’a’:          //otherwise, ch is invalid
    case ’A’: x++;
        break;
    default :
        BadChar(ch);
        break;
}
```

break Statement

Description

The break statement exits the innermost enclosing control statement (for, while, do, switch) from any point within the body. The break statement bypasses normal termination from an expression. If the break occurs in a nested loop, control returns to the previous nesting level.

Syntax

break-statement:
    break ;

Example

```
//Get 100 values. Stop immediately if the value is 0.
unsigned char i;
for(i = 0; i < 100; i++)
{
    x = GetValue();
    if(x == 0)
        break;
    HandleValue(x);
}
```

continue Statement

Description

The continue statement allows a program to skip to the end of a for, while, or do statement without exiting the loop.

Syntax

continue-statement:
    continue ;

Example

```
//Get 100 values. If the value is 0,
//ignore it and go on.
unsigned char i;
for (i = 0; i < 100; i++)
{
      x = GetValue;
      if (x == 0)
         continue;
      HandleValue(x);
}
```

Arrays and Strings

An array is a list of related variables of the same data type. Strings are arrays of characters with some special rules.

Topics discussed in this section include:

- Arrays
- Strings
- Initializing Arrays

Arrays

Description

An array is a list of variables that are all of the same type and can be referenced through the same name. An individual variable in the array is called an array element. When an array is declared, C defines the first element to be at an index of 0. If the array has 50 elements, the last element is at an index of 49.

C stores arrays in contiguous memory locations. The first element is at the lowest address. An array element can be used anywhere a variable or constant would be used.

Syntax

declarator:
    declarator array-declarator

array-declarator:
    [*constant-expression* **]**
    array-declarator **[** *constant-expression* **]**

Example

```
#define SIZE 10
unsigned char i, num[SIZE];
for(i = 0; i < SIZE; i++)
      num[i] = i;
```

To copy the contents of one array into another, copy each individual element from the first array into the second array. The following example shows one method of copying the array a[ ] into b[ ] assuming that each array has 10 elements.

```
for(i=0;i<10;i++)
      b[i] = a[i];
```

Strings

Description

A common one-dimensional array is the string. C does not have a built-in string data type. Instead, a string is defined as a null (0) terminated character array. The size of the character array must include the terminating null. All string constants are automatically null terminated.

Example

```
char String[80];
int i;
.
.
.
for(i = 0; (i < 80) && !String[i]; i++)
      HandleChar(String[i]);
```

Initializing Arrays

Description

C allows pre-initialization of arrays.

Syntax

initialized-declarator:
    declarator **= {** *value-list* **}**

value-list:
    { *value-list* **}**
    constant-expression-list

constant-expression-list:
    constant-expression
    constant-expression-list **,** *constant-expression*

Example

The following example shows a 5 element integer array initialization.

```
int i[5] = {1,2,3,4,5};
```

The element i[0] has a value of 1 and the element i[4] has a value of 5.

A string (character array) can be initialized in two ways. One method is to make a list of each individual character:

```
char str[4]={'a','b','c', 0};
```

The second method is to use a string constant:

```
char name[5]="John";
```

A null is automatically appended at the end of "John". When initializing an entire array, the array size may be omitted:

```
char Version[] = "V1.0";
```

Because the PICmicro family of microcontrollers uses separate program memory and data memory address busses in their design, MPLAB-C17 requires ANSI extensions to distinguish between data located in ROM and data located in RAM. The ANSI/ISO C standard allows for code and data to be in separate address spaces, but this is not sufficient when it is required to locate data in the code space as well. To this purpose, MPLAB-C17 introduces the `rom` and `ram` qualifiers. Syntactically, these qualifiers bind to identifiers just as the const and volatile qualifiers do in strict ANSI C.

The primary use of ROM data is for static strings. In keeping with this, MPLAB-C17 automatically places all string literals in ROM. The type of a string literal is "array of char located in ROM." For example, a string table in ROM can be declared as:

```
       rom const char table[][20] = { "string 1", "string 2",
                                      "string 3", "string 4"
};
       rom const char *rom table2[] = { "string 1", "string 2",
                                        "string 3", "string 4"
};
```

The declaration of `table` declares an array of four strings that are each 20 characters long, and so takes 40 words of program memory. Table2 is declared as an array of pointers to ROM. The `rom` qualifier after the * places the array of pointers in ROM as well. All of the strings in table2 are 9 bytes long, and the array is four elements long, so `table2` takes (9*4+4*2)/2 = 22 words of program memory. Accesses to `table2` may often be less efficient than accesses to `table`, however, because of the additional level of indirection required by the pointer.

An important consequence of the separate ROM and RAM address spaces for MPLAB-C17 is that pointers to data in ROM and pointers to data in RAM are not compatible. That is, two pointer types are not compatible unless they point to objects of compatible types and the objects they point to are located in the same address space. For example, a pointer to a string in ROM and a pointer to a string in RAM are not compatible because they refer to different address spaces. To copy data from ROM to RAM, it must be done explicitly. For simple types, this entails only a simple assignment, but for arrays and other complex data-types it may require more.

For example, a function to copy a string from ROM to RAM could be written as follows.

```
void str2ram(static char *dest, static char rom *src)
{
       while( (*dest++ = *src++) != '\0' )
              ;
} /* end str2ram */
```

As an example, the following code will send a ROM string to USART1 on a PIC17C756 using the PICmicro C libraries. The library function to send a string to the USART, `putsUSART1(const char *str)`, takes a pointer to a string as its argument, but that string must be in ram.

METHOD 1: COPY THE ROM STRING TO A RAM BUFFER BEFORE SENDING

```
rom char mystring[] = "Send me to the USART";
void foo( void )
{
       char strbuffer[21];
       str2ram( strbuffer, mystring );
       putsUSART1( strbuffer );
}
```

METHOD 2: MODIFY THE LIBRARY ROUTINE TO READ FROM A ROM STRING.

```
/* The only changes required to the library routine is to change
 * the name so the new routine does not conflict with the
   original
 * routine and to add the rom qualifier to the parameter.
 */
void putsUSART1_rom( static const rom char *data )
{
       do                  // Send characters up to the null
       {                   // Write a byte to the UASRT
              while(BusyUSART1());
              putcUSART1(*data);
       } while(*data++);
} /* end putsUSART1_rom */
```

Pointers

This section covers one of the most important and powerful features of C, pointers. A pointer is a variable that contains the location of an object.

The topics covered in this section are:

- Introduction to Pointers
- Pointers and Arrays
- Pointer Arithmetic
- Passing Pointers to Functions

ROM and RAM pointers in MPLAB-C17

Pointer arithmetic is complicated by the ROM paging and RAM banking of the PICmicro MCU. Pointers are assumed to be RAM pointers unless declared as ROM.

```
rom int *p;          // ROM pointer
char *q;             // RAM pointer (default)
ram char *r;         // RAM pointer (explicitly declared)
char * rom * pp;     // RAM pointer to a ROM char pointer
```

RAM pointers are 16-bit values. ROM pointers are 24-bit values if they point to 8-bit objects. ROM pointers are

Introduction to Pointers

Description

A pointer is an object that holds the location of another object or a NULL constant.

For example, if a pointer variable called Var1 contains the address of a variable called Var2, then Var1 points to Var2. If Var2 is a variable at address 100 in memory, then Var1 would contain the value 100.

Syntax

declarator:
 ***** *type-qualifier-list declarator*

The two special operators that are associated with pointers are the asterisk (*) and the ampersand (&). The address of a variable can be accessed by preceding the variable with the & operator. The * operator returns the value stored at the address pointed to by the variable.

Example

```
void main(void)
{
      unsigned char *Var1, Var2, Var3;

      Var2 = 6;
      Var1 = &Var2;
      Var3 = Var2;                //These two do
      Var3 = *Var1;               //the same thing.
}
```

The first statement declares three variables: Var1, which is an integer pointer, and Var2 and Var3, which are integers. The next statement assigns the value of 6 to Var2. Then the address of Var2 (&Var2) is assigned to the pointer variable Var1. Finally, the value of Var2 is assigned to Var3 in two ways: first by accessing Var2 directly, then by accessing Var2 through the pointer Var1.

Pointer Arithmetic

Description

In general, pointers may be treated like other variables. However, there are a few rules and exceptions. In addition to the * and & operators, there are only four other operators that can be applied to pointer variables: +, ++, -, --.

An important point to remember when performing pointer arithmetic is that the value of the pointer is adjusted according to the size of the data type it is pointing to. If a pointer's data type requires five memory bytes, "incrementing" the pointer actually increases the value of the pointer by five. Similarly, "adding" three to the pointer increases the value of the pointer by fifteen (three times five).

Example

```
unsigned char *p, *q, r[30] ;
.
.
p = r + 20;//p points to element 20 of r
q = p - 5//q points to element 15 of r
p++;  //p points to element 21 of r
```

It is possible to increment or decrement either the pointer itself or the object to which it points. Pointers may also be used in relational operations.

Passing Pointers to Functions

Description

A pointer may be passed to a function just like any other variable.

Example

```
void incby10(unsigned char *n)
{
      *n += 10;
}
void main(void)
{
      unsigned char *p;
      unsigned char i = 0;

      p=&i;
      incby10(p);     //i equals 10
      incby10(&i);    //i equals 20
}
```

Structures and Unions

Structures are a group of related variables. Unions are a group of variables, often of differing types, that share the same memory space.

This section covers:

- Introduction to Structures
- Introduction to Unions
- Nesting Structures
- Bit-fields

Syntax

struct-or-union-type-name:
    struct-or-union identifier
    struct-or-union identifier **{** *member-declaration-list* **}**
    struct-or-union **{** *member-declaration-list* **}**


```
member-declaration-list:
      member-declaration
      member-declaration-list member-declaration
member-declaration:
      member-declaration-specifiers declarator-list ;
member-declaration-specifiers:
      member-declaration-specifier
      member-declaration-specifiers member-declaration-
      specifier
member-declaration-specifier:
      type-name
      const
      volatile
      near
      far
```

Structures and Debugging in MPLAB
User-defined data constructs are not fully described in the symbolic information file from the linker, and you may not be able to use the names of elements of structures when debugging in MPLAB.

Introduction to Structures

Structures and unions allow the storage and manipulation of related data together rather than in separate variables. Structures located in ROM must have all elements word aligned.

Description

A structure is a group of related items that can be accessed through a common name. Each item within a structure has its own data type, which can be different from the other data types.

Example

The following example is for a card catalog in a library.

```
struct catalog_tag
{
      char author[40];
      char title[40];
      char pub[40];
      unsigned int date;
      unsigned char rev;
} card;
```

In this example, the tag of the structure is catalog. It is not the name of a variable, only the name of the type of structure. The variable card is declared as a structure of type catalog. The following shows what the structure catalog looks like in memory.

author	40 bytes
title	40 bytes
pub	40 bytes
date	2 bytes

rev	1 byte

To access any member of a structure, specify the name of the variable and the name of the member separated by a period. For example, to change the revision member of the structure catalog, use the following:

```
card.rev='a';
```

To access the third character in the title, use the following:

```
ThirdChar = card.title[2];
```

Introduction to Unions

Description

A union is a memory block that is shared by two or more variables, which can be of any data type. A union resembles a structure, but its memory usage is fundamentally different. In a structure, the elements are arranged sequentially. In a union, all of the elements begin at the same address, making the size of the union equal to the size of the largest element.

Syntax

The `<union-name>` is the tag of the union, and the `<variable-list>` contains the names of the variables that have a data type of `<union-name>`.

Accessing members of a union is the same as accessing members of a structure.

Example

Because an int is two bytes, a char is one byte, and a long is four bytes, the union below is stored in memory as shown:

```
union u_tag
{
      int i;
      char c[3];
      long l;
} temp;
```

where:

```
<------------  i ------------>

<-----c[0]----><---- c[1]----><---- c[2]----><---- c[3]-----
>

<--------------------------- l ---------------------------
>
```

location 0	location 1	location 2	location 3

An example of saving space is shown below:

```
struct type_tag
{
      enum { VARIABLE, CONSTANT } type;
```

```
        union
        {
            char *variable_name;
            int   constant_value;
        } value;
} variable_or_constant;

void function( struct type_tag var_or_const )
{
        int constant;
        char *variable;

        switch( var_or_const.type )
        {
            case VARIABLE:
                    variable = var_or_const.value.variable_name;
                    break;
            case CONSTANT:
                    constant = var_or_const.value.constant_value;
                    break;
        }
}
```

Based on the type of data stored in `struct type_tag`, the access of the data is different. A union allows the data for the two types to share space.

An example of using a union to access memory as two different data types is shown below:

```
union MergeData
{
        short int TwoInts[2];
        long OneLong;
};
```

The above union accesses memory as two integers or as one long integer.

Nesting Structures

Description

A structure member can have a data type that is another structure. This is referred to as a nested structure.

Example

```
struct Memory
{
        int RAMSize;
        int ROMSize;
};
```

```
struct PIC
{
      char Name[12];
      struct Memory MemSizes;
};
```

Members of a structure or union define a separate name space, Meaning that two different structures can have the same names for their members.

Example

```
struct struct_tag_1{
      int a;
      int b;
      char c;
} struct_1;
struct struct_tag_2
{
      char d;
      int  a;
      int  b;
} struct_2;
```

`struct_1.a` references the first two bytes of a structure of type `struct tag_1.`

`struct_2.a` references the second and third bytes of a structure of type `struct tag_2.`

`struct_2.c` and `struct_1.d` would produce an error because the referenced member is not part of the structure's definition.

Bit-fields

Description

Bit-fields allow the specification of 1-bit wide elements of a structure.

Syntax

```
struct <struct_name>

{
<int type> <member1> : <bit-width>;
<int type> <member2> : <bit-width>;
:
<int type> <membern> : <bit-width>;
}
```

Example

See Special Function Registers section in Chapter 4.

Chapter 4. MPLAB-C17 and PICmicro™ MCU Programming

Introduction

This section discusses specific details for PICmicro MCUs when using MPLAB-C17.

Highlights

- Processor header and assembly definition files
- Software Stack
- C startup code
- Interrupts
- Internal Assembler

Processor Header and Assembly Definition Files

Each PICmicro device has two files associated with it, a processor header file, and a processor assembly file. The assembly file contains declarations for all the special function registers on the device. Every assembly file is associated with a C header file that contains, among other things, external declarations for the special function registers.

Special Function Registers

Special function registers are defined in the processor assembly file. For example, here port A is defined in the processor assembly file P17C44.ASM as:

```
BANK0_SFR_SEC   DATA   H'010'
PORTAbits
PORTA   RES   1    ;  010h
DDRB    RES   1    ;  011h
.
.
and so on.
```

The first line specifies the file register bank where port A is located and the starting address for that bank. Port A has two labels PORTAbits and PORTA both referring to the same location (in this case 010h in bank 0). So the above definition reserves 1 byte for PORTA and PORTAbits at location 010h.

In P17C44.H, port A is declared as:

```
volatile extern far unsigned char PORTA;
```

and as:

```
extern far volatile union
{
  struct
  {
      unsigned RA0:1;     /* Bit 0                    */
      unsigned RA1:1;
      unsigned RA2:1;
      unsigned RA3:1;
      unsigned RA4:1;
      unsigned RA5:1;
      unsigned :1;
      unsigned NOT_RBPU:1;
  };
  struct
  {
      unsigned INT:1;     /* Alternate name for bit 0 */
      unsigned T0CKI:1;   /* Alternate name for bit 1 */
      unsigned :6;        /* pad next 6 locatons */
  };
} PORTAbits;
```

The first declaration specifies that PORTA is a byte (unsigned char) whereas the second one declares PORTAbits as a union of bit-addressible structures. Since individual bits in a special function register may have more than one function (and hence more than one name), there are multiple structure definitions inside the union all referring to the same register. Respective bits in all structure definitions refer to the same bit in the register. Where a bit has only one function for its position is simply padded in other structure definitions. For example, bits 2 through 7 on port A are simply padded in the second structure definition using the statement (unsigned :6).

When using a special function register such as port A, write the following statements:

```
PORTA         = 0x34; /* Assigns the value 0x34 to the whole
                                         port */
PORTAbits.INT = 1;    /* Sets the INT pin high */
PORTAbits.RA0 = 1;    /* Sets the RA0 pin high, same as
                                         above statement */
```

The 'extern' modifier is needed since the variables are declared in the processor assembly definition file. The 'volatile' modifier tells the compile that it cannot assume that port A retains values assigned to it. The 'far' modifier specifies that the port needs a bank switching instruction prior to access.

Specific Instruction Macros for PICmicro MCUs

There are certain instructions on PICmicro MCUs that may need to execute from the C code. They can be included as inline assembler instructions but for convenience they are also available as macros in C. They are listed in the following table:

Table 4:

Instruction Macro	Action
Nop ( )	Executes a no operation (NOP)
ClrWdtT ( )	Clears the watchdog timer (CLRWDT).
Sleep ( )	Executes a SLEEP instruction
Rlcf(*var*)	Rotates '*var*' to the left through the carry bit
Rlncf(*var)*	Rotates '*var*' to the left without going through the carry bit.
Rrcf(*var*)	Rotates '*var*' to the right through the carry bit.
Rlncf(*var*)	Rotates '*var*' to the right without going through the carry bit.
Swapf(*var*)	Swaps the upper and lower nibble of '*var*'

Note: 'var' must be an 8-bit quantity (i.e. char) and not located on the stack.

Interrupt Support Macros

All PIC17CXXX header files have four macros for installing interrupt service routines to the four interrupt vectors available. Call these macros as part of setting up the interrupt handler functions. Specify which C function should act as the interrupt handling function for a particular interrupt vector. For more information on how interrupts are handled by MPLAB-C17, please refer to the 'interrupts' section below. Interrupt support macros are listed in the following table:

Table 5:

Macro	Action
Install_INT(*func*)	Sets '*func*' as the handler for the INT interrupt.
Install_TMR0(*func)*	Sets '*func*' as the handler for the TMR0 interrupt.
Install_T0CKI(*func*)	Sets '*func*' as the handler for the T0CKI interrupt.
Install_PIV(*func*)	Sets '*func*' as the handler for the PIV interrupt.

Software Stack

The compiler uses a software stack for storing local variables, and for passing arguments to and returning values from functions. The software stack should not be confused with the hardware stack that the PICmicro MCU uses for storing return addresses during function calls and interrupts. Define a software stack in the linker script for the processor by placing a command similar to the following:

```
stack   size = 0x20
```

This reserves 32 bytes in the general purpose RAM area for the software stack. The size of the software stack required by a program varies with the complexity of the program. The following considerations should be kept in mind:

- One location of the stack will be reserved by the compiler for use as the Stack Pointer.
- When nesting function calls, all arguments and local variables of the calling function will remain on the stack. Therefore, the stack must be large enough to accommodate the requirements by all functions in a calling sequence.

C Startup Code

The C start up code is an assembly file that is assembled and linked with your C files. It performs four main tasks:

1. Sets up the software stack used by the compiler.
2. Optionally calls a function called __STARTUP() upon reset.
3. Optionally calls the code which copies initialized data from program memory to data memory.
4. Transfers control to the C function main() which is the entry point for C programs.

There are two C startup code files for the PIC17CXXX family. The first is C0S17.ASM which uses short GOTOs and CALLs. C0S17.ASM should be assembled and linked with the small model (code less than 8K). The other startup file is C0L17.ASM which uses long jumps and LCALLs. C0L17.ASM should be used with projects targeting memory larger than 8K.

Stack initialization

The stack initialization simply points the compiler stack pointer to the right location in data memory.

__STARTUP()

To execute some code immediately after a device reset before any other code generated by the compiler is executed, optionally create a function by the name __STARTUP(). This will be the first code executed upon a reset. To use a __STARTUP() function in a program:

1. Define a __STARTUP() function in a C program as follows:

```
void __STARTUP(void)
{
// Initialize some registers to 0
DDRB = 0;
DDRC = 0;
}
```

2. In C0L17.ASM and C0S17.ASM, uncomment the line:

```
#DEFINE  USE_STARTUP
```

3. Compile the source file, assemble C0L17.ASM or C0S17.ASM and link them.

Note that since __STARTUP() is executed before the stack is initialized, ’t’ variables may not be used

Initialized Data Support

When declaring initialized data (such as int x = 5;) the variable is allocated in data memory but the value is stored in program memory. Before the data is usable in any program, the values must be copied from program memory into the variable in data memory. C0L17.ASM and C0S17.ASM perform this task by calling a routine that does just that. The size of the initialization code is approximately 50 words. Therefore, to only initialize a few variables, do not use that feature and initialize the variables manually in the code. If initializing many variables (10 or more integers or 20 or more characters) as they are declared, then the initialization code is the better option in terms of code size. To use initialized data in the program:

1. Uncomment the following line in C0L17.ASM or C0S17.ASM
   `#DEFINE  USE_INITDATA`
2. Assemble C0L17.ASM or C0S17.ASM and IDATA17.ASM (or use IDATA17.o directly).
3. Link the above files with the C object code.

Branching to main()

After the startup code optionally calls __STARTUP() and/or copies initialized data, and sets up the stack, it calls the main() function of the C program. There are no arguments passed to main().

Default Options for the Startup Code

The startup code files are provided in object format as C0L17.O and C0S17.O. These two files are assembled with the following options:

- Initialized data support is on (i.e. USE_INITDATA is defined).
- __STARTUP() support is off (i.e. USER_STARTUP is commented out).

To change the behavior of the startup code, assemble the files after making the necessary changes. Choose 17CXX as the processor type when assembling C0L17.ASM and C0S17.ASM. The resulting object file will be usable for any PIC17CXXX project.

Interrupts

MPLAB-C17 provides interrupt support macros and code for saving and restoring context during interrupt handling. The use of such macros and code are optional. Elect to do interrupt handling in assembler to reduce latency and minimize overhead.

Each PICmicro MCU processor has two interrupt support assembly files. One is for the small model and the other for the large model as before. These files contain code fragments that save critical special function registers, call the interrupt handling function, and returns from the interrupt. The registers are saved as follows:

- First ALUSTA is saved primarily to preserve the Z bit. The saved ALUSTA can go in any bank (since BSR isn't known at that time) but always at location 0xFF. The interrupt support code reserves location 0xFF in all banks for save_ALUSTA.
- Second, PCLATH is saved at location 0xFE or the equivalent location in the same manner as with ALUSTA. The interrupt support code reserves location 0xFE in all banks for save_PCLATH.
- Finally BSR and WREG are saved in bank 0 at locations 0xFD and 0xFC, respectively. The interrupt support code reserves locations 0xFD and 0xFC in bank 0 for save_BSR and save_WREG.

Here is how the highest addresses in data memory would look if an interrupt occurred while BSR was pointing to bank 2 on the PIC17C756. (For the PIC17C44 only banks 0 and 1 are used.)

Startup code supplied with MPLAB-C17 does not support nested interrupts.

Table 6:

	Bank 0	**Bank 1**	**Bank 2**	**Bank 3**
0xFB	<Available>	<Available>	<Available>	<Available>
0xFC	**save_BSR**	<Available>	<Available>	<Available>
0xFD	**save_WREG**	<Reserved>	<Reserved>	<Reserved>
0xFE	**save_ALUSTA**	<Reserved>	<Reserved>	<Reserved>
0xFF	**save_PCLATH**	<Reserved>	<Reserved>	<Reserved>

The ALUSTA, PCLATH, BSR, and WREG are the registers that absolutely need to be saved before we branch to the interrupt service function. However, there are other registers used by the compiler that are worth saving under certain circumstances. The following is an example that uses the Timer 0 Overflow Interrupt.


```
#include <p17c44.h>

unsigned char x;

void __TMR0()
{
   x++;
   PORTB = x;
}

void main()
{
   x = 1;

// Install interrupt handler for timer 0 interrupt
   Install_TMR0(__TMR0);

// Set prescale value for TMR0
   T0STA = 0b11100110;

// Unmask TMR0 overflow interrupt
   INTSTA = 0b00000010;

// Enable all unmasked interrupts
   CPUSTA = 0;

// Set Port B in o/p mode
   DDRB = 0;

   while(1)
   {

   // Loop and wait for an interrupt to take place!
   }
}
```

Install _TMR0 (_TMR0) sets the function __TMR0() as the interrupt handler for Timer 0 overflow interrupts. Then the appropriate prescale value, interrupt flag, and global interrupt enable flag are set. The program enters into an infinite loop when it reaches the while(1) statement. When Timer 0 overflows, program control goes to the __TMR0() function where the value of 'x' is sent to PORT B and possibly displayed on LEDs.

In this simple program the PICmicro MCU wasn't doing much at the time the interrupt occurred. Therefore do not save any more registers in addition to what the compiler interrupt code saved. However, in a more complex application, the interrupt may occur at any point in the program. Therefore other registers may need to be saved. The best way to find out is to look at the generated code for the interrupt handling function. Find out which registers are used by the compiler inside the function and make sure to save them at the beginning and restore them at the end of the function. Looking at the following example's generated code, determine that registers PRODL and PRODH are used both inside and outside the interrupt function.

```
#include <p17c44.h>
```

```
#pragma udata intSave = 0xFa
  unsigned char save_PRODL;      // 0xF9
  unsigned char save_1F;         // 0xFA
  unsigned char save_1E;         //  0xFB

#pragma udata anywhere
  unsigned char x, y;

void __TMR0()
{
_asm
  movpf PRODL, save_PRODL
  movpf PRODH,  save_1E
  movpf ,  save_1F
_endasm
  x++;
  PORTB = x;
  y = y * x;
_asm
  movlr 0                  // Switch to bank 0
  movfp save_PRODL, PRODL
  movfp save_1E, PRODH
  movfp save_1F,
_endasm
}

void main()
{
   x = y = 1;

   Install_TMR0(__TMR0);

// Set prescale value for TMR0
   T0STA = 0b11100110;

// Unmask TMR0 overflow interrupt
   INTSTA = 0b00000010;

// Enable all unmasked interrupts
   CPUSTA = 0;

// Set Port B in o/p mode
   DDRB = 0;

   while(1)
   {
     x = x * 5;
   }
}
```

The registers PRODH and PRODL are saved in save_1F, save_1E, and save_PRODL, respectively. These variables are declared globally and allocated at locations 0xFa to 0xFB in bank 0 using the #pragma udata directive. This places them at the end of the bank just before save_B and guarantees they are in bank 0. Since BSR is cleared in the interrupt support code, don't do any bank switching to save those three registers. However, clear the BSR (using MOVLR 00) before restoring them as the interrupt function code could have switched banks.

The following are merely guidelines as to what the compiler might be using for certain tasks. However, the best guarantee that the context is saved and restored correctly is by looking at generated code.

1. **WREG:** This is necessary if the program is doing anything other than looping when an interrupt occurs. It is best to save WREG at all times.
2. **FSR0, FSR1:** Save FSR0 if the interrupt handling function uses arrays or pointers.
3. **PRODL, PRODH:** Save these registers if performing multiplication in the interrupt function. The compiler potentially uses PRODL and PRODH if it is evaluating a complex expression.
4. **TBLPTRL, TBLPTRH:** These two registers are used for table read and write operations. However, the compiler rarely uses them for temporary storage. In general, it is not recommended to do table reads or writes in the interrupt functions if done elsewhere in the program. Table reads and writes use the 16-bit TBLAT register for latching data transferred from and to program memory. Since TBLAT is not an addressable register it cannot be saved or restored during interrupts.

Internal Assembler

MPLAB-C17 has an internal assembler using a syntax similar to MPASM. The block of assembly code must begin with "_asm" and end with "_endasm". The syntax within the block is

```
<instruction> [arg1][, arg2][, arg3]
```

Comments must be C or C++ type notation.
Example:

```
_asm
/* User assembly code */
movlw 7       // Load 7 into WREG
movwf PORTB   // and send it to PORTB
_endasm
```

It is generally recommended to limit the use of inline assembly to a minimum. To write large fragments of assembly code, use the standalone assembler and link the modules to the C modules using MPLINK.

NOTES:

Chapter 5. Using MPLAB-C17 with Other Microchip Tools

Introduction

This chapter describes how to use MPLAB-C17 with other Microchip development tools.

Highlights

- **MPLAB IDE**
- **MPLAB-SIM**
- **PROCMD**
- **PICSTART® Plus and PRO MATE™ II**

MPLAB IDE

Why You Would Want to Use MPLAB Tools	The MPLAB IDE provides the ability to do source level debugging in C, and a Project Manager that allows programmers to edit and compile MPLAB-C17 source code. The MPLAB IDE interfaces with the PICMASTER® emulator and the MPLAB-SIM simulator for debugging source code.
The MPLAB IDE Software Version	3.40 or later
MPLAB-C17 Command Line Parameters Needed	None.
Files Types Shared between the MPLAB IDE and MPLAB-C17	Common Object Description (*.COD), List File (*.LST), Error File (*.ERR)
Setup Required	Project > Make Setup
Method of Opening Source Files from the MPLAB IDE	From the MPLAB IDE Main Menu: *Project > Open Project.* Open the source file from the project window. From the MPLAB IDE Main Menu: *File > Open Source*
Integration Description	The MPLAB IDE extracts the machine code and symbolic information from the *.COD file.
Special Considerations	None

MPLAB-SIM Simulator

Why You Would Want to Use the MPLAB-SIM Simulator Tools	The MPLAB-SIM Simulator allows programmers to simulate discrete events in an application by imitating the operation of the microcontroller. Thus, MPLAB-SIM assists in the debugging of the general logic of software.
MPLAB-SIM Software Version	5.10 or greater
MPLAB-C17 Command Line Parameters Needed	None
Files Types Shared between MPLAB-SIM and MPLAB-C17	Machine Code (*.HEX), Common Object Description (*.COD), List File (*.LST)
Setup Required	All *.HEX, *.COD, and *.LST files must be placed in the current MPLAB-SIM directory.
Method of Opening Source Files from MPLAB-SIM	Same as MPLAB
Integration Description	MPLAB-SIM gets machine code from *.HEX files, and gets symbols and source/list file correspondence from *.COD files. MPLAB-SIM uses *.LST files to show code while disassembling, single-stepping, and tracing.
Special Considerations	The PIC17CXXX family requires a hex file output format of INHX32 if configuration bits or program words above address 0x7FFF are specified.

PROCMD

Why You Would Want to Use PROCMD Tools	PROCMD enables development engineers to program Microchip PICmicro eight-bit microcontroller devices in a DOS environment.
PROCMD Software Version	All
MPLAB-C17 Command Line Parameters Needed	None
Files Types Shared between PROCMD and MPLAB-C17	Machine Code (*.HEX)
Setup Required	None
Integration Description	PROCMD programs the contents of the *.HEX file into the microcontroller.
Special Considerations	The PIC17CXXX family uses the INHX32 file format when programming. The other families use the INHX8M file format.

PICSTART Plus and PRO MATE II

Why You Would Want to Use PICSTART Plus or PRO MATE II	The PICSTART Plus or PRO MATE II device programmer enables users to quickly and easily program PICmicro microcontroller devices.
PICSTART Plus or PRO MATE II Software Version	All
MPLAB-C17 Command Line Parameters Needed	None
Files Types Shared between PICSTART Plus or PRO MATE II and MPLAB-C17	Machine Code (*.HEX)
Setup Required	None

Method of Opening Source Files from PICSTART Plus or PRO MATE II	Same as MPLAB
Integration Description	PICSTART Plus and PRO MATE II program the contents of the *.HEX file into the microcontroller.
Special Considerations	The PIC17CXXX family uses the INHX32 file format when programming. The other families use the INHX8M file format.

Chapter 6. Mixing Assembly Language and C Modules

Introduction

This section describes how to use assembly language and C modules together. It gives examples of using C variables and functions in assembly code and examples of using assembly language variables and functions in C.

Highlights

This chapter covers the following topics:

- **C calling convention**
- **Mixing assembly language and C variables and functions**

C calling convention

The following example shows how to call an assembly function with a parameter. Most of the work is done in the file 'call_asm.asm' where the parameter is taken off of the software stack. 'call_c.c' calls the 'asm_function' with a parameter.

```
// File CALL_C.C
unsigned char asm_function( unsigned char a );

unsigned char x;

void main( void )
{

   x = asm_function( 0xff );
}


; File CALL_ASM.ASM
   LIST P=17C756

   EXTERN _stack
   GLOBAL asm_function

MYCODE CODE
asm_function
   banksel _stack       ; Get the stack pointer into 0x00
   movfp   _stack, 0x01
   decf    0x01, f      ; Point FSR1 at the argument
   movfp   0x00, 0x0a   ; Get the argument
   decf    0x0a, f

                        ; The convention is that we return
                        ; with
                        ; FSR0 pointing at the return value.
```



```
                        ; We'll just reuse the space
                        ; allocated for
                        ; the argument since we're already
                        ; pointed there.
   movwf   0x00         ; Store the return value
   return

   END
```

Mixing assembly language and C variables and functions

The following example shows how to use variables and functions in both assembly language and C regardless of where they are originally defined. The file 'EX_ASM.ASM' defines 'asm_function' and 'asm_variable' as required to use them in a linked C file. The assembly file also shows how to call a C function, 'main', and how to access a C defined variable, 'c_variable'. The file 'EX_C.C' defines 'main' and 'c_variable' to be used in the assembly language file. The C file also shows how to call an assembly function, 'asm_function', and how to access the assembly defined variable, 'asm_variable'.

```
; file: EX_ASM.ASM
LIST P=17C44

    EXTERN main        ; defined in C module
    EXTERN c_variable  ; also defined in C module

MYCODE CODE

asm_function
    movlw 0xff
    movwf c_variable        ; put 0xffff in the C declared
                              variable
    movwf c_variable+1
    return

    GLOBAL  asm_function    ; export so linker can see it

MYDATA UDATA
asm_variable    RES 2   ; 2 byte variable
    GLOBAL  asm_variable    ; export so linker can see it

    END

// file: EX_C.C
extern unsigned asm_variable;
extern near void asm_function( void );

extern void main( void );

unsigned c_variable;

void main(void)
{
    asm_function();     // will modify 'c_variable'

    asm_variable = 0x1234;

}
```


Chapter 7. ANSI Implementation Issues

Introduction

This section describes the behavior of MPLAB-C17 where the ANSI standard X3.159-1989 describes the behavior as *implementation defined.* The text below in italic font is taken directly from the ANSI standard with the appropriate section in parentheses.

Highlights

This chapter covers ANSI-implementation issues for the following categories:

- **Identifiers**
- **Characters**
- **Integers**
- **Floating Point**
- **Arrays and Pointers**
- **Registers**
- **Structures and Unions**
- **Bit-Fields**
- **Enumerations**
- **Switch statements**
- **Preprocessor directives**

Identifiers

The number of significant initial characters (beyond 31) in an identifier without external linkage (3.1.2)

The number of significant initial characters (beyond 6) in an identifier with external linkage (3.1.2)

Whether case distinctions are significant in an identifier with external linkage (3.1.2)

All MPLAB-C17 identifiers have 31 significant characters. Case distinctions are significant in an identifier with external linkage.

Characters

The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character (3.1.3.4)

The value of the integer character constant is the 8-bit value of the first character. Wide characters are not supported.

Whether a 'plain' char has the same range of values as signed char or unsigned char (3.2.1.1)

A 'plain' `char` has the same range of values as a `signed char.`

Integers

The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented (3.2.1.2)

When converting from a larger integer type to a smaller integer type, the high order bits of the value are discarded and the remaining bits are interpreted according to the type of the smaller integer type. When converting from an unsigned integer to a signed integer of equal size, the bits of the unsigned integer are simply re-interpreted according to the rules for a signed integer of that size.

The results of bitwise operations on signed integers (3.3)

The bitwise operators are applied to the signed integer as if it were an unsigned integer of the same type. i.e., the sign bit is treated as any other bit.

The sign of the remainder on integer division (3.3.5)

The remainder has the same sign as the quotient.

The result of a right shift of a negative-valued signed integral type (3.3.7)

The value is shifted as if it were an unsigned integral type of the same size. i.e., the sign bit is not propagated.

Floating Point

The representations and sets of values of the various types of floating point numbers (3.1.2.5)

The direction of truncation when an integral number is converted to a floating point number that cannot exactly

r*epresent the original value (3.2.1.3)*

The direction of truncation or rounding when a floating point number is converted to a narrower floating point number (3.2.1.4)

No floating point types are supported in MPLAB-C17 at this time.

Arrays and Pointers

The type of integer required to hold the maximum size of an array - that is, the type of the sizeof operator, size_t (3.3.3.4, 4.1.1)

`size_t` is defined as an `unsigned int`.

*The result o*f casting a pointer to an integer, or vice-versa (3.3.4)

The integer will contain the binary value used to represent the pointer. If the pointer is larger than the integer, the representation will be truncated to fit in the integer.

The type of integer required to hold the difference between two pointers to elements of the same array, ptrdiff_t (3.3.6, 4.1.1)

`ptrdiff_t` is defined as an `unsigned int.`

Registers

The extent to which objects can actually be placed in registers by use of the register storage class specifier (3.5.1)

The `register` storage class specifier is ignored.

Structures and Unions

A member of a union object is accessed using a member of a different type (3.3.2.3)

The value of the member is the bits residing at the location for the member interpreted as the type of the member being accessed.

The padding and alignment of members of structures (3.5.2.1)

Members of structures and unions are aligned on byte boundaries.

Bit-Fields

Whether a 'plain' int bit-field is treated as a signed int or as an unsigned int bit-field (3.5.2.1)

A 'plain' `int` bit-field is treated as an `unsigned int` bit-field.

The order of allocation of bit-fields within a unit (3.5.2.1)

Bit-fields are allocated from least significant bit to most significant bit in order of occurrence.

Whether a bit-field can straddle a storage-unit boundary (3.5.2.1)

A bit-field cannot straddle a storage unit boundary.

Enumerations

*The integer type chosen to re*present the values of an enumeration type (3.5.2.2)

`signed int` is used to represent the values of an enumeration type.

Switch statement

The maximum number of case values in a switch statement (3.6.4.2)

The maximum number of values is limited only by target memory.

Preprocessing directives

The method for locating includable source files (3.8.2)

Includable source files specified via the `#include <filename>` mechanism are searched for in the path specified in the MCC_INCLUDE environment variable. The MCC_INCLUDE environment variable contains a semi-colon delimited list of directories to search.

The support for quoted names for includable source files (3.8.2)

Includable source files specified via the `#include "filename"` mechanism are searched for in the current directory and then in the path specified in the MCC_INCLUDE environment variable. The MCC_INCLUDE environment variable contains a semi-colon delimited list of directories to search.

The behavior on each recognized #pragma directive (3.8.6)

Each #pragma directive is listed in Chapter 3.

Chapter 8. Libraries

1.0 Introduction

This chapter documents functions that are in libraries and pre-compiled object files that can be included in an application. The source code for all of these functions is included with MPLAB-C17 in the \MCC\SRC directory. See the "MPASM User's Guide with MPLINK and MPLIB" for more information about libraries.

1.1 Highlights

This chapter consists of these sections:

- MPLAB-C17 Library Functions and Pre-Compiled Object Files Overview
    - Hardware, Software, Standard Libraries
    - Math Libraries
    - Interrupt Handler Code
    - Register File Definitions
    - Start Up Code
    - Initialized Data Move Code
- Libraries
    - Hardware Peripheral Library
    - Software Peripheral Library
    - General Software Library
    - Math Library

1.2 MPLAB-C17 Library Functions and Pre-Compiled Object Files Overview

The pre-compiled libraries are included in the \MCC\LIB directory. These can be linked directly into an application with MPLINK. These files were precompiled in the C:\MCC\SRC directory at Microchip. A warning message will be generated by MPLINK if the compiler has been installed in a different location. This warning means that source code from the libraries will not show in the .LST file and can not be stepped through when using MPLAB., since the debug info does not point to the location of the source files for the libraries.

To include the library code in the .LST file and to be able to single step through library functions, use the batch file BUILDALL.BAT in the \MCC\SRC directory to rebuild the files. Then execute the batch file COPY2LIB to copy the newly compiled files into the \MCC\LIB directory.

When building an application, usually one file from each of the following categories will be needed to successfully link.

1.2.1 Hardware, Software, Standard Libraries

Memory Model	PIC17C42A	PIC17C43	PIC17C44	PIC17C756
Small	PMC42AS.LIB	PMC43S.LIB	PMC44S.LIB	PMC756S.LIB
Medium	PMC42AM.LIB	PMC43M.LIB	PMC44M.LIB	PMC756M.LIB
Compact	PMC42AC.LIB	PMC43C.LIB	PMC44C.LIB	PMC756C.LIB
Large	PMC42AL.LIB	PMC43L.LIB	PMC44L.LIB	PMC756L.LIB

These are the main library files as described in Section 2.0, 3.0 and 4.0 of this chapter, and this file should be included by the linker when building a project using any of these library functions described in this chapter except the math libraries listed in Section 5.0. The source code for these libraries is in \MCC\SRC\PMC.

1.3 Pre-Compiled Math Libraries

All processors and memory models:	CMATH17.LIB

This file contains the math libraries. The source files can be found in \MCC\SRC\MATH. This file is the same for all memory models and all PIC17CXXX PICmicros™. See Section 5.0 in this chapter for more information.

1.3.1 Interrupt Handler Code

These pre-compiled object files contain the interrupt code. These may be customized for specific applications. The source code for these pre-compiled objects is in \MCC\SRC\STARTUP.

Memory Model	PIC17C42A	PIC17C43	PIC17C44	PIC17C756
Small	INT42AS.O	INT43S.O	INT44S.O	INT756S.O
Medium	INT42AM.O	INT43M.O	INT44M.O	INT756M.O
Compact	INT42AC.O	INT43C.O	INT44C.O	INT756C.O
Large	INT42AL.O	INT43L.O	INT44L.O	INT756L.O

1.3.2 Register File Definitions

These files contain the PICmicro special function register definitions for each processor supported. These are the same for all memory models. The source code can be found in \MCC\SRC\PROCESSOR

1.3.3 Start Up Code

PIC17C42A	PIC17C43	PIC17C44	PIC17C756
P17C42A.O	P17C43.O	P17C44.O	P17C756.O

These files contain the start up code for the compiler. This code initializes the C software stack, calls the routines in IDATA17.O to initialize data (see below), and jumps to the start of the application function, main(). These files will work for all PIC17CXXX PICmicros. The source code is in \MCC\SRC\STARTUP. If the application uses more than one page (8k) of program memory, the Large model should be used.

Memory Model	
Small	C0S17.O
Large	C0L17.O

1.3.4 Initialized Data Move Code

All processors and memory models:	IDATA17.O

This assembly code copies initialized data from ROM to RAM upon system start up. This code is required if variables are set to a value when they are first defined. This file is the same for all memory models and all PIC17CXXX PICmicros. The source code is in \MCC\SRC\STARTUP.

Here is an example of data that will need to be initialized on system startup:

```
int my_data = 0x1234;
unsigned char my_char = "a";
```

To avoid the overhead of this initialization code, set variable values at run time:

```
     int my_data;
     unsigned char my_char;
Void main (void)
     …
     …
     my_data = 0x1234;
     my_char = "a";
     …
     …
```

2.0 Hardware Peripheral Library

2.1 A/D Convertor Functions

BusyADC

Device: PIC17C756

Function: Returns the value of the GO bit in the ADCON0 register.

Syntax:
```
#include <adc16.h>
char BusyADC (void);
```

Remarks: This function returns the value of the GO bit in the ADCON0 register. If the value is equal to 1, then the A/D is busy converting. If the value is equal to 0, then the A/D is done converting.

Return Value: This function returns a char with value either 0 (done) or 1 (busy).

Filename: adcbusy.c

See also: None.

CloseADC

Device: PIC17C756

Function: This function disables the A/D convertor.

Syntax:
```
#include <adc16.h>
void CloseADC (void);
```

Remarks: This function first disables the A/D convertor by clearing the ADON bit in the ADCON0 register. It then disables the A/D interrupt by clearing the ADIE bit in the PIE2 register.

Return Value: None.

Filename: adcclose.c

See also: None.

ConvertADC

Device: PIC17C756

Function: Starts an A/D conversion by setting the GO bit in the ADCON0 register.

Syntax:
```
#include <adc16.h>
void ConvertADC (void);
```

Remarks: This function sets the GO bit in the ADCON0 register.

Return Value: None.

Filename: adcconv.c

See also: None.

OpenADC

Device: PIC17C756

Function: Configures the A/D convertor.

Syntax:
```
#include <adc16.h>
void OpenADC (unsigned char config,
unsigned char channel);
```

Remarks: This function resets the A/D related Special Function Registers to the POR state and then configures the clock, interrupts, justification, voltage reference source, number of analog/ digital I/Os, and current channel.

The value of *config* can be a combination of the following values (defined in adc16.h):

A/D Interrupts

ADC_INT_ON	Interrupts ON
ADC_INT_OFF	Interrupts OFF

A/D clock source

ADC_FOSC_8	Fosc/8
ADC_FOSC_32	Fosc/32
ADC_FOSC_64	Fosc/64
ADC_FOSC_RC	Internal RC Oscillator

A/D result justification

ADC_RIGHT_JUST
ADC_LEFT_JUST

A/D voltage reference source

ADC_VREF_EXT	Vref from I/O pins
ADC_VREF_INT	Vref from AVdd pin

A/D analog/digital I/O configuration

ADC_ALL_ANALOG	All channels analog
ADC_ALL_DIGITAL	All channels digital
ADC_11ANA_1DIG	11 analog, 1 digital
ADC_10ANA_2DIG	10 analog, 2 digital
ADC_9ANA_3DIG	9 analog, 3 digital
ADC_8ANA_4DIG	8 analog, 4 digital
ADC_6ANA_6DIG	6 analog, 6 digital
ADC_4ANA_8DIG	4 analog, 8 digital

The value of *channel* can be one of the following values (defined in adc16.h):

ADC_CH0	Channel 0
ADC_CH1	Channel 1
ADC_CH2	Channel 2
ADC_CH3	Channel 3
ADC_CH4	Channel 4
ADC_CH5	Channel 5
ADC_CH6	Channel 6
ADC_CH7	Channel 7
ADC_CH8	Channel 8
ADC_CH9	Channel 9
ADC_CH10	Channel 10
ADC_CH11	Channel 11

Return Value: None.

Filename: adcopen.c

See also: None.

Code Example:

```
#include <p17c756.h>
#include <adc16.h>
#include <stdlib.h>
#include <delays.h>
#include <usart16.h>
   void main(void)
   {
      int result;
      char str[7];
      // configure A/D convertor
      OpenADC(ADC_INT_OFF&ADC_FOSC_32&ADC_
              RIGHT_JUST&ADC_VREF_INT&ADC_
               ALL_ANALOG,ADC_CH0);
      // configure USART
      OpenUSART1(USART_TX_INT_
                 OFF&USART_RX_
                 INT_OFF&USART_ASYNCH_
                 MODE&USART_EIGHT_
                  BIT&USART_CONT_RX);
      Delay10TCYx(5);      // Delay for 50TCY
      ConvertADC();        // Start Conversion
      while(BusyADC());    // Done Converting?
      result = ReadADC();  // read result
      itoa(result,str);    // convert to string
      putsUSART1(str);     // Write string to USART
      CloseADC();          // Close Modules
```

```
    CloseUSART1();
    return;
}
```

ReadADC

Device: PIC17C756

Function: Reads the result of an A/D conversion.

Syntax:
```
#include <adc16.h>
int ReadADC (void);
```

Remarks: This function reads the 16-bit result of an A/D conversion.

Return Value: This function returns the 16-bit signed result of the A/D conversion. If the `ADFM` bit in `ADCON1` is set, then the result is always right justified leaving the MSbs cleared. If the `ADFM` bit is cleared, then the result is left justified where the LSbs are cleared.

Filename: adcread.c

See also: None

SetChanADC

Device: PIC17C756

Function: Selects a specific A/D channel.

Syntax:
```
#include <adc16.h>
void SetChanADC (unsigned char channel);
```

Remarks: This function first clears the channel select bits in the `ADCON0` register, which selects channel 0. It then ORs the value *channel* with `ADCON0` register.

The value of *channel* can be one of the following values (defined in adc16.h):

ADC_CH0	Channel 0
ADC_CH1	Channel 1
ADC_CH2	Channel 2
ADC_CH3	Channel 3
ADC_CH4	Channel 4
ADC_CH5	Channel 5
ADC_CH6	Channel 6
ADC_CH7	Channel 7
ADC_CH8	Channel 8
ADC_CH9	Channel 9
ADC_CH10	Channel 10
ADC_CH11	Channel 11

Return Value: None.

Filename: adcset.c

2.2 Input Capture Functions

CloseCapture1
CloseCapture2
CloseCapture3
CloseCapture4

Device: CloseCapture1 - PIC17C4X, PIC17C756
CloseCapture2 - PIC17C4X, PIC17C756
CloseCapture3 - PIC17C756
CloseCapture4 - PIC17C756

Function: This function disables the specified input capture.

Syntax:
```
#include <captur16.h>
void CloseCapture1 (void);
void CloseCapture2 (void);
void CloseCapture3 (void);
void CloseCapture4 (void);
```

Remarks: This function simply disables the interrupt of the specified input capture.

Return Value: None.

Filename: cp1close.c
cp2close.c
cp3close.c
cp4close.c

See also: None.

OpenCapture1
OpenCapture2
OpenCapture3
OpenCapture4

Device: OpenCapture1 - PIC17C4X, PIC17C756
OpenCapture2 - PIC17C4X, PIC17C756
OpenCapture3 - PIC17C756
OpenCapture4 - PIC17C756

Function: This function configures the specified input capture.

Syntax:
```
#include <captur16.h>
void OpenCapture1 (unsigned char config);
void OpenCapture2 (unsigned char config);
void OpenCapture3 (unsigned char config);
void OpenCapture4 (unsigned char config);
```

Remarks: This function first resets the capture module to the POR state and then configures the specified input capture for edge detection, i.e., every falling edge, every rising edge, every fourth rising edge, or every sixteenth rising edge.

Capture1 has the ability to become a period register for Timer3.

The value of *config* can be a combination of the following values (defined in captur16.h):

All OpenCapture functions
- CAPTURE_INT_ON Interrupts ON
- CAPTURE_INT_OFF Interrupts OFF

OpenCapture1
- C1_EVERY_FALL_EDGE
- C1_EVERY_RISE_EDGE
- C1_EVERY_4_RISE_EDGE
- C1_EVERY_16_RISE_EDGE
- CAPTURE1_PERIOD
- CAPTURE1_CAPTURE

OpenCapture2
- C2_EVERY_FALL_EDGE
- C2_EVERY_RISE_EDGE
- C2_EVERY_4_RISE_EDGE
- C2_EVERY_16_RISE_EDGE

OpenCapture3
- C3_EVERY_FALL_EDGE
- C3_EVERY_RISE_EDGE
- C3_EVERY_4_RISE_EDGE
- C3_EVERY_16_RISE_EDGE

OpenCapture4
- C4_EVERY_FALL_EDGE
- C4_EVERY_RISE_EDGE
- C4_EVERY_4_RISE_EDGE
- C4_EVERY_16_RISE_EDGE

The capture functions use a structure to indicate overflow status of each of the capture modules. This structure is called CapStatus and has the following bit fields:

```
    struct capstatus
{
    unsigned Cap1OVF:1;
    unsigned Cap2OVF:1;
    unsigned Cap3OVF:1;
```



```
        unsigned Cap4OVF:1;
        unsigned :4;
     } CapStatus;
```

In addition to opening the capture, Timer3 must also be opened with an **OpenTimer3** (...) statement before any of the captures will operate.

Return Value: None.

Filename: cp1open.c
cp2open.c
cp3open.c
cpopen4.c

See also: **Timer3.**

Code Example:

```
#include <p17c756.h>
#include <captur16.h>
#include <timers16.h>
#include <usart16.h>
   void main(void)
   {
      unsigned int result;
      char str[7];
      // Configure Capture1
      OpenCapture1(C1_EVERY_4_RISE_EDGE
                   &CAPTURE1_CAPTURE);
      // Configure Timer3
      OpenTimer3(TIMER_INT_OFF&T3_SOURCE_INT);
      // Configure USART
      OpenUSART1(USART_TX_INT_OFF&USART_RX_
                 INT_OFF&USART_ASYNCH_MODE&
                 USART_EIGHT_BIT&USART_CONT_RX);
   while(!PIR1bits.CA1IF);       // Wait for event
   result = ReadCapture1();      // read result
   uitoa(result,str);// convert to string
   if(!CapStatus.Cap1OVF)
   {
      putsUSART1(str);           // write string
      CloseCapture1();           // to USART
   }
CloseTimer3();
CloseUSART1();
return;
}
```

ReadCapture1
ReadCapture2
ReadCapture3
ReadCapture4

Device: ReadCapture1 - PIC17C4X, PIC17C756
ReadCapture2 - PIC17C4X, PIC17C756
ReadCapture3 - PIC17C756
ReadCapture4 - PIC17C756

Function: Reads the result of a capture event from the specified input capture.

Syntax:

```
#include <captur16.h>
unsigned int ReadCapture1 (void);
unsigned int ReadCapture2 (void);
unsigned int ReadCapture3 (void);
unsigned int ReadCapture4 (void);
```

Remarks: This function reads the value of the respective input capture SFRs.
Capture1: `CA1L,CA1H`
Capture2: `CA2L,CA2H`
Capture3: `CA3L,CA3H`
Capture4: `CA4L,CA4H`

Return Value: This function returns the result of the capture event. The value is a 16-bit unsigned integer.

Filename: cap1read.c
cap2read.c
cap3read.c
cap4read.c

See also: None.

2.3 I²C Functions

AckI2C

Device: PIC17C756

Function: Generates I²C bus Acknowledge condition.

Syntax:
```
#include <i2c16.h>
void AckI2C (void);
```

Remarks: This function generates an I²C bus Acknowledge condition.

Return Value: None.

Filename: acki2c.c

See also: None.

CloseI2C

Device: PIC17C756

Function: Disables the SSP module.

Syntax:
```
#include <i2c16.h>
void CloseI2C (void);
```

Remarks: Pin I/O returns under control Port register settings.

Return Value: None.

Filename: closei2c.c

See also: None.

DataRdyI2C

Device: PIC17C756

Function: Provides status back to user if the SSPBUF register contains data.

Syntax:
```
#include <i2c16.h>
unsigned char DataRdyI2C (void);
```

Remarks: Determines if there is a byte to be read from the SPBUF register.

Return Value: This function returns 1 if there is data in the SSPBUF register else returns 0 which indicates no data in SSPBUF register.

Filename: dtrdyi2c.c

See also: None.

getsI2C

Device: PIC17C756

Function: This function is used to write a predetermined data string length to the I^2C bus.

Syntax:
```
#include <i2c16.h>
unsigned char getsI2C (unsigned char far
*rdptr,unsigned char length);
```

Remarks: **Master I^2C mode:** This routine writes a predefined data string length to the I^2C bus. Each byte is retrieved via a call to the **getcI2C** function. The actual called function body is termed **ReadI2C**. **ReadI2C** and **getcI2C** refer to the same function via a `#define` statement in the i2c16.h file.

Slave I^2C mode: This routine writes a predefined data string length to the I^2C bus. Each byte is retrieved by reading the `SSPBUF` register. There is a time-out period which can be adjusted so as to prevent the slave from waiting forever for data reception.

Return Value: **Master I^2C mode:** This function returns 0 if all bytes have been sent.

Slave I^2C mode: This function returns -1 if the slave device timed-out waiting for a data byte else it returns 0 if the master I^2C device sent a Not Ack condition.

Filename: getsi2c.c

See also: ReadI2C

IdleI2C

Device: PIC17C756

Function: Generates wait condition until I^2C bus is idle.

Syntax:
```
#include <i2c16.h>
```
void IdleI2C (void);

Remarks: This function checks the `R/W` bit of the `SSPSTAT` register and the `SEN, RSEN, PEN, RCEN` and `ACKEN` bits of the `SSPCON2` register. When the state of any of these bits is a logic 1 the function loops on itself. When all of these bits are clear the function terminates and returns to the calling function. The **IdleI2C** function is required since the hardware I^2C peripheral does not allow for spooling of bus sequences/actions. The I^2C

peripheral must be in an idle state before any I^{2}C operation can be initiated or a bus collision will be generated.

Return Value: None.

Filename: idlei2c.c

See also: None.

NotAckI2C

Device: PIC17C756

Function: Generates I^{2}C bus Not Acknowledge condition.

Syntax:
```
#include <i2c16.h>
void NotAckI2C (void);
```

Remarks: This function generates an I^{2}C bus *Not Acknowledge* condition.

Filename: noacki2c.c

Return Value: None.

See also: None.

OpenI2C

Device: PIC17C756

Function: Configures the SSP module.

Syntax:
```
#include <i2c16.h>
void OpenI2C (unsigned char sync_mode,
unsigned char slew);
```

Remarks: OpenI2C resets the SSP module to the POR state and then configures the module for master/slave mode and slew rate enable/disable.

The value of function parameter *sync_mode* can be one of the following values defined in i2c16.h:

SLAVE_7	I2C Slave mode, 7-bit address
SLAVE_10	I2C Slave mode, 10-bit address
MASTER	I2C Master mode

The value of function parameter *slew* can be one of the following values defined in i2c16.h:

SLEW_OFF	Slew rate disabled for 100kHz mode
SLEW_ON	Slew rate enabled for 400kHz mode

Return Value: None.

Filename: openi2c.c

See also: None.

CODE EXAMPLES:

The following are simple code examples illustrating the SSP module configured for I^2C master communication. The routines illustrate I^2C communications with a Microchip 24LC01B I^2C EE Memory Device. In all the examples provided no error checking utilizing the function return value is implemented.

The basic I^2C routines for the hardware I^2C module of the PIC17C756 such as StartI2C, StopI2C, AckI2C, NotAckI2C, RestartI2C, putcI2C, getcI2C, putsI2C, getsI2C, etc., are utilized within the specialized EE I^2C routines such as EESequentialRead or EEPageWrite.

```
   #include "p17cxx.h"
   #include "i2c16.h"
   // FUNCTION PROTOTYPES
   void main(void);
   // POINTERS and ARRAYS
     unsigned char arraywr[] = {1,2,3,4,5,6,7,8,0};
     //24LC01B page write
   // unsigned char arraywr[] = {1,2,3,4,5,6,7,8,9,10,
   //                            11,12,13,14,15,16,0};
   //                            24LC04B page write
   unsigned char far *wrptr = arraywr;
   unsigned char arrayrd[80];
   unsigned char far *rdptr = arrayrd;
   unsigned char temp;

   //*****************************************************
   #pragma code _main=0x100
   void main(void)
   {
     OpenI2C(MASTER, SLEW_ON);//initialize I2C module
     SSPADD = 9;            //400Khz Baud clock(9)
@16MHz
                           //100khz Baud clock(39) @16MHz
     temp = 0;
     while(1)
{
       temp = EEByteWrite(0xA0, 0x30, 0xA5);
       temp = EEAckPolling(0xA0);
       temp = EECurrentAddRead(0xA1);
       temp = EEPageWrite(0xA0, 0x70, wrptr);
       temp = EEAckPolling(0xA0);
       temp = EESequentialRead(0xA0, 0x70, rdptr, 15);
       temp = EERandomRead(0xA0,0x30);
       CloseI2C();
   }
}
```

putsI2C

Device: PIC17C756

Function: This function is used to write out a data string to the I^2C bus.

Syntax:

```
#include <i2c16.h>
unsigned char putsI2C (unsigned char far
*wrptr);
```

Remarks: **Master I^2C mode:** This routine writes a data string to the I^2C bus until a null character is reached. Each byte is written via a call to the putcI2C function. The actual called function body is termed **WriteI2C**. **WriteI2C** and **putcI2C** refer to the same function via a #define statement in the i2c16.h file.

Slave I^2C mode: This routine writes a string out to the I^2C bus until a null character is reached. Each byte is placed directly in the SSPBUF register and the **putcI2C** routine is not called.

Return Value: **Master I^2C Mode:** This function returns 1 if the slave I^2C device responded with a *Not Ack* which terminated the data transfer. The function returns 0 if the null character was reached in the data string.

Slave I^2C mode: This function returns -1 if the master I^2C device responded with a *Not Ack* which terminated the data transfer. The function returns 0 if the null character was reached in the data string

Filename: putsi2c.c

See also: WriteI2C

ReadI2C

Device: PIC17C756

Function: This function is used to read a single byte from the I^2C bus.

Syntax:

```
#include <i2c16.h>
unsigned char ReadI2C (void);
```

Remarks: This function reads in a single byte from the I^2C bus.

Return Value: The return value is the data byte read from the I^2C bus.

Filename: readi2c.c

See also: **getsI2C** ;

RestartI2C

Device: PIC17C756

Function: Generates I^2C bus restart condition.

Syntax:
```
#include <i2c16.h>
unsigned char RestartI2C (void);
```

Remarks: This function generates an I^2C bus restart condition.

Return Value: This function returns -1 if there was a bus collision error or returns 0 if the bus restart condition completed without error.

Filename: rstrti2c.c

See also: None.

StartI2C

Device: PIC17C756

Function: Generates I^2C bus start condition.

Syntax:
```
#include <i2c16.h>
unsigned char StartI2C (void);
```

Remarks: This function generates a I^2C bus start condition.

Return Value: This function returns -1 if there was a bus collision error or returns 0 if the bus start condition completed without error.

Filename: starti2c.c

See also: None.

StopI2C

Device: PIC17C756

Function: Generates I^2C bus stop condition.

Syntax:
```
#include <i2c16.h>
unsigned char StopI2C (void);
```

Remarks: This function generates an I^2C bus stop condition.

Return Value: This function returns -1 if there was a bus collision error or returns 0 if the bus stop condition completed without error.

Filename: stopi2c.c

See also: None.

WriteI2C

Device: PIC17C756

Function: This function is used to write out a single data byte to the I^2C
bus device.

Syntax:
```
#include <i2c16.h>
unsigned char WriteI2C (unsigned char data_out);
```

Remarks: This function writes out a single data byte to the I^2C bus device.

Return Value: This function returns -1 if there was a write collision else it returns a 0.

Filename: writei2c.c

See also: **putsI2C**

> **Note:** The routines to follow are specialized and specific to EE I^2C memory devices such as, but not limited to, the Microchip 24LC01B. Each of the routines depicted below utilize the previous basic 'C' routines in a composite standalone function.

EEAckPolling

Device: PIC17C756

Function: This function is used to generate the acknowledge polling sequence for Microchip EE I^2C memory devices.

Syntax:
```
#include <i2c16.h> unsigned char
EEAckPolling (unsigned char
control);
```

Remarks: This function is used to generate the acknowledge polling sequence for Microchip EE I^2C memory devices. This routine can be used for I^2C EE memory device which utilize acknowledge polling.

Return Value: The return value is -1 if there bus collision error else return 0.

File name: i2ceeap.c

See also: None.

EEByteWrite

Device: PIC17C756

Function: This function is used to write a single byte to the I^{2}C bus.

Syntax:
```
#include <i2c16.h>
unsigned char EEByteWrite (unsigned char
control,unsigned char address, unsigned
char data);
```

Remarks: This function writes a single data byte to the I^{2}C bus. This routine can be used for any Microchip I^{2}C EE memory device which requires only 1 byte of address information.

Return Value: The return value is -1 if there was a bus collision error, -2 if there was a not ack error else returns 0 if there were no errors.

File name: i2ceebw.c

See also: None.

EECurrentAddRead

Device: PIC17C756

Function: This function is used to read a single byte from the I^{2}C bus.

Syntax:
```
#include <i2c16.h>
unsigned char EECurrentAddRead (unsigned
char control);
```

Remarks: This function reads in a single byte from the I^{2}C bus. The address location of the data to read is that of the current pointer within the I^{2}C EE device. The memory device contains an address counter that maintains the address of the last word accessed, incremented by one.

Return Value: The return value is -1 if there was a bus collision error, -2 if there was a not ack error else returns the contents of the SSPBUF register.

File name: i2ceecar.c

See also: EERandomRead

EEPageWrite

Device: PIC17C756

Function: This function is used to write a string of data to the I^2C EEb device.

Syntax:

```
#include <i2c16.h>
unsigned char EEPageWrite (unsigned char
control, unsigned char address, unsigned
char far *wrptr);
```

Remarks: This function writes a predetermined string length of data to the I^2C EE memory device. The length of the data string to read is passed as a function parameter.

Return Value: The return value is -1 if there was a bus collision error, -2 if there was a not ack error else returns 0 if there were no errors.

File name: i2ceepw.c

See also: None.

EERandomRead

Device: PIC17C756

Function: This function is used to read a single byte from the I^2C bus.

Syntax:

```
#include <i2c16.h>
unsigned char EERandomRead (unsigned char
control, unsigned char address);
```

Remarks: This function reads in a single byte from the I^2C bus. The routine can be used for Microchip I^2C EE memory devices which only require 1 byte of address information.

Return Value: The return value is -1 if there was a bus collision error, -2 if there was a not ack error else returns the contents of the SSPBUF register.

File name: i2ceerr.c

See also: None.

EESequentialRead

Device: PIC17C756

Function: This function is used to read in a string of data from the I^2C bus.

Syntax:

```
#include <i2c16.h>
unsigned char EESequentialRead (unsigned
char control, unsigned char address,
unsigned char
far *rdptr, unsigned char length);
```

Remarks: This function reads in a predefined string length of data from the I^2C bus. The routine can be used for Microchip I^2C EE memory devices which only require 1 byte of address information. The length of the data string to read in is passed as a function parameter. The function parameter 'control' is the defining address of the I^2C memory device.

Return Value: The return value is -1 if there was a bus collision error, -2 if there was a not ack error else returns 0 if there were no errors.

File name: i2ceesr.c

See also: None.

2.4 Interrupt Functions

Disable

Device: PIC17C4X, PIC17C756

Function: Disables global interrupts.

Syntax:
```
#include <int16.h>
void Disable (void);
```

Remarks: This function disables global interrupts by setting the `GLINTD` bit of the `CPUSTA` register.

Return Value: None.

Filename: disable.c

See also: None.

Enable

Device: PIC17C4X, PIC17C756

Function: Enables global interrupts.

Syntax:
```
#include <int16.h>
void Enable (void);
```

Remarks: This function enables global interrupts by clearing the `GLINTD` bit of the `CPUSTA` register.

Return Value: None.

Filename: enable.c

See also: None.

2.5 I/O Port Functions

ClosePORTB

Device: PIC17C4X, PIC17C756

Function: Disables the interrupts and internal pull-up resistors for `PORTB.`

Syntax:

```
#include <portb16.h>
void ClosePORTB (void);
```

Remarks: This function disables the `PORTB` interrupt on change by clearing the `RBIE` bit in the `PIE` register. It also disables the internal pull-up resistors by clearing the `NOT_RBPU` bit in the `PORTA` register.

Return Value: None.

Filename: pbclose.c

See also: None.

CloseRA0INT

Device: PIC17C4X, PIC17C756

Function: Disables the RA0/INT pin interrupt.

Syntax:

```
#include <int16.h>
void CloseRA0INT (void);
```

Remarks: This function disables the RA0/INT pin interrupt by clearing the `INTE` bit in the `INTSTA` register.

Return Value: None.

Filename: ra0close.c

See also: None.

DisablePullups

Device: PIC17C4X, PIC17C756

Function: Disables the internal pull-up resistors on PORTB.

Syntax:
```
#include <portb16.h>
void DisablePullups (void);
```

Remarks: This function disables the internal pull-up resistors on PORTB by clearing the NOT_RBPU bit in the PORTA register.

Return Value: None.

Filename: pulldis.c

See also: None.

EnablePullups

Device: PIC17C4X, PIC17C756

Function: Enables the internal pull-up resistors on PORTB.

Syntax:
```
#include <portb16.h>
void EnablePullups (void);
```

Remarks: This function enables the internal pull-up resistors on PORTB by setting the NOT_RBPU bit in the PORTA register.

Return Value: None.

Filename: pullen.c

See also: None.

OpenPORTB

Device: PIC17C4X, PIC17C756

Function: Configures the interrupts and internal pull-up resistors on PORTB.

Syntax:
```
#include <portb16.h>
void OpenPORTB (unsigned char config);
```

Remarks: This function configures the interrupts and internal pull-up resistors on PORTB.

The value of *config* can be a combination of the following values (defined in portb16.h):

PORTB_CHANGE_INT_ON	Interrupt ON
PORTB_CHANGE_INT_OFF	Interrupt OFF
PORTB_PULLUPS_ON	pull-up resistors enabled
PORTB_PULLUPS_OFF	pull-up resistors disabled

Return Value: None.

Filename: pbopen.c

See also: None.

OpenRA0INT

Device: PIC17C4X, PIC17C756

Function: Configures the external interrupt pin RA0/INT.

Syntax:
```
#include <int16.h>
void OpenRA0INT (unsigned char config);
```

Remarks: This function configures the RA0/INT pin for external interrupt for interrupt on/off and edge select.

The value of *config* can be a combination of the following values (defined in int16.h):

INT_ON	Interrupt ON
INT_OFF	Interrupt OFF
INT_RISE_EDGE	Interrupt on rising edge
INT_FALL_EDGE	Interrupt on falling edge

Return Value: None.

Filename: ra0open.c

See also: None.

2.6 Microwire® Functions

CloseMwire

Device: PIC17C756

Function: Disables the SSP module.

Syntax:
```
#include <mwire16.h>
void CloseMwire (void);
```

Remarks: Pin I/O returns under control `DDRx` and `PORTx` register settings.

Return Value: None.

Filename: closmwir.c

See also: None.

DataRdyMwire

Device: PIC17C756

Function: Provides status back to user if the Microwire device has completed the internal write cycle.

Syntax:
```
#include <mwire16.h>
unsigned char DataRdyMwire (void);
```

Remarks: Determines if Microwire device is ready.

Return Value: This function returns 1 if the Microwire device is ready else returns 0 which indicates that the internal write cycle is not complete or there could be a bus error.

Filename: drdymwir.c

See also: None.

getsMwire

Device: PIC17C756

Function: This routine reads a string from the Microwire device.

Syntax:
```
#include <mwire16.h>
void getsMwire (unsigned char far *rdptr,
unsigned char length);
```

Remarks: This function is used to read a predetermined length of data from a Microwire device. User must first issue start bit, opcode and address before reading a data string.

Return Value: None.

Filename: getsmwir.c

See also: None.

OpenMwire

Device: PIC17C756

Function: Configures the SSP module.

Syntax:
```
#include <mwire16.h>
void OpenMwire (unsigned char sync_mode);
```

Remarks: OpenMwire resets the SSP module to the POR state and then configures the module for Microwire communications.

The value of the function parameter *sync_mode* can be one of the following values defined in mwire16.h:

FOSC_4	clock = Fosc/4
FOSC_16	clock = Fosc/16
FOSC_64	clock = Fosc/64
FOSC_TMR2	clock = TMR2 output/2

Return Value: None.

Filename: openmwir.c

See also: None.

CODE EXAMPLES:

The following are simple code examples illustrating the SSP module communicating with a Microchip 93LC66 Microwire EE Memory Device. In all the examples provided no error checking utilizing the value returned from a function is implemented.

```
#include "p17c756.h"
#include "mwire16.h"

// 93LC66 x 8
```



```
// FUNCTION PROTOTYPES
void main(void);
void ew_enable(void);
void erase_all(void);
void busy_poll(void);
void write_all(unsigned char data);
void byte_read(unsigned char address);
void read_mult(unsigned char address, unsigned char
far *rdptr, unsigned char length);
void write_byte(unsigned char address, unsigned char
data);
unsigned char arrayrd[20];
unsigned char far *rdptr = arrayrd;
unsigned char var;

// DEFINE 93LC66 MACROS
#define   READ      0x0C
#define   WRITE     0x0A
#define   ERASE     0x0E
#define   EWEN1     0x09
#define   EWEN2     0x80
#define   ERAL1     0x09
#define   ERAL2     0x00
#define   WRAL1     0x08
#define   WRAL2     0x80
#define   EWDS1     0x08
#define   EWDS2     0x00
#define W_CS  PORTAbits.RA2
#pragma code _main=0x100
void main(void)
{
   W_CS = 0;                //ensure CS is negated
   OpenMwire(FOSC_16);      //enable SSP perpiheral
   ew_enable();             //send erase/write enable
   write_byte(0x13, 0x34); //write byte
(address,data)
   busy_poll();
   Nop();
   byte_read(0x13);         //read single byte
(address)
   read_mult(0x10, rdptr, 10); //read multiple bytes
   erase_all();             //erase entire array
   CloseMwire();            //disable SSP peripheral
   }

void busy_poll(void)
{
   W_CS = 1;
   do
```

```
        {
           var = DataRdyMwire();//test for busy/ready
        }  while(var != 0);
        W_CS = 0;
}
     void write_byte(unsigned char address, unsigned char
     data)
     {
        W_CS = 1;
        putcMwire(WRITE);//write command
        putcMwire(address);//address
        putcMwire(data);//write single byte
        W_CS = 0;
     }

     void byte_read(unsigned char address)
     {
        W_CS = 1;
        getcMwire(READ,address);//read one byte
        W_CS = 0;
     }

     void read_mult(unsigned char address, unsigned char
     far *rdptr, unsigned char length)
     {
        W_CS = 1;
        putcMwire(READ);        //read command
        putcMwire(address);     //address (A7 - A0)
        getsMwire(rdptr, length);//read multiple bytes
     W_CS = 0;
     }

     void ew_enable(void)
     {
        W_CS = 1;//assert chip select
        putcMwire(EWEN1);//enable write command byte 1
        putcMwire(EWEN2);//enable write command byte 2
        W_CS = 0;//negate chip select
     }

     void erase_all(void)
     {
        W_CS = 1;
        putcMwire(ERAL1);//erase all command byte 1
        putcMwire(ERAL2);//erase all command byte 2
        W_CS = 0;
     }
```


ReadMwire

Device: PIC17C756

Function: This function is used to read a single byte from a Microwire device.

Syntax:
```
#include <mwire16.h>
unsigned char ReadMwire (unsigned char
high_byte, unsigned char low_byte);
```

Remarks: This function reads in a single byte from a Microwire device. The start bit, opcode and address compose the high and low bytes passed into this function.

Return Value: The return value is the data byte read from the Microwire device.

Filename: readmwir.c

See also: None.

WriteMwire

Device: PIC17C756

Function: This function is used to write out a single data byte.

Syntax:
```
#include <mwire16.h>
unsigned char WriteMwire (unsigned char
data_out);
```

Remarks: This function writes out single data byte to a Microwire device utilizing the SSP module.

Return Value: This function returns -1 if there was a write collision else it returns a 0.

Filename: writmwir.c

See also: None.

2.7 Pulse Width Modulation Functions

ClosePWM1
ClosePWM2
ClosePWM3

Device: ClosePWM1 - PIC17C4X, PIC17C756
ClosePWM2 - PIC17C4X, PIC17C756
ClosePWM3 - PIC17C756

Function: This function disables the specified PWM channel.

Syntax:
```
#include <pwm16.h>
void ClosePWM1 (void);
void ClosePWM2 (void);
void ClosePWM3 (void);
```

Remarks: This function simply disables the specified PWM channel by clearing the `PWMxON` bit in the respective `TCON2` or `TCON3`
registers.

Return Value: None.

Filename: pw1close.c
pw2close.c
pw3close.c

See also: None.

OpenPWM1
OpenPWM2
OpenPWM3

Device: OpenPWM1 - PIC17C4X, PIC17C756
OpenPWM2 - PIC17C4X, PIC17C756
OpenPWM3 - PIC17C756

Function: Configures the specified PWM channel.

Syntax:
```
#include <pwm16.h>
void OpenPWM1 (char period);
void OpenPWM2 (unsigned char config,
               char period);
void OpenPWM3 (unsigned char config,
               char period);
```

Remarks: This function configures the specified PWM channel for period and for time base. PWM1 uses only Timer1. PWM2 and PWM3 can use either Timer1 or Timer2. Timer1 and Timer2 must be set up as individual 8-bit timers for PWM mode to work correctly.

The value of *period* can be any value from 0x00 to 0xff. This value determines the PWM frequency by using the following formula:

Period1 = [(PR1)+1] x 4 x Tosc

Period2 = [(PR1)+1] x 4 x Tosc
= [(PR2)+1] x 4 x Tosc

Period3 = [(PR1)+1] x 4 x Tosc
= [(PR2)+1] x 4 x Tosc

The value of *config* can be one of the following values (defined in captur16.h):

OpenPWM2
OpenPWM3
T1_SOURCETimer1 is clock source
T2_SOURCETimer2 is clock source

In addition to opening the PWM, Timer1 or Timer2 must also be opened with an **OpenTimer1(...)** or **OpenTimer2(...)** statement before any of the PWMs will operate.

Return Value: None.

Filename: pw1open.c
pw2open.c
pw3open.c

See also: Timer1, Timer2.

Code Example:

```
#include <p17c756.h>
#include <pwm16.h>
#include <timers16.h>
void main(void)
{
   int i;
SetDCPWM2(0);                      //set duty cycle
OpenPWM2(T1_SOURCE,0xff);          //open PW2
OpenTimer1(TIMER_INT_OFF&T1_SOURCE_
            INT&T1_T2_8BIT);       //open timer
for(i=0;i<1024;i++)
{
   while(!PIR1bits.TMR1IF);
   PIR1bits.TMR1IF = 0;
   SetDCPWM2(i);                   //slowly increment
                                     duty cycle
}
ClosePWM2();                       //close modules
CloseTimer1();
return;
}
```

SetDCPWM1
SetDCPWM2
SetDCPWM3

Device: SetDCPWM1 - PIC17C4X, PIC17C756
SetDCPWM2 - PIC17C4X, PIC17C756
SetDCPWM3 - PIC17C756

Function: Writes a new dutycycle value to the specified PWM channel dutycycle registers.

Syntax:

```
#include <pwm16.h>
void SetDCPWM1 (unsigned int dutycycle);
void SetDCPWM2 (unsigned int dutycycle);
void SetDCPWM3 (unsigned int dutycycle);
```

Remarks: This function writes the new value for *dutycycle* to the specified PWM channel dutycycle registers.

PWM1: `PW1DCL,PW1DCH`

PWM2: `PW2DCL,PW2DCH`

PWM3: `PW3DCL,PW3DCH`

The value of *dutycycle* can be any 10-bit number. Only the lower 10-bits of *dutycycle* are written into the dutycycle registers.

The dutycycle, or more specifically the high time of the PWM waveform, can be calculated from the following formula:

PWMx Dutycycle = (DCx<9:0>) x Tosc

where DCx<9:0> is the 10-bit value from the `PWxDCH:PWxDCL` registers.

The maximum resolution of the PWM waveform can be calculated from the period using the following formula:

Resolution (bits) = log(Fosc/Fpwm) / log(2)

Return Value: None.

Filename: pw1set.c
pw2set.c
pw3set.c

See also: None.

2.8 Reset Functions

isBOR

Device: PIC17C756

Function: Detects a reset condition due to the Brown-out Reset circuit.

Syntax:
```
#include <reset16.h>
char isBOR (void);
```

Remarks: This function detects if the microcontroller was reset due to the Brown-out Reset circuit. This condition is indicated by the following status bits:

$\overline{POR}$ = 1
$\overline{BOR}$ = 0
$\overline{TO}$ = don’t care
$\overline{PD}$ = don’t care

Include the statement `#define BOR_ENABLED` in the header file reset16.h. After the definitions have been made, compile the reset16.c file. Refer to Chapter 2 of this manual (DS51112A) for information on compilers. Refer to the MPASM User’s Guide with MPLINK and MPLIB (DS33014F) for information on linking.

Return Value: This function returns 1 if the reset was due to the Brown- out Reset circuit, otherwise 0 is returned.

Filename: reset16.c

See also: None.

isMCLR

Device: PIC17C756

Function: Detects if a MCLR reset during normal operation occurred.

Syntax:
```
#include <reset16.h>
char isMCLR (void);
```

Remarks: This function detects if the microcontroller was reset via the MCLR pin while in normal operation. This situation is indicated by the following status bits:

$\overline{POR}$ = 1
$\overline{BOR}$ = 1 if Brown-out is enabled
$\overline{TO}$ = 1 if WDT is enabled
$\overline{PD}$ = 1

Return Value: This function returns 1 if the reset was due to MCLR during normal operation, otherwise 0 is returned.

Filename: reset16.c

See also: None.

isPOR

Device: PIC17C4X, PIC17C756

Function: Detects a Power-on Reset condition.

Syntax:

```
#include <reset16.h>
char isPOR (void);
```

Remarks: This function detects if the microcontroller just left a Power-on Reset. This condition is indicated by the following status bits:

PIC17C4X
$\overline{\text{TO}} = 1$
$\overline{\text{PD}} = 1$

This condition also for MCLR reset during normal operation and `CLRWDT` instruction executed

PIC17C756
$\overline{\text{POR}} = 0$
$\overline{\text{BOR}} = 0$
$\overline{\text{TO}} = 1$
$\overline{\text{PD}} = 1$

Return Value: This function returns 1 if the device just left a Power-on Reset, otherwise 0 is returned.

Filename: reset16.c

See also: None.

isWDTTO

Device: PIC17C4X, PIC17C756

Function: Detects a reset condition due to the WDT during normal operation.

Syntax:

```
#include <reset16.h>
char isWDTTO (void);
```

Remarks: This function detects if the microcontroller was reset due to the WDT during normal operation. This condition is indicated by the following status bits:

PIC17C4X
$\overline{\text{TO}} = 0$
$\overline{\text{PD}} = 1$

PIC17C756
POR = 1
BOR = 1
TO = 0
PD = 1

Include the statement #define WDT_ENABLED in the header file reset16.h. After the definitions have been made, compile the reset16.c file. Refer to Chapter 2 of this manual (DS51112) for information on compilers. Refer to the MPASM User's Guide with MPLINK and MPLIB (DS33014F) for information on linking.

Return Value: This function returns 1 if the reset was due to the WDT during normal operation, otherwise 0 is returned.

Filename: reset16.c

See also: None.

isWDTWU

Device: PIC17C4X, PIC17C756

Function: Detects when the WDT wakes up the device from SLEEP.

Syntax:

```
#include <reset16.h>
char isWDTWU (void);
```

Remarks: This function detects if the microcontroller was brought out of SLEEP by the WDT. This condition is indicated by the following status bits:

PIC17C4X
TO = 0
PD = 0

PIC17C756
POR = 1
BOR = 1
TO = 0
PD = 0

Include the statement #define WDT_ENABLED in the header file reset16.h. After the definitions have been made, compile the reset16.c file. Refer to Chapter 2 of this manual (DS51112B) for information on compilers. Refer to the MPASM User's Guide with MPLINK and MPLIB (DS33014F) for information on linking.

Return Value: This function returns 1 if device was brought out of SLEEP by the WDT, otherwise 0 is returned.

Filename: reset16.c

See also: None.

sWU

Device: PIC17C4X, PIC17C756

Function: Detects if the microcontroller was just waken up from `SLEEP` via the MCLR pin or interrupt.

Syntax:

```
#include <reset16.h>
char isWU (void);
```

Remarks: This function detects if the microcontroller was brought out of `SLEEP` by the MCLR pin or an interrupt. This condition is indicated by the following status bits:

PIC17C4X

$\overline{TO}$ = 1
$\overline{PD}$ = 0

PIC17C756

$\overline{POR}$ = 1
$\overline{BOR}$ = 1
$\overline{TO}$ = 1
$\overline{PD}$ = 0

Return Value: This function returns 1 if the device was brought out of `SLEEP` by the MCLR pin or an interrupt, otherwise 0 is returned.

Filename: reset16.c

See also: None.

StatusReset

Device: PIC17C756

Function: Sets the $\overline{POR}$ and $\overline{BOR}$ bits in the `CPUSTA` register.

Syntax:

```
#include <reset16.h>
void StatusReset (void);
```

Remarks: This function sets the `POR` and `BOR` bits in the `CPUSTA` register. These bits must be set in software after a Power-on Reset has occurred.

Return Value: None.

Filename: reset16.c

See also: None.

2.9 i SPI™ Functions

CloseSPI

Device: PIC17C756

Function: Disables the SSP module.

Syntax: #include <spi16.h>
void CloseSPI (void);

Remarks: This function disables the SSP module. Pin I/O returns under the control of the `DDRx` and `PORTx` Registers.

Return Value: None.

Filename: closespi.c

See also: None.

DataRdySPI

Device: PIC17C756

Function: Determines if the `SSPBUF` contains data.

Syntax:

```
#include <spi16.h>
unsigned char DataRdySPI (void);
```

Remarks: This function determines if there is a byte to be read from the `SSPBUF` register.

Return Value: This function returns 1 if there is data in the `SSPBUF` register else returns a 0.

Filename: dtrdyspi.c

See also: None.

getsSPI

Device: PIC17C756

Function: Reads in data string from the SPI bus.

Syntax:

```
#include <spi16.h>
void getsSPI (unsigned char far *rdptr,
unsigned char length);
```

Remarks: This function reads in a predetermined data string length from the SPI bus. The length of the data string to read in is passed as a function parameter. Each byte is retrieved via a call to the **getcSPI** function. The actual called function body is termed **ReadSPI**. **ReadSPI** and **getcSPI** refer to the same function via a `#define` statement in the spi16.h file.

Return Value: None.

Filename: getsspi.c

See also: **ReadSPI**

OpenSPI

Device: PIC17C756

Function: Initializes the SSP module.

Syntax:

```
#include <spi16.h>
void OpenSPI (unsigned char sync_mode, unsigned
char bus_mode, unsigned char smp_phase);
```

Remarks: This function setups the SSP module for use with a SPI bus device.

Return Value: None.

Filename: openspi.c

See also: None.

The value of *sync_mode*, *bus_mode* and *smp_phase* parameters can be one of the following values defined in spi16.h:

sync_mode:

FOSC_4	SPI Master mode, clock = Fosc/4
FOSC_16	SPI Master mode, clock = Fosc/16
FOSC_64	SPI Master mode, clock = Fosc/64
FOSC_TMR2	SPI Master mode, clock = TMR2 output/2
SLV_SSON	SPI Slave mode, /SS pin control enabled
SLV_SSOFF	SPI Slave mode, /SS pin control disabled

bus_mode:

MODE_00	Setting for SPI bus Mode 0,0
MODE_01	Setting for SPI bus Mode 0,1
MODE_10	Setting for SPI bus Mode 1,0
MODE_11	Setting for SPI bus Mode 1,1

smp_phase:

SMPEND	Input data sample at end of data out
SMPMID	Input data sample at middle of data out

CODE EXAMPLE:

The following are simple code examples illustrating the SSP module communicating with a Microchip 24C080 SPI EE Memory Device. In all the examples provided no error checking utilizing the value returned from a function is implemented.

```
#include <p17c756.h>
#include <spi16.h>
// FUNCTION PROTOTYPES
void main(void);
void set_wren(void);
void busy_polling(void);
unsigned char status_read(void);
void status_write(unsigned char data);
void byte_write(unsigned char addhigh, unsigned char
               addlow, unsigned char data);
void page_write(unsigned char addhigh, unsigned char
                addlow, unsigned char far *wrptr);
void array_read(unsigned char addhigh, unsigned char
                addlow, unsigned char far *rdptr,
                unsigned char count);
                unsigned char byte_read
               (unsigned char addhigh,
                unsigned char addlow);
unsigned char arraywr[] = {1,2,3,4,5,6,7,8,9,10,11,
                     12,13,14,15,16,0};
                  //24C040/080/160 page write size
unsigned char far *wrptr = arraywr;
unsigned char arrayrd[32];
unsigned char far *rdptr = arrayrd;
unsigned char var;
#define SPI_CS  PORTAbits.RA2
//**************************************************
#pragma code _main=0x100
void main(void)
{
 SPI_CS = 1;         //ensure SPI memory device Chip
                         Select is reset
 OpenSPI(FOSC_16, MODE_00, SMPEND);
 set_wren();
 status_write(0);
 busy_polling();
 set_wren();
 byte_write(0x00, 0x61, 'E');
 busy_polling();
 var = byte_read(0x00, 0x61);
 set_wren();
 page_write(0x00, 0x30, wrptr);
 busy_polling();
```



```
  array_read(0x00, 0x30, rdptr, 16);
  var = status_read();
  CloseSPI();
  while(1);
}
void set_wren(void)
{
  SPI_CS = 0;             //assert chip select
  var = putcSPI(WREN);    //send write enable command
  SPI_CS = 1;             //negate chip select
}
void page_write (unsigned char addhigh, unsigned char
                 addlow, unsigned char far *wrptr)
{
  SPI_CS = 0;             //assert chip select
  var = putcSPI(WRITE);   //send write command
  var = putcSPI(addhigh); //send high byte of address
  var = putcSPI(addlow);  //send low byte of address
  putsSPI(wrptr);         //send data byte
  SPI_CS = 1;             //negate chip select
}
void array_read (unsigned char addhigh, unsigned char
                 addlow, unsigned char far
                 *rdptr,nsigned char count)
{
  SPI_CS = 0;             //assert chip select
  var = putcSPI(READ);    //send read command
  var = putcSPI(addhigh); //send high byte of address
  var = putcSPI(addlow);  //send low byte of address
  getsSPI(rdptr, count);  //read multiple bytes
  SPI_CS = 1;
}
void byte_write (unsigned char addhigh, unsigned char
                  addlow, unsigned char data)
{
  SPI_CS = 0;             //assert chip select
  var = putcSPI(WRITE);   //send write command
  var = putcSPI(addhigh); //send high byte of address
  var = putcSPI(addlow);  //send low byte of address
  var = putcSPI(data);    //send data byte
  SPI_CS = 1;             //negate chip select
}
unsigned char byte_read (unsigned char addhigh,
                         unsigned
                         char addlow)
{
  SPI_CS = 0;             //assert chip select
  var = putcSPI(READ);    //send read command
  var = putcSPI(addhigh); //send high byte of address
```

```
  var = putcSPI(addlow);  //send low byte of address
  var = getcSPI();        //read single byte
  SPI_CS = 1;
  return (var);
}
unsigned char status_read (void)
{
  SPI_CS = 0;             //assert chip select
  var = putcSPI(RDSR);    //send read status command
var = getcSPI();          //read data byte
  SPI_CS = 1;             //negate chip select
  return (var);
}
void status_write (unsigned char data)
{
  SPI_CS = 0;
  var = putcSPI(WRSR);    //write status command
  var = putcSPI(data);    //status byte to write
  SPI_CS = 1;             //negate chip select
}
void busy_polling (void)
{
  do
     {
     SPI_CS = 0;          //assert chip select
     var = putcSPI(RDSR); //send read status command
     var = fetcSPI();     //read data byte
     SPI_CS = 1;          //negate chip select
  }while (var & 0x01);    //stay in loop until
                             notbusy
}
```

putsSPI

Device: PIC17C756

Function: Writes data string out to the SPI bus.

Syntax:
```
#include <spi16.h>
void putsSPI (unsigned char far *wrptr);
```

Remarks: This function writes out a data string to the SPI bus device. The routine is terminated by reading a null character in the data string.

Return Value: None.

Filename: putsspi.c

See also: None.

ReadSPI

Device: PIC17C756

Function: Reads a single byte from the SSPBUF register.

Syntax:
```
#include <spi16.h>
unsigned char ReadSPI (void);
```

Remarks: This function initiates a SPI bus cycle for the acquisition of a byte of data. **ReadSPI** and **getcSPI** refer to the same function via a `#define` statement in the spi16.h file.

Return Value: This function returns a byte of data read during a SPI read cycle.

Filename: readspi.c

See also: None.

WriteSPI

Device: PIC17C756

Function: Writes a single byte of data out to the SPI bus.

Syntax:
```
#include <spi16.h>
unsigned char WriteSPI (unsigned char
data_out);
```

Remarks: This function writes a single data byte out and then checks for a write collision. **WriteSPI** and **putcSPI** refer to the same function via a `#define` statement in the spi16.h file.

Return Value: This function returns -1 if a write collision occurred else a 0 if no write collision.

Filename: writespi.c

See also: None.

2.10 Timer Functions

CloseTimer0
CloseTimer1
CloseTimer2
CloseTimer3

Device: PIC17C4X, PIC17C756

Function: This function disables the specified timer.

Syntax:

```
#include <timers16.h>
void CloseTimer0 (void);
void CloseTimer1 (void);
void CloseTimer2 (void);
void CloseTimer3 (void);
```

Remarks: This function simply disables the interrupt and the specified timer.

Return Value: None.

Filename: t0close.c
t1close.c
t2close.c
t3close.c

See also: None.

OpenTimer0
OpenTimer1
Opentimer2
OpenTimer3

Device: PIC17C4X, PIC17C756

Function: Configures the specified timer.

Syntax:

```
#include <timers16.h>
void OpenTimer0 (unsigned char config);
void OpenTimer1 (unsigned char config);
void OpenTimer2 (unsigned char config);
void OpenTimer3 (unsigned char config);
```

Remarks: This function configures the specified timer for interrupts, internal/external clock source, prescaler, etc.

Timer0 -> 16-bit
Timer1 -> 8-bit
Timer2 -> 8-bit
Timer3 -> 16-bit

Timer0 has a programmable prescaler from 1:1 to 1:256. Timer1 and Timer2 can be concatenated to be a 16-bit timer.

The value of *config* can be a combination of the following values (defined in timers16.h):

All OpenTimer functions

TIMER_INT_ON	Interrupts ON
TIMER_INT_OFF	Interrupts OFF

OpenTimer0

T0_EDGE_FALL	External clock on falling edge	
T0_EDGE_RISE	External clock on rising edge	
T0_SOURCE_EXT	External clock source (I/O pin)	
T0_SOURCE_INT	Internal clock source (Tosc)	
T0_PS_1_1	Prescale ->	1:1
T0_PS_1_2		1:2
T0_PS_1_4		1:4
T0_PS_1_8		1:8
T0_PS_1_16		1:16
T0_PS_1_32		1:32
T0_PS_1_64		1:64
T0_PS_1_128		1:128
T0_PS_1_256		1:256

OpenTimer1

T1_SOURCE_EXT	External clock source (I/O pin)
T1_SOURCE_INT	Internal clock source (Tosc)
T1_T2_8BIT	Timer1 and Timer2 individual 8-bit timers
T1_T2-16BIT	Timer1 and Timer2 one 16-bit timer

OpenTimer2

T2_SOURCE_EXT	External clock source (I/O pin)
T2_SOURCE_INT	Internal clock source (Tosc)

OpenTimer3

T3_SOURCE_EXT	External clock source (I/O pin)
T3_SOURCE_INT	Internal clock source (Tosc)

Return Value: None.

Filename: t0open.c
t1open.c
t2open.c
3open.c

See also: None.

CODE EXAMPLE:

```
#include <p17c756.h>
#include <timers16.h>
#include <usart16.h>
void main (void)
{
   int result;
   char str[7];
   // configure timer0

OpenTimer0(TIMER_INT_OFF&T0_SOURCE_NT&T0_PS_1_32);
   // configure USART
   OpenUSART1(USART_TX_INT_OFF&USART_RX_
              INT_OFF&USART_ASYNCH_MODE&
               USART_EIGHT_BIT&USART_CONT_RX);
   while(1)
   {
         while(!PORTBbits.RB3);//wait for RB3 high
         result = ReadTimer0();//read timer
         if(result>0xc000)
         break;
         WriteTimer0(0);//write new value
         uitoa(result,str);//convert to string
         putsUSART1(str);//print string
         }
   CloseTimer0();//close modules
   CloseUSART1();
   return;
   }
```

ReadTimer0
ReadTimer1
ReadTimer2
ReadTimer3
ReadTimer1_2

Device: PIC17C4X, PIC17C756

Function: Reads the contents of the specified timer register(s).

Syntax:

```
#include <timers16.h>
unsigned int ReadTimer0 (void);
unsigned char ReadTimer1 (void);
unsigned char ReadTimer2 (void);
unsigned int ReadTimer3 (void);
unsigned int ReadTimer1_2 (void);
```

Remarks: This function reads the value of the respective timer register(s).

Timer0:	`TMR0L,TMR0H`
Timer1:	`TMR1`
Timer2:	`TMR2`
Timer3:	`TMR3L,TMR3H`
Timer1_2:	`TMR2:TMR1`

Return Value: These functions returns the value of the timer register(s) which may be 8-bits or 16-bits.

Timer0:	int (16-bits)
Timer1:	char (8-bits)
Timer2:	char (8-bits)
Timer3:	int (16-bits)
Timer1_2:	int (16-bits)

Filename: t0read.c
t1read.c
t2read.c
t3read.c

See also: None.

WriteTimer0
WriteTimer1
WriteTimer2
WriteTimer3
WriteTimer1_2

Device: PIC17C4X, PIC17C756

Function: Reads the contents of the specified timer register(s).

Syntax:

```
#include <timers16.h>
void WriteTimer0 (unsigned int timer);
void WriteTimer1 (unsigned char timer);
void WriteTimer2 (unsigned char timer);
void WriteTimer3 (unsigned int timer);
void WriteTimer1_2 (unsigned int timer);
```

Remarks: This function writes the value *timer* to the respective timer register(s).

Timer0:	`TMR0L,TMR0H`
Timer1:	`TMR1`
Timer2:	`TMR2`
Timer3:	`TMR3L,TMR3H`
Timer1_2:	`TMR2:TMR1`

These functions write a value to the timer register(s) which may be 8-bits or 16-bits.

Timer0:	int (16-bits)

Timer1:	char (8-bits)
Timer2:	char (8-bits)
Timer3:	int (16-bits)
Timer1_2:	int (16-bits)

Return Value: None.

Filename: t0write.c
t1write.c
t2write.c
t3write.c

See also: None.

2.11 USART Functions

BusyUSART1
BusyUSART2

Device: BusyUSART1: PIC17C4X, PIC17C756
BusyUSART2: PIC17C756

Function: Returns the status of the TRMT flag bit in the TXSTA? register.

Syntax:
```
#include <usart16.h>
char BusyUSART1 (void);
Char BusyUSART2 (void);
```

Remarks: This function returns the status of the TRMT flag bit in the TXSTA? register.

Return Value: If the USART transmitter is busy, a value of 1 is returned. If the USART receiver is idle, then a value of 0 is returned.

Filename: u1busy.c
u2busy.c

See also: None.

CloseUSART1
CloseUSART2

Device: CloseUSART1: PIC17C4X, PIC17C756
CloseUSART2: PIC17C756

Function: Disables the specified USART.

Syntax:
```
#include <usart16.h>
void CloseUSART1 (void);
void CloseUSART2 (void);
```

Remarks: This function disables the specified USARTs interrupts, transmitter, and receiver.

Return Value: None.

Filename: u1close.c
u2close.c

See also: None.

DataRdyUSART1
DataRdyUSART2

Device: DataRdyUSART1: PIC17C4X, PIC17C756
DataRdyUSART2: PIC17C756

Function: Returns the status of the `RCIF` flag bit in the `PIR` register.

Syntax:
```
#include <usart16.h>
char DataRdyUSART1 (void);
char DataRdyUSART2 (void);
```

Remarks: This function returns the status of the `RCIF` flag bit in the `PIR`
register.

Return Value: If data is available, a value of 1 is returned. If data is not available, then a value of 0 is returned.

Filename: u1drdy.c
u2drdy.c

See also: None.

getcUSART1
getcUSART2

Device: getcUSART1: PIC17C4X, PIC17C756
getcUSART2: PIC17C756

Function: Reads one character from the specified USART.

Syntax:
```
#include <usart16.h>
char getcUSART1 (void);
char getcUSART2 (void);
```

Remarks: This function performs the same function as **ReadUSARTx**. Please refer to the description of that function.

Return Value: The next received character from the specified USART.

Filename: #define in usart16.h

See also: ReadUSART1, ReadUSART2.

getsUSART1
getsUSART2

Device: getsUSART1: PIC17C4X, PIC17C756
getsUSART2: PIC17C756

Function: Reads a string of characters until the specified number of characters have been read.

Syntax:
```
#include <usart16.h>
void getsUSART1 (char *buffer, unsigned
                  char len);
void getsUSART2 (char *buffer, unsigned
                  char len);
```

Remarks: This function waits for and reads *len* number of characters out of the specified USART. There is no timeout when waiting for characters to arrive. After *len* characters have been written to the string, a null character is appended to the end of the string.

The value of *buffer* is a pointer to the string where incoming characters are to be stored. The length of this string should be at least *len* + 1.

The value of *len* is limited to the available amount of RAM locations remaining in any one bank - 1. There must be one extra location to store the null character.

Return Value: None.

Filename: u1gets.c
u2gets.c

See also: None.

OpenUSART1
OpenUSART2

Device: OpenUSART1: PIC17C4X, PIC17C756
OpenUSART2: PIC17C756

Function: Configures the specified USART module.

Syntax:
```
#include <usart16.h>
void OpenUSART1 (unsigned char config,
                 char spbrg);
void OpenUSART2 (unsigned char config,
                 char spbrg);
```


Remarks: This function configures the USART module for interrupts, baud rate, sync or async operation, 8- or 9-bit mode, master or slave mode, and single or continuous reception.

The value of *config* can be a combination of the following values (defined in usart16.h):

USART_TX_INT_ON	Transmit interrupt ON
USART_TX_INT_OFF	Transmit interrupt OFF
USART_RX_INT_ON	Receive interrupt ON
USART_RX_INT_OFF	Receive interrupt OFF
USART_ASYNCH_MODE	Asynchronous Mode
USART_SYNCH_MODE	Synchronous Mode
USART_EIGHT_BIT	8-bit transmit/receive
USART_NINE_BIT	9-bit transmit/receive
USART_SYNC_SLAVE	Synchronous slave mode
USART_SYNC_MASTER	Synchronous master mode
USART_SINGLE_RX	Single reception
USART_CONT_RX	Continuous reception

The value of *spbrg* determines the baud rate of the USART. The formulas for baud rate are:

asynchronous mode:	FOSC/(64 (*spbrg* + 1))
synchronous mode:	FOSC/(4 (*spbrg* + 1))

Return Value: None.

Filename: u1open.c
u2open.c

See also: None.

Code Example:

```
#include <p17c756.h>
#include <usart16.h>
void main(void)
{
      // configure USART
      OpenUSART1(USART_TX_INT_OFF&USART_RX_
                 INT_OFF&USART_ASYNCH_MODE&
                 USART_EIGHT_BIT&USART_
                 CONT_RX);
while(1)
{
      while(!PORTAbits.RA0)   //wait for RA0 high
      WriteUSART1(PORTD);     //write value of PORTD
      if(PORTD == 0x80)
      break;
}
```

```
    CloseUSART1();
    return;
}
```

putcUSART1
putcUSART2

Device: putcUSART1: PIC17C4X, PIC17C756
putcUSART2: PIC17C756

Function: Writes one character to the specified USART.

Syntax:
```
#include <usart16.h>
void putcUSART1 (char data);
void putcUSART2 (char data);
```

Remarks: This function performs the same function as **WriteUSARTx**. Please refer to the description of that function.

Return Value: None.

Filename: #define in usart16.h

See also: WriteUSART1, WriteUSART2.

putsUSART1
putsUSART2

Device: putsUSART1: PIC17C4X, PIC17C756
putsUSART2: PIC17C756

Function: Writes a string of characters to the USART including the null character.

Syntax:
```
#include <usart16.h>
void putsUSART1 (char *data);
void putsUSART2 (char *data);
```

Remarks: This function writes a string of data to the USART including the null character.

The value of *data* is a pointer to a string in contiguous RAM locations within the same bank.

Return Value: None.

Filename: u1puts.c
u2puts.c

See also: None.

ReadUSART1
ReadUSART2

Device: ReadUSART1: PIC17C4X, PIC17C756
ReadUSART2: PIC17C756

Functio Reads a byte out of the USART receive buffer, including the 9th bit if enabled.

Syntax:
```
#include <usart16.h>
char ReadUSART1 (void);
char ReadUSART2 (void);
```

Remarks: This function reads a byte out of the USART receive buffer. The 9th bit is recorded as well as the status bits. The status bits and the 9th data bits are saved in a union named `USART_Status` with the following declaration:

```
union USART
{
unsigned char val;
struct
{
      unsigned RX1_NINE:1;
      unsigned TX1_NINE:1;
      unsigned FRAME_ERROR1:1;
      unsigned OVERRUN_ERROR1:1;
      unsigned RX2_NINE:1;
      unsigned TX2_NINE:1;
      unsigned FRAME_ERROR2:1;
      unsigned OVERRUN_ERROR2:1;
} ;
};
```

The 9th bit is recorded only if 9-bit mode is enabled. The status bits are always recorded.

Return Value: This function returns the next character in the USART's receive buffer.

Filename: u1read.c
u2read.c

See also: getcUSART1, getcUSART2.

WriteUSART1
WriteUSART2

Device: WriteUSART1: PIC17C4X, PIC17C756
WriteUSART2: PIC17C756

Function: Writes a byte to the USART transmit buffer, including the 9th bit if enabled.

Syntax:
```
#include <usart16.h>
void WriteUSART1 (char data);
void WriteUSART2 (char data);
```

Remarks: This function writes a byte to the USART transmit buffer. The
9th bit is written as well. The 9th data bits are saved in a union named `USART_Status` with the following declaration:
```
union USART
      {
      unsigned char val;
      struct
      {
      unsigned RX1_NINE:1;
      unsigned TX1_NINE:1;
      unsigned FRAME_ERROR1:1;
      unsigned OVERRUN_ERROR1:1;
      unsigned RX2_NINE:1;
      unsigned TX2_NINE:1;
      unsigned FRAME_ERROR2:1;
      unsigned OVERRUN_ERROR2:1;
      } ;
      };
```
The 9th bit is used only if 9-bit mode is enabled.

The value of *data* can be any number from 0x00 to 0xff.

Return Value: None.

Filename: u1write.c
u2write.c

See also: **putcUSART1, putcUSART2.**

Chapter 8. Libraries

3.0 Software Peripheral Library

3.1 External LCD Functions

BusyXLCD

Device: PIC17C4X, PIC17C756

Function: Returns the status of the busy flag of the Hitachi HD44780 LCD controller.

Syntax:

```
#include <xlcd.h>
unsigned char BusyXLCD (void);
```

Remarks: This function returns the status of the busy flag of the Hitachi HD44780 LCD controller.

Return Value: This function returns 0 if the LCD controller is not busy, otherwise 1 is returned.

Filename: xlcd.c

See also: None.

OpenXLCD

Device: PIC17C4X, PIC17C756

Function: Configures the I/O pins and initializes the Hitachi HD44780 LCD controller.

Syntax:

```
#include <xlcd.h>
void OpenXLCD (unsigned char lcdtype);
```

Remarks: This function configures the I/O pins used to control the Hitachi HD44780 LCD controller. It also initializes this controller.

The I/O pin definitions that must be made to ensure that the external LCD operates correctly are:

Control I/O pin definitions

```
RW_PIN      PORTxbits.Rx?
TRIS_RW     DDRxbits.Rx?
RS_PIN      PORTxbits.Rx?
TRIS_RS     DDRxbits.Rx?
E_PIN       PORTxbits.Rx?
TRIS_E      DDRxbits.Rx?
x is the PORT, ? is the pin number
```

Data Port definitions

```
DATA_PORT   PORTx
            TRIS_DATA_PORTDDRx
```

The control pins can be on any port and are not required to be on the same port. The data interface must be defined as either 4-bit or 8-bit. The 8-bit interface is defined when a `#define BIT8` is included in the header file xlcd.h. If no define is included, then the 4-bit interface is included. When in 8-bit data interface mode, all 8 pins must be on the same port. When in 4-bit data interface mode, the 4 pins must be either the high or low nibble of a single port. When in 4-bit interface mode, the high nibble is specified by including `#define UPPER` in the header file xlcd.h. Otherwise, the lower nibble is specified by commenting this line out.

After these definitions have been made, the user must compile xlcd.c into an object to be linked.
Please refer to Chapter 2 of this manual (DS51112A) for information on compilers. Please refer to the MPASM User's Guide with MPLINK and MPLIB (DS33014F) for information on linking.

The value of *lcdtype* can be one of the following values (defined in xlcd.h):

Function Set defines

FOUR_BIT	4-bit data interface mode
EIGHT_BIT	8-bit data interface mode
LINE_5X7	5x7 characters, single line display
LINE_5X10	5x10 characters display
LINES_5X7	5x7 characters, multiple line display

This function also requires three external routines to be provided by the user for specific delays:

DelayFor18TCY()	18 Tcy delay
DelayPORXLCD()	15ms delay
DelayXLCD()	5ms delay

Return Value: None.

Filename: xlcd.c

See also: None.

Code Example:

```
#include <p17c756.h>
#include <xlcd.h>
#include <delays.h>
#include <usart16.h>
```



```
    void DelayFor18TCY(void)
    {
          Nop;
          Nop;
          Nop;
          Nop;
          Nop;
          Nop;
          Nop;
          Nop;
          Nop;
          Nop;
          Nop;
          Nop;
          return;
    }
    void DelayPORXLCD(void)
    {
          Delay1KTCYx(60);           //Delay of 15ms
          return;
    }
    void DelayXLCD(void)
    {
          Delay1KTCYx(20);           //Delay of 5ms
          return;
    }
    void main(void)
    {
          char data;
          // configure external LCD
          OpenXLCD(EIGHT_BIT&LINES_5X7);
          // configure USART
          OpenUSART1(USART_TX_INT_OFF&
                     USART_RX_INT_OFF&
                     USART_ASYNCH_MODE&USART_
                     EIGHT_BIT&USART_CONT_RX);
    while(1)
    {
          while(!DataRdyUSART1());   //wait for data
          data = ReadUSART1();       //read data
          WriteDataXLCD(data);       //write to LCD
          if(data==’Q’)
                  break;
    }
    CloseADC();                      //close modules
    CloseUSART1();
    return;
}
```

putcXLCD

Device: PIC17C4X, PIC17C756

Function: Writes one byte of data to the Hitachi HD44780 LCD controller.

Syntax:
```
#include <xlcd.h>
void putcXLCD (char data);
```

Remarks: This function performs the same function as **WriteDataXLCD**. Please refer to the description of that function.

Return Value: None.

Filename: #define in xlcd.h

See also: None.

putsXLCD

Device: PIC17C4X, PIC17C756

Function: Writes a string of characters to the Hitachi HC44780 LCD controller.

Syntax:
```
#include <xlcd.h>
void putsXLCD (char *buffer);
```

Remarks: This functions writes a string of characters located in *buffer* to the Hitachi HD44780 LCD controller. It stops transmission after the character before the null character, i.e. the null character is not sent.

Return Value: None.

Filename: xlcd.c

See also: None.

ReadAddrXLCD

Device: PIC17C4X, PIC17C756

Function: Reads the address byte from the Hitachi HD44780 LCD controller.

Syntax:
```
#include <xlcd.h>
unsigned char ReadAddrXLCD (void);
```

Remarks: This function reads the address byte from the Hitachi HD44780 LCD controller. The user must first check to see if the LCD controller is busy by calling the **BusyXLCD()** function.

The address read from the controller is for the character generator RAM or the display data RAM depending on the previous **Set??RamAddr()** function that was called.

Return Value: This function returns an 8-bit which is the 7-bit address in the lower 7-bits of the byte and the BUSY status flag in the 8th bit.

Bit7							Bit0
BF	A6	A5	A4	A3	A2	A1	A0

Filename: xlcd.c

See also: **SetCGRamAddr, SetDDRamAddr.**

ReadDataXLCD

Device: PIC17C4X, PIC17C756

Function: Reads a data byte from the Hitachi HD44780 LCD controller.

Syntax:

```
#include <xlcd.h>
char ReadDataXLCD (void);
```

Remarks: This function reads a data byte from the Hitachi HD44780 LCD controller. The user must first check to see if the LCD controller is busy by calling the **BusyXLCD()** function.

The data read from the controller is for the character generator RAM or the display data RAM depending on the previous **Set??RamAddr()** function that was called.

Return Value: This function returns the 8-bit data value.

Filename: xlcd.c

See also: **SetCGRamAddr, SetDDRamAddr.**

3.1.1 SetCGRamAddr

Device: PIC17C4X, PIC17C756

Function: Sets the character generator address.

Syntax:
```
#include <uart16.h>
void SetCGRamAddr (unsigned char CGaddr);
```

Remarks: This function sets the character generator address of the Hitachi HD44780 LCD controller. The user must first check to see if the controller is busy by calling the **BusyXLCD()** function.

Return Value: None.

Filename: xlcd.c

See also: None.

SetDDRamAddr

Device: PIC17C4X, PIC17C756

Function: Sets the display data address.

Syntax:
```
#include <uart16.h>
void SetDDRamAddr (unsigned char DDaddr);
```

Remarks: This function sets the display data address of the Hitachi HD44780 LCD controller. The user must first check to see if the controller is busy by calling the **BusyXLCD()** function.

Return Value: None.

Filename: xlcd.c

See also: None.

WriteCmdXLCD

Device: PIC17C4X, PIC17C756

Function: Writes a command to the Hitachi HD44780 LCD controller.

Syntax:
```
#include <xlcd.h>
void WriteCmdXLCD (unsigned char cmd);
```

Remarks: This function writes the command byte to the Hitachi HD44780 LCD controller. The user must first check to see if the LCD controller is busy by calling the **BusyXLCD()** function.

The value of *cmd* can be one of the following values (defined in xlcd.h):

Function Set defines

FOUR_BIT	4-bit data interface mode
EIGHT_BIT	8-bit data interface mode
LINE_5X7	5x7 characters, single line display
LINE_5X10	5x10 characters display
LINES_5X7	5x7 characters, multiple line display

Display ON/OFF control defines

DON	Display on
DOFF	Display off
CURSOR_ON	Cursor on
CURSOR_OFF	Cursor off
BLINK_ON	Blinking cursor on
BLINK_OFF	Blinking cursor off

Cursor or Display shift defines

SHIFT_CUR_LEFT	Cursor shifts to the left
SHIFT_CUR_RIGHT	Cursor shifts to the right
SHIFT_DISP_LEFT	Display shifts to the left
SHIFT_DISP_RIGHT	Display shifts to the right

The above defines can not be mixed. The only commands that can be issued are function set, display control, and cursor/display shift control.

Return Value: None.

Filename: xlcd.c

See also: None.

WriteDataXLCD

Device: PIC17C4X, PIC17C756

Function: Writes a data byte from the Hitachi HD44780 LCD controller.

Syntax:

```
#include <xlcd.h>
void WriteDataXLCD (char data);
```

Remarks: This function writes a data byte to the Hitachi HD44780 LCD controller. The user must first check to see if the LCD controller is busy by calling the **BusyXLCD()** function.

The data read from the controller is for the character generator RAM or the display data RAM depending on the previous **Set??RamAddr()** function that was called.

The value of *data* can be any 8-bit value, but should correspond to the character RAM table of the HD44780 LCD controller.

Return Value: None.

Filename: xlcd.c

See also: SetCGRamAddr, SetDDRamAddr.

3.2 Software I²C Functions

Clock_test

Device: PIC17CXXX

Function: Generates delay for slave clock stretching.

Syntax:
```
#include <swi2c16.h>
void Clock_test (void);
```

Remarks: This function is called to allow for slave clock stretching. The delay time may need to be adjusted per application requirements. If at the end of the delay period the clock line is low, a bit field in the global structure BUS_STATUS (BUS_STATUS.clk) is set to 1. If the clock line is high at the end of the delay, this bit field is a 0.

```
far ram union i2cbus_state
{
  struct
  {
     unsigned busy:1;  bus state is busy
     unsigned clk :1;  clock timeout or
                       failure
     unsigned ack :1;  acknowledge error
                       or not ACK
     unsigned     :5;  bit padding
  };
  unsigned char dummy;dummy variable
} BUS_STATUS;          efine union/struct
```

Return Value: None.

Filename: swckti2c.c

See also: None.

SWAckI2C

Device: PIC17CXXX

Function: Generates I²C bus acknowledge condition.

Syntax:
```
#include <swi2c16.h>
void SWAckI2C (void);
```

Remarks: This function is called to generate an I²C bus acknowledge sequence. A bit field in the global structure BUS_STATUS (BUS_STATUS.ack) is set to 1 if the slave device did not ack. This error condition could also indicate a bus error on the SDA line. If no error occurred this bit field is a 0.

```
far ram union i2cbus_state
{
 struct
 {
 unsigned busy :1;  bus state is busy
 unsigned clk  :1;  clock timeout or
                    failure
 unsigned ack  :1;  acknowledge error or
                    not ACK
 unsigned      :5;  bit padding
 };
 unsigned char dummy;dummy variable
} BUS_STATUS;       define union/struct
```

Return Value: None.

Filename: swacki2c.c

See also: None.

SWGetsI2C

Device: PIC17CXXX

Function: Reads in data string via software I^2C implementation.

Syntax: #include <swi2c16.h>
unsigned char SWGetsI2C (unsigned char far **rdptr*,
unsigned char *length*);

Remarks: This function reads in a predetermined data string length. Each byte is retrieved via a call to the **SWGetcI2C** function. **SWGetcI2C** and **SWReadI2C** refer to the same function via a `#define` statement in the swi2c16.h file.

Return Value: This function returns -1 if all bytes have been received and the master generated a *not ack* bus condition.

Filename: swgtsi2c.c

See also: None.

SWPutsI2C

Device: PIC17CXXX

Function: Writes out data string via software I^2C implementation.

Syntax:

```
#include <swi2c16.h>
unsigned char SWPutsI2C (unsigned char
far *wrdptr);
```


Remarks: This function writes out a data string until a null character is evaluated. Each byte is written via a call to the SWPutcI2C function. The actual called function body is termed **SWWriteI2C**. **SWPutcI2C** and **SWWriteI2C** refer to the same function via a `#define` statement in the swi2c16.h file.

Return Value: This function returns -1 if there was an error else returns a 0.

Filename: swptsi2c.c

See also: None.

CODE EXAMPLES:

The following are simple code examples illustrating a software I^2C implementation communicating with a Microchip 24LC01B I^2C EE Memory Device. In all the examples provided no error checking utilizing the value returned from a function is implemented. The port pins used are defined in the swi2c16.h file and must be set per application requirments.

```
#include <p17cxx.h>
#include <swi2c16.h>
#include <delays.h>
extern far ram union i2cbus_state
{
   struct
   {
   unsigned busy :1;       // bus state is busy
   unsigned clk  :1;       // clock timeout or
failure
   unsigned ack  :1;       // acknowledge error or
not ACK
   unsigned      :5;       // bit padding
   };
   unsigned char dummy;
} BUS_STATUS;

// FUNCTION PROTOTYPES
void main(void);
void byte_write(void);
void page_write(void);
void current_address(void);
void random_read(void);
void sequential_read(void);
void ack_poll(void);
unsigned char warr[] = {8,7,6,5,4,3,2,1,0};
unsigned char rarr[15];
unsigned char far *rdptr = rarr;
unsigned char far *wrptr = warr;
unsigned char var;
#define W_CS  PORTA.2
```

```
//***************************************************
#pragma code _main=0x100
void main(void)
{
   byte_write();
   ack_poll();
   page_write();
   ack_poll();
   Nop();
   sequential_read();
   Nop();
   while (1);
}

void byte_write(void)
{
   SWStartI2C();
   var = SWPutcI2C(0xA0);  // control byte
   swAckI2C();
   var = SWPutcI2C(0x10);  // word address
   swAckI2C();
   var = SWPutcI2C(0x66);  // data
   SWAckI2C();
   SWStopI2C();
}

void page_write(void)
{
   SWStartI2C();
   var = SWPutcI2C(0xA0);  // control byte
   SWAckI2C();
   var = SWPutcI2C(0x20);  // word address
   SWAckI2C();
   var = SWPutsI2C(wrptr); // data
   SWStopI2C();
}

void sequential_read(void)
{
   SWStartI2C();
   var = SWPutcI2C(0xA0);  // control byte
   SWAckI2C();
   var = SWPutcI2C(0x00);  // address to read from
   SWAckI2C();
   SWRestartI2C();
   var = SWPutcI2C(0xA1);
   SWAckI2C();
```

```
      var = SWGetsI2C(rdptr,9);
      SWStopI2C();
   }

   void current_address(void)
   {
      SWStartI2C();
      SWPutcI2C(0xA1);        // control byte
      SWAckI2C();
      SWGetcI2C();            // word address
      SWNotAckI2C();
      SWStopI2C();
   }

   void ack_poll(void)
   {
      SWStartI2C();
      var = SWPutcI2C(0xA0);  // control byte
      SWAckI2C();
      while (BUS_STATUS.ack)
      {
      BUS_STATUS.ack = 0;
      SWRestartI2C();
      var = SWPutcI2C(0xA0);  // data
      SWAckI2C();
      }
      SWStopI2C();
}
```

SWReadI2C

Device: PIC17CXXX

Function: Reads a single data byte via software I^{2}C implementation.

Syntax:
```
#include <swi2c16.h>
unsigned char SWReadI2C (void);
```

Remarks: This function reads in a single data byte by generating the appropriate signals on the predefined I^{2}C clock line.

Return Value: This function returns the acquired I^{2}C data byte. If there was an error in this function, the return value will be -1. This condition can be evaluated by testing the bit field `BUS_STATUS.clk`. If this bit field is 1, then there was an error, else it is a 0.

Filename: swgtci2c.c

See also: None.

SWRestartI2C

Device: PIC17CXXX

Function: Generates I^2C restart bus condition.

Syntax:
```
#include <swi2c16.h>
void SWRestartI2C (void);
```

Remarks: This function is called to generate an I^2C bus restart condition.

Return Value: None.

Filename: swrsti2c.c

See also: None.

SWStartI2C

Device: PIC17CXXX

Function: Generates I^2C bus start condition.

Syntax:
```
#include <swi2c16.h>
void SWStartI2C (void);
```

Remarks: This function is called to generate an I^2C bus start condition.

Return Value: None.

Filename: swstri2c.c

See also: None.

SWStopI2C

Device: PIC17CXXX

Function: Generates I^2C bus stop condition.

Syntax:
```
#include <swi2c16.h>
void SWStopI2C (void);
```

Remarks: This function is called to generate an I^2C bus stop condition.

Return Value: None.

Filename: swstpi2c.c

See also: None.

SWWriteI2C

Device: PIC17CXXX

Function: Writes out single data byte via software I^2C implementation.

Syntax:
```
#include <swi2c16.h>
unsigned char SWWriteI2C (unsigned char
data_out);
```

Remarks: This function writes out a single data byte to the predefined data pin. The clock and data pins are user defined in the swi2c16.h file and must be set per application requirements. **SWWriteI2C** and **SWPutcI2C** refer to the same function via a `#define` statement in the swi2c16.h file.

Return Value: This function returns -1 if there was an error condition else returns a 0.

Filename: swptci2c.c

See also: None.

3.3 Software SPI Functions

SWClearCSSPI

Device: PIC17C4X, PIC17C756

Function: Clears the chip select (CS) pin that is specified in the swspi16.h header file.

Syntax:
```
#include <swspi16.h>
void SWClearCSSPI (void);
```

Remarks: This function clears the I/O pin that is specified in swspi16.h to be the chip select (CS) pin for the software SPI.

Return Value: None.

Filename: swspi16.c

See also: SWSetCSSPI.

SWOpenSPI

Device: PIC17C4X, PIC17C756

Function: Configures the I/O pins for the software SPI.

Syntax:
```
#include <swspi16.h>
void SWOpenSPI (void);
```

Remarks: This function configures the I/O pins used for the software SPI to the correct input or ouput state and logic level. The I/O pins used for chip select (CS), data in (DIN), data out (DOUT), and serial clock (SCK) must be defined in the header file swspi16.h.

The definitions that must be made to ensure that the software SPI operates correctly are:

I/O pin definitions
```
SW_CS_PIN          PORTxbits.Rx?
TRIS_SW_CS_PIN     DDRxbits.Rx?
SW_DIN_PIN         PORTxbits.Rx?
TRIS_SW_DIN_PIN    DDRxbits.Rx?
SW_DOUT_PIN        PORTxbits.Rx?
TRIS_SW_DOUT_PIN   DDRxbits.Rx?
SW_SCK_PIN         PORTxbits.Rx?
TRIS_SW_SCK_PIN    DDRxbits.Rx?
```
x is the PORT, ? is the pin number

SPI Mode
```
#define MODE0 or
#define MODE1 or
#define MODE2 or
#define MODE3
```
Only one of the `MODEx` can be defined.

After these definitions have been made, compile the software SPI files into an DS51112B) for information on compilers. Refer to the MPASM User's Guide with MPLINK and MPLIB (DS33014F) for information on linking.

Return Value: None.

Filename: swspi16.c

See also: None.

Code Example:

```
#include <p17c756.h>
#include <swspi16.h>
#include <delays.h>
void main(void)
{
```



```
        char address;
        // configure software SPI
        OpenSWSPI();
        for(address=0;address<0x10;address++)
        {
            ClearCSSWSPI();     //clear CS pin
            WriteSWSPI(0x02);   //send write cmd
            WriteSWSPI(address);//send address h
            WriteSWSPI(address);//send address low
            SetCSSWSPI();       //set CS pin
            Delay10KTCYx(50);   //wait 5000,000TCY
        }
        return;
}
```

SWputcSPI

Device: PIC17C4X, PIC17C756

Function: Reads/writes one byte of data out the software SPI.

Syntax: #include <swspi16.h>
char SWputcSPI (char *data*);

Remarks: This function performs the same function as **SWWriteSPI()**. Refer to the description of that function.

Return Value: None.

Filename: swspi16.c

See also: None.

SWSetCSSPI

Device: PIC17C4X, PIC17C756

Function: Sets the chip select (CS) pin that is specified in the swspi16.h header file.

Syntax:
```
#include <swspi16.h>
void SWSetCSSPI (void);
```

Remarks: This function sets the I/O pin that is specified in swspi16.h to be the chip select (CS) pin for the software SPI.

Return Value: None.

Filename: swspi16.c

See also: SWClearCSSPI.

SWWriteSPI

Device: PIC17C4X, PIC17C756

Function: Reads/writes one byte of data out the software SPI.

Syntax:
```
#include <swspi16.h>
char SWWriteSPI (char data);
```

Remarks: This function writes the specified byte of data out the software SPI and returns the byte of data that was read. This function does not provide any control of the chip select pin (CS).

Return Value: This function returns the byte of data that was read from the data in (DIN) pin of the software SPI.

Filename: swspi16.c

See also: None.

3.4 Software UART Functions

getcUART

Device: PIC17C4X, PIC17C756

Function: Reads one byte of data from the software UART.

Syntax:
```
#include <uart16.h>
char getcUART (void);
```

Remarks: This function performs the same function as **ReadUART()**. Please refer to the description of that function.

Return Value: None.

Filename: uart16.c

See also: **ReadUART**

getsUART

Device: PIC17C4X, PIC17C756

Function: Reads a string of characters from the software UART.

Syntax:
```
#include <uart16.h>
void getsUART (char *buffer, unsigned
char len);
```

Remarks: This function reads a string of characters from the software UART and places them in *buffer*. The number of characters read is given in the variable *len*.

The value of *len* can be any 8-bit value, but is restricted to the maximum size of an array within any bank of RAM.

Return Value: None.

Filename: uart16.c

See also: None.

OpenUART

Device: PIC17C4X, PIC17C756

Function: Configures the I/O pins for the software UART.

Syntax:
```
#include <uart16.h>
void OpenUART (void);
```

Remarks: This function configures the I/O pins used for the software UART to the correct input or ouput state and logic level. The I/O pins used for receive data (RXD) and transmit data (TXD) must be defined in the header file uart16_a.asm.

The definitions that must be made to ensure that the software UART operates correctly are:

```
I/O pin definitions
  SWTXDequPORTx
  SWTXDpinequ?
  TRIS_SWTXDequDDRx
  SWRXDequPORTx
  SWRXDpinequ?
  TRIS_SWRXDequDDRx
  UART_PORT_BSRequb
  x is the PORT, ? is the pin number,
  b is the PORTx bank
```

After these definitions have been made, compile the software ART files into an object to be linked. Refer to Chapter 2 of this manual (DS51112A) for information on compilers. Refer to the MPASM User's Guide with MPLINK and MPLIB (DS33014F) for information on linking.

Return Value: None.

Filename: uart16.c

See also: None.

Code Example:

```
#include <p17c756.h>
#include <uart16.h>
     void main(void)
     {
        char data
        // configure software UART
        OpenUART();
        while(1)
        {
        data = ReadUART();//read a byte
        WriteUART(data);//bounce it back
```

```
    }
    return;
}
```

putcUART

Device: PIC17C4X, PIC17C756

Function: Writes one byte of data out the software UART.

Syntax:
```
#include <uart16.h>
void putcUART (char data);
```

Remarks: This function performs the same function as **WriteUART()**. Refer to the description of that function.

Return Value: None.

Filename: uart16.c

See also: **WriteUART**

putsUART

Device: PIC17C4X, PIC17C756

Function: Writes a string of characters to the software UART.

Syntax:
```
#include <uart16.h>
void getsUART (char *buffer);
```

Remarks: This function writes a string of characters to the software UART. The entire string including the null is sent to the UART.

Return Value: None.

Filename: uart16.c

See also: None.

ReadUART

Device: PIC17C4X, PIC17C756

Function: Reads one byte of data out the software UART.

Syntax:
```
#include <uart16.h>
char ReadUART (void);
```

Remarks: This function reads a byte of data out the software UART and returns the byte of data.

Return Value: This function returns the byte of data that was read from the receive data (RXD) pin of the software UART.

Filename: uart16.c

See also: **getcUART**

WriteUART

Device: PIC17C4X, PIC17C756

Function: Writes one byte of data out the software UART.

Syntax:
```
#include <uart16.h>
void WriteUART (char data);
```

Remarks: This function writes the specified byte of data out the software UART.

The value of *data* can be any 8-bit value.

Return Value: None.

Filename: uart16.c

See also: putcUART

4.0 General Software Library

4.1 Character Classification Functions

isalnum

Device: PIC17C4X, PIC17C756

Function: Alphanumeric character classification.

Syntax:
```
#include <ctype.h>
char isalnum (char ch);
```

Remarks: This function determines if *ch* is an alphanumeric character in the ranges of:

A to Z	(0x41 to 0x5A)
a to z	(0x61 to 0x7A)
0 to 9	(0x30 to 0x39)

Return Value: This function returns 1 when the argument is within the specified range of values, otherwise 0 is returned.

Filename: isalnum.c

See also: None.

isalpha

Device: PIC17C4X, PIC17C756

Function: Alphabetical character classification.

Syntax:
```
#include <ctype.h>
char isalpha (char ch);
```

Remarks: This function determines if *ch* is a valid character of the alphabet in the ranges of:

A to Z	(0x41 to 0x5A)
a to z	(0x61 to 0x7A)

Return Value: This function returns 1 when the argument is within the specified range of values, otherwise 0 is returned.

Filename: isalpha.c

See also: None.

isascii

Device: PIC17C4X, PIC17C756

Function: ASCII character classification.

Syntax:
```
#include <ctype.h>
char isascii (char ch);
```

Remarks: This function determines if *ch* is an ASCII character which has a range of 0x00 to 0x7F.

Return Value: This function returns 1 when the argument is within the specified range of values, otherwise 0 is returned.

Filename: isascii.c

See also: None.

iscntrl

Device: PIC17C4X, PIC17C756

Function: Control character classification.

Syntax:
```
#include <ctype.h>
char iscntrl (char ch);
```

Remarks: This function determines if *ch* is a control character in the ranges of:

0x00 to 0x1F
0x7f

Return Value: This function returns 1 when the argument is within the specified range of values, otherwise 0 is returned.

Filename: iscntrl.c

See also: None.

isdigit

Device: PIC17C4X, PIC17C756

Function: Numeric character classification.

Syntax:
```
#include <ctype.h>
char isdigit (char ch);
```

Remarks: This function determines if *ch* is an numeric character in the ranges of:

0 to 9 (0x30 to 0x39)

Return Value: This function returns 1 when the argument is within the specified range of values, otherwise 0 is returned.

Filename: isdigit.c

See also: None.

islower

Device: PIC17C4X, PIC17C756

Function: Lower-case alphabetical character classification.

Syntax:
```
#include <ctype.h>
char isalnum (char ch);
```

Remarks: This function determines if *ch* is a lower-case alphabetical character in the ranges of:

a to z(0x61 to 0x7A)

Return Value: This function returns 1 when the argument is within the specified range of values, otherwise 0 is returned.

Filename: islower.c

See also: None.

isupper

Device: PIC17C4X, PIC17C756

Function: Upper-case alphabetical character classification.

Syntax:
```
#include <ctype.h>
char isupper (char ch);
```

Remarks: This function determines if *ch* is an upper-case alphabetical character in the ranges of:

A to Z	(0x41 to 0x5A)

Return Value: This function returns 1 when the argument is within the specified range of values, otherwise 0 is returned.

Filename: isupper.c

See also: None.

isxdigit

Device: PIC17C4X, PIC17C756

Function: Hexadecimal character classification.

Syntax:
```
#include <ctype.h>
char isalnum (char ch);
```

Remarks: This function determines if *ch* is a hexadecimal character in the ranges of:

A to F	(0x41 to 0x46
a to f	(0x61 to 0x66)
0 to 9	(0x30 to 0x39)

Return Value: This function returns 1 when the argument is within the specified range of values, otherwise 0 is returned.

Filename: isxdig.c

See also: None.

4.2 Number and Text Conversion Functions

atob

Device: PIC17C4X, PIC17C756

Function: Converts a string to an 8-bit signed byte.

Syntax:
```
#include <stdlib.h>
char atob (char *string);
```

Remarks: This function converts the ASCII *string* into an 8-bit signed byte. It first finds the length of the *string* by searching for the null character. If the string length is greater than 5 characters, this function returns 0. It then starts processing the *string* into the 8-bit signed byte (-128 to 127).

Return Value: 8-bit signed byte for all strings with 5 characters or less (-128 to 127). 0 for all strings greater than 5 characters.

Filename: atob.c

See also: None.

atoi

Device: PIC17C4X, PIC17C756

Function: Converts a string to an 16-bit signed integer.

Syntax:
```
#include <stdlib.h>
int atoi(char *string);
```

Remarks: This function converts the ASCII *string* into an 16-bit signed integer. It first finds the length of the *string* by searching for the null character. If the string length is greater than 7 characters, this function returns 0. It then starts processing the *string* into the 16-bit signed integer (-32768 to 32767).

Return Value: 16-bit signed integer for all strings with 7 characters or less (-32768 to 32767). 0 for all strings greater than 7 characters.

Filename: atoi.c

See also: None.

atoub

Device: PIC17C4X, PIC17C756

Function: Converts a string to an 8-bit unsigned byte.

Syntax:
```
#include <stdlib.h>
unsigned char atoub (char *string);
```

Remarks: This function converts the ASCII *string* into an 8-bit unsigned byte. It first finds the length of the *string* by searching for the null character. If the string length is greater than 4 characters, this function returns 0. It then starts processing the *string* into the 8-bit unsigned byte (0 to 255).

Return Value: 8-bit unsigned byte for all strings with 4 characters or less (0 to 255). 0 for all strings greater than 4 characters.

Filename: atoub.c

See also: None.

atoui

Device: PIC17C4X, PIC17C756

Function: Converts a string to an 16-bit unsigned integer.

Syntax:
```
#include <stdlib.h>
unsigned int atoui (char *string);
```

Remarks: This function converts the ASCII *string* into an 16-bit unsigned integer. It first finds the length of the *string* by searching for the null character. If the string length is greater than 6 characters, this function returns 0. It then starts processing the *string* into the 16-bit unsigned integer. (0 to 65535)

Return Value: 16-bit unsigned integer for all strings with 6 characters or less (0 to 65535). 0 for all strings greater than 6 characters

Filename: atoui.c

See also: None.

Chapter 8. Libraries

btoa

Device: PIC17C4X, PIC17C756

Function: Converts an 8-bit signed byte to string.

Syntax:

```
#include <stdlib.h>
void btoa (char value, char *string);
```

Remarks: This function converts the 8-bit signed byte in the argument *value* to a ASCII string representation. The *string* must be long enough to hold the ASCII representation which is:

number(3) + sign(1) + null(1) = 5

The conversion process uses the minimum amount of characters in the string. Some examples are:

-120	5 characters
-57	4 characters
-6	3 characters
0	2 characters
29	3 characters
107	4 characters

Return Value: None.

Filename: btoa.c

See also: None.

itoa

Device: PIC17C4X, PIC17C756

Function: Converts an 16-bit signed integer to string.

Syntax:

```
#include <stdlib.h>
void itoa (int value, char *string);
```

Remarks: This function converts the 16-bit signed integer in the argument *value* to a ASCII *string* representation. The *string* must be long enough to hold the ASCII representation which is:

number(5) + sign(1) + null(1) = 7

The conversion process uses the minimum amount of characters in the string. Some examples are:

-24290	7 characters
-6183	6 characters
-120	5 characters
-57	4 characters
-6	3 characters
0	2 characters

29	3 characters
107	4 characters
1187	5 characters
32000	6 characters

Return Value: None.

Filename: itoa.c

See also: None.

toascii

Device: PIC17C4X, PIC17C756

Function: Converts a character to an ASCII character

Syntax:

```
#include <ctype.h>
char toascii (char ch);
```

Remarks: This function converts *ch* to a valid ASCII character by setting the MSB bit7 to a zero.

Return Value: This function returns the converted ASCII character.

Filename: toascii.c

See also: None.

tolower

Device: PIC17C4X, PIC17C756

Function: Converts a character to a lower-case alphabetical ASCII character.

Syntax

```
#include <ctype.h>
char tolower (char ch);
```

Remarks: This function converts *ch* to a lower-case alphabetical ASCII character provided that the argument is a valid upper-case alphabetical character.

Return Value: This function returns a lower-case character if the argument was upper-case to begin with, otherwise the original character is returned.

Filename: tolower.c

See also: None.

toupper

Device: PIC17C4X, PIC17C756

Function: Converts a character to a upper-case alphabetical ASCII character.

Syntax:
```
#include <ctype.h>
char toupper (char ch);
```

Remarks: This function converts *ch* to a upper-case alphabetical ASCII character provided that the argument is a valid lower-case alphabetical character.

Return Value: This function returns a lower-case character if the argument was upper-case to begin with, otherwise the original character is returned.

Filename: toupper.c

See also: None.

ubtoa

Device: PIC17C4X, PIC17C756

Function: Converts an 8-bit unsigned byte to string.

Syntax:
```
#include <stdlib.h>
void ubtoa (unsigned char value, char
*string);
```

Remarks: This function converts the 8-bit unsigned byte in the argument *value* to a ASCII *string* representation. The *string* must be long enough to hold the ASCII representation which is:

number(3) + null(1) = 4

The conversion process uses the minimum amount of characters in the string. Some examples are:

0	2 characters
29	3 characters
107	4 characters
255	4 characters

Return Value: None.

Filename: ubtoa.c

See also: None.

uitoa

Device: PIC17C4X, PIC17C756

Function: Converts an 16-bit unsigned integer to string.

Syntax:
```
#include <stdlib.h>
void uitoa (unsigned int value, char
*string);
```

Remarks: This function converts the 16-bit unsigned integer in the argument *value* to a ASCII *string* representation. The *string* must be long enough to hold the ASCII representation which is:

number(2) + null(1) = 6

The conversion process uses the minimum amount of characters in the string. Some examples are:

0	2 characters
29	3 characters
107	4 characters
3481	5 characters
57912	6 characters

Return Value: None.

Filename: uitoa.c

See also: None.

4.3 Delay Functions

Delay1TCY

Device: PIC17C4X, PIC17C756

Function: Delay of 1 instruction cycle (Tcy).

Syntax:
```
#include <delays.h>
void Delay1TCY (void);
```

Remarks: This function is actually a `#define` for the `Nop()` instruction. When encountered in the source code, the compiler simply inserts a `Nop()`.

Return Value: None.

Filename: #define in delays.h

See also: None.

Delay10TCY

Device: PIC17C4X, PIC17C756

Function: Delay of 10 instruction cycles (Tcy).

Syntax:
```
#include <delays.h>
void Delay10TCY (void);
```

Remarks: This function creates a delay of 10 instruction cycles.

Return Value: None.

Filename: dy1otcy.c

See also: None.

Delay10TCYx

Device: PIC17C4X, PIC17C756

Function: Delay of multiples of 10 instruction cycles (Tcy).

Syntax:
```
#include <delays.h>
void Delay10TCYx (unsigned char unit);
```

Remarks: This function creates delays of multiples of 10 instruction cycles.

The value of *unit* can be any 8-bit value from 2 to 255 or 0. A value of 0 represents sending 256 to the function.

Return Value: None.

Filename: dy1otcyx.c

See also: None.

Delay100TCYx

Device: PIC17C4X, PIC17C756

Function: Delay of multiples of 100 instruction cycles (Tcy).

Syntax:
```
#include <delays.h>
void Delay100TCYx (unsigned char unit);
```

Remarks: This function creates delays of multiples of 100 instruction cycles.

The value of *unit* can be any 8-bit value from 0 to 255. A value of 0 represents sending 256 to the function.

Return Value: None.

Filename: dy100tcx.c

See also: None.

Delay1KTCYx

Device: PIC17C4X, PIC17C756

Function: Delay of multiples of 1000 instruction cycles (Tcy).

Syntax:
```
#include <delays.h>
void Delay1KTCYx (unsigned char unit);
```

Remarks: This function creates delays of multiples of 1000 instruction cycles.

The value of *unit* can be any 8-bit value from 0 to 255. A value of 0 represents sending 256 to the function.

Return Value: None.

Filename: dy1ktcyx.c

See also: None.

Delay10KTCYx

Device: PIC17C4X, PIC17C756

Function: Delay of multiples of 10000 instruction cycles (Tcy).

Syntax:
```
#include <delays.h>
void Delay10KTCYx (unsigned char unit);
```

Remarks: This function creates delays of multiples of 10000 instruction cycles.

The value of *unit* can be any 8-bit value from 0 to 255. A value of 0 represents sending 256 to the function.

Return Value: None.

Filename: dy10ktcx.c

See also: None.

4.4 Memory and String Manipulation Functions

memcmp

Device: PIC17C4X, PIC17C756

Function: Compares the contents of two arrays of bytes.

Syntax:
```
#include <mem.h>
signed char memcmp (char *buf1, char
*buf2, unsigned char memsize);
```

Remarks: This function compares the first *memsize* number of elements in *buf1* to the first *memsize* number of elements in *buf2* and returns if the buffers are less than, equal to, or greater than each other.

Return Value: -1 if buf1 < buf2
0 if buf1 == buf2
1 if buf1 > buf2

Filename: memcmp.c

See also: None.

memcpy

Device: PIC17C4X, PIC17C756

Function: Copies the contents of the source buffer into the destination buffer.

Syntax:
```
#include <mem.h>
void memcmp (char *dest, char *src,
unsigned char memsize);
```

Remarks: This function copies the first *memsize* number of elements in *src* to the array *dest*.

Return Value: None.

Filename: memcpy.c

See also: None.

memset

Device: PIC17C4X, PIC17C756

Function: Copies the specified character into the destination array.

Syntax:
```
#include <mem.h>
void memcmp (char *dest, char value,
unsigned char memsize);
```

Remarks: This function copies the character *value* into the first *memsize* elements of the array *dest*.

Return Value: None.

Filename: memset.c

See also: None.

strcat

Device: PIC17C4X, PIC17C756

Function: Concatenates the source string to the end of the destination string.

Syntax:
```
#include <string.h>
void strcat (char *dest, char *src);
```

Remarks: This function copies the string in *src* to the end of the string in dest. The *src* string starts at the null in *dest*. A null character is added to the end of the resulting string in *dest*.

Return Value: None.

Filename: strcat.c

See also: None.

strcmp

Device: PIC17C4X, PIC17C756

Function: Compares two strings.

Syntax:
```
include <string.h>
signed char strcmp (char *str1, char
*str2);
```

Remarks: This function compares the string in *str1* to the string in *str2* and returns if *str1* is less than, equal to, or greater than *str2*.

Return Value: -1 if str1 < str2
0 if str1 == str2
1 if str1 > str2

Filename: strcmp.c

See also: None.

strcpy

Device: PIC17C4X, PIC17C756

Function: Copies the source string into the destination string.

Syntax:
```
#include <string.h>
void strcpy (char *dest, char *src);
```

Remarks: This function copies the string in *src* to *dest*. Characters in src are copied until the null character is reached. The string *dest* is null terminated.

Return Value: None.

Filename: strcpy.c

See also: None.

strlen

Device: PIC17C4X, PIC17C756

Function: Returns the length of the string.

Syntax:
```
#include <string.h>
unsigned char strlen (char *str);
```

Remarks: This function determines the length of the string minus the null character.

Return Value: This function returns the length of the string in an unsigned 8-bit byte.

Filename: strlen.c

See also: None.

strlwr

Device: PIC17C4X, PIC17C756

Function: Converts all upper-case characters in a string to lower-case.

Syntax:
```
#include <string.h>
void strlwr (char *str);
```

Remarks: This function converts all upper-case characters in str to lower-case characters. All characters that are not upper-case (A to Z) are not affected.

Return Value: None.

Filename: strlwr.c

See also: None.

strset

Device: PIC17C4X, PIC17C756

Function: Copies the specified character into all characters in a string.

Syntax:
```
#include <string.h>
void memcmp (char *str, char ch);
```

Remarks: This function copies the character in *ch* to all characters in the string up to the null character.

Return Value: None.

Filename: strset.c

See also: None.

strupr

Device: PIC17C4X, PIC17C756

Function: Converts all lower-case characters in a string to upper-case.

Syntax:
```
#include <string.h>
void strupr (char *str);
```

Remarks: This function converts all lower-case characters in str to upper-case characters. All characters that are not lower-case (a to z) are not affected.

Return Value: None.

Filename: strupr.c

See also: None.

5.0 Math Library

5.1 32-bit Integer and 32-bit Floating Point Math Libraries

The math libraries are included in the \MCC\SRC\MATH folder. These are assembly language routines and can be included and linked with your application. Use the BUILD.BAT file to build a library of all routines.

5.1.1 Functions

FXM3232U	32-bit unsigned integer multiplication
FXM3232S	32-bit signed integer multiplication
FXD3232U	32-bit unsigned integer division
FXD3232S	32-bit signed integer division
FPM32	32-bit floating point multiplication
FPD32	32-bit floating point division
FLO3232U	32-bit unsigned integer to 32-bit floating point conversion
FLO3232S	32-bit signed integer to 32-bit floating point conversion
FLO1632U	16-bit unsigned integer to 32-bit floating point conversion
FLO1632S	16-bit signed integer to 32-bit floating point conversion
FLO0832U	8-bit unsigned integer to 32-bit floating point conversion
FLO0832S	8-bit signed integer to 32-bit floating point conversion
INT3232	32-bit floating point to 32-bit integer conversion

5.1.2 Calling Convention

The math libraries expect arguments to be provided in the locations AARG and BARG and provide their results in AARG. For example, an integer argument in AARG uses AARG0, AARG1, AARG2, and AARG3. A floating point argument in AARG uses AEXP, AARG0, AARG1, and AARG2. Integer division functions provide the remainder in REM0, REM1, REM2, and REM3.

5.1.3 Example

Given two 32-bit signed integers, `int1` and `int2`, the following code will multiply the two numbers and place the result in `int1`. Banking and paging considerations have been omitted for clarity.

```
MOVFP int1,           WREG        ; Load AARG
MOVWF AARG0
MOVFP int1+1,         WREG
MOVWF AARG1
MOVFP int1+2,         WREG
MOVWF AARG2
MOVPF int1+3,         WREG
MOVWF AARG3
MOVFP int2,           WREG
MOVWF BARG0                       ; Load BARG
MOVFP int2+1,         WREG
MOVWF BARG1
MOVFP int2+2,         WREG
MOVWF BARG2
MOVPF int2+3,         WREG
MOVWF BARG3
CALL FXM3232S                     ; Perform the multiply
MOVFP AARGB0,         WREG        ; Save the result
MOVWF int1
MOVFP AARGB1,         WREG
MOVWF int1+1
MOVFP AARGB2,         WREG
MOVWF int1+2
MOVFP AARGB3,         WREG
MOVWF int1+3
```

NOTES:

Appendix A. Porting Code from MPLAB-C to MPLAB-C17

Introduction

This appendix provides guidelines for migrating code from MPLAB-C to the MPLAB-C17 compiler.

External Differences

These are the main differences that will require changes to the source code when porting an application from MPLAB-C to MPLAB-C17:

- **Software stack** - Allows reuse of memory
- **Pointers** - Far RAM pointers are 16-bits, near RAM pointers are 8-bits, word-aligned ROM pointers are 16-bits, and pointers to 8-bit wide data in ROM are 24-bits.
- **File Locations** - File locations are searched in a more conventional order.
- **MPLIB** - The librarian now can create true library modules.
- **`#pragmas`** are different.
- **Bit fields** are implemented as described in the ANSI standard, and currently limited to one bit. Bits operator is no longer supported.

Internal Differences

Internally, MPLAB-C17 is radically different from MPLAB-C. Among the differences:

- **Software Stack** - This allows more than two parameters to be passed to a function, and allows re-use of memory.
- **Pointers** - Pointers are now 16 bits for RAM and 16 bits for ROM, with 24 bits used for byte data in ROM.
- **Interrupts** - Interrupts are handled in a more general way. Start up code sets up interrupts and initialized data.
- The compiler uses and reserves the shared memory area in RAM
- The compiler runs as a 32-bit console application under Win 95 or NT or as a 32-bit DOS Extended Program under Windows 3.x.

Porting Code

From the differences listed above, refer to the detailed sections below for more information. For reference, there is an example at the end of this section which shows an application written in MPLAB-C converted to MPLAB-C17.

Data Types

The table below outlines the differences between MPLAB-C and MPLAB-C17 data types.

Type	MPLAB-C	MPLAB-C17
char	8-bit (default: unsigned)	8-bit (default: signed)
int	8-bit (switchable to 16-bit)	16-bit
short	8-bit (switchable to 16-bit)	16-bit
long	16-bit	32-bit (future support)
float	N/A	32-bit Microchip modified IEEE754 (future support)
double	N/A	same as float
ANSI bit-fields	no	yes
registerw	W register	N/A
registerx	FSR register	N/A
bits	8-bit	N/A

Code written for MPLAB-C may require the following changes to variables:

- Change 'char' to 'unsigned char' since characters are signed by default.
- Change 'int' to 'char'. If the +i option was used in MPLAB-C (i.e. 16-bit ints), then no change is needed.
- Change 'long' to 'int'.
- The types 'registerw' and 'registerx' are no longer supported. Use WREG and FSR directly but note that they are extremely volatile.

`bits` data type

If the ’bits’ data type is used, add the following structure definition at the top of the file (or in a header file):

```
typedef struct bits_tag
{
  unsigned int b0: 1;
  unsigned int b1: 1;
  unsigned int b2: 1;
  unsigned int b3: 1;
  unsigned int b4: 1;
  unsigned int b5: 1;
  unsigned int b6: 1;
  unsigned int b7: 1;
} bits;
```

Then all references to variables of type **`bits`** must be as follows:

Use: `x.b2 = 1;`

This sets bit `’2’` of `’x’.`

In place of: `x.2   = 1;`

This syntax is no longer supported.

Variable Allocation

General

MPLAB-C17 encourages the use of local variables instead of global variables for RAM conservation. Local variables are allocated on the software stack. Therefore, RAM locations used by local variables are reusable, conversely, the use of global variables conserves ROM.

Using @ to allocate variables at absolute locations

The @ operator is not supported. To access memory at a specific location use a pointer as follows:

```
char *p = 0x35;       // ’p’ points to location 0x35
     *p = 0xF0;       // send value 0xF0 to location 0x35
      p = 0x41;       // ’p’ now points to location 0x41
```

The above code uses the same pointer to access more than one absolute RAM location. To access a fixed location the following syntax can be used since it generates a shorter machine code sequence.

```
#define FIXED35 (* ((char *) 0x35) )

FIXED35 = 0xF0;       // Location 35 now has 0xF0
```

More than one location can be referenced at a time. For example, to write the value `0x1234` in locations `0x40` and `0x41`, use the following construct:

```
#define  LOC4041  (* ( (int *) 0x40) )
LOC4041 = 0x1234;     // Now 0x40 contains 0x34  and location
                         0x41 contains 0x12
```

Please note the following:

1. Locations defined using the above method bypass all variable allocation error checking. Make sure that these locations are not used by other variables.
2. Since these locations are defined as macros, they are not included in the symbol table. Therefore these locations cannot be added to a watch window in MPLAB.

Using @ to allocate local variables in global scratch locations no longer needed

In MPLAB-C17 local variables follow proper scoping rules but are allocated as 'static'. To reuse the space allocated for local variables in MPLAB-C, the use of the @ sign to reuse global RAM was suggested as follows:

```
unsigned char Temp;  // Global variable
void main()
{
  unsigned char Counter @ Temp;
          .
          .
}
void function1()
{
  unsigned char Index @ Temp;
          .
          .
}
```

The above method is no longer needed or supported.

Function arguments using shared global variables

In MPLAB-C function arguments were allocated in RAM and were not reused. A non-standard method for reusing those locations is to declare global variables with the same names as the arguments as follows:

```
char a, b;

func1(a,b) { /* code for func1() */ }

func2(a,b) { /* code for func2() */ }
```

The above syntax is no longer supported nor needed since MPLAB-C17 allocates function arguments on the stack. The space used by these arguments is reused once the function goes out of scope.

Use #PRAGMA IDATA, UDATA, ROMDATA to allocate specific addresses for data

Variables can be located at fixed addresses in memory with the following declarations:

```
#pragma idata GPR2
unsigned char temp1 = 0x40;
unsigned char temp2 = 0x80;
```

This will cause the two variables temp1 and temp2 to be allocated in the area defined by GPR2, the second bank of general purpose registers and will initialize their values on start up to `0x40` and `0x80`.

To allocate variables with uninitialized data use:

```
#pragma udata GPR0
unsigned char temp4,temp5;
```

To allocate storage for data in ROM, use:

```
#pragma romdata
char temp[] = "This is a message";
```

This puts the string into the current code page.

(Refer to chapter 3 for the `#pragma` preprocessor directives.)

Code Allocation

Allocating code at a specific address using ORG or #pragma memory ROM

MPLAB-C allowed the allocation of pieces of code at absolute locations. This was done either by using the assembler directive ORG, or the compiler directive `#pragma memory ROM`.

The method is no longer supported. To allocate code at a specific location, compile it in a separate module. Then, in the linker script, specify that this module is allocated in an absolute section. Specify the absolute address where a section is to be allocated in the linker script file. Refer to MPASM with MPLINK and MPLIB User's Guide for further instructions on creating absolute sections.

You can also change the code section by using

```
#pragma code mycode=0x300
```

This will change the allocation of subsequent code into the new section called `mycode` which begins at address 0x300.

Access to pre-loaded code in ROM

MPLAB-C17 does not support using the @ sign with a function prototype to enable access to code that is pre-loaded in program memory. To access code that is hard-coded at specific locations (such as A/D calibration constants on PIC14000 that are at address 0xFC0 and up), use function pointers:

```
unsigned char (*AtodCalibration)() ;

AtodCalibration = 0xFC0;  // assign the address

k1 = AtodCalibration();   // Call and read first constant
```

Header Files and Libraries

Header file inclusion

In MPLAB-C17, the behavior of #include directive has changed to a more conventional usage:

#include <filename.ext> searches the path defined by the environment variable MCC_INCLUDE only. The compiler will not search for the file in the DOS path like MPLAB-C.

#include "filename.ext" searches the current directory for the filename and if it doesn't find it, uses the path defined by the environment variable MCC_INCLUDE.

Libraries

In MPLAB-C libraries were created by enclosing C code between a #pragma library, and a #pragma endlibrary directives. Then include files were created with prototypes to the library functions. To use the functions, use the #include directive to include the header file at the top of the file, and include the library at the end.

In MPLAB-C17 libraries are created using MPLIB, the librarian. Object modules can be added or removed to libraries with MPLIB. MPLAB-C17 allows more conventional library access, so including the source library is no longer required. To use a library function in an application, use the #include directive to include the header file that contains the prototypes for that function in the appropriate source file. Then the library will need to be linked with MPLINK. Please refer to MPASM with MPLINK and MPLIB User's Guide for more information.

Appendix A. Porting Code from MPLAB-C to MPLAB-C17

The use of `const`

The use of the keyword `const` has changed since version 1.2.1 MPLAB-C. It no longer means that the data object is stored in ROM but rather it follows the ANSI specification and specifics that its contents cannot be modified. To place a data object in ROM, explicitly use the `rom` keyword. For example:

```
const char Msg[]     = "Hello world!\n"; // Allocated in
                                         //  RAM.

const rom char Msg[] = "Hello world!\n"; // Allocated in
                                         //  ROM and
                                         //  cannot be
                                         //  modified.

rom char Msg[]= "Hello world!\n";   // Allocated in
                                         //  ROM and
                                         //  can be
                                         //  modified.
```

Since program memory can be written on PIC17CXXX devices, placing an object in ROM doesn't necessarily mean it's read-only. On such devices, both the `rom` and `const` keywords must be used if the object is to be declared read-only.

Inline assembler support

In MPLAB-C17 the inline assembler has a different syntax from MPLAB-C.

To assemble a single instruction, place that instruction after the `_asm` directive. For example:

```
_asm  MOVLW 0x01 // Put a comment following double
                 //   forward slashes
```

If code has multiple assembly instructions enclosed between `#asm` and `#endasm`, change it to use `_asm` and `_endasm` instead.

For example:

```
#asm
         MOVLW 9      ;Move 9 into W
         ADDWF 0x1A   ;Add 26 to W
         MOVWF  PORTB ;Move W to PORTB
#endasm
```

must be changed to:

```
_asm
         MOVLW  9     // Move 9 into W
         ADDWF 0x1A   // Add 26 to W
```

```
         MOWVF  PORTB // Move W to PORTB
_endasm
```

The MPASM assembler directives or labels cannot be used. GOTOs that jump to a C label are valid.

For example:

```
_asm
       . . .          // some assembler code
       goto MyLabel   // jump to a C label
_endasm
MyLabel:              // C label
                x++;
_asm
      // more assembly code
_endasm
```

For the features of a full-macro assembler, separate the assembly routines in a separate file, assemble them using MPASM, and then link the resulting object file with the C program. For more information, refer to Chapter 6 and the MPASM with MPLINK and MPLIB User's Guide.

`Switch..case` support

ANSI C `switch..case` statements are supported but MPLAB-C extensions are not. Ranges, values separated by commas, and variables in 'case' statements are not supported. Code that uses these extensions will need to be modified.

For example:

```
switch(x)
{
  case 0..4:  /* Range of numbers - not supported */
                  ProcessNumbers();
                  break;
  case 'a','b': /* Values separated by commas - not supported */
                  ProcessAB();
                  break;
  case y:     /* Variable - not supported */
                  ProcessY();
                  break;
}
```

must be changed to:


```
switch(x)
{
  case 0:
  case 1:
  case 2:
  case 3:
  case 4: /* Range of numbers */
                 ProcessNumbers();
                 break;

  case 'a':
  case 'b': /* Values separated by commas */
                 ProcessAB();
                 break;

          /* Variables in a 'case' label expression
              are not allowed */
}
```

#pragma directives

#pragma directives are, by definition, implementation specific. None of the **#pragma** directives defined in MPLAB-C are valid directives in MPLAB-C17. These are the **#pragmas** for MPLAB-C17. Refer to Chapter 3 for more details:

- **nocontext** - Disable stack frame code for following function.
- **nosaveregs** - Disable save/restore of working registers for the next function.
- **list** - Turn on list file generation.
- **nolist** - Turn off list file generation
- **code** - For the following data, change to the specified code section
- **idata** - For the following data, change to the specified initialized data section.
- **udata** - For the following data, change to the specified uninitialized data section.
- **romdata** - For the following data, change to the specified ROM section.
- **varlocate** - For the following data, tell the compiler that it resides in the specified bank.

Porting Code from MPLAB-C to MPLAB-C17 Checksheet

- ☐ Search for `org` statements and replace with section directives #pragma `code`
- ☐ Change header file names to MPLAB-C17 standard
- ☐ Search for @ operator and replace with pointer or allow MPLINK to allocate space
- ☐ Search for `long` and replace with `int`
- ☐ Remove library `include` references and add library to MPLINK
- ☐ Scan for `bits` directive and replace with ANSI bit structures
- ☐ Change bit access to SFR's (PORTA.1) to ANSI format (`PORTAbits.RA1`).
- ☐ Scan for `int` and `short` usage
  - ☐ Does it require 16-bits? No, then change to `char`
  - ☐ Does it require sign? No then change to `unsigned`
- ☐ Scan for `char` usage
  - ☐ Does it require `sign` bit? No, then change to `unsigned`
- ☐ Scan for `#asm` and `#endasm` and change to `_asm` and `_endasm`
- ☐ Change "`;`" comments in `#asm` segments to "`//`" comments
- ☐ Check that `switch..case` statements do not have ranges or commas
- ☐ Search for `const` statements and add `rom` keyword to keep data in ROM

Example Code Ported from MPLAB-C to MPLAB-C17

The following two listings are included as a reference for converting code from MPLAB-C to MPLAB-C17. The first file compiles under MPLAB-C and the second is a translated version that will compile under MPLAB-C17.

MPLAB-C Portion of Header File Example

```
/*********************************************************************************
*   from PICmicro C Libraries                                                    *
*   Written and Tested using MPLAB-C                                             *
*********************************************************************************
*   Filename:                               xlcd.h                               *
*********************************************************************************


// DATA_PORT defines the port on which the LCD
// data lines are connected to
#define DATA_PORT PORTF
#define TRIS_DATA_PORT TRISF

// Control Signals
#define RS 1                                                // Register Select bit
#define RW 0                                                // Read/Write bit
#define E  6                                                // Clock bit

/ CTRL_PORT defines the port where the control
// lines are connected
#define RW_PIN   PORTG.RW                                   // Port for RW
#define TRIS_RW  TRISG.RW                                   // TRIS for RW
#define RS_PIN   PORTG.RS                                   // Port for RS
#define TRIS_RS  TRISG.RS                                   // TRIS for RS
#define E_PIN    PORTF.E                                    // PORT for E
#define TRIS_E   TRISF.E                                    // TRIS for E

// Display ON/OFF Control defines
#define DON                     0b00001111                  // Display on
#define DOFF                    0b00001011                  // Display off
#define CURSOR_ON               0b00001111                  // Cursor on
#define CURSOR_OFF              0b00001101                  // Cursor off
#define BLINK_ON                0b00001111                  // Cursor Blink
#define BLINK_OFF               0b00001110                  // Cursor No Blink
```


MPLAB-C17 Portion of Header File Example

```
*********************************************************************************
*   from PICmicro C Libraries                                                   *
*   Written and Tested using MPLAB-C17                                          *
*********************************************************************************
*   Filename:                              xlcd.h                               *
*********************************************************************************


// DATA_PORT defines the port on which the LCD
// data lines are connected to
#define DATA_PORT PORTF
#define TRIS_DATA_PORT DDRF

// Control Signals
#define RS 1                                                // Register Select bit
#define RW 0                                                // Read/Write bit
#define E  6                                                // Clock bit

// CTRL_PORT defines the port where the control
// lines are connected
#define RW_PIN  PORTGbits.RG0                               // Port for RW
#define TRIS_RW  DDRGbits.RG0                               // TRIS for RW
#define RS_PIN  PORTGbits.RG1                               // Port for RS
#define TRIS_RS  DDRGbits.RG1                               // TRIS for RS
#define E_PIN   PORTFbits.RF6                               // PORT for E
#define TRIS_E   DDRFbits.RF6                               // TRIS for E

// Display ON/OFF Control defines
#define DON                     0b00001111                  // Display on
#define DOFF                    0b00001011                  // Display off
#define CURSOR_ON               0b00001111                  // Cursor on
#define CURSOR_OFF              0b00001101                  // Cursor off
#define BLINK_ON                0b00001111                  // Cursor Blink
#define BLINK_OFF               0b00001110                  // Cursor No Blink
```

MPLAB-C Source File Example

```
#pragma library

#pragma option +l

/******************************************************************************
* Selected code from PICmicro C Libraries V1.00 (BETA)
* This demonstrates how the code would
*    be written for MPLAB-C
* Some of the conditional assembly and comments from the original
*  library file were removed for this example
******************************************************************************/


void SetCGRamAddr(char CGaddr)
{
                                                    // Lower nibble interface
      TRIS_DATA_PORT = TRIS_DATA_PORT & 0xf0;        // Make nibble input
      DATA_PORT = DATA_PORT & 0xf0;                  // and write upper nibble
      DATA_PORT = DATA_PORT | (((CGaddr |0b01000000)>>4) & 0x0f);

      RW_PIN = 0;                                    // Set control signals
      RS_PIN = 0;
      DelayFor18TCY();
      E_PIN = 1;                                     // Clock cmd and address in
      DelayFor18TCY();
      E_PIN = 0;


                                                    // Lower nibble interface
      DATA_PORT = DATA_PORT & 0xf0;                  // Write lower nibble
      DATA_PORT = DATA_PORT | (CGaddr&0x0f);

      DelayFor18TCY();
      E_PIN = 1;                                     // Clock cmd and address in
      DelayFor18TCY();
      E_PIN = 0;


                                                    // Lower nibble interface
      TRIS_DATA_PORT = TRIS_DATA_PORT | 0x0f;        // Make inputs

      return;
}
```


MPLAB-C17 Source File Example

```
#include <p17c756.h>
#include "xlcd.h"
/*********************************************************************************
*   Selected code from PICmicro C Libraries V2.00 (BETA)                          *
*   Written and Tested using MPLABC V2.00                                         *
*********************************************************************************

void SetCGRamAddr(char CGaddr)
{
                                                        // Lower nibble interface
     TRIS_DATA_PORT &= 0xf0;                            // Make nibble input
     DATA_PORT &= 0xf0;                                 // and write upper nibble
     DATA_PORT |= (((CGaddr |0b01000000)>>4) & 0x0f);

     RW_PIN = 0;                                        // Set control signals
     RS_PIN = 0;
     DelayFor18TCY();
     E_PIN = 1;                                         // Clock cmd and address in
     DelayFor18TCY();
     E_PIN = 0;

                                                        // Lower nibble interface
     DATA_PORT &= 0xf0;                                 // Write lower nibble
     DATA_PORT |= (CGaddr&0x0f);

     DelayFor18TCY();
     E_PIN = 1;                                         // Clock cmd and address in
     DelayFor18TCY();
     E_PIN = 0;

                                                        // Lower nibble interface
     TRIS_DATA_PORT |= 0x0f;                            // Make inputs

     return;
}
```

NOTES:

Appendix B. ASCII Character Set

Introduction

This appendix contains the ASCII character set.

ASCII Character Set

Most Significant Character

Least Significant Character

<table>
<tr><th>Hex</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr>
<tr><td>0</td><td>NUL</td><td>DLE</td><td>Space</td><td>0</td><td>@</td><td>P</td><td>‘</td><td>p</td></tr>
<tr><td>1</td><td>SOH</td><td>DC1</td><td>!</td><td>1</td><td>A</td><td>Q</td><td>a</td><td>q</td></tr>
<tr><td>2</td><td>STX</td><td>DC2</td><td>"</td><td>2</td><td>B</td><td>R</td><td>b</td><td>r</td></tr>
<tr><td>3</td><td>ETX</td><td>DC3</td><td>#</td><td>3</td><td>C</td><td>S</td><td>c</td><td>s</td></tr>
<tr><td>4</td><td>EOT</td><td>DC4</td><td>$</td><td>4</td><td>D</td><td>T</td><td>d</td><td>t</td></tr>
<tr><td>5</td><td>ENQ</td><td>NAK</td><td>%</td><td>5</td><td>E</td><td>U</td><td>e</td><td>u</td></tr>
<tr><td>6</td><td>ACK</td><td>SYN</td><td>&</td><td>6</td><td>F</td><td>V</td><td>f</td><td>v</td></tr>
<tr><td>7</td><td>Bell</td><td>ETB</td><td>’</td><td>7</td><td>G</td><td>W</td><td>g</td><td>w</td></tr>
<tr><td>8</td><td>BS</td><td>CAN</td><td>(</td><td>8</td><td>H</td><td>X</td><td>h</td><td>x</td></tr>
<tr><td>9</td><td>HT</td><td>EM</td><td>)</td><td>9</td><td>I</td><td>Y</td><td>i</td><td>y</td></tr>
<tr><td>A</td><td>LF</td><td>SUB</td><td>*</td><td>:</td><td>J</td><td>Z</td><td>j</td><td>z</td></tr>
<tr><td>B</td><td>VT</td><td>ESC</td><td>+</td><td>;</td><td>K</td><td>[</td><td>k</td><td>{</td></tr>
<tr><td>C</td><td>FF</td><td>FS</td><td>,</td><td><</td><td>L</td><td>\</td><td>l</td><td>|</td></tr>
<tr><td>D</td><td>CR</td><td>GS</td><td>-</td><td>=</td><td>M</td><td>]</td><td>m</td><td>}</td></tr>
<tr><td>E</td><td>SO</td><td>RS</td><td>.</td><td>></td><td>N</td><td>^</td><td>n</td><td>~</td></tr>
<tr><td>F</td><td>SI</td><td>US</td><td>/</td><td>?</td><td>O</td><td>_</td><td>o</td><td>DEL</td></tr>
</table>

NOTES:

Appendix C. Detailed MPLAB-C17 Example

Introduction

This appendix gives an example of actual working source code with comments included. This example is included on the distribution disk along with other examples not included in this User's Guide.

Highlights

This appendix presents the following example:

- **Flashing LEDs**

Flashing LEDs

```
// File: 17c42a.h

#ifndef __17C42A_H
#define __17C42A_H

extern unsigned char INDF0;
extern unsigned char FSR0;
extern unsigned char PCL;
extern unsigned char PCLATH;
extern unsigned char ALUSTA;
extern unsigned char T0STA;
extern unsigned char CPUSTA;
extern unsigned char INTSTA;
extern unsigned char INDF1;
extern unsigned char FSR1;
extern unsigned char WREG;
extern unsigned int  TMR0;   /* same location as TMR0L/H */
extern unsigned char TMR0L;
extern unsigned char TMR0H;
extern unsigned int  TBLPTR; /* same location as TBLPTRL/H */
extern unsigned char TBLPTRL;
extern unsigned char TBLPTRH;
extern unsigned char BSR;

extern struct
{
    unsigned C:1;
    unsigned DC:1;
    unsigned Z:1;
    unsigned OV:1;
    unsigned FS0:1;
    unsigned FS1:1;
    unsigned FS2:1;
    unsigned FS3:1;
} ALUSTAbits;

/* Bank 0 SFR's */
extern far  unsigned char PORTA;
extern near unsigned char DDRB;
extern far  unsigned char PORTB;
extern far  unsigned char RCSTA;
extern far  unsigned char RCREG;
```

```
extern far  unsigned char TXSTA;
extern far  unsigned char TXREG;
extern far  unsigned char SPBRG;

extern far union
{
    struct
    {
        unsigned RA0:1;     /* Bit 0                      */
        unsigned RA1:1;
        unsigned RA2:1;
        unsigned RA3:1;
        unsigned RA4:1;
        unsigned RA5:1;
        unsigned :1;
        unsigned NOT_RBPU:1;
    };
    struct
    {
        unsigned INT:1;     /* Alternate name for bit 0 */
        unsigned T0CKI:1;   /* Alternate name for bit 1 */
        unsigned :6;        /* pad it */
    };
} PORTAbits;
extern far union
{
    struct
    {
        unsigned RCD8:1;
        unsigned OERR:1;
        unsigned FERR:1;
        unsigned :1;
        unsigned CREN:1;
        unsigned SREN:1;
        unsigned RC8:1;
        unsigned SPEN:1;
    };
    struct
    {
        unsigned :6;
        unsigned RC9:1;     /* Alternate name for bit 6 */
    };
} RCSTAbits;

extern far union
{
    struct
    {
        unsigned TXD8:1;
        unsigned TRMT:1;
        unsigned :1;
        unsigned :1;
        unsigned SYNC:1;
        unsigned TXEN:1;
        unsigned TX8:1;
        unsigned CSRC:1;
    };
    struct
    {
        unsigned :6;
        unsigned TX9:1;     /* Alternate name for bit 6 */
    };
} TXSTAbits;
```

```
/* Bank 1 SFR's */
extern near unsigned char DDRC;
extern far  unsigned char PORTC;
extern near unsigned char DDRD;
extern far  unsigned char PORTD;
extern near unsigned char DDRE;
extern far unsigned char PORTE;
extern far unsigned char PIR;
extern far unsigned char PIE;

/* Bank 2 SFR's */
extern far unsigned char TMR1;
extern far unsigned char TMR2;
extern far unsigned int  TMR3; /* same location as TMR3L/H */
extern far unsigned char TMR3L;
extern far unsigned char TMR3H;
extern far unsigned char PR1;
extern far unsigned char PR2;
extern far unsigned int  PR3;  /* same location as PR3L/H  */
extern far unsigned char PR3L;
extern far unsigned char PR3H;

/* Bank 3 SFR's */
extern far unsigned char PW1DCL;
extern far unsigned char PW2DCL;
extern far unsigned char PW1DCH;
extern far unsigned char PW2DCH;
extern far unsigned int  CA2; /* same location as CA2L/H  */
extern far unsigned char CA2L;
extern far unsigned char CA2H;
extern far unsigned char TCON1;
extern far unsigned char TCON2;

#endif

; File: 17C42a.asm
LIST P=17C42A

SFR0   UDATA

       GLOBAL  INDF0, FSR0, PCL, PCLATH, ALUSTA, T0STA, CPUSTA
       GLOBAL  INTSTA, INDF1, FSR1, WREG, TMR0, TMR0L, TMR0H
       GLOBAL  TBLPTR, TBLPTRL, TBLPTRH
       GLOBAL  ALUSTAbits, T0STAbits, CPUSTAbits

INDF0           RES   1    ;  0x000
FSR0            RES   1    ;  0x001
PCL             RES   1    ;  0x002
PCLATH          RES   1    ;  0x003
ALUSTAbits
ALUSTA          RES   1    ;  0x004
T0STAbits
T0STA           RES   1    ;  0x005
CPUSTAbits
CPUSTA          RES   1    ;  0x006
INTSTA          RES   1    ;  0x007
INDF1           RES   1    ;  0x008
FSR1            RES   1    ;  0x009
WREG            RES   1    ;  0x00A
TMR0
TMR0L           RES   1    ;  0x00B
TMR0H           RES   1    ;  0x00C
TBLPTR
TBLPTRL         RES   1    ;  0x00D
```

```
TBLPTRH         RES   1    ;  0x00E
BSR             RES   1    ;  0x00F

;----- Bank 0 Special Function Registers --------------------

PORTA
PORTAbits       RES   1    ;  0x010
DDRB            RES   1    ;  0x011
PORTB           RES   1    ;  0x012
RCSTAbits
RCSTA           RES   1    ;  0x013
RCREG           RES   1    ;  0x014
TXSTAbits
TXSTA           RES   1    ;  0x015
TXREG           RES   1    ;  0x016
SPBRG           RES   1    ;  0x017

        GLOBAL  PORTA, DDRB, PORTB, RCSTAbits, RCSTA, RCREG
        GLOBAL  TXSTAbits, TXSTA, TXREG, SPBRG
        GLOBAL  PORTAbits

;----- Bank 1 Special Function Registers --------------------


SFR1   UDATA

    GLOBAL  DDRC, PORTC, PORTCbits, DDRD, PORTD, PORTDbits
    GLOBAL  DDRE, PORTE, PORTEbits, PIR, PIE

DDRC            RES 1   ;   0X110
PORTC
PORTCbits       RES 1   ;   0x111
DDRD            RES 1   ;   0X112
PORTD
PORTDbits       RES 1   ;   0x113
DDRE            RES 1   ;   0X114
PORTE
PORTEbits       RES 1   ;   0x115
PIR             RES 1   ;   0x116
PIE             RES 1   ;   0x117

    END

// File:  LED42.C

#include "17C42A.H"

#define ROLF( Bank, Address ) _asm   movlb  Bank    _endasm \
                              _asm  rlcf    Address _endasm

#define SetBank _asm    movlb   0x01    _endasm

/* Prototypes */
void main( void );
void delay( void );
void WriteToPORTA( void );
void WriteToPORTB( void );
void WriteToPORTC( void );
void WriteToPORTD( void );
void FlashAll( unsigned char * );


unsigned    char    i;
unsigned    char    count1;
unsigned    char    count2;
unsigned    char    flashcount;
```

```
#pragma nocontext
#pragma nosaveregs

void main( void )
{
       PORTB = 0xff;   // CLEAR PORT B register
       DDRB  = 0x00;   // Set Port B as Output
       PORTC = 0xff;   // Clear Port C Register
       DDRC  = 0x00;   // Set Port C as output
       PORTD = 0xff;   // Clear Port D Register
       DDRD  = 0x00;   // Set Port D as output

       FlashAll( &flashcount );

       goto    main;

} /* end main */

#pragma nocontext
#pragma nosaveregs

void WriteToPORTA()
{
       for( i = 2; i < 4; i++ )
       {
           PORTA = 0xff;
           PORTA = ~( 1 << i );
           delay();
       } /* end for */

    PORTA = 0xff;
} /* WriteToPORTA */

#pragma nocontext
#pragma nosaveregs

void WriteToPORTB()
{
       for( i = 1; i != 0; i += i )
       {
           PORTB = 0xff;
           PORTB = ~i;
           delay();
       } /* end for */

       PORTB = 0xff;
} /* end WriteToPORTB */

#pragma nocontext
#pragma nosaveregs

void WriteToPORTC()
{
       PORTC = 0xfe;

       do
       {
           delay();
           ALUSTA |= 0x01;
       ROLF( 1, PORTC );
       }while( ALUSTAbits.C );

} /* end WriteToPORTC */
```

```
#pragma nocontext
#pragma nosaveregs

void WriteToPORTD()
{
       for( i = 0; i < 8; i++ )
       {
           PORTD = 0xff;
           PORTD = ~( 1 << i );
           delay();
       } /* end for */

    PORTD = 0xff;
} /* end WriteToPORTD */

void FlashAll( unsigned char *flashcount )
{
    for( *flashcount = 0; *flashcount < 5; *flashcount++ )
    {
        PORTB       = 0X00;
        PORTC       = 0X00;
        PORTD       = 0X00;
        delay();

        PORTB       = 0XFF;
        PORTC       = 0XFF;
        PORTD       = 0XFF;
        delay();
    } /* end for */

} /* end FlashAll */
```


Linker File to Link Flashing LEDs Example

```
// File: led42.lkr
// Example Linker Command File For a PIC17C42A
//
// The Linker supports the following command line options:
//   -o <filename>    : specify output file 'filename'
//   -m <filename>    : create map file 'filename'
//   -L <libpath>     : additional library directory for
                           search path
//   -s               : strip symbol table and line info
                           from output
//
// The linker command file is used:
//   1) To specify an additional directory for the library
         search path
//   2) To specify the object files for linking
//   3) To include additional linker command files
//   4) To define the target's memory architecture
//   5) To locate sections within the target's memory
//
// The following statement specifies an additional directory
// for the library search path:
//   LIBPATH 'libpath' ['libpath'...]
// where,
//   'libpath' is an absolute path to the directory containing
// a library.  Note, more than one path can specified in a
   single
// LIBPATH statement.
//
//
// The following statement specifies object files for linking:
//   FILES 'objfile' ['objfile'...]
// where,
//   'objfile' is an object file.  Note, more than one object
     file can be
// specified in a single FILES statement.
//
//
// The following statement includes an additional linker
   command file:
//   INCLUDE 'cmdfile'
// where,
//   'cmdfile' is the name of the linker cmd file to include.
     Note,
// command line options in an included linker cmd file are
   ignored.
//
//
// The following statements define portions of the target's
   memory
// by specifiying a name for a block of memory, its starting
   address,
// and its ending address:
//   DATABANK   NAME='memName' START='addr' END='addr'
//   CODEPAGE   NAME='memName' START='addr' END='addr'
//   SHAREBANK  NAME='memName' START='addr' END='addr'
// where,
//   'memName'  is any ASCII string used to identify a
     DATABANK,
//              CODEPAGE, or SHAREBANK
//   'addr'       is a decimal or hexadecimal number
```

```
                    specifying an address
// The SHAREBANK statement identifies a region in RAM which
   is mapped across
// mulitple banks. Note, a SHAREBANK statement should be
   given for each bank that
// shares a region and each of these statements should have
   the same NAME.
//
//
// The following statement defines a section by specifying
   its name,
// the block of memory in which to load the section, and
   optionally,
// the block of memory in which to run the section:
//   SECTION  NAME='secName' LOAD='memName' RUN='memName'
// where,
//   'secName'  is an ASCII string used to identify a
     SECTION, this is the
//              same name for the section in the COFF file
//   'memName'  is a previously defined DATABANK or CODEPAGE
// The optional run block allows sections which contain
   initialized data
// to be stored in a CODEPAGE (ROM) and copied to a DATABANK
   (RAM) at runtime.

CODEPAGE    NAME=reset_vector   START=0x0000   END=0x0007
// Reset Vector
CODEPAGE    NAME=page0          START=0x0022   END=0x1FFF
// On chip memory

DATABANK    NAME=sfr0           START=0x00     END=0x1F
PROTECTED
DATABANK    NAME=sfr1           START=0x0110   END=0x117
PROTECTED
DATABANK    NAME=gpr0           START=0x20     END=0x7F
// GPRs Bank 0
DATABANK    NAME=stack          START=0x80     END=0xFF
// Stack RAM

SECTION     NAME=SFR0           RAM=sfr0
// Data segments defined
SECTION     NAME=SFR1           RAM=sfr1
// in 17C42A.asm
SECTION     NAME=.bss_t.o       RAM=gpr0
// .bss section resides in RAM
STACK SIZE=0x7F RAM=stack
```


Appendix D. PIC17CXXX Instruction Set

Introduction

This appendix gives the instruction set for the PIC17CXXX device family.

Highlights

This appendix presents the following reference information:

- **PIC17CXXX Instruction Set**

PIC17CXXX Instruction Set

The PIC17CXXX, Microchip’s high-performance 8-bit microcontroller family, uses a 16-bit wide instruction set. The PIC17CXXX instruction set consists of 58 instructions, each a single 16-bit wide word. Most instructions operate on a file register, f, and the working register, W (accumulator). The result can be directed either to the file register or the W register or to both in the case of some instructions. Some devices in this family also include hardware multiply instructions. A few instructions operate solely on a file register (BSF for example).

Table D.1: PIC17CXXX Literal and Control Operations

Mnemonic		Description	Function
MOVFP	f,p	Move f to p	f → p
MOVLB	k	Move literal to BSR	k → BSR
MOVLP	k	Move literal to RAM page select	k → BSR <7:4>
MOVPF	p,f	Move p to f	p → W
MOVWF	f	Move W to F	W → f
TABLRD	t,i,f	Read data from table latch into file f, then update table latch with 16-bit contents of memory location addressed by table pointer	TBLATH → f if t=1, TBLATL → f if t=0; ProgMem(TBLPTR) →TBLAT TBLPTR+1 →TBLPTR if i=1

Table D.1: PIC17CXXX Literal and Control Operations (Continued)

Mnemonic		Description	Function
TABLWT	t,i,f	Write data from file f to table latch and then write 16-bit table latch to program memory location addressed	f → TBLATH if t = 1, f → TBLATL if t = 0; TBLAT →ProgMem(TBLPTR); TBLPTR+1 → TBLPTR if i=1
TLRD	t,f	Read data from table latch into file f (table latch unchanged)	TBLATH → f if t = 1 TBLATL → f if t =
TLWT	t,f	Write data from file f	f → TBLATH if t = 1 f → TBLATL if t = 0
ADDLW	k	Add literal to W	(W + k) → W
ADDWF	f,d	Add W to F	(W + f) → d
ADDWFC	f,d	Add W and Carry to f	(W + f + C) → d
ANDLW	k	AND Literal and W	(W .AND. k) → W
ANDWF	f,d	AND W with f	(W .AND. f) → d
CLRF	f,d	Clear f and Clear d	0x00 → f,0x00 → d
COMF	f,d	Complement f	.NOT. f → d
DAW	f,d	Dec. adjust W, store in f,d	W adjusted → f and d
DECF	f,d	Decrement f	(f - 1) → f and d
INCF	f,d	Increment f	(f + 1) → f and d
IORLW	k	Inclusive OR literal with W	(W .OR. k) → W
IORWF	f,d	Inclusive or W with f	(W .OR. f) → d
MOVLW	k	Move literal to W	k → W
MULLW	k	Multiply literal and W	(k x W) → PH, PL
MULWF	f	Multiply W and f	(W x f) → PH, PL
NEGW	f,d	Negate W, store in f and d	(W + 1) → f,(W + 1) → d
RLCF	f,d	Rotate left through carry	
RLNCF	f,d	Rotate left (no carry)	

Table D.1: PIC17CXXX Literal and Control Operations (Continued)

Mnemonic	Description	Function
RRCF f,d	Rotate right through carry	
RRNCF f,d	Rotate right (no carry)	
SETF f,d	Set f and Set d	0xff → f,0xff → d
SUBLW k	Subtract W from literal	(k - W) → W
SUBWF f,d	Subtract W from f	(f - W) → d
SUBWFB f,d	Subtract from f with	(f - W - c) → d
SWAPF f,d	Swap ff	(0:3) → d(4:7), f(4:7) → d(0:3)
XORLW k	Exclusive OR literal	(W .XOR. k) → W
XORWF f,d	Exclusive OR W with f	(W .XOR. f) → d

Table D.2: PIC17CXXX Bit Handling Instructions

Mnemonic	Description	Function
BCF f,b	Bit clear f	0 → f(b)
BSF f,b	Bit set f	1 → f(b)
BTFSC f,b	Bit test, skip if clear	skip if f(b) = 0
BTFSS f,b	Bit test, skip if set	skip if f(b) = 1
BTG f,b	Bit toggle f	.NOT. f(b) → f(b)

Table D.3: PIC17CXXX Program Control Instructions

Mnemonic	Description	Function
CALL k	Subroutine call (within 8k page)	PC+1 → TOS,k → PC(12:0), k(12:8) → PCLATH(4:0), PC(15:13) → PCLATH(7:5)
CPFSEQ f	Compare f/w, skip if f = w	f-W, skip if f = W
CPFSGT f	Compare f/w, skip if f > w	f-W, skip if f > W

Table D.3: PIC17CXXX Program Control Instructions (Continued)

Mnemonic	Description	Function
CPFSLT f	Compare f/w, skip if f< w	f-W, skip if f < W
DECFSZ f,d	Decrement f, skip if 0	(f-1) $\rightarrow$ d, skip if 0
DCFSNZ f,d	Decrement f, skip if not 0	(f-1) $\rightarrow$ d, skip if not 0
GOTO k	Unconditional branch (within 8k)	k $\rightarrow$ PC(12:0) k(12:8) $\rightarrow$ f3(4:0),
INFSNZ f,d	Increment f, skip if not zero	(f+1) $\rightarrow$ d, skip if not 0
LCALL k	Long Call (within 64k)	(PC+1) $\rightarrow$ TOS; k $\rightarrow$ PCL,
RETFIE	Return from interrupt, enable interrupt	(f3) $\rightarrow$ PCH:k $\rightarrow$ PCL
RETLW k	Return with literal in W	k $\rightarrow$ W, TOS $\rightarrow$ PC, (f3 unchanged)
RETURN	Return from subroutine	TOS $\rightarrow$ PC
TSTFSZ f	Test f, skip if zero	skip if f = 0

PIC17CXXX Special Control Instructions

Mnemonic	Description	Function
CLRWDT	Clear watchdog timer	0 $\rightarrow$ WDT,0 $\rightarrow$ WDT prescaler, 1 $\rightarrow$ PD, 1 $\rightarrow$ TO
NOP	No operation	None
SLEEP	Enter Sleep Mode	Stop oscillator, power down, 0 $\rightarrow$ WDT, 0 $\rightarrow$ WDT Prescaler 1 $\rightarrow$ PD, 1 $\rightarrow$ TO

Appendix E. References

Introduction

This appendix gives references that may be helpful in programming with MPLAB-C17.

Highlights

This appendix lists the following reference types:

- **General C Information**
- **C Standards Information**

References

American National Standard for Information Systems – *Programming Language – C*. American National Standards Institute (ANSI), 11 West 42nd. Street, New York, New York, 10036.

This standard specifies the form and establishes the interpretation of programs expressed in the programming language C. Its purpose is to promote portability, reliability, maintainability, and efficient execution of C language programs on a variety of computing systems.

Harbison, Samuel P., and Steele, Guy L., *C A Reference Manual*, Fourth Edition, Prentice-Hall, Englewood Cliffs, New Jersey 07632

A best selling authoritative reference for the C programming language.

Kernighan, Brian W., and Ritchie, Dennis M. *The C Programming Language*, Second Edition. Prentice Hall, Englewood Cliffs, New Jersey 07632

Presents a concise exposition of C as defined by the ANSI standard. This book is an excellent reference for C programmers.

NOTES:

Appendix F. On-Line Support

Introduction

Microchip provides on-line support via the Microchip World Wide Web (WWW) site.

The web site is used by Microchip as a means to make files and information easily available to customers. To view the site, the user must have access to the Internet and a web browser, such as Netscape Navigator or Microsoft Internet Explorer. Files are also available for FTP download from our FTP site.

Connecting to the Microchip Internet Web Site

The Microchip web site is available by using your favorite Internet browser to attach to:

www.microchip.com

The file transfer site is available by using an FTP service to connect to:

ftp://ftp.futureone.com/pub/microchip

The web site and file transfer site provide a variety of services. Users may download files for the latest Development Tools, Datasheets, Application Notes, User's Guides, Articles and Sample Programs.

In addition to technical documentation, a variety of Corporate information is also available:

- Microchip Sales Offices, Distributors and Factory Representatives
- Latest Microchip Press Releases
- Technical Support Section with Frequently Asked Questions
- Design Tips
- Device Errata
- Job Postings
- Microchip Consultant Program Member Listing
- Links to other useful Web Sites related to Microchip Products

Software Releases

Software products released by Microchip are referred to by version numbers. Version numbers use the form:

`xx.yy.zz`

Where `xx` is the major release number, `yy` is the minor number, and `zz` is the intermediate number.

Intermediate Release

Intermediate released software represents changes to a released software system and is designated as such by adding an intermediate number to the version number. Intermediate changes are represented by:

- Bug Fixes
- Special Releases
- Feature Experiments

Intermediate released software does not represent our most tested and stable software. Typically, it will not have been subject to a thorough and rigorous test suite, unlike production released versions. Therefore, customers should use these versions with care, and only in cases where the features provided by an intermediate release are required.

Intermediate releases are primarily available through the Microchip Web Site.

Production Release

Production released software is software shipped with tool products. Example products are PRO MATE II, PICSTART Plus, and PICMASTER. The Major number is advanced when significant feature enhancements are made to the product. The minor version number is advanced for maintenance fixes and minor enhancements. Production released software represents Microchip's most stable and thoroughly tested software.

There will always be a period of time when the Production Released software is not reflected by products being shipped until stocks are rotated. You should always check the Microchip Web Site for the current production release.

Systems Information and Upgrade Hot Line

The Systems Information and Upgrade Line provides system users a listing of the latest versions of all of Microchip's development systems software products. Plus, this line provides information on how customers can receive any currently available upgrade kits. The Hot Line Numbers are:

1-800-755-2345 for U.S. and most of Canada

1-602-786-7302 for the rest of the world

These phone numbers are also listed on the "Important Information" sheet that is shipped with all development systems. The hot line message is updated whenever a new software version is added to the Microchip Web Site, or when a new upgrade kit becomes available.

NOTES:

NOTES:

Index

MICROCHIP

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
Microchip Technology Inc.
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 602-786-7200 Fax: 602-786-7277
Technical Support: 602-786-7627
Web: http://www.microchip.com

Atlanta
Microchip Technology Inc.
500 Sugar Mill Road, Suite 200B
Atlanta, GA 30350
Tel: 770-640-0034 Fax: 770-640-0307

Boston
Microchip Technology Inc.
5 Mount Royal Avenue
Marlborough, MA 01752
Tel: 508-480-9990 Fax: 508-480-8575

Chicago
Microchip Technology Inc.
333 Pierce Road, Suite 180
Itasca, IL 60143
Tel: 630-285-0071 Fax: 630-285-0075

Dallas
Microchip Technology Inc.
14651 Dallas Parkway, Suite 816
Dallas, TX 75240-8809
Tel: 972-991-7177 Fax: 972-991-8588

Dayton
Microchip Technology Inc.
Two Prestige Place, Suite 150
Miamisburg, OH 45342
Tel: 937-291-1654 Fax: 937-291-9175

Los Angeles
Microchip Technology Inc.
18201 Von Karman, Suite 1090
Irvine, CA 92612
Tel: 714-263-1888 Fax: 714-263-1338

New York
Microchip Technology Inc.
150 Motor Parkway, Suite 202
Hauppauge, NY 11788
Tel: 516-273-5305 Fax: 516-273-5335

San Jose
Microchip Technology Inc.
2107 North First Street, Suite 590
San Jose, CA 95131
Tel: 408-436-7950 Fax: 408-436-7955

Toronto
Microchip Technology Inc.
5925 Airport Road, Suite 200
Mississauga, Ontario L4V 1W1, Canada
Tel: 905-405-6279 Fax: 905-405-6253

ASIA/PACIFIC

Hong Kong
Microchip Asia Pacific
RM 3801B, Tower Two
Metroplaza
223 Hing Fong Road
Kwai Fong, N.T., Hong Kong
Tel: 852-2-401-1200 Fax: 852-2-401-3431

India
Microchip Technology Inc.
India Liaison Office
No. 6, Legacy, Convent Road
Bangalore 560 025, India
Tel: 91-80-229-0061 Fax: 91-80-229-0062

Japan
Microchip Technology Intl. Inc.
Benex S-1 6F
3-18-20, Shinyokohama
Kohoku-Ku, Yokohama-shi
Kanagawa 222 Japan
Tel: 81-45-471- 6166 Fax: 81-45-471-6122

Korea
Microchip Technology Korea
168-1, Youngbo Bldg. 3 Floor
Samsung-Dong, Kangnam-Ku
Seoul, Korea
Tel: 82-2-554-7200 Fax: 82-2-558-5934

Shanghai
Microchip Technology
RM 406 Shanghai Golden Bridge Bldg.
2077 Yan'an Road West, Hong Qiao District
Shanghai, PRC 200335
Tel: 86-21-6275-5700
Fax: 86 21-6275-5060

Singapore
Microchip Technology Taiwan
Singapore Branch
200 Middle Road
#07-02 Prime Centre
Singapore 188980
Tel: 65-334-8870 Fax: 65-334-8850

ASIA/PACIFIC (CONTINUED)

Taiwan, R.O.C
Microchip Technology Taiwan
10F-1C 207
Tung Hua North Road
Taipei, Taiwan, ROC
Tel: 886-2-2717-7175 Fax: 886-2-2545-0139

EUROPE

United Kingdom
Arizona Microchip Technology Ltd.
505 Eskdale Road
Winnersh Triangle
Wokingham
Berkshire, England RG41 5TU
Tel: 44-1189-21-5858 Fax: 44-1189-21-5835

France
Arizona Microchip Technology SARL
Zone Industrielle de la Bonde
2 Rue du Buisson aux Fraises
91300 Massy, France
Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79

Germany
Arizona Microchip Technology GmbH
Gustav-Heinemann-Ring 125
D-81739 Müchen, Germany
Tel: 49-89-627-144 0 Fax: 49-89-627-144-44

Italy
Arizona Microchip Technology SRL
Centro Direzionale Colleoni
Palazzo Taurus 1 V. Le Colleoni 1
20041 Agrate Brianza
Milan, Italy
Tel: 39-39-6899939 Fax: 39-39-6899883

1/13/98

Microchip received ISO 9001 Quality System certification for its worldwide headquarters, design, and wafer fabrication facilities in January 1997. Our field-programmable PICmicro™ 8-bit MCUs, Serial EEPROMs, related specialty memory products and development systems conform to the stringent quality standards of the International Standard Organization (ISO).