MPLAB® C17 C COMPILER LIBRARIES

Note the following details of the code protection feature on PICmicro® MCUs.

- The PICmicro family meets the specifications contained in the Microchip Data Sheet.
- Microchip believes that its family of PICmicro microcontrollers is one of the most secure products of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the PICmicro microcontroller in a manner outside the operating specifications contained in the data sheet. The person doing so may be engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable”.
- Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our product.

If you have any further questions about this matter, please contact the local sales office nearest to you.

Microchip received QS-9000 quality system certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona in July 1999 and Mountain View, California in March 2002. The Company’s quality system processes and procedures are QS-9000 compliant for its PICmicro® 8-bit MCUs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, non-volatile memory and analog products. In addition, Microchip’s quality system for the design and manufacture of development systems is ISO 9001 certified.

MPLAB® C17 C COMPILER LIBRARIES

Table of Contents

MPLAB® C17 C COMPILER LIBRARIES

Preface

INTRODUCTION

The purpose of this document is to provide detailed information on the libraries and precompiled object files that may be used with Microchip's MPLAB® C17 C Compiler.

HIGHLIGHTS

Items discussed in this chapter are:

- About this Guide
- Warranty Registration
- Recommended Reading
- Troubleshooting
- Microchip On-Line Support
- Customer Change Notification Service
- Customer Support

ABOUT THIS GUIDE

Document Layout

This document describes MPLAB C17 libraries and precompiled object files. For a detailed discussion about using MPLAB C17 or MPLAB® IDE, refer to Recommended Reading later in this chapter.

The document layout is as follows:

- **Chapter 1: Library/Precompiled Object Overview** – describes the libraries and precompiled object files available.
- **Chapter 2: Hardware Peripheral Library** – describes each hardware peripheral library function.
- **Chapter 3: Software Peripheral Library** – describes each software peripheral library function.
- **Chapter 4: General Software Library** – describes each general software library function.
- **Chapter 5: Math Library** – discusses the math library functions.
- **Glossary** – A glossary of terms used in this guide.
- **Index** – Cross-reference listing of terms, features and sections of this document.
- **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 manual uses the following documentation conventions:

Table: Documentation Conventions

Description	Represents	Examples
Code (Courier font):		
Plain characters	Sample code Filenames and paths	`#define START` `c:\autoexec.bat`
Angle brackets: < >	Variables	`<label>, <exp>`
Square brackets [ ]	Optional arguments	`MPASMWIN [main.asm]`
Curly brackets and pipe character: { | }	Choice of mutually exclusive arguments; An OR selection	`errorlevel {0|1}`
Lower case characters in quotes	Type of data	`"filename"`
Ellipses...	Used to imply (but not show) additional text that is not relevant to the example	`list ["list_option..., "list_option"]`
0xnnn	A hexadecimal number where n is a hexadecimal digit	`0xFFFF, 0x007A`
Italic characters	A variable argument; it can be either a type of data (in lower case characters) or a specific example (in uppercase characters).	`char isascii (char,` *`ch`*`);`
Interface (Arial font):		
Underlined, italic text with right arrow	A menu selection from the menu bar	*File > Save*
Bold characters	A window or dialog button to click	**OK**, **Cancel**
Characters in angle brackets < >	A key on the keyboard	<Tab>, <Ctrl-C>
Documents (Arial font):		
Italic characters	Referenced books	*MPLAB IDE User's Guide*

Documentation Updates

All documentation becomes dated, and this user's guide is no exception. Since MPLAB IDE, MPLAB C17 and other Microchip tools are constantly evolving to meet customer needs, some actual dialogs and/or tool descriptions may differ from those in this document. Please refer to our web site to obtain the latest documentation available.

Documentation Numbering Conventions

Documents are numbered with a "DS" number. The number is located on the bottom of each page, in front of the page number. The numbering convention for the DS Number is: DSXXXXXA,

where:

XXXXX	=	The document number.
A	=	The revision level of the document.

WARRANTY REGISTRATION

Please complete the enclosed Warranty Registration Card and mail it promptly. Sending in your Warranty Registration Card entitles you to receive new product updates. Interim software releases are available at the Microchip web site.

RECOMMENDED READING

This document describes the MPLAB C17 C Compliler libraries and precompiled object files. For more information on the MPLAB C17 C compliler, the operation of MPLAB IDE and the use of other tools, the following are recommended reading.

README.C17

For the latest information on using MPLAB C17 C Compiler, read the README.C17 file (ASCII text) included with the software. This README file contains update information that may not be included in this document.

README.XXX

For the latest information on other Microchip tools (MPLAB IDE, MPLINK™ linker, etc.), read the associated README files (ASCII text file) included with the MPLAB IDE software.

MPLAB C17 C Compiler User's Guide (DS51290)

Comprehensive guide that describes the installation, operation and features of Microchip's MPLAB C17 C compiler for PIC17CXXX devices.

MPLAB IDE User's Guide (DS51025)

Comprehensive guide that describes installation and features of Microchip's MPLAB Integrated Development Environment (IDE), as well as the editor and simulator functions in the MPLAB IDE environment.

MPASM™ User's Guide with MPLINK™ and MPLIB™ (DS33014)

This user's guide describes how to use the Microchip PICmicro® MCU MPASM assembler, the MPLINK object linker and the MPLIB object librarian.

Technical Library CD-ROM (DS00161)

This CD-ROM contains comprehensive application notes, data sheets and technical briefs for all Microchip products. To obtain this CD-ROM, contact the nearest Microchip Sales and Service location (see back page).

Microchip Web Site

Our web site (www.microchip.com) contains a wealth of documentation. Individual data sheets, application notes, tutorials and user's guides are all available for easy download. All documentation is in Adobe™ Acrobat (pdf) format.

Microsoft® Windows® Manuals

This manual assumes that users are familiar with the Microsoft® Windows® operating system. Many excellent references exist for this software program, and should be consulted for general operation of Windows.

TROUBLESHOOTING

See the README files for information on common problems not addressed in this user's guide.

MICROCHIP ON-LINE SUPPORT

Microchip provides on-line support on the Microchip web site at:

http://www.microchip.com

A file transfer site is also available by using an FTP service connecting to:

ftp://ftp.microchip.com

The web site and file transfer site provide a variety of services. Users may download files for the latest development tools, data sheets, application notes, user' guides, articles and sample programs. A variety of Microchip specific business information is also available, including listings of Microchip sales offices and distributors. Other information available on the web site includes:

- Latest Microchip press releases
- Technical support section with FAQs
- Design tips
- Device errata
- Job postings
- Microchip consultant program member listing
- Links to other useful web sites related to Microchip products
- Conferences for products, development systems, technical information and more
- Listing of seminars and events

CUSTOMER CHANGE NOTIFICATION SERVICE

Microchip started the customer notification service to help customers stay current on Microchip products with the least amount of effort. Once you subscribe, you will receive email notification whenever we change, update, revise or have errata related to your specified product family or development tool of interest.

Go to the Microchip web site (www.microchip.com) and click on Customer Change Notification. Follow the instructions to register.

The Development Systems product group categories are:

- Compilers
- Emulators
- In-Circuit Debuggers
- MPLAB IDE
- Programmers

Here is a description of these categories:

Compilers - The latest information on Microchip C compilers and other language tools. These include the MPLAB C17, MPLAB C18 and MPLAB C30 C Compilers; MPASM and MPLAB ASM30 assemblers; MPLINK and MPLAB LINK30 linkers; and MPLIB and MPLAB LIB30 librarians.

Emulators - The latest information on Microchip in-circuit emulators. This includes the MPLAB ICE 2000.

In-Circuit Debuggers - The latest information on Microchip in-circuit debuggers. These include the MPLAB ICD and MPLAB ICD 2.

MPLAB - The latest information on Microchip MPLAB IDE, the Windows Integrated Development Environment for development systems tools. This list is focused on the MPLAB IDE, MPLAB SIM simulator, MPLAB IDE Project Manager and general editing and debugging features.

Programmers - The latest information on Microchip device programmers. These include the PRO MATE® II device programmer and PICSTART® Plus development programmer.

CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

- Distributors
- Local Sales Office
- Field Application Engineers (FAEs)
- Corporate Applications Engineers (CAEs)
- Systems Information and Upgrade Hot Line

Customers should call their distributor or field application engineer (FAE) for support. Local sales offices are also available to help customers. See the last page of this document for a listing of sales offices and locations.

Corporate applications engineers (CAEs) may be contacted at (480) 792-7627.

Systems Information and Upgrade Line

The Systems Information and Upgrade Information Line provides system users with 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 the most current upgrade kits. The Information Line Numbers are:

1-800-755-2345 for U.S. and most of Canada.

1-480-792-7302 for the rest of the world.

NOTES:

Chapter 1. Library/Precompiled Object Overview

1.1 INTRODUCTION

This chapter gives an overview of the MPLAB C17 libraries and precompiled object files that can be included in an application.

1.2 HIGHLIGHTS

This chapter is organized as follows:

- MPLAB C17 Libraries
  - Hardware, Software and Standard Libraries
  - Math Library
- MPLAB C17 Precompiled Object Files
  - Start Up Code
  - Initialization Code
  - Interrupt Handler Code
  - Special Function Register Definitions

1.3 MPLAB C17 LIBRARIES

A library is a collection of functions grouped for reference and ease of linking. See the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) for more information about creating and maintaining libraries.

When building an application, usually one file from 1.3.1 "Hardware, Software and Standard Libraries" will be needed to successfully link. Be sure to chose the library that corresponds to your selected device and memory model. For more information on memory models, see the *MPLAB C17 C Compiler User's Guide* (DS51290).

For functions contained in MPLAB C17 libraries, all parameters sent to these functions are classified as static and therefore are passed in global RAM. The first variable is always passed in the `PROD` register if declared as static, i.e., 8 bits in `PRODL` and 16 bits in `PRODH:PRODL`.

The MPLAB C17 libraries are included in the `c:\mcc\lib` directory, where `c:\mcc` is the compiler install directory. These can be linked directly into an application with MPLINK object linker.

These files were precompiled in the `c:\mcc\src` directory at Microchip. If you chose **not** to install the compiler and related files in the `c:\mcc` directory (ex: `c:\c17\src`, `d:\mcc\src`, etc.), a warning message will be generated by MPLINK linker stating that source code from the libraries will not show in the `.lst` file and can not be stepped through when using MPLAB IDE. This results from MPLINK linker looking for the library source files in the absolute path of `c:\mcc\src.`

To include the library code in the `.lst` file and to be able to single step through library functions, use the batch file (`.bat`) in the `src` directory to rebuild the files. Then copy the newly compiled files into the `lib` directory.

1.3.1 Hardware, Software and Standard Libraries

Device	Memory Model			
	Small	**Medium**	**Compact**	**Large**
PIC17C42A	pmc42as.lib	pmc42am.lib	pmc42ac.lib	pmc42al.lib
PIC17C43	pmc43s.lib	pmc43m.lib	pmc43c.lib	pmc43l.lib
PIC17C44	pmc44s.lib	pmc44m.lib	pmc44c.lib	pmc44l.lib
PIC17C752	pmc752s.lib	pmc752m.lib	pmc752c.lib	pmc752l.lib
PIC17C756A	pmc756as.lib	pmc756am.lib	pmc756ac.lib	pmc756al.lib
PIC17C756	pmc756s.lib	pmc756m.lib	pmc756c.lib	pmc756l.lib
PIC17C762	pmc762s.lib	pmc762m.lib	pmc762c.lib	pmc762l.lib
PIC17C766	pmc766s.lib	pmc766m.lib	pmc766c.lib	pmc766l.lib

These are the main MPLAB C17 library files that contain the functions described in the following three chapters.

- Hardware functions are described in Chapter 2.
- Software functions are described in Chapter 3.
- General functions are described in Chapter 4.

When you wish to use any of the functions described in these chapters, include the appropriate above library as part of your project.

The source code for these libraries may be found in `c:\mcc\src\pmc`, where `c:\mcc` is the compiler install directory.

1.3.2 Math Library

Device	All Memory Models
PIC17CXXX	cmath17.lib

This library file contains the available math functions described in detail in Chapter 5. When you wish to use any of the functions described in this chapter, include the math library as part of your project.

The source code for this library can be found in `c:\mcc\src\math`, where `c:\mcc` is the compiler install directory.

1.4 MPLAB C17 PRECOMPILED OBJECT FILES

Precompiled object files are useful inclusions when building applications. These files have already been compiled and tested, so may be used as “plug-ins” to serve a specific function in your code development.

When building an application, usually one file from each of the following subsections will be needed to successfully link. Be sure to chose the file that corresponds to your selected device and memory model. For more information on memory models, see the *MPLAB C17 C Compiler User’s Guide* (DS51290).

These files are included in the `c:\mcc\lib` directory, where `c:\mcc` is the compiler install directory. They can be linked directly into an application with MPLINK linker.

1.4.1 Start Up Code

Device	Memory Model	
	Small	Compact/Medium/Large
PIC17CXXX	c0s17.o	c0l17.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 (`c0l17.o` only), and jumps to the start of the application function, `main()`.

If the application uses more than one page (8k) of program memory, then `c0l17.o` should be used.

The source code may be found in `c:\mcc\src\startup`, where `c:\mcc` is the compiler install directory.

1.4.2 Initialization Code

Device	All Memory Models
PIC17CXXX	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.

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";
     :
```

The source code may be found in `c:\mcc\src\startup`, where `c:\mcc` is the compiler install directory.

1.4.3 Interrupt Handler Code

Device	Memory Model	
	Small	Compact/Medium/Large
PIC17C42A	int42as.o	int42al.o
PIC17C43	int43s.o	int43l.o
PIC17C44	int44s.o	int44l.o
PIC17C752	int752s.o	int752l.o
PIC17C756A	int756as.o	int756al.o
PIC17C756	int756s.o	int756l.o
PIC17C762	int762s.o	int762l.o
PIC17C766	int766s.o	int766l.o

These precompiled object files contain useful interrupt code. These may be customized for specific applications.

The source code for these precompiled objects can be found in `c:\mcc\src\startup`, where `c:\mcc` is the compiler install directory.

1.4.4 Special Function Register Definitions

Device	All Memory Models
PIC17C42A	p17c42a.o
PIC17C43	p17c43.o
PIC17C44	p17c44.o
PIC17C752	p17c752.o
PIC17C756A	p17c756a.o
PIC17C756	p17c756.o
PIC17C762	p17c762.o
PIC17C766	p17c766.o

These files contain the PICmicro MCU special function register definitions for each processor supported.

The source code can be found in `c:\mcc\src\proc`, where `c:\mcc` is the compiler install directory.

Chapter 2. Hardware Peripheral Library

2.1 INTRODUCTION

This chapter documents hardware peripheral library functions. The source code for all of these functions is included with MPLAB C17 in the `c:\mcc\src\pmc` directory, where `c:\mcc` is the compiler install directory.

See the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) for more information about building libraries.

2.2 HIGHLIGHTS

This chapter is organized as follows:

- A/D Converter Functions
- Input Capture Functions
- I^2C® Functions
- Interrupt Functions
- Port B Functions
- Microwire® Functions
- Pulse Width Modulation (PWM) Functions
- Reset Functions
- SPI™ Functions
- Timer Functions
- USART Functions

2.3 A/D CONVERTER FUNCTIONS

This section contains a list of individual functions and an example of use of the functions in this section. Functions may be implemented as macros.

2.3.1 Function Descriptions

BusyADC

Device:	PIC17C756
Function:	Returns the value of the `GO` bit in the `ADCON0` register.
Include:	`adc16.h`
Prototype:	`char BusyADC (void);`
Arguments:	None
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).
File Name:	`adcbusy.c`
Code Example:	`while (BusyACD());`

CloseADC

Device:	PIC17C756
Function:	This function disables the A/D convertor.
Include:	`adc16.h`
Prototype:	`void CloseADC (void);`
Arguments:	None
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
File Name:	`adcclose.c`
Code Example:	`CloseADC`();

ConvertADC

Device:	PIC17C756
Function:	Starts an A/D conversion by setting the `GO` bit in the `ADCON0` register.
Include:	`adc16.h`
Prototype:	`void ConvertADC (void);`
Arguments:	None
Remarks:	This function sets the `GO` bit in the `ADCON0` register.
Return Value:	None
File Name:	`adcconv.c`
Code Example:	`ConvertADC();`

OpenADC

Device: PIC17C756

Function: Configures the A/D convertor.

Include: `adc16.h`

Prototype: `void OpenADC (static unsigned char config, static unsigned char channel);`

Arguments: **config**

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

channel

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

OpenADC (Continued)

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.

Return Value: None

File Name: `adcopen.c`

Code Example:

```
OpenADC(ADC_INT_OFF&ADC_FOSC_32&
        ADC_RIGHT_JUST&ADC_VREF_INT&
        ADC_ALL_ANALOG,ADC_CH0);
```

ReadADC

Device: PIC17C756

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

Include: `adc16.h`

Prototype: `int ReadADC (void);`

Arguments: None

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.

File Name: `adcread.c`

Code Example:

```
int result;
result = ReadADC();
```

SetChanADC

Device: PIC17C756

Function: Selects a specific A/D channel.

Include: `adc16.h`

Prototype: `void SetChanADC (static unsigned char channel);`

Arguments: **channel**

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

SetChanADC (Continued)

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.
Return Value:	None
File Name:	`adcset.c`
Code Example:	`SetChanADC(ADC_CH0);`

2.3.2 Example of Use

```
#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, 25);
    Delay10TCYx(5);     // Delay for 50TCY
    ConvertADC();       // Start Conversion
    result = ReadADC(); // read result
    itoa(result,str);   // convert to string
    putsUSART1(str);    // Write string to USART
    CloseADC();       // Close Modules
    CloseUSART1();
    return;
 }
```

2.4 INPUT CAPTURE FUNCTIONS

This section contains a list of individual functions and an example of use of the functions in this section. Functions may be implemented as macros.

2.4.1 Function Descriptions

CloseCapture1
CloseCapture2
CloseCapture3
CloseCapture4

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

Function: This function disables the specified input capture.

Include: `captur16.h`

Prototype:
```
void CloseCapture1 (void);
void CloseCapture2 (void);
void CloseCapture3 (void);
void CloseCapture4 (void);
```

Arguments: None

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

Return Value: None

File Name:
```
cp1close.c
cp2close.c
cp3close.c
cp4close.c
```

Code Example: `CloseCapture1();`

OpenCapture1
OpenCapture2
OpenCapture3
OpenCapture4

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

Function: This function configures the specified input capture.

Include: `captur16.h`

Prototype:
```
void OpenCapture1 (static unsigned char config);
void OpenCapture2 (static unsigned char config);
void OpenCapture3 (static unsigned char config);
void OpenCapture4 (static unsigned char config);
```


OpenCapture1
OpenCapture2
OpenCapture3
OpenCapture4 (Continued)

Arguments: **config**

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

All OpenCapture functions

CAPTURE_INT_ONInterrupts ON
CAPTURE_INT_OFFInterrupts 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

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 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.

OpenCapture1
OpenCapture2
OpenCapture3
OpenCapture4 (Continued)

Return Value: None

File Name:
```
cp1open.c
cp2open.c
cp3open.c
cpopen4.c
```

Code Example:
```
OpenCapture1(C1_EVERY_4_RISE_EDGE&CAPTURE 1_CAPTURE);
```

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.

Include: `captur16.h`

Prototype:
```
unsigned int ReadCapture1 (void);
unsigned int ReadCapture2 (void);
unsigned int ReadCapture3 (void);
unsigned int ReadCapture4 (void);
```

Arguments: None

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.

File Name:
```
cap1read.c
cap2read.c
cap3read.c
cap4read.c
```

Code Example:
```
unsigned int result;
result = ReadCapture1();
```

2.4.2 Example of Use

```
#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, 25);
 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;
}
```

2.5 I²C® FUNCTIONS

This section contains a list of individual functions and an example of use of the functions in this section. Functions may be implemented as macros.

2.5.1 Function Descriptions

AckI2C

Device:	PIC17C756
Function:	Generates I²C bus Acknowledge condition.
Include:	`i2c16.h`
Prototype:	`void AckI2C (void);`
Arguments:	None
Remarks:	This function generates an I²C bus Acknowledge condition.
Return Value:	None
File Name:	`acki2c.c`
Code Example:	`AckI2C();`

CloseI2C

Device:	PIC17C756
Function:	Disables the SSP module.
Include:	`i2c16.h`
Prototype:	`void CloseI2C (void);`
Arguments:	None
Remarks:	Pin I/O returns under control of Port register settings.
Return Value:	None
File Name:	`closei2c.c`
Code Example:	`CloseI2C();`

DataRdyI2C

Device:	PIC17C756
Function:	Provides status back to user if the `SSPBUF` register contains data.
Include:	`i2c16.h`
Prototype:	`unsigned char DataRdyI2C (void);`
Arguments:	None
Remarks:	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 0 which indicates no data in `SSPBUF` register.
File Name:	`dtrdyi2c.c`
Code Example:	`if (!DataRdyI2C());`

getcI2C

Function: This function operates identically to **ReadI2C**.

File Name: `#define in i2c16.h`

getsI2C

Device: PIC17C756

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

Include: `i2c16.h`

Prototype: unsigned char getsI2C (static unsigned char far **rdptr*, static unsigned char *length*);

Arguments: **rdptr**
Character type pointer to PICmicro RAM for storage of data read from I^2C device.
length
Number of bytes to read from I^2C device.

Remarks: **Master I^2C mode:** This routine reads a predefined data string length from 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 reads a predefined data string length from 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.

File Name: `getsi2c.c`

Code Example:

```
unsigned char string[15];
unsigned char far *ptrstring;
ptrstring = string;
getsI2C(ptrstring, 15);
```

IdleI2C

Device:	PIC17C756
Function:	Generates wait condition until I^{2}C bus is idle.
Include:	`i2c16.h`
Prototype:	`void IdleI2C (void);`
Arguments:	None
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^{2}C peripheral does not allow for spooling of bus sequences. The I^{2}C peripheral must be in an idle state before an I^{2}C operation can be initiated or a write collision will be generated.
Return Value:	None
File Name:	`idlei2c.c`
Code Example:	`IdleI2C();`

NotAckI2C

Device:	PIC17C756
Function:	Generates I^{2}C bus Not Acknowledge condition.
Include:	`i2c16.h`
Prototype:	`void NotAckI2C (void);`
Arguments:	None
Remarks:	This function generates an I^{2}C bus *Not Acknowledge* condition.
Return Value:	None
File Name:	`noacki2c.c`
Code Example:	`NotAckI2C();`

OpenI2C

Device:	PIC17C756
Function:	Configures the SSP module.
Include:	`i2c16.h`
Prototype:	void OpenI2C (static unsigned char *sync_mode*, static unsigned char *slew*);
Arguments:	**sync_mode** The value of function parameter *sync_mode* can be one of the following values defined in `i2c16.h:` SLAVE_7 — I^{2}C Slave mode, 7-bit address SLAVE_10 — I^{2}C Slave mode, 10-bit address MASTER — I^{2}C Master mode

OpenI2C (Continued)

slew
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

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

Return Value: None

File Name: `openi2c.c`

Code Examples: `OpenI2C(MASTER, SLEW_ON);`

putcI2C

Function: This function operates identically to **WriteI2C**.

File Name: `#define in i2c16.h`

putsI2C

Device: PIC17C756

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

Include: `i2c16.h`

Prototype: unsigned char putsI2C (static unsigned char far **wrptr*);

Arguments: **wrptr**
Character type pointer to data objects in PICmicro RAM. The data objects are written to the I^2C device.

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

File Name: `putsi2c.c`

Code Example:

```
unsigned char string[] = "data to send";
unsigned char far *ptrstring;
ptrstring = string;
putsI2C(ptrstring);
```

ReadI2C

Device:	PIC17C756
Function:	This function is used to read a single byte (one character) from the I^2C bus.
Include:	`i2c16.h`
Prototype:	`unsigned char ReadI2C (void);`
Arguments:	None
Remarks:	This function reads in a single byte from the I^2C bus. This function performs the same function as **getcI2C**.
Return Value:	The return value is the data byte read from the I^2C bus.
File Name:	`readi2c.c`
Code Example:	`unsigned char value;` `value = ReadI2C();`

RestartI2C

Device:	PIC17C756
Function:	Generates I^2C bus restart condition.
Include:	`i2c16.h`
Prototype:	`void RestartI2C (void);`
Arguments:	None
Remarks:	This function generates an I^2C bus restart condition.
Return Value:	None
File Name:	`rstrti2c.c`
Code Example:	`RestartI2C();`

StartI2C

Device:	PIC17C756
Function:	Generates I^2C bus start condition.
Include:	`i2c16.h`
Prototype:	`void StartI2C (void);`
Arguments:	None
Remarks:	This function generates a I^2C bus start condition.
Return Value:	None
File Name:	`starti2c.c`
Code Example:	`StartI2C();`

StopI2C

Device:	PIC17C756
Function:	Generates I^2C bus stop condition.
Include:	`i2c16.h`
Prototype:	`void StopI2C (void);`
Arguments:	None
Remarks:	This function generates an I^2C bus stop condition.
Return Value:	None
File Name:	`stopi2c.c`
Code Example:	`StopI2C();`

WriteI2C

Device:	PIC17C756
Function:	This function is used to write out a single data byte (one character) to the I^2C bus device.
Include:	`i2c16.h`
Prototype:	`unsigned char WriteI2C (static unsigned char data_out);`
Arguments:	**data_out** A single data byte to be written to the I^2C bus device.
Remarks:	This function writes out a single data byte to the I^2C bus device. This function performs the same function as **putcI2C**.
Return Value:	This function returns -1 if there was a write collision else it returns a 0.
File Name:	`writei2c.c`
Code Example:	`WriteI2C('a');`

> **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.

Include: `i2c16.h`

Prototype: `unsigned char EEAckPolling (static unsigned char control);`

Arguments: **control**
EEPROM control / bus device select address byte.

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`

Code Example:

```
temp = EEAckPolling(0xA0);
```

EEByteWrite

Device: PIC17C756

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

Include: `i2c16.h`

Prototype: `unsigned char EEByteWrite (static unsigned char control, static unsigned char address, static unsigned char data);`

Arguments: **control**
EEPROM control / bus device select address byte.
address
EEPROM internal address location.
data
Data to write to EEPROM address specified in function parameter address.

Remarks: This function writes a single data byte to the I^2C bus. This routine can be used for any Microchip I^2C 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`

Code Example:

```
temp = EEByteWrite(0xA0, 0x30, 0xA5);
```

EECurrentAddRead

Device: PIC17C756

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

Include: `i2c16.h`

Prototype:
```
unsigned char EECurrentAddRead
(static unsigned char control);
```

Arguments: **control**
EEPROM control / bus device select address byte.

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`

Code Example:
```
temp = EECurrentAddRead(0xA1);
```

EEPageWrite

Device: PIC17C756

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

Include: `i2c16.h`

Prototype:
```
unsigned char EEPageWrite (static unsigned char
control, static unsigned char address, static
unsigned char far *wrptr);
```

Arguments: **control**
EEPROM control / bus device select address byte.
address
EEPROM internal address location.
wrptr
Pointer to character type data objects in PICmicro RAM. The data objects pointed to by *wrptr* will be written to the I^{2}C bus.

Remarks: This function writes a null terminated string of data objects to the I^{2}C EE 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: `i2ceepw.c`

Code Example:
```
temp = EEPageWrite(0xA0, 0x70, wrptr);
```


EERandomRead

Device: PIC17C756

Function: This function is used to read a single byte from the I²C bus.

Include: `i2c16.h`

Prototype:
```
unsigned char EERandomRead (static unsigned char
control, static unsigned char address);
```

Arguments:
control
EEPROM control / bus device select address byte.
address
EEPROM internal address location.

Remarks: This function reads in a single byte from the I²C bus. The routine can be used for Microchip I²C 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`

Code Example:
```
temp = EERandomRead(0xA0,0x30);
```

EESequentialRead

Device: PIC17C756

Function: This function is used to read in a string of data from the I²C bus.

Include: `i2c16.h`

Prototype:
```
unsigned char EESequentialRead (static unsigned char
control, static unsigned char address, static
unsigned char far *rdptr, static unsigned char
length);
```

Arguments:
control
EEPROM control / bus device select address byte.
address
EEPROM internal address location.
rdptr
Character type pointer to PICmicro RAM area for placement of data read from EEPROM device.
length
Number of bytes to read from EEPROM device.

Remarks: This function reads in a predefined string length of data from the I²C bus. The routine can be used for Microchip I²C 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.

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`

Code Example:
```
temp = EESequentialRead(0xA0, 0x70, rdptr, 15);
```

2.5.2 Example of Use

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 Prototype
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) @16 MHz
                           //100khz Baud clock(39) @16 MHz
 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);
 }
}
```

2.6 INTERRUPT FUNCTIONS

This section contains a list of individual functions and an example of use of the functions in this section. Functions may be implemented as macros.

2.6.1 Function Descriptions

CloseRA0INT

Device:	PIC17C4X, PIC17C756
Function:	Disables the RA0/INT pin interrupt.
Include:	`int16.h`
Prototype:	`void CloseRA0INT (void);`
Arguments:	None
Remarks:	This function disables the RA0/INT pin interrupt by clearing the `INTE` bit in the `INTSTA` register.
Return Value:	None
File Name:	`ra0close.c`
Code Example:	`CloseRA0INT();`

Disable

Device:	PIC17C4X, PIC17C756
Function:	Disables global interrupts.
Include:	`int16.h`
Prototype:	`void Disable (void);`
Arguments:	None
Remarks:	This function disables global interrupts by setting the `GLINTD` bit of the `CPUSTA` register.
Return Value:	None
File Name:	`disable.c`
Code Example:	`Disable();`

Enable

Device:	PIC17C4X, PIC17C756
Function:	Enables global interrupts.
Include:	`int16.h`
Prototype:	`void Enable (void);`
Arguments:	None
Remarks:	This function enables global interrupts by clearing the `GLINTD` bit of the `CPUSTA` register.
Return Value:	None
File Name:	`enable.c`
Code Example:	`enable();;`

OpenRA0INT

Device:	PIC17C4X, PIC17C756
Function:	Configures the external interrupt pin RA0/INT.
Include:	`int16.h`
Prototype:	`void OpenRA0INT (static unsigned char config);`
Arguments:	**config** 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
Remarks:	This function configures the RA0/INT pin for external interrupt for interrupt on/off and edge select.
Return Value:	None
File Name:	`ra0open.c`
Code Example:	`OpenRA0INT(INT_ON);`

2.6.2 Example of Use

```
   #include<p17C756.h>
   #include<int16.h>

void INT_ISR(void)
{
   PORTB++;              // increment data register
}

void main(void)
{
   Install_INT(INT_ISR); // install INT pin interrupt vector

   PORTB = 0x00;         // clear PORTB data register
   DDRB = 0x00;          // config PORTB as outputs

   // enable external interrupt and detect rising edge
   OpenRA0INT(INT_ON & INT_RISE_EDGE);


   Enable();             // enable global interrupts

   while(PORTB != 0xFF); // wait for interrupt and check
                         // for max value of PORTB register
   Disable();            // disable global interrupts
   CloseRA0INT();        // turn off INT pin interrupt
}
```

2.7 PORT B FUNCTIONS

This section contains a list of individual functions and an example of use of the functions in this section. Functions may be implemented as macros.

2.7.1 Function Descriptions

ClosePORTB

Device:	PIC17C4X, PIC17C756
Function:	Disables the interrupts and internal pull-up resistors for `PORTB`.
Include:	`portb16.h`
Prototype:	`void ClosePORTB (void);`
Arguments:	None
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 setting the `NOT_RBPU` bit in the `PORTA` register.
Return Value:	None
File Name:	`pbclose.c`
Code Example:	`ClosePORTB();`

DisablePullups

Device:	PIC17C4X, PIC17C756
Function:	Disables the internal pull-up resistors on `PORTB`.
Include:	`portb16.h`
Prototype:	`void DisablePullups (void);`
Arguments:	None
Remarks:	This function disables the internal pull-up resistors on `PORTB`. by setting the `NOT_RBPU` bit in the `PORTA` register.
Return Value:	None
File Name:	`pulldis.c`
Code Example:	`DisablePullups();`

EnablePullups

Device:	PIC17C4X, PIC17C756
Function:	Enables the internal pull-up resistors on `PORTB`.
Include:	`portb16.h`
Prototype:	`void EnablePullups (void);`
Arguments:	None
Remarks:	This function enables the internal pull-up resistors on `PORTB` by clearing the `NOT_RBPU` bit in the `PORTA` register.
Return Value:	None
File Name:	`pullen.c`
Code Example:	`EnablePullups();`

OpenPORTB

Device: PIC17C4X, PIC17C756

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

Include: `portb16.h`

Prototype: `void OpenPORTB (static unsigned char config);`

Arguments: **config**
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_OFFI	nterrupt OFF
PORTB_PULLUPS_ON	pull-up resistors enabled
PORTB_PULLUPS_OFF	pull-up resistors disabled

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

Return Value: None

File Name: `pbopen.c`

Code Example:

```
OpenPORTB(PORTB_CHANGE_INT_ON);
```

2.7.2 Example of Use

```
   #include<p17C756.h>
   #include<int16.h>
   #include<portb16.h>

unsigned char Key;

void PIV_ISR(void)
{
   if(PIR1bits.RBIF)          // ensure PORTB interrupt
                              // got us here
   {
      Key = ~(PORTB & 0xF0);  // keep track of scan row

      DDRB = 0x0F;            // switch I/O drive to
                              // scan column
      PORTB = 0x00;

      Key += ~(PORTB & 0x0F); // add in scan column
      PIR1bits.RBIF = 0;      // reset interrupt flag
   }
}

void main(void)
{
   unsigned char PressedKey;

   Install_PIV(PIV_ISR); // install peripheral
                         // interrupt vector

   DDRB = 0xF0;        // set lower nibble to output
                       // upper nibble to input to scan row
   Key = 0x00;         // reset key scan register

   PORTB = PORTB;      // read PORTB to clear mismatch
   PIR1bits.RBIF = 0; // clear RBIF to ensure no interrupt

   // enable PORTB interrupt on change
   OpenPORTB(PORTB_CHANGE_INT_ON);

   EnablePullups();   // enable internal pullups

   Enable();          // enable global interrupts

   while(1)
   {
      while(Key==0x00);

      switch(Key)
      {
         case 0x11:  PressedKey = '1';
                     break;
         case 0x12:  PressedKey = '2';
                     break;
         case 0x14:  PressedKey = '3';
                     break;
         case 0x18:  PressedKey = '4';
                     break;
```

```
            case 0x21:  PressedKey = '5';
                        break;
            case 0x22:  PressedKey = '6';
                        break;
            case 0x24:  PressedKey = '7';
                        break;
            case 0x28:  PressedKey = '8';
                        break;

            case 0x41:  PressedKey = '9';
                        break;
            case 0x42:  PressedKey = '0';
                        break;
            case 0x44:  PressedKey = '*';
                        break;
            case 0x48:  PressedKey = '#';
                        break;

            case 0x81:  PressedKey = 'A';
                        break;
            case 0x82:  PressedKey = 'B';
                        break;
            case 0x84:  PressedKey = 'C';
                        break;
            case 0x88:  PressedKey = 'D';
                        break;

            default:    PressedKey = ' ';
                        break;
        }

        Key = 0x00;
    }
}
```

2.8 MICROWIRE® FUNCTIONS

This section contains a list of individual functions and an example of use of the functions in this section. Functions may be implemented as macros.

2.8.1 Function Descriptions

CloseMwire

Device:	PIC17C756
Function:	Disables the SSP module.
Include:	`mwire16.h`
Prototype:	`void CloseMwire (void);`
Arguments:	None
Remarks:	Pin I/O returns under control `DDRx` and `PORTx` register settings.
Return Value:	None
File Name:	`closmwir.c`
Code Example:	`CloseMwire();`

DataRdyMwire

Device:	PIC17C756
Function:	Provides status back to user if the Microwire device has completed the internal write cycle.
Include:	`mwire16.h`
Prototype:	`unsigned char DataRdyMwire (void);`
Arguments:	None
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.
File Name:	`drdymwir.c`
Code Example:	`while (!DataRdyMwire());`

getcMwire

Function:	This function operates identically to **ReadMwire**.
File Name:	`#define in mwire16.h`

getsMwire

Device: PIC17C756

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

Include: `mwire16.h`

Prototype:
```
void getsMwire (static unsigned char far *rdptr,
static unsigned char length);
```

Arguments:
rdptr
Pointer to PICmicro RAM area for placement of writing data read from Microwire device.
length
Number of bytes to read from Microwire device.

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

File Name: `getsmwir.c`

Code Example:
```
unsigned char arrayrd[20];
unsigned char far *rdptr = arrayrd;
getsMwire(rdptr, 10);
```

OpenMwire

Device: PIC17C756

Function: Configures the SSP module.

Include: `mwire16.h`

Prototype:
```
void OpenMwire (static unsigned char sync_mode);
```

Arguments:
sync_mode
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

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

Return Value: None

File Name: `openmwir.c`

Code Examples:
```
OpenMwire(FOSC_16);
```

putcMwire

Function: This function operates identically to **WriteMwire**.

File Name: `#define in mwire16.h`

ReadMwire

Device:	PIC17C756
Function:	This function is used to read a single byte (one character) from a Microwire device.
Include:	`mwire16.h`
Prototype:	`unsigned char ReadMwire (static unsigned char` *`high_byte`*`, static unsigned char` *`low_byte`*`);`
Arguments:	**high_byte** First byte of 16-bit instruction word. **low_byte** Second byte of 16-bit instruction word.
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. This function operates identically to **getcMwire**.
Return Value:	The return value is the data byte read from the Microwire device.
File Name:	`readmwir.c`
Code Example:	`ReadMwire(0x03, 0x00);`

WriteMwire

Device:	PIC17C756
Function:	This function is used to write out a single data byte (one character).
Include:	`mwire16.h`
Prototype:	`unsigned char WriteMwire (static unsigned char` *`data_out`*`);`
Arguments:	**data_out** Single byte of data to write to Microwire device.
Remarks:	This function writes out single data byte to a Microwire device utilizing the SSP module.This function operates identically to **putcMwire**.
Return Value:	This function returns -1 if there was a write collision, else it returns a 0.
File Name:	`writmwir.c`
Code Example:	`WriteMwire(0xFF);`

2.8.2 Example of Use

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 Prototype
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  EWEN  10x09
#define  EWEN  20x80
#define  ERAL  10x09
#define  ERAL  20x00
#define  WRAL  10x08
#define  WRAL  20x80
#define  EWDS  10x08
#define  EWDS  20x00
#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;
}
```

2.9 PULSE WIDTH MODULATION FUNCTIONS

This section contains a list of individual functions and an example of use of the functions in this section. Functions may be implemented as macros.

2.9.1 Function Descriptions

ClosePWM1
ClosePWM2
ClosePWM3

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

Function: This function disables the specified PWM channel.

Include: `pwm16.h`

Prototype:
```
void ClosePWM1 (void);
void ClosePWM2 (void);
void ClosePWM3 (void);
```

Arguments: None

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

Return Value: None

File Name:
`pw1close.c`
`pw2close.c`
`pw3close.c`

Code Example: `ClosePWM2();`

OpenPWM1
OpenPWM2
OpenPWM3

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

Function: Configures the specified PWM channel.

Include: `pwm16.h`

Prototype:
```
void OpenPWM1 (static char period);
void OpenPWM2 (static unsigned char
config, static char period);
void OpenPWM3 (static unsigned char
config, static char period);
```

Arguments: **config**
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

OpenPWM1 OpenPWM2 OpenPWM3 (Continued)

period
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

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.

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

File Name: `pw1open.c`
`pw2open.c`
`pw3open.c`

Code Example:

```
OpenPWM2(T1_SOURCE,0xff);
```

SetDCPWM1 SetDCPWM2 SetDCPWM3

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

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

Include: `pwm16.h`

Prototype:

```
void SetDCPWM1 (static unsigned int dutycycle);
void SetDCPWM2 (static unsigned int dutycycle);
void SetDCPWM3 (static unsigned int dutycycle);
```

Arguments: dutycycle
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:
PWM x Dutycycle = (DCx<9:0>) x TOSC
where DCx<9:0> is the 10-bit value from the `PWxDCH:PWxDCL` registers.

SetDCPWM1
SetDCPWM2
SetDCPWM3 (Continued)

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 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

File Name: `pw1set.c`
`pw2set.c`
`pw3set.c`

Code Example: `SetDCPWM2(0);`

2.9.2 Example of Use

```
#include <p17c756.h>
#include <pwm16.h>
#include <timers16.h>
void main(void)
{
 int i;
 //set duty cycle
 SetDCPWM2(0);
 //open PW2
 OpenPWM2(T1_SOURCE,0xff);
 //open timer
 OpenTimer1(TIMER_INT_OFF&T1_SOURCE_INT&T1_T2_8BIT);
 for(i=0;i<1024;i++)
 {
  while(!PIR1bits.TMR1IF);
  PIR1bits.TMR1IF = 0;
  SetDCPWM2(i);  //slowly increment duty cycle
   }
 ClosePWM2();    //close modules
 CloseTimer1();
 return;
}
```

2.10 RESET FUNCTIONS

This section contains a list of individual functions and an example of use of the functions in this section. Functions may be implemented as macros.

2.10.1 Function Descriptions

isBOR

Device: PIC17C756

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

Include: `reset16.h`

Prototype: `char isBOR (void);`

Arguments: None

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.h` file. Refer to Chapter 2 of this manual for information on compilers. Refer to the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) 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.

File Name: `reset16.c`

Code Example:
```
if(isBOR());
 then ...
```

isMCLR

Device: PIC17C756

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

Include: `reset16.h`

Prototype: `char isMCLR (void);`

Arguments: None

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.

File Name: `reset16.c`

Code Example:
```
if(isMCLR());
 then ...
```

isPOR

Device: PIC17C4X, PIC17C756

Function: Detects a Power-on Reset condition.

Include: `reset16.h`

Prototype: `char isPOR (void);`

Arguments: None

Remarks: This function detects if the microcontroller just left a Power-on Reset. This condition is indicated by the following status bits:
PIC17C4X
$\overline{TO} = 1$
$\overline{PD} = 1$
This condition also for MCLR reset during normal operation and `CLRWDT` instruction executed.
PIC17C756
$\overline{POR} = 0$
$\overline{BOR} = 0$
$\overline{TO} = 1$
$\overline{PD} = 1$

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

File Name: `reset16.c`

Code Example:
```
if(isPOR());
 then ...
```

isWDTTO

Device: PIC17C4X, PIC17C756

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

Include: `reset16.h`

Prototype: `char isWDTTO (void);`

Arguments: None

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

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 for information on compilers. Refer to the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) for information on linking.

isWDTTO (Continued)

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

File Name: `reset16.c`

Code Example:
```
while(!isWDTTO());
```

isWDTWU

Device: PIC17C4X, PIC17C756

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

Include: `reset16.h`

Prototype: `char isWDTWU (void);`

Arguments: None

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

PIC17C756
- $\overline{POR}$ = 1
- $\overline{BOR}$ = 1
- $\overline{TO}$ = 0
- $\overline{PD}$ = 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 for information on compilers. Refer to the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) for information on linking.

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

File Name: `reset16.c`

Code Example:
```
if(isWDTWU());
 then ...
```

isWU

Device: `PIC17C4X, PIC17C756`

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

Include: `reset16.h`

Prototype: `char isWU (void);`

Arguments: `None`

isWU (Continued)

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.

File Name: `reset16.c`

Code Example:
```
if(isWU());
 then ...
```

StatusReset

Device: PIC17C756

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

Include: `reset16.h`

Prototype: `void StatusReset (void);`

Arguments: None

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

Return Value: None

File Name: `reset16.c`

Code Example:
```
if(StatusReset());
 then ...
```

2.10.2 Example of Use

There are no interdependencies between reset functions. See individual function code examples.

2.11 SPI™ FUNCTIONS

This section contains a list of individual functions and an example of use of the functions in this section. Functions may be implemented as macros.

2.11.1 Function Descriptions

CloseSPI

Device:	PIC17C756
Function:	Disables the SSP module.
Include:	`spi16.h`
Prototype:	`void CloseSPI (void);`
Arguments:	None
Remarks:	This function disables the SSP module. Pin I/O returns under the control of the `DDRx` and `PORTx` registers.
Return Value:	None
File Name:	`closespi.c`
Code Example:	`CloseSPI();`

DataRdySPI

Device:	PIC17C756
Function:	Determines if the `SSPBUF` contains data.
Include:	`spi16.h`
Prototype:	`unsigned char DataRdySPI (void);`
Arguments:	None
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.
File Name:	`dtrdyspi.c`
Code Example:	`while (!DataRdySPI());`

getcSPI

Function:	This function operates identically to **ReadSPI**.
File Name:	`#define in spi16.h`

getsSPI

Device:	PIC17C756
Function:	Reads in data string from the SPI bus.
Include:	`spi16.h`
Prototype:	`void getsSPI (`static unsigned char far **rdptr*, static unsigned char *length*);

getsSPI (Continued)

Arguments: **rdptr**
Character type pointer to PICmicro RAM area for placement of data read from SPI device.
length
Number of bytes to read from SPI device.

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

File Name: `getsspi.c`

Code Example:
```
unsigned char far *wrptr;
getsSPI(wrptr, 10);
```

OpenSPI

Device: PIC17C756

Function: Initializes the SSP module.

Include: `spi16.h`

Prototype:
```
void OpenSPI (static unsigned char sync_mode, static
unsigned char bus_mode, static unsigned char
smp_phase);
```

Arguments: 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_4SPI	Master mode, clock = FOSC/4
FOSC_16SPI	Master mode, clock = FOSC/16
FOSC_64SPI	Master mode, clock = FOSC/64
FOSC_TMR2SPI	Master mode, clock = TMR2 output/2
SLV_SSONSPI	Slave mode, /SS pin control enabled
SLV_SSOFFSPI	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

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

Return Value: None

File Name: `openspi.c`

Code Example:
```
OpenSPI(FOSC_16, MODE_00, SMPEND);
```

putcSPI

Function: This function operates identically to **WriteSPI**.

File Name: `#define in spi16.h`

putsSPI

Device: PIC17C756

Function: Writes data string out to the SPI bus.

Include: `spi16.h`

Prototype: `void putsSPI (static unsigned char far *wrptr);`

Arguments: **wrptr**
Pointer to character type data objects in PICmicro RAM. Those objects pointed to by *wrptr* will be written to the SPI bus.

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

File Name: `putsspi.c`

Code Example:

```
unsigned char far *wrptr = "Hello!";
putsSPI(wrptr);
```

ReadSPI

Device: PIC17C756

Function: Reads a single byte (one character) from the `SSPBUF` register.

Include: `spi16.h`

Prototype: `unsigned char ReadSPI (void);`

Arguments: None

Remarks: This function initiates a SPI bus cycle for the acquisition of a byte of data. This function operates identically to **getcSPI**.

Return Value: This function returns a byte of data read during a SPI read cycle.

File Name: `readspi.c`

Code Example:

```
char x;
x = ReadSPI();
```

WriteSPI

Device: PIC17C756

Function: Writes a single byte of data (one character) out to the SPI bus.

Include: `spi16.h`

Prototype: `unsigned char WriteSPI (static unsigned char data_out);`

Arguments: **data_out**
Single byte to write to SPI device on bus.

WriteSPI (Continued)

Remarks:	This function writes a single data byte out and then checks for a write collision. This function operates identically to **putcSPI**.
Return Value:	This function returns -1 if a write collision occurred else a 0 if no write collision.
File Name:	`writespi.c`
Code Example:	`WriteSPI('a');`

2.11.2 Example of Use

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 Prototype
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,
                 unsigned 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
}
```

2.12 TIMER FUNCTIONS

This section contains a list of individual functions and an example of use of the functions in this section. Functions may be implemented as macros.

2.12.1 Function Descriptions

CloseTimer0
CloseTimer1
CloseTimer2
CloseTimer3

Device: PIC17C4X, PIC17C756

Function: This function disables the specified timer.

Include: `timers16.h`

Prototype:
```
void CloseTimer0 (void);
void CloseTimer1 (void);
void CloseTimer2 (void);
void CloseTimer3 (void);
```

Arguments: None

Remarks: This function simply disables the interrupt and the specified timer.

Return Value: None

File Name:
```
t0close.c
t1close.c
t2close.c
t3close.c
```

Code Example: `CloseTimer0();`

OpenTimer0
OpenTimer1
OpenTimer2
OpenTimer3

Device: PIC17C4X, PIC17C756

Function: Configures the specified timer.

Include: `timers16.h`

Prototype:
```
void OpenTimer0 (static unsigned char config);
void OpenTimer1 (static unsigned char config);
void OpenTimer2 (static unsigned char config);
void OpenTimer3 (static unsigned char config);
```

Arguments: **config**

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
OpenTimer1
OpenTimer2
OpenTimer3 (Continued)

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)

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.

Return Value: None

File Name:
```
t0open.c
t1open.c
t2open.c
t3open.c
```

Code Example:
```
OpenTimer0(TIMER_INT_OFF&T0_SOURCE_NT&T0_PS_1_32);
```

ReadTimer0
ReadTimer1
ReadTimer2
ReadTimer3
ReadTimer1_16

Device: PIC17C4X, PIC17C756

Function: Reads the contents of the specified timer register(s).

Include: `timers16.h`

Prototype:

```
unsigned int  ReadTimer0 (void);
unsigned char ReadTimer1 (void);
unsigned char ReadTimer2 (void);
unsigned int  ReadTimer3 (void);
unsigned int  ReadTimer1_16 (void);
```

Arguments: None

Remarks: This function reads the value of the respective timer register(s).

Timer0:	`TMR0L,TMR0H`
`Timer1:`	`TMR1`
`Timer2:`	`TMR2`
`Timer3:`	`TMR3L,TMR3H`
`Timer1_16:`	`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_16:	int (16-bits)

File Name:

```
t0read.c
t1read.c
t2read.c
t3read.c
t12read.c
```

Code Example:

```
unsigned int result;
result = ReadTimer0();
```

WriteTimer0
WriteTimer1
WriteTimer2
WriteTimer3
WriteTimer1_16

Device: PIC17C4X, PIC17C756

Function: Reads the contents of the specified timer register(s).

Include: `timers16.h`

Prototype:

```
void WriteTimer0 (static unsigned int timer);
void WriteTimer1 (static unsigned char timer);
void WriteTimer2 (static unsigned char timer);
void WriteTimer3 (static unsigned int timer);
void WriteTimer1_16 (static unsigned int timer);
```

Arguments: **timer**

This function writes the value *timer* to the respective timer register(s).

Timer0:	`TMR0L,TMR0H`
`Timer1:`	`TMR1`
`Timer2:`	`TMR2`
`Timer3:`	`TMR3L,TMR3H`
`Timer1_16:`	`TMR2:TMR1`

Remarks: 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_16:	int (16-bits)

Return Value: None

File Name:

`t0write.c`
`t1write.c`
`t2write.c`
`t3write.c`
t12write.c

Code Example:

```
WriteTimer0(0);
```

2.12.2 Example of Use

```
#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, 25);
 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;
}
```


2.13 USART FUNCTIONS

This section contains a list of individual functions and an example of use of the functions in this section. Functions may be implemented as macros.

2.13.1 Function Descriptions

BusyUSART1
BusyUSART2

Device:	BusyUSART1: PIC17C4X, PIC17C756 BusyUSART2: PIC17C756
Function:	Returns the status of the `TRMT` flag bit in the `TXSTA?` register.
Include:	`usart16.h`
Prototype:	`char BusyUSART1 (void);` `char BusyUSART2 (void);`
Arguments:	None
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.
File Name:	`u1busy.c` `u2busy.c`
Code Example:	`while (BusyUSART1());`

CloseUSART1
CloseUSART2

Device:	CloseUSART1: PIC17C4X, PIC17C756 CloseUSART2: PIC17C756
Function:	Disables the specified USART.
Include:	`usart16.h`
Prototype:	`void CloseUSART1 (void);` `void CloseUSART2 (void);`
Arguments:	None
Remarks:	This function disables the specified USARTs interrupts, transmitter and receiver.
Return Value:	None
File Name:	`u1close.c` `u2close.c`
Code Example:	`CloseUSART1();`

DataRdyUSART1
DataRdyUSART2

Device:	DataRdyUSART1: PIC17C4X, PIC17C756 DataRdyUSART2: PIC17C756
Function:	Returns the status of the `RCIF` flag bit in the `PIR` register.
Include:	`usart16.h`
Prototype:	`char DataRdyUSART1 (void);` `char DataRdyUSART2 (void);`
Arguments:	None
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.
File Name:	`u1drdy.c` `u2drdy.c`
Code Example:	`while (!DataRdyUSART1());`

getcUSART1
getcUSART2

Function:	This function operates identically to **ReadUSARTx**.
File Name:	`#define` in `usart16.h`

getsUSART1
getsUSART2

Device:	getsUSART1 :PIC17C4X, PIC17C756 getsUSART2: PIC17C756
Function:	Reads a string of characters until the specified number of characters have been read.
Include:	`usart16.h`
Prototype:	`void getsUSART1 (static char *buffer, static unsigned char len);` `void getsUSART2 (static char *buffer, static unsigned char len);`
Arguments:	**buffer** 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. **len** 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.

getsUSART1
getsUSART2 (Continued)

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.

Return Value: None

File Name: `u1gets.c`
`u2gets.c`

Code Example:
```
char x[10];
getsUSART2(x,5);
```

OpenUSART1
OpenUSART2

Device: OpenUSART1: PIC17C4X, PIC17C756
OpenUSART2: PIC17C756

Function: Configures the specified USART module.

Include: `usart16.h`

Prototype:
```
void OpenUSART1 (static unsigned char config, static
char spbrg);
void OpenUSART2 (static unsigned char config, static
char spbrg);
```

Arguments: **config**

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	8-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

spbrg

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))

OpenUSART1
OpenUSART2 (Continued)

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.
Return Value:	None
File Name:	`u1open.c` `u2open.c`
Code Example:	`OpenUSART1(USART_TX_INT_OFF&USART_RX_INT_OFF&USART_ASYNCH_MODE&USART_EIGHT_BIT&USART_CONT_RX, 25);`

putcUSART1
putcUSART2

Function:	This function operates identically to **WriteUSARTx**.
File Name:	`#define` in `usart16.h`

putrsUSART1
putrsUSART2

Device:	putrsUSART1: PIC17C4X, PIC17C756 putrsUSART2: PIC17C756
Function:	Writes a string of characters in ROM to the USART including the null character.
Include:	`usart16.h`
Prototype:	`void putrsUSART1 (static const rom char *data);` `void putrsUSART2 (static const rom char *data);`
Arguments:	**data** The value of *data* is a pointer to a string in contiguous RAM locations within the same bank.
Remarks:	This function writes a string of data in program memory to the USART, including the null character.
Return Value:	None
File Name:	`u1putrs.c` `u2putrs.c`
Code Example:	`rom char mybuff [20];` `putrsUSART1(mybuff);`

putsUSART1
putsUSART2

Device:	putsUSART1: PIC17C4X, PIC17C756 putsUSART2: PIC17C756
Function:	Writes a string of characters to the USART including the null character.
Include:	`usart16.h`
Prototype:	`void putsUSART1 (static char *data);` `void putsUSART2 (static char *data);`
Arguments:	**data** The value of *data* is a pointer to a string in contiguous RAM locations within the same bank.
Remarks:	This function writes a string of data to the USART including the null character.
Return Value:	None
File Name:	`u1puts.c` `u2puts.c`
Code Example:	`char mybuff [20];` `putsUSART1(mybuff);`

ReadUSART1
ReadUSART2

Device:	ReadUSART1: PIC17C4X, PIC17C756 ReadUSART2: PIC17C756
Function:	Reads a byte (one character) out of the USART receive buffer, including the 9th bit if enabled.
Include:	`usart16.h`
Prototype:	`char ReadUSART1 (void);` `char ReadUSART2 (void);`
Arguments:	None

ReadUSART1
ReadUSART2 (Continued)

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. This function operates identically to **getcUSARTx**.

Return Value: This function returns the next character in the USART receive buffer.

File Name: `u1read.c`
`u2read.c`

Code Example:

```
char x;
x = ReadUSART2();
```

WriteUSART1
WriteUSART2

Device: WriteUSART1: PIC17C4X, PIC17C756
WriteUSART2: PIC17C756

Function: Writes a byte (one character) to the USART transmit buffer, including the 9th bit if enabled.

Include: `usart16.h`

Prototype:

```
void WriteUSART1 (static char data);
void WriteUSART2 (static char data);
```

Arguments: **data**
The value of *data* can be any number from 0x00 to 0xff.

WriteUSART1
WriteUSART2 (Continued)

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.
This function operates identically to **putcUSARTx**.

Return Value: None

File Name: `u1write.c` `u2write.c`

Code Example:

```
char x;
WriteUSART2(x);
```

2.13.2 Example of Use

```
#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, 25);
 while(1)
 {
  while(!PORTAbits.RA0)//wait for RA0 high
  WriteUSART1(PORTD);//write value of PORTD
  if(PORTD == 0x80)
   break;
 }
 CloseUSART1();
 return;
}
```


NOTES:

Chapter 3. Software Peripheral Library

3.1 INTRODUCTION

This chapter documents software peripheral library functions. The source code for all of these functions is included with MPLAB-C17 in the `c:\mcc\src\pmc` directory, where `c:\mcc` is the compiler install directory.

See the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) for more information about building libraries.

3.2 HIGHLIGHTS

This chapter is organized as follows:

- External LCD Functions
- Software I^{2}C Functions
- Software SPI Functions
- Software UART Functions

3.3 EXTERNAL LCD FUNCTIONS

This section contains a list of individual functions and an example of use of the functions in this section. Functions may be implemented as macros.

3.3.1 Function Descriptions

BusyXLCD

Device:	PIC17C4X, PIC17C756
Function:	Returns the status of the busy flag of the Hitachi HD44780 LCD controller.
Include:	xlcd.h
Prototype:	unsigned char BusyXLCD (void);
Arguments:	None
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.
File Name:	`xlcd.c`
Code Example:	`while ( BusyXLCD() );`

OpenXLCD

Device: PIC17C4X, PIC17C756

Function: Configures the I/O pins and initializes the Hitachi HD44780 LCD controller.

Include: `xlcd.h`

Prototype: `void OpenXLCD (static unsigned char lcdtype);`

Arguments: **lcdtype**
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

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?
```

where `x` is the PORT, `?` is the pin number

Data Port definitions

```
DATA_PORT       PORTx
TRIS_DATA_PORT  DDRx
```

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 the *MPLAB CXX Compiler User's Guide* (DS51217) for information on the compilers and to the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) for information on linking.

This function also requires three external routines to be provided by the user for specific delays:

DelayFor18TCY()	18 Tcy delay
DelayPORXLCD()	15 ms delay
DelayXLCD()	5 ms delay

OpenXLCD (Continued)

Return Value: None

File Name: `xlcd.c`

Code Example: `OpenXLCD(EIGHT_BIT&LINES_5X7);`

putrsXLCD

Device: PIC17C4X, PIC17C756

Function: Writes a string of characters in ROM to the Hitachi HD44780 LCD controller.

Include: `xlcd.h`

Prototype: `void putrsXLCD (static rom char *buffer);`

Arguments: **buffer**
Pointer to characters to be written to the LCD controller.

Remarks: This functions writes a string of characters located in program memory 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

File Name: `xlcd.c`

Code Example:

```
rom char mybuff [20];
putrsXLCD(mybuff);
```

putcXLCD

Function: This function operates identically to **WriteDataXLCD**.

File Name: `#define` in `xlcd.h`

putsXLCD

Device: PIC17C4X, PIC17C756

Function: Writes a string of characters to the Hitachi HD44780 LCD controller.

Include: `xlcd.h`

Prototype: `void putsXLCD (static char *buffer);`

Arguments: **buffer**
Pointer to characters to be written to the LCD controller.

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

File Name: `xlcd.c`

Code Example:

```
char mybuff [20];
putsXLCD(mybuff);
```

ReadAddrXLCD

Device: PIC17C4X, PIC17C756

Function: Reads the address byte from the Hitachi HD44780 LCD controller.

Include: `xlcd.h`

Prototype: `unsigned char ReadAddrXLCD (void);`

Arguments: None

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
```

File Name: `xlcd.c`

Code Example:

```
char addr;
while ( BusyXLCD() );
addr = ReadAddrXLCD();
```

ReadDataXLCD

Device: PIC17C4X, PIC17C756

Function: Reads a data byte from the Hitachi HD44780 LCD controller.

Include: `xlcd.h`

Prototype: `char ReadDataXLCD (void);`

Arguments: None

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.

File Name: `xlcd.c`

Code Example:

```
char data;
while ( BusyXLCD() );
data = ReadAddrXLCD();
```


SetCGRamAddr

Device: PIC17C4X, PIC17C756

Function: Sets the character generator address.

Include: `xlcd.h`

Prototype: `void SetCGRamAddr (static unsigned char CGaddr);`

Arguments: **CGaddr**
Character generator address.

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

File Name: `xlcd.c`

Code Example:

```
char cgaddr = 0x1F;
while ( BusyXLCD() );
SetCGRamAddr(cgaddr);
```

SetDDRamAddr

Device: PIC17C4X, PIC17C756

Function: Sets the display data address.

Include: `xlcd.h`

Prototype: `void SetDDRamAddr (static unsigned char DDaddr);`

Arguments: **DDaddr**
Display data address.

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

File Name: `xlcd.c`

Code Example:

```
char ddaddr = 0x10;
while ( BusyXLCD() );
SetDDRamAddr(ddaddr);
```

WriteCmdXLCD

Device: PIC17C4X, PIC17C756

Function: Writes a command to the Hitachi HD44780 LCD controller.

Include: `xlcd.h`

Prototype: `void WriteCmdXLCD (static unsigned char cmd);`

Arguments: **cmd**
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

WriteCmdXLCD (Continued)

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.

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.

Return Value: None

File Name: `xlcd.c`

Code Example:

```
while ( BusyXLCD() );
WriteCmdXLCD(EIGHT_BIT&LINES_5X7);
WriteCmdXLCD(DON);
WriteCmdXLCD(SHIFT_DISP_LEFT);
```

WriteDataXLCD

Device: PIC17C4X, PIC17C756

Function: Writes a data byte (one character) from the Hitachi HD44780 LCD controller.

Include: `xlcd.h`

Prototype: `void WriteDataXLCD (static char data);`

Arguments: **data**
The value of *data* can be any 8-bit value, but should correspond to the character RAM table of the HD44780 LCD controller.

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.
This function operates identically to **putcXLCD**.

Return Value: None

File Name: `xlcd.c`

Code Example:

```
char data;
data = ReadUSART1();
WriteDataXLCD(data);
```

3.3.2 Example of Use

```
#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, 25);
 while(1)
 {
  while(!DataRdyUSART1()); //wait for data
  data = ReadUSART1();     //read data
  WriteDataXLCD(data);     //write to LCD
  if(data=='Q')
   break;
 }
 CloseXLCD();              //close modules
 CloseUSART1();
 return;
}
```

3.4 SOFTWARE I²C FUNCTIONS

This section contains a list of individual functions and an example of use of the functions in this section. Functions may be implemented as macros.

3.4.1 Function Descriptions

Clock_test

Device: PIC17CXXX

Function: Generates delay for slave clock stretching.

Include: `swi2c16.h`

Prototype: `void Clock_test (void);`

Arguments: None

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; define union/struct
```

Return Value: None

File Name: `swckti2c.c`

Code Example: `Clock_test();`

SWAckI2C

Device: PIC17CXXX

Function: Generates I²C bus acknowledge condition.

Include: `swi2c16.h`

Prototype: `void SWAckI2C (void);`

Arguments: None

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.

SWAckI2C (Continued)

```
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
```

This function operates identically to **SWNotAckI2C**.

Return Value: None

File Name: `swacki2c.c`

Code Example: `SWAckI2C();`

SWGetcI2C

Function: This function operates identically to **SWReadI2C**.

File Name: `#define in swi2c16.h`

SWGetsI2C

Device: PIC17CXXX

Function: Reads in data string via software I^2C implementation.

Include: `swi2c16.h`

Prototype: `unsigned char SWGetsI2C (static unsigned char far *rdptr, static unsigned char length);`

Arguments: **rdptr**
Character type pointer to PICmicro RAM for storage of data read from I^2C device.
length
Number of bytes to read from I^2C bus.

Remarks: This function reads in a predetermined data string *length*. Each byte is retrieved via a call to the **SWGetcI2C** function.

Return Value: This function returns -1 if all bytes have been received and the master generated a *not ack* bus condition.

File Name: `swgtsi2c.c`

Code Example:

```
char x[10];
SWGetsI2C(x,5);
```

SWNotAckI2C

Function: This function operates identically to **SWAckI2C**.

File Name: `#define` in `swi2c16.h`

SWPutcI2C

Function: This function operates identically to **SWWriteI2C**.

File Name: `#define in swi2c16.h`

SWPutsI2C

Device: PIC17CXXX

Function: Writes out data string via software I^{2}C implementation.

Include: `swi2c16.h`

Prototype:
```
unsigned char SWPutsI2C
(static unsigned char far *wrdptr);
```

Arguments: **wrdptr**
Character type pointer to data objects in PICmicro RAM. The data objects are written to the I^{2}C device.

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.

File Name: `swptsi2c.c`

Code Examples:
```
char mybuff [20];
SWPutsI2C(mybuff);
```

SWReadI2C

Device: PIC17CXXX

Function: Reads a single data byte (one character) via software I^{2}C implementation.

Include: `swi2c16.h`

Prototype: `unsigned char SWReadI2C (void);`

Arguments: None

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. This function operates identically to **SWGetcI2C**.

File Name: `swgtci2c.c`

Code Example:
```
char x;
x = SWReadI2C();
```

SWRestartI2C

Device:	PIC17CXXX
Function:	Generates I^2C restart bus condition.
Include:	`swi2c16.h`
Prototype:	`void SWRestartI2C (void);`
Arguments:	None
Remarks:	This function is called to generate an I^2C bus restart condition.
Return Value:	None
File Name:	`swrsti2c.c`
Code Example:	`SWRestartI2C();`

SWStartI2C

Device:	PIC17CXXX
Function:	Generates I^2C bus start condition.
Include:	`swi2c16.h`
Prototype:	`void SWStartI2C (void);`
Arguments:	None
Remarks:	This function is called to generate an I^2C bus start condition.
Return Value:	None
File Name:	`swstri2c.c`
Code Example:	`SWStartI2C();`

SWStopI2C

Device:	PIC17CXXX
Function:	Generates I^2C bus stop condition.
Include:	`swi2c16.h`
Prototype:	`void SWStopI2C (void);`
Arguments:	None
Remarks:	This function is called to generate an I^2C bus stop condition.
Return Value:	None
File Name:	`swstpi2c.c`
Code Example:	`SWStopI2C();`

SWWriteI2C

Device:	PIC17CXXX
Function:	Writes out single data byte via software I²C implementation.
Include:	`swi2c16.h`
Prototype:	`unsigned char SWWriteI2C (static unsigned char data_out);`
Arguments:	**data_out** Single data byte to be written to the I²C device.
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. This function operates identically to **SWPutcI2C**.
Return Value:	This function returns -1 if there was an error condition else returns a 0.
File Name:	`swptci2c.c`
Code Example:	`char x;` `SWWriteI2C(x);`

3.4.2 Example of Use

The following are simple code examples illustrating a software I²C implementation communicating with a Microchip 24LC01B I²C 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 requirements.

```
#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 Prototype
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();
}
```

3.5 SOFTWARE SPI FUNCTIONS

This section contains a list of individual functions and an example of use of the functions in this section. Functions may be implemented as macros.

3.5.1 Function Descriptions

ClearSWCSSPI

Device:	PIC17C4X, PIC17C756
Function:	Clears the chip select (CS) pin that is specified in the `swspi16.h` header file.
Include:	`swspi16.h`
Prototype:	`void SWClearCSSPI (void);`
Arguments:	None
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
File Name:	`swspi16.c`
Code Example:	`ClearSWCSSPI();`

OpenSWSPI

Device: PIC17C4X, PIC17C756

Function: Configures the I/O pins for the software SPI.

Include: `swspi16.h`

Prototype: `void SWOpenSPI (void);`

Arguments: None

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?
```

where `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.

OpenSWSPI (Continued)

After these definitions have been made, compile the software SPI files into an executable. For information on compilers, refer to the *MPLAB CXX Compiler User's Guide* (DS51217). Refer to the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) for information on linking.

Return Value: None

File Name: `swspi16.c`

Code Example: `OpenSWSPI();`

putcSWSPI

Function: This function operates identically to **WriteSWSPI**.

File Name: `#define in swspi16.h`

SetSWCSSPI

Device: PIC17C4X, PIC17C756

Function: Sets the chip select (CS) pin that is specified in the `swspi16.h` header file.

Include: `swspi16.h`

Prototype: `void SWSetCSSPI (void);`

Arguments: None

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

File Name: `swspi16.c`

Code Example: `SetSWCSSPI();`

WriteSWSPI

Device: PIC17C4X, PIC17C756

Function: Reads/writes one byte of data out the software SPI.

Include: `swspi16.h`

Prototype: `char SWWriteSPI (static char data);`

Arguments: **data**
Byte of data written to software SPI.

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).
This function operates identically to **putcSWSPI**.

Return Value: This function returns the byte of data that was read from the data in (DIN) pin of the software SPI.

File Name: `swspi16.c`

Code Example:

```
char addr;
WriteSWSPI(addr);
```

3.5.2 Example of Use

```
#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;
}
```

3.6 SOFTWARE UART FUNCTIONS

This section contains a list of individual functions and an example of use of the functions in this section. Functions may be implemented as macros.

3.6.1 Function Descriptions

getcUART

Function: This function operates identically to **ReadUART**.

File Name: `#define in uart16.h`

getsUART

Device: PIC17C4X, PIC17C756

Function: Reads a string of characters from the software UART.

Include: `uart16.h`

Prototype: `void getsUART (static char *buffer, static unsigned char len);`

Arguments:
buffer
Pointer to the string of characters read from the software UART.
len
Number of characters read from the software UART. 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.

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*.

Return Value: None

File Name: `uart16_c.c`

Code Example:
```
char x[10];
getsUART(x,5);
```

OpenUART

Device: PIC17C4X, PIC17C756

Function: Configures the I/O pins for the software UART.

Include: `uart16.h`

Prototype: `void OpenUART (void);`

Arguments: None

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:

OpenUART (Continued)

I/O pin definitions

```
SWTXD          equ   PORTx
SWTXDpin       equ   ?
TRIS_SWTXD     equ   DDRx
SWRXD          equ   PORTx
SWRXDpin       equ   ?
TRIS_SWRXD     equ   DDRx
UART_PORT_BSR  equ   b
```

where `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 the *MPLAB CXX Compiler User's Guide* (DS51217) for information on compilers. Refer to the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) for information on linking.

Return Value: None

File Name: `uart16_c.c`

Code Example: `OpenUART();`

putcUART

Function: This function operates identically to **WriteUART**.

File Name: `#define in uart16.h`

putsUART

Device: PIC17C4X, PIC17C756

Function: Writes a string of characters to the software UART.

Include: `uart16.h`

Prototype: `void getsUART (static char *buffer);`

Arguments: **buffer**
Pointer to characters written to data string of software UART.

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

File Name: `uart16_c.c`

Code Example:

```
char mybuff [20];
putsUART(mybuff);
```

ReadUART

Device:	PIC17C4X, PIC17C756
Function:	Reads one byte of data out the software UART.
Include:	`uart16.h`
Prototype:	`char ReadUART (void);`
Arguments:	None
Remarks:	This function reads a byte of data out the software UART and returns the byte of data. This function operates identically to **getcUART**.
Return Value:	This function returns the byte of data that was read from the receive data (RXD) pin of the software UART.
File Name:	`uart16_a.asm`
Code Example:	`char x;` `x = ReadUART();`

WriteUART

Device:	PIC17C4X, PIC17C756
Function:	Writes one byte of data out the software UART.
Include:	`uart16.h`
Prototype:	`void WriteUART (static char data);`
Arguments:	**data** Byte of data written to software UART. The value of *data* can be any 8-bit value.
Remarks:	This function writes the specified byte of data out the software UART. This function operates identically to **putcUART**.
Return Value:	None
File Name:	`uart16_a.asm`
Code Example:	`char x;` `WriteUART(x);`

3.6.2 Example of Use

```
#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;
}
```


Chapter 4. General Software Library

4.1 INTRODUCTION

This chapter documents general software library functions. The source code for all of these functions is included with MPLAB-C17 in the `c:\mcc\src\pmc` directory, where `c:\mcc` is the compiler install directory.

See the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) for more information about building libraries.

4.2 HIGHLIGHTS

This chapter is organized as follows:

- Character Classification Functions
- Number and Text Conversion Functions
- Delay Functions
- Memory and String Manipulation Functions

4.3 CHARACTER CLASSIFICATION FUNCTIONS

4.3.1 Function Descriptions

isalnum

Device:	PIC17C4X, PIC17C756
Function:	Alphanumeric character classification.
Include:	`ctype.h`
Prototype:	`char isalnum (static char` *`ch`*`);`
Arguments:	**ch** Character.

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.
File Name:	`isalnum.c`

isalpha

Device: PIC17C4X, PIC17C756

Function: Alphabetical character classification.

Include: `ctype.h`

Prototype: `char isalpha (static char ch);`

Arguments: **ch**
Character.

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.

File Name: `isalpha.c`

isascii

Device: PIC17C4X, PIC17C756

Function: ASCII character classification.

Include: `ctype.h`

Prototype: `char isascii (static char ch);`

Arguments: **ch**
Character.

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.

File Name: `isascii.c`

iscntrl

Device: PIC17C4X, PIC17C756

Function: Control character classification.

Include: `ctype.h`

Prototype: `char iscntrl (static char ch);`

Arguments: **ch**
Character.

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.

File Name: `iscntrl.c`

isdigit

Device:	PIC17C4X, PIC17C756
Function:	Numeric character classification.
Include:	`ctype.h`
Prototype:	`char isdigit (static char ch);`
Arguments:	**ch** Character.
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.
File Name:	`isdigit.c`

islower

Device:	PIC17C4X, PIC17C756
Function:	Lower case alphabetical character classification.
Include:	`ctype.h`
Prototype:	`char islower (static char ch);`
Arguments:	**ch** Character.
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.
File Name:	`islower.c`

isupper

Device:	PIC17C4X, PIC17C756
Function:	Upper case alphabetical character classification.
Include:	`ctype.h`
Prototype:	`char isupper (static char ch);`
Arguments:	**ch** Character.
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.
File Name:	`isupper.c`

isxdigit

Device: PIC17C4X, PIC17C756

Function: Hexadecimal character classification.

Include: `ctype.h`

Prototype: `char isxdigit (static char ch);`

Arguments: **ch**
Character.

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.

File Name: `isxdig.c`

4.4 NUMBER AND TEXT CONVERSION FUNCTIONS

atob

Device:	PIC17C4X, PIC17C756
Function:	Converts a string to an 8-bit signed byte.
Include:	`stdlib.h`
Prototype:	`char atob (static char *string);`
Arguments:	**string** Pointer to ASCII 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.
File Name:	`atob.c`

atoi

Device:	PIC17C4X, PIC17C756
Function:	Converts a string to an 16-bit signed integer.
Include:	`stdlib.h`
Prototype:	`int atoi(static char *string);`
Arguments:	**string** Pointer to ASCII 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.
File Name:	`atoi.c`

atoub

Device:	PIC17C4X, PIC17C756
Function:	Converts a string to an 8-bit unsigned byte.
Include:	`stdlib.h`
Prototype:	`unsigned char atoub (static char *string);`
Arguments:	**string** Pointer to ASCII 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.
File Name:	`atoub.c`

atoui

Device:	PIC17C4X, PIC17C756
Function:	Converts a string to an 16-bit unsigned integer.
Include:	`stdlib.h`
Prototype:	`unsigned int atoui (static char *string);`
Arguments:	**string** Pointer to ASCII 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
File Name:	`atoui.c`

btoa

Device:	PIC17C4X, PIC17C756
Function:	Converts an 8-bit signed byte to string.
Include:	`stdlib.h`
Prototype:	`void btoa (static char value, static char *string);`
Arguments:	**value** An 8-bit signed byte. **string** Pointer to ASCII 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

btoa (Continued)

	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
File Name:	`btoa.c`

itoa

Device:	PIC17C4X, PIC17C756
Function:	Converts an 16-bit signed integer to string.
Include:	`stdlib.h`
Prototype:	`void itoa (static int` *`value`*`, static char *`*`string`*`);`
Arguments:	**value** An 8-bit signed byte. **string** Pointer to ASCII 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
File Name:	`itoa.c`

toascii

Device:	PIC17C4X, PIC17C756
Function:	Converts a character to an ASCII character
Include:	`ctype.h`
Prototype:	`char toascii (static char ch);`
Arguments:	**ch** Character.
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.
File Name:	`toascii.c`

tolower

Device:	PIC17C4X, PIC17C756
Function:	Converts a character to a lower case alphabetical ASCII character.
Include:	`ctype.h`
Prototype:	`char tolower (static char ch);`
Arguments:	**ch** Character.
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.
File Name:	`tolower.c`

toupper

Device:	PIC17C4X, PIC17C756
Function:	Converts a character to a upper case alphabetical ASCII character.
Include:	`ctype.h`
Prototype:	`char toupper (static char ch);`
Arguments:	**ch** Character.
Remarks:	This function converts *ch* to a upper case alphabetical ASCII character provided that the argument is a valid lower case alphabetical haracter.
Return Value:	This function returns a lower case character if the argument was upper case to begin with, otherwise the original character is returned.
File Name:	`toupper.c`

ubtoa

Device: PIC17C4X, PIC17C756

Function: Converts an 8-bit unsigned byte to string.

Include: `stdlib.h`

Prototype: `void ubtoa (static unsigned char value, static char *string);`

Arguments: **value**
An 8-bit signed byte.
string
Pointer to ASCII 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

File Name: `ubtoa.c`

uitoa

Device: PIC17C4X, PIC17C756

Function: Converts an 16-bit unsigned integer to string.

Include: `stdlib.h`

Prototype: `void uitoa (static unsigned int value, static char *string);`

Arguments: **value**
An 8-bit signed byte.
string
Pointer to ASCII 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

File Name: `uitoa.c`

4.5 DELAY FUNCTIONS

Delay1TCY

Device:	PIC17C4X, PIC17C756
Function:	Delay of 1 instruction cycle (Tcy).
Include:	`delays.h`
Prototype:	`void Delay1TCY (void);`
Arguments:	None
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
File Name:	`#define` in `delays.h`

Delay10TCY

Device:	PIC17C4X, PIC17C756
Function:	Delay of 10 instruction cycles (Tcy).
Include:	`delays.h`
Prototype:	`void Delay10TCY (void);`
Arguments:	None
Remarks:	This function creates a delay of 10 instruction cycles.
Return Value:	None
File Name:	`dy10tcy.asm`

Delay10TCYx

Device:	PIC17C4X, PIC17C756
Function:	Delay of multiples of 10 instruction cycles (Tcy).
Include:	`delays.h`
Prototype:	`void Delay10TCYx (static unsigned char` *`unit`*`);`
Arguments:	**unit** 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.
Remarks:	This function creates delays of multiples of 10 instruction cycles.
Return Value:	None
File Name:	`dy1otcyx.asm`

Delay100TCYx

Device:	PIC17C4X, PIC17C756
Function:	Delay of multiples of 100 instruction cycles (Tcy).
Include:	`delays.h`
Prototype:	`void Delay100TCYx (static unsigned char` *`unit`*`);`
Arguments:	**unit** 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.
Remarks:	This function creates delays of multiples of 100 instruction cycles.
Return Value:	None
File Name:	`dy100tcx.asm`

Delay1KTCYx

Device:	PIC17C4X, PIC17C756
Function:	Delay of multiples of 1000 instruction cycles (Tcy).
Include:	`delays.h`
Prototype:	`void Delay1KTCYx (static unsigned char` *`unit`*`);`
Arguments:	**unit** 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.
Remarks:	This function creates delays of multiples of 1000 instruction cycles.
Return Value:	None
File Name:	`dy1ktcyx.asm`

Delay10KTCYx

Device:	PIC17C4X, PIC17C756
Function:	Delay of multiples of 10000 instruction cycles (Tcy).
Include:	`delays.h`
Prototype:	`void Delay10KTCYx (static unsigned char` *`unit`*`);`
Arguments:	**unit** 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.
Remarks:	This function creates delays of multiples of 10000 instruction cycles.
Return Value:	None
File Name:	`dy10ktcx.asm`

4.6 MEMORY AND STRING MANIPULATION FUNCTIONS

memcmp

Device:	PIC17C4X, PIC17C756
Function:	Compares the contents of two arrays of bytes.
Include:	`mem.h`
Prototype:	`signed char memcmp (static char *buf1, static char *buf2, static unsigned char memsize);`
Arguments:	**buf1** Pointer to first array. **buf2** Pointer to second array. **memsize** Number of elements to be compared in arrays.
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
File Name:	`memcmp.c`

memcpy

Device:	PIC17C4X, PIC17C756
Function:	Copies the contents of the source buffer into the destination buffer.
Include:	`mem.h`
Prototype:	`void memcpy (static char *dest, static char *src, static unsigned char memsize);`
Arguments:	**dest** Pointer to destination array. **src** Pointer to source array. **memsize** Number of elements of *src* array copied into *dest*.
Remarks:	This function copies the first *memsize* number of elements in *src* to the array *dest*.
Return Value:	None
File Name:	`memcpy.c`

memset

Device: PIC17C4X, PIC17C756

Function: Copies the specified character into the destination array.

Include: `mem.h`

Prototype: `void memset (static char *dest, static char value, static unsigned char memsize);`

Arguments:
dest
Pointer to destination array.
value
Character value to be copied.
memsize
Number of elements of *dest* into which *value* is copied.

Remarks: This function copies the character *value* into the first *memsize* elements of the array *dest*.

Return Value: None

File Name: `memset.c`

strcat

Device: PIC17C4X, PIC17C756

Function: Concatenates the source string to the end of the destination string.

Include: `string.h`

Prototype: `void strcat (static char *dest, static char *src);`

Arguments:
dest
Pointer to destination array.
src
Pointer to source array.

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

File Name: `strcat.c`

strcmp

Device:	PIC17C4X, PIC17C756
Function:	Compares two strings.
Include:	`string.h`
Prototype:	signed char strcmp (static char **str1*, static char **str2*);
Arguments:	**str1** Pointer to first string. **str2** Pointer to second string.
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
File Name:	`strcmp.c`

strcpy

Device:	PIC17C4X, PIC17C756
Function:	Copies the source string into the destination string.
Include:	`string.h`
Prototype:	`void strcpy (static char *dest, static char *src);`
Arguments:	**dest** Pointer to destination string. **src** Pointer to source string.
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
File Name:	`strcpy.c`

strlen

Device:	PIC17C4X, PIC17C756
Function:	Returns the length of the string.
Include:	`string.h`
Prototype:	`unsigned char strlen (static char *str);`
Arguments:	**str** Pointer to string.
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.
File Name:	`strlen.c`

strlwr

Device:	PIC17C4X, PIC17C756
Function:	Converts all upper case characters in a string to lower case.
Include:	`string.h`
Prototype:	`void strlwr (static char *str);`
Arguments:	**str** Pointer to string.
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
File Name:	`strlwr.c`

strset

Device:	PIC17C4X, PIC17C756
Function:	Copies the specified character into all characters in a string.
Include:	`string.h`
Prototype:	`void strset (static char *str, static char ch);`
Arguments:	**str** Pointer to string. **ch** Character.
Remarks:	This function copies the character in *ch* to all characters in the string up to the null character.
Return Value:	None
File Name:	`strset.c`

strupr

Device:	PIC17C4X, PIC17C756
Function:	Converts all lower case characters in a string to upper case.
Include:	`string.h`
Prototype:	`void strupr (static char *str);`
Arguments:	**str** Pointer to string.
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
File Name:	`strupr.c`

NOTES:

Chapter 5. Math Library

5.1 INTRODUCTION

This chapter documents math library functions. For more information on math libraries, see the *Embedded Control Handbook, Volume 2* (DS00167). See the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) for more information on creating and using libraries in general.

5.2 HIGHLIGHTS

This chapter is organized as follows:

- 32-Bit Integer and 32-Bit Floating Point Math Libraries
- Decimal/Floating Point and Floating Point/Decimal Conversions

5.3 32-BIT INTEGER AND 32-BIT FLOATING POINT MATH LIBRARIES

The math routines used by MPLAB-C17 are based on the Microchip Application Note AN575. Source code for the routines may be found in the `c:\mcc\src\math` directory, where `c:\mcc` is the compiler install directory. These source files have been compiled into object code and added to a library called `cmath17.lib`, which may be found in the `c:\mcc\lib` folder. The `cmath17.lib` file must be included during the linking process when using floating point or 32-bit integer routine function calls in your C code.

The mathematical functions performed by the floating point library routines are: 32-bit signed and unsigned integer multiplication; 32-bit signed and unsigned integer division; 32-bit floating point multiplication and division. The routines also contain conversion functions to go from 8, 16 and 32-bit signed and unsigned integers to 32-bit floating point, as well as a 32-bit floating point conversion to 32-bit integer. Calling conventions will be discussed later.

5.3.1 Floating Point Representation

Floating point numbers are represented in a modified IEEE-754 format. This format allows the floating-point routines to take advantage of the processor architecture and reduce the amount of overhead required in the calculations. The representation is shown below:

Format	Exponent	Mantissa 0	Mantissa 1	Mantissa 2
IEEE-754	`sxxx xxxx`	`yxxx xxxx`	`xxxx xxxx`	`xxxx xxxx`
Microchip	`xxxx xxxy`	`sxxx xxxx`	`xxxx xxxx`	`xxxx xxxx`

where `s` is the sign bit, `y` is the LSb of the exponent and `x` is a placeholder for the mantissa and exponent bits.

The two formats may be easily converted from one to the other by simple a manipulation of the Exponent and Mantissa 0 bytes. The following C code shows an example of this operation.

EXAMPLE 5-1: IEEE-754 TO MICROCHIP

```
Rlcf(AARGB0);
Rlcf(AEXP);
Rrcf(AARGB0);
```

EXAMPLE 5-2: MICROCHIP TO IEEE-754

```
Rlcf(AARGB0);
Rrcf(AEXP);
Rrcf(AARGB0);
```

5.3.2 Variables Used by the Floating Point Libraries

Several 8-bit RAM registers are used by the math routines to hold the operands for and results of floating point and integer operations. Since there may be two operands required for a floating point operation (such as multiplication or division), there are two sets of exponent and mantissa registers reserved. AEXP and BEXP hold the exponent for arguments A and B respectively while AARGB0, AARGB1 and AARGB2 or BARGB0, BARGB1 and BARGB2 hold the mantissa.

> **Note:** The MSB of the mantissa is stored in the AARGB0 or BARGB0 byte. Results of the floating point routines are placed in the AEXP and AARGB0:2 registers.

For 32-bit integers, AARGB0, AARGB1, AARGB2 and AARGB3 or BARGB0, BARGB1, BARGB2 and BARGB3 are used to hold the operands. Results of integer operations will be placed in AARGB0, AARGB1, AARGB2 and AARGB3. In the case of 32-bit division, the remainder is placed in an additional set of registers, REMB0, REMB1, REMB2 and REMB3. The MSB of the 32-bit integer is contained in AARGB0, BARGB0 or REMB0.

5.3.3 Calling the Math Functions

Before calling a math operation, the exponent and/or mantissa operands must be set up by your C code. For those operations that require two arguments (such as division or multiplication), both sets of arguments must be initialized. Once initialization is complete, the math function may be called using standard C function calls. The operands of the math routine are not passed as arguments to the function.Table 5-1 shows the syntax, operation, operand(s) required and where to extract the result of the operation.

TABLE 5-1: MATH FUNCTIONS

Syntax	Operation	Operand(s)	Result In
FXM3232U()	A•B (unsigned 32-bit integers)	A, B	A
FXM3232S()	A•B (signed 32-bit integers)	A, B	A
FXD3232U()	A/B (unsigned 32-bit integers)	A, B	A, REM
FXD3232S()	A/B (signed 32-bit integers)	A, B	A, REM
FPM32()	A•B (32-bit floating point)	A, B	A
FPD32()	A/B (32-bit floating point)	A, B	A
FLO3232U()	32-bit unsigned int to 32-bit floating point	A	A
FLO3232S()	32-bit signed int to 32-bit floating point	A	A
FLO1632U()	16-bit unsigned int to 32-bit floating point	A	A
FLO1632S()	16-bit signed int to 32-bit floating point	A	A
FLO0832U()	8-bit unsigned int to 32-bit floating point	A	A
FLO0832S()	8-bit signed int to 32-bit floating point	A	A
INT3232()	32-bit floating point to 32-bit integer	A	A

5.3.4 Example

Given two 32-bit signed integers, `int1` (AARG) and `int2` (BARG), the following code will multiply the two numbers and place the result in `int1` (AARG). Banking and paging considerations have been omitted for clarity. Include this code into your C program as inline assembly code.

```
MOVFP int1,      WREG    ; Load AARG
MOVWF AARGB0
MOVFP int1+1,    WREG
MOVWF AARGB1
MOVFP int1+2,    WREG
MOVWF AARGB2
MOVPF int1+3,    WREG
MOVWF AARGB3
MOVFP int2,      WREG
MOVWF BARGB0              ; Load BARG
MOVFP int2+1,    WREG
MOVWF BARGB1
MOVFP int2+2,    WREG
MOVWF BARGB2
MOVPF int2+3,    WREG
MOVWF BARGB3
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
```

5.4 DECIMAL/FLOATING POINT AND FLOATING POINT/DECIMAL CONVERSIONS

The details of how decimal numbers are converted to floating point numbers and how floating point numbers are converted to decimal numbers are discuss in the following sections.

5.4.1 Converting Decimal to Microchip Floating Point

There are several methods that will allow the conversion of decimal (base 10) numbers to Microchip floating point format. Microchip provides a PC utility called `FPREP.EXE`, which will convert decimal numbers to floating point for use in the math library routines. This utility may be download from the Microchip web site along with the AN575 source code.

Alternatively, the floating point equivalent to decimal numbers may be calculated longhand. To calculate the floating point via a longhand method, both the exponent and mantissa must be found.

To find the exponent, the following formulae are used:

EQUATION 5-1:

$$2^{Z} = A_{10}$$

EQUATION 5-2:

$$\mathrm{Exp} = \mathrm{int}(Z)$$

where Z is the fractional exponent, A_{10} is the original decimal number, and Exp is the integer portion of Z.

To solve for the exponent, first begin by rearranging Equation 5-1 to solve for Z.

$$Z = \frac{\ln(A_{10})}{\ln(2)}$$

The absolute value of Z is then rounded to the next larger absolute value integer to yield the value of Exp. Finally a bias value of 0x7F is added to convert Exp to Microchip floating point format.

Next, the mantissa is determined. The exponent value just determined must be removed from the original decimal number, using division.

EQUATION 5-3:

$$x = \frac{A_{10}}{2^{Z}}$$

where x is the fractional portion of the mantissa, and A_{10} and Z are values as described above.

> **Note:** x will always be a value greater than 1.

To determine the binary representation of the mantissa, x is compared in turn to decreasing powers of 2, starting with 2^{0} and decreasing to 2^{-23}. If x is greater than or equal to the power of 2 currently being compared, a ‘1’ is placed in the corresponding bit position of the binary representation and the power of 2 value is subtracted from x.

The new x is then used for the next decreasing power of 2 comparison. If x is less than the power of 2 currently being compared, a ‘0’ is placed in the bit position and no subtraction occurs. The same value of x is used to compare to the next power of 2 value.

This process repeats until all 24 bits have been determined or until subtraction yields an x value of 0. Finally, to convert this 24-bit value to Microchip floating point format, the MSb is substituted with the sign of the original decimal number, i.e., ‘1’ for negative or ’0’ for positive.

To demonstrate the method of conversion, the same example as in AN575 will be used, where A_{10} = 0.15625.

First, find the exponent:

$$2^Z = 0.15625$$

$$Z = \frac{\ln(0.15625)}{\ln(2)} = -2.6780719$$

$$\text{Exp} = \text{int}(Z) = -3$$

Next calculate the fractional portion of the mantissa:

$$x = \frac{0.15625}{2^{-3}} = 1.25$$

And then the binary representation:

$x = 1.25 \geq 2^0?$	Yes	bit = 1	x = 1.25 - 1 = 0.25
$x = 0.25 \geq 2^{-1}?$	No	bit = 0	x = 0.25
$x = 0.25 \geq 2^{-2}?$	Yes	bit = 1	x = 0.25 - 0.25 = 0
$x = 0$	Process complete		

Therefore, the binary representation is:

A_2=1.01000000000000000000000.

Finally, convert to Microchip floating point format by placing the proper sign bit in the MSb of the mantissa and add `0x7F` to the calculated exponent. The Microchip floating point representation of 0.156256 is then `0x7C200000`.For more details on the floating point conversion, please consult AN575.

5.4.2 Converting Microchip Floating-Point to Decimal

The process of converting floating-point number to decimal is relatively simple and can be done by hand (or using a calculator) to check your results.To convert from floating point to decimal, the following formula is used:

EQUATION 5-4:

$$A_{10} = 2^{\text{Exp}} \cdot A_2$$

where *Exp* is the unbiased exponent and *A* is the binary expansion of the mantissa.

Some processing of the values stored in AEXP and AARGB0:2 must be performed in order to use the above formula. The exponent is stored in a biased format, which simply means that `0x7F` has been added to the true exponent that of the number. To extract the exponent to be used in the above calculation, subtract `0x7F` from the value stored in AEXP.

The sign bit is stored in the MSB of the mantissa. To allow the full 24-bit precision of the mantissa, the MSB is assumed to be 1 explicitly, once the sign bit is stripped out. To calculate A_2, a simple binary expansion is used, as shown in the formula below. Since the MSB is explicitly 1, the expansion will always contain the term 2^0.

EQUATION 5-5:

$$A_2 = 2^0 + (\mathrm{Bit22}) \cdot 2^{-1} + (\mathrm{Bit21}) \cdot 2^{-2} + \ldots + (\mathrm{Bit0}) \cdot 2^{-23}$$

As in AN575, we will use the example of the decimal number 50.2654824574. which has a floating point representation of `0x84490FDB`, with the biased exponent being `0x84` and the mantissa (including sign bit) being `0x490FDB`. The unbiased exponent is calculated to be Exp = `0x84` - `0x7F` = `0x05`. To process the mantissa, it is first translated to binary format and the MSB is set to prepare for the expansion.

0x490FDB =

$0100\ 1001\ 0000\ 1111\ 1101\ 1011_2 \rightarrow$

$1100\ 1001\ 0000\ 1111\ 1101\ 1011_2$

The expansion is then performed according to Equation 5-5.

$A_2 = 2^0 + 2^{-1} + 2^{-4} + 2^{-7} + 2^{-12} + 2^{-13} + 2^{-14} + 2^{-15} + 2^{-16} + 2^{-17} + 2^{-19} + 2^{-20} + 2^{-22} + 2^{-23}$

$A_2 = 1.570796371$

Finally, to calculate the actual floating point number, the exponent and expanded mantissa are plugged into the conversion formula (Equation 5-4).

$A_{10} = 2^0 \bullet 1.570796371$

$A_{10} = 50.26548387$

The result of these calculations are accurate out to about 5 decimal places, with rounding and calculation errors creating some degree of uncertainty for the remaining decimal places. For more details on the sources of error, please consult AN575.

Glossary

A

Absolute Section (MPLINK Linker)

A section with a fixed (absolute) address that can not be changed by the linker.

Access RAM - PIC18CXXX Devices Only

Special general purpose registers on PIC18CXXX devices that allow access regardless of the setting of the bank select bit (BSR).

Alpha Character

Alpha characters are those characters that are letters of the arabic alphabet (a, b, …, z, A, B, …, Z).

ANSI

American National Standards Institute, which is an organization responsible for formulating and approving computer-related standards in the United States.

Alphanumeric

Alphanumeric characters are comprised of alpha characters and decimal digits (0,1, …, 9).

Application

A set of software and hardware usually designed to be a product controlled by a PICmicro® microcontroller.

ASCII

American Standard Code for Information Interchange is character set encoding using 7 binary digits to represent each character. It includes upper and lower case letters, digits, symbols and control characters.

Assembler (Assemblers)

A language tool that translates assembly source code into machine code.

Assembly Language (Assemblers)

A programming language that is once removed from machine language. Machine languages consist entirely of numbers and are difficult for humans to read and write. Assembly languages enable a programmer to use names (mnemonics) instead of numbers.

Assigned Section (MPLINK™ Linker)

A section which has been assigned to a target memory block in the linker command file. The linker allocates an assigned section into its specified target memory block.

Asynchronous Stimulus (Simulators)

Data generated to simulate external inputs to a simulator device.

B

Breakpoint – Hardware (MPLAB® ICE 2000, MPLAB ICD, MPLAB ICD 2)

An event whose execution will cause a halt.

Breakpoint – Software (Debuggers)

An address where execution of the firmware will halt. Usually achieved by a special break opcode.

Build (MPLAB® IDE v5.xx/v6.xx)

The compilation and linking of all the source files for an application.

C

C (Compilers)

A high level programming language that may be used to develop applications for microcontrollers, especially high-end device families.

Calibration Memory

A special function register or registers used to hold values for calibration of a PICmicro® microcontroller on-board RC oscillator or other device peripherals.

COFF (MPLAB ASM30, Linkers)

Common Object File Format. An object file format that contains machine code and debugging information.

Command Line Interface

Command line interface refers to executing a program on the command line with options.

Compiler (Compilers)

A language tool that translates source code into assembly code.

Configuration Bits

Special-purpose bits programmed to set PICmicro® microcontroller modes of operation. A configuration bit may or may not be preprogrammed.

Control Directives (Assemblers)

Control directives in an assembler permit code to be conditionally assembled.

Cross Reference File (Linkers)

A file that references a table of symbols and a list of files that references the symbol. If the symbol is defined, the first file listed is the location of the definition. The remaining files contain references to the symbol.

D

Data Directives (Assemblers)

Data directives are those that control the assembler's allocation of program or data memory and provide a way to refer to data items symbolically; that is, by meaningful names.

Data Memory

On a PICmicro MCU device, data memory (RAM) is comprised of General Purpose Registers (GPRs) and Special Function Registers (SFRs). Some devices also have EEPROM data memory.

Directives

Directives provide control of the language tool's operation.

Download

Download is the process of sending data from a host to another device, such as an emulator, programmer or target board.

DSC

See Digital Signal Controller.

DSP

See Digital Signal Processing.

E

EEPROM

Electrically Erasable Programmable Read Only Memory. A special type of PROM that can be erased electrically. Data is written or erased one byte at a time. EEPROM retains its contents even when power is turned off.

EPROM

Erasable Programmable Read Only Memory. A programmable read-only memory that can be erased usually by exposure to ultraviolet radiation.

Emulation (MPLAB ICE 2000)

The process of executing software loaded into emulation memory as if it were firmware residing on a microcontroller device.

Emulation Memory (MPLAB ICE 2000)

Program memory contained within the emulator.

Emulator (MPLAB ICE 2000)

Hardware that performs emulation.

Emulator System (MPLAB ICE 2000)

The MPLAB ICE 2000 emulator system includes the pod, processor module, device adapter, cables and MPLAB IDE software.

Event (MPLAB IDE v5.xx/v6.xx)

A description of a bus cycle which may include address, data, pass count, external input, cycle type (fetch, R/W) and time stamp. Events are used to describe triggers and breakpoints.

Executable Code

Software that is ready to be loaded for execution.

Export (MPLAB IDE v5.xx/v6.xx)

Send data out of the MPLAB IDE in a standardized format.

Expressions

Expressions are used in the operand field of the source line and may contain constants, symbols, or combinations of constants and/or symbols separated by arithmetic or logical operators. Each constant or symbol may be preceded by a plus or minus to indicate a positive or negative expression.

Extended Microcontroller Mode - PIC17CXXX and PIC18CXXX Devices Only

In extended microcontroller mode, on-chip program memory as well as external memory is available. Execution automatically switches to external if the program memory address is greater than the internal memory space of the PIC17CXXX or PIC18CXXX device.

External Input Line (MPLAB ICE 2000)

An external input signal logic probe line (TRIGIN) for setting an event based upon external signals.

External Label (Linkers)

A label that has external linkage.

External Linkage (Linkers)

A function or variable has external linkage if it can be referenced from outside the module in which it is defined.

External RAM - PIC17CXXX and PIC18CXXX Devices Only

Off-chip Read/Write memory.

External Symbol (Linkers)

A symbol for an identifier which has external linkage.

External Symbol Definition (Linkers)

A symbol for a function or variable defined in the current module.

External Symbol Reference (Linkers)

A symbol which references a function or variable defined outside the current module.

External Symbol Resolution (Linkers)

A process performed by the linker in which external symbol definitions from all input modules are collected in an attempt to resolve all external symbol references. Any external symbol references which do not have a corresponding definition cause a linker error to be reported.

F

File Registers

On-chip general purpose and special function registers.

Flash

A type of EEPROM where data is written or erased in blocks instead of bytes.

FNOP

Forced No Operation. A forced NOP cycle is the second cycle of a two-cycle instruction. Since the PICmicro microcontroller architecture is pipelined, it prefetches the next instruction in the physical address space while it is executing the current instruction. However, if the current instruction changes the program counter, this prefetched instruction is explicitly ignored, causing a forced NOP cycle.

G

GPR

General Purpose Register. The portion of PICmicro MCU data memory (RAM) available for general use, e.g., program-specific variables.

H

Halt (MPLAB IDE v5.xx/v6.xx)

A stop of program execution. Executing Halt is the same as stopping at a breakpoint.

HEX Code

Executable instructions assembled or compiled from source code into hexadecimal format code. HEX code is contained in a HEX file.

HEX File

An ASCII file containing hexadecimal addresses and values (HEX code) suitable for programming a device.

High Level Language (Language Tools)

A language for writing programs that is of a higher level of abstraction from the processor than assembly code. High level languages (such as C) employ a compiler to translate statements into machine instructions that the target processor can execute.

I

ICD

In-Circuit Debugger. MPLAB ICD and MPLAB ICD 2 are Microchip's in-circuit debuggers for PIC16F87X and PIC18FXXXX devices, respectively. These ICDs work with MPLAB IDE.

ICE

In-Circuit Emulator. MPLAB ICE 2000 is Microchip's in-circuit emulator that works with MPLAB IDE. PICMASTER (Obsolete product) and ICEPIC (Third Party product) are other ICE devices.

IDE

Integrated Development Environment. A software application that is used for firmware development. The MPLAB IDE integrates a project manager, an editor, language tools, debug tools, programmers and an assortment of other tools within one Windows® application. A user developing an application can write code, compile, debug and test an application without leaving the MPLAB IDE desktop.

Identifier

A function or variable name.

Import (MPLAB IDE v5.xx/v6.xx)

Bring data into the MPLAB IDE from an outside source, such as from a HEX file.

Initialized Data (Language Tools)

Data which is defined with an initial value. In C,

```
int myVar=5
```

defines a variable which will reside in an initialized data section.

Instructions (Language Tools)

A sequence of bits that tells a central processing unit to perform a particular operation and can contain data to be used in the operation.

Instruction Set (Language Tools)

The collection of machine language instructions that a particular processor understands.

Internal Linkage (Linkers)

A function or variable has internal linkage if it can not be accessed from outside the module in which it is defined.

International Organization for Standardization

An organization that sets standards in many businesses and technologies, including computing and communications.

Interrupt

An asynchronous event that suspends normal processing and temporarily diverts the flow of control through an "interrupt handler" routine.

Interrupts may be caused by both hardware (I/O, timer, machine check) and software (supervisor, system call or trap instruction).

In general the computer responds to an interrupt by storing the information about the current state of the running program; storing information to identify the source of the interrupt; and invoking a first-level interrupt handler. This is usually a kernel level privileged process that can discover the precise cause of the interrupt (e.g. if several devices share one interrupt) and what must be done to keep operating system tables (such as the process table) updated. This first-level handler may then call another handler, e.g. one associated with the particular device which generated the interrupt.

Interrupt Handler

A routine which is executed when an interrupt occurs. Interrupt handlers typically deal with low-level events in the hardware of a computer system such as a character arriving at a serial port or a tick of a real-time clock. Special case is required when writing an interrupt handler to ensure that either the interrupt which triggered the handler's execution is masked out (inhibited) until the handler is done, or the handler is written in a re-entrant fashion so that multiple concurrent invocations will not interfere with each other.

If interrupts are masked then the handler must execute as quickly as possible so that important events are not missed. This is often arranged by splitting the processing associated with the event into "upper" and "lower" halves. The lower part is the interrupt handler which masks out further interrupts as required, checks that the appropriate event has occurred (this may be necessary if several events share the same interrupt), services the interrupt, e.g. by reading a character from a UART and writing it to a queue, and re-enabling interrupts.

The upper half executes as part of a user process. It waits until the interrupt handler has run. Normally the operating system is responsible for reactivating a process which is waiting for some low-level event. It detects this by a shared flag or by inspecting a shared queue or by some other synchronization mechanism. It is important that the upper and lower halves do not interfere if an interrupt occurs during the execution of upper half code. This is usually ensured by disabling interrupts during critical sections of code such as removing a character from a queue.

Interrupt Request

The name of an input found on many processors which causes the processor to suspend normal instruction execution temporarily and to start executing an interrupt handler routine. Such an input may be either "level sensitive" - the interrupt condition will persist as long as the input is active or "edge triggered" - an interrupt is signaled by a low-to-high or high-to-low transition on the input. Some processors have several interrupt request inputs allowing different priority interrupts.

Interrupt Service Routine

User-generated code that is entered when an interrupt occurs. The location of the code in program memory will usually depend on the type of interrupt that has occurred.

IRQ

See Interrupt Request.

ISO

See International Organization for Standardization.

ISR

See Interrupt Service Routine.

L

Librarian (Librarians)

A language tool that creates and manipulates libraries.

Library (Librarians)

A library is a collection of relocatable object modules. It is created by assembling multiple source files to object files, and then using the librarian to combine the object files into one library file. A library can be linked with object modules and other libraries to create executable code.

Linker (Linkers)

A language tool that combines object files and libraries to create executable code, resolving references from one module to another.

Linker Script Files (Linkers)

Linker script files are the command files of a linker. They define linker options and describe available memory on the target platform.

Listing Directives (Assemblers)

Listing directives are those directives that control the assembler listing file format. They allow the specification of titles, pagination and other listing control.

Listing File (Assemblers)

A listing file is an ASCII text file that shows the machine code generated for each C source statement, assembly instruction, assembler directive, or macro encountered in a source file.

Logic Probes (MPLAB ICE 2000)

Up to 14 logic probes can be connected to some Microchip emulators. The logic probes provide external trace inputs, trigger output signal, +5V and a common ground.

M

Machine Code

The representation of a computer program that is actually read and interpreted by the processor. A program in machine code consists of a sequence of machine instructions (possibly interspersed with data). Instructions are binary strings. The collection of all possible instructions for a particular processor is known as its "instruction set".

Machine Language

A set of instructions for a specific central processing unit, designed to be usable by a processor without being translated. Also called machine code.

Macro (Assemblers)

A collection of assembler instructions that are included in the assembly code when the macro name is encountered in the source code. Macros must be defined before they are used; forward references to macros are not allowed.

All statements following a `MACRO` directive and prior to an `ENDM` directive are part of the macro definition. Labels used within the macro must be local to the macro so the macro can be called repetitively.

Macro Directives (Assemblers)

Directives that control the execution and data allocation within macro body definitions.

Make Project (MPLAB IDEv5.xx/v6.xx)

A command that rebuilds an application by recompiling only those source files that have changed since the last complete compilation.

MCU

Microcontroller Unit. An abbreviation for microcontroller. Also μC.

Memory Models (Compilers)

(C17): Versions of libraries and/or precompiled object files based on a device's memory (RAM/ROM) size and structure.

(C18): A description that specifies the size of pointers that point to program memory.

Microcontroller

A highly integrated chip that contains all the components comprising a controller. Typically this includes a CPU, RAM, some form of ROM, I/O ports and timers. Unlike a general-purpose computer, which also includes all of these components, a microcontroller is designed for a very specific task – to control a particular system. As a result, the parts can be simplified and reduced, which cuts down on production costs.

Microcontroller Mode - PIC17CXXX and PIC18CXXX Devices Only

One of the possible program memory configurations of the PIC17CXXX and PIC18CXXX families of microcontrollers. In microcontroller mode, only internal execution is allowed. Thus, only the on-chip program memory is available in microcontroller mode.

Microprocessor Mode - PIC17CXXX and PIC18CXXX Devices Only

One of the possible program memory configurations of the PIC17CXXX and PIC18CXXX families of microcontrollers. In microprocessor mode, the on-chip program memory is not used. The entire program memory is mapped externally.

Mnemonics

Instructions that are translated directly into machine code. Mnemonics are used to perform arithmetic and logical operations on data residing in program or data memory of a microcontroller. They can also move data in and out of registers and memory as well as change the flow of program execution. Also referred to as Opcodes.

MPASM Assembler

Microchip Technology's relocatable macro assembler. MPASM assembler is a command-line or Windows-based PC application that provides a platform for developing assembly language code for Microchip's PICmicro microcontroller (MCU) families, KeeLoq devices and Microchip memory devices. Generically, MPASM assembler will refer to the entire development platform including the macro assembler and utility functions.

MPASM assembler will translate source code into either object or executable code. The object code created by the assembler may be turned into executable code through the use of the MPLINK linker.

MPLAB C1X

Refers to both the MPLAB C17 and MPLAB C18 C compilers from Microchip. MPLAB C17 is the C compiler for PIC17CXXX devices and MPLAB C18 is the C compiler for PIC18CXXX and PIC18FXXXX devices.

MPLAB ICD and MPLAB ICD 2

Microchip's in-circuit debuggers, for PIC16F87X and PIC18FXXX devices, respectively. The ICDs work with MPLAB IDE. The main component of each ICD is the module. A complete system consists of a module, header, demo board, cables and MPLAB IDE Software.

MPLAB ICE 2000

Microchip's in-circuit emulator that works with MPLAB IDE.

MPLAB IDE

The name of the main executable program that supports the IDE.

(IDE5): MPLAB IDE v5.xx has a built-in project manager, editor and simulator (MPLAB SIM) and support for an emulator or debugger. The MPLAB IDE software resides on the PC host. The executable (`mplab.exe`) calls many other files. MPLAB IDE v5.xx and lower is a 16-bit application.

(IDE6): MPLAB IDE v6.xx has a built-in project manager, editor and support for debug and programming tools. The MPLAB IDE software resides on the PC host. The executable calls many other files. MPLAB IDE v6.xx and higher is a 32-bit application.

MPLAB SIM

Microchip's simulator that works with MPLAB IDE in support of PICmicro MCU devices.

MPLIB Object Librarian

MPLIB librarian is an object librarian for use with COFF object modules created using either MPASM assembler (mpasm or mpasmwin v2.0) or MPLAB C1X C compilers.

MPLIB librarian will combine multiple object files into one library file. Then the librarian can be used to manipulate the object files within the created library.

MPLINK Object Linker

MPLINK linker is an object linker for the Microchip MPASM assembler and the Microchip MPLAB C17 or C18 C compilers. MPLINK linker also may be used with the Microchip MPLIB librarian. MPLINK linker is designed to be used with MPLAB IDE, though it does not have to be.

MPLINK linker will combine object files and libraries to create a single executable file.

MPSIM Simulator

The DOS version of Microchip's MPLAB SIM simulator.

MRU

Most Recently Used. Refers to files and windows available to be selected from MPLAB IDE main pull down menus.

N

Nesting Depth

The maximum level to which macros can include other macros.

Node (MPLAB IDE v5.xx)

MPLAB IDE project component.

Non Real-Time

Refers to the processor at a breakpoint or executing single step instructions or MPLAB IDE being run in simulator mode.

Non-Volatile Storage

A storage device whose contents are preserved when its power is off.

NOP

No Operation. An instruction that has no effect when executed except to advance the program counter.

O

Object Code

The machine code generated by a source code language processor such as an assembler or compiler. A file of object code may be immediately executable or it may require linking with other object code files, e.g. libraries, to produce a complete executable program.

Object File

A module which may contain relocatable code or data and references to external code or data. Typically, multiple object modules are linked to form a single executable output. Special directives are required in the source code when generating an object file. The object file contains object code.

Object File Directives

Directives that are used only when creating an object file.

Off-Chip Memory - PIC17CXXX and PIC18CXXX Devices Only

Off-chip memory refers to the memory selection option for the PIC17CXXX or PIC18CXXX device where memory may reside on the target board, or where all program memory may be supplied by the Emulator. The Memory tab accessed from *Options > Development Mode* provides the Off-Chip Memory selection dialog box.

Opcodes

Operational Codes. See Mnemonics.

Operators

Arithmetic symbols, like the plus sign '+' and the minus sign '-', that are used when forming well-defined expressions. Each operator has an assigned precedence.

OTP

One Time Programmable. EPROM devices that are not in windowed packages. Since EPROM needs ultraviolet light to erase its memory, only windowed devices are erasable.

P

Pass Counter (MPLAB IDE v5.xx/v6.xx)

A counter that decrements each time an event (such as the execution of an instruction at a particular address) occurs. When the pass count value reaches zero, the event is satisfied. You can assign the Pass Counter to break and trace logic, and to any sequential event in the complex trigger dialog.

PC

Personal Computer or Program Counter.

PC Host

Any IBM® or compatible Personal Computer running Windows® 3.1x or Windows 95/98, Windows NT®, or Windows 2000.

PICmicro® MCUs

PICmicro® microcontrollers (MCUs) refers to all Microchip microcontroller families.

PICSTART Plus Programmer

A device programmer from Microchip. Programs 8, 14, 28 and 40 pin PICmicro® microcontrollers. Must be used with MPLAB® IDE Software.

Pod (MPLAB ICE 2000)

The external emulator box that contains emulation memory, trace memory, event and cycle timers and trace/breakpoint logic.

Power-on-Reset Emulation (MPLAB ICE 2000)

A software randomization process that writes random values in data RAM areas to simulate uninitialized values in RAM upon initial power application.

Pragma (Compilers)

A standardized form of comment which has meaning to a compiler. It may use a special syntax or a specific form within the normal comment syntax. A pragma usually conveys non-essential information, often intended to help the compiler to optimize the program.

Precedence

The concept that some elements of an expression are evaluated before others; (i.e., * and / before + and -). In MPASM assembler, operators of the same precedence are evaluated from left to right. Use parentheses to alter the order of evaluation.

Program Counter

A register that specifies the current execution address for emulation and simulation.

Program Memory

The memory area in a microcontroller where instructions are stored. Also, the memory in the emulator or simulator containing the downloaded target application firmware.

Programmer

A device used to program electrically programmable semiconductor devices such as microcontrollers.

Project (MPLAB IDE v5.xx/v6.xx)

A set of source files and instructions to build the object and executable code for an application.

PRO MATE II Programmer

A device programmer from Microchip. Programs all PICmicro microcontrollers and most memory and Keeloq devices. Can be used with MPLAB IDE or stand-alone.

Prototype System

A term referring to a user's target application, or target board.

PWM Signals

Pulse Width Modulation Signals. Certain PICmicro MCU devices have a PWM peripheral.

Q

Qualifier

An address or an address range used by the Pass Counter or as an event before another operation in a complex trigger.

R

Radix

The number base, HEX, or decimal, used in specifying an address and for entering data in the *Window > Modify* command.

RAM

Random Access Memory (Data Memory).

Raw Data

The binary representation of code or data associated with a section.

Real-Time

When released from the halt state in the emulator or MPLAB ICD mode, the processor runs in real-time mode and behaves exactly as the normal chip would behave. In real-time mode, the real-time trace buffer of MPLAB ICE is enabled and constantly captures all selected cycles, and all break logic is enabled. In the emulator or MPLAB ICD, the processor executes in real-time until a valid breakpoint causes a halt, or until the user halts the emulator.

In the simulator real-time simply means execution of the microcontroller instructions as fast as they can be simulated by the host CPU.

Recursion

The concept that a function or macro, having been defined, can call itself. Great care should be taken when writing recursive macros; it is easy to get caught in an infinite loop where there will be no exit from the recursion.

Relaxation

The process of converting an instruction to an identical, but smaller instruction. This is useful for saving on code size. The assembler currently knows how to RELAX a CALL instruction into an RCALL instruction. This is done when the symbol that is being called is within +/- 32k instruction words from the current instruction.

Relocatable Section (Linkers)

A section whose address is not fixed (absolute). The linker assigns addresses to relocatable sections through a process called relocation.

Relocation (Linkers)

A process performed by the linker in which absolute addresses are assigned to relocatable sections and all identifier symbol definitions within the relocatable sections are updated to their new addresses.

ROM

Read Only Memory (Program Memory).

Run

The command that releases the emulator from halt, allowing it to run the application code and change or respond to I/O in real time.

S

Section (Linkers)

An portion of code or data which has a name, size and address.

SFR

See Special Function Registers.

Shared Section (MPLINK Linker)

A section which resides in a shared (non-banked) region of data RAM.

Shell (MPASM Assembler)

The MPASM assembler shell is a prompted input interface to the macro assembler. There are two MPASM assembler shells: one for the DOS version and one for the Windows® version.

Simulator

A software program that models the operation of PICmicro microcontrollers.

Single Step (MPLAB IDE v5.xx/v6.xx)

This command steps though code, one instruction at a time. After each instruction, MPLAB IDE updates register windows, watch variables and status displays so you can analyze and debug instruction execution.

You can also single step C compiler source code, but instead of executing single instructions, MPLAB IDE will execute all assembly level instructions generated by the line of the high level C statement.

Skew (MPLAB ICE 2000)

The information associated with the execution of an instruction appears on the processor bus at different times. For example, the executed Opcodes appears on the bus as a fetch during the execution of the previous instruction, the source data address and value and the destination data address appear when the Opcodes is actually executed, and the destination data value appears when the next instruction is executed. The trace buffer captures the information that is on the bus at one instance. Therefore, one trace buffer entry will contain execution information for three instructions. The number of captured cycles from one piece of information to another for a single instruction execution is referred to as the skew.

Skid (MPLAB ICE 2000, MPLAB ICD, MPLAB ICD 2))

When a hardware breakpoint is used to halt the processor, one or more additional instructions may be executed before the processor halts. The number of extra instructions executed after the intended breakpoint is referred to as the skid.

Source Code - Assembly

Source code consists of PICmicro MCU instructions and MPASM assembler directives and macros that will be translated into machine code by an assembler.

Source Code - C

A program written in the high level language called “C” which will be converted into PICmicro MCU machine code by a compiler. Machine code is suitable for use by a PICmicro MCU or Microchip development system product like MPLAB IDE.

Source File - Assembly

The ASCII text file of PICmicro MCU instructions and MPASM assembler directives and macros (source code) that will be translated into machine code by an assembler. It is an ASCII file that can be created using any ASCII text editor.

Source File - C

The ASCII text file containing C source code that will be translated into machine code by a compiler. It is an ASCII file that can be created using any ASCII text editor.

Special Function Registers

Registers that control I/O processor functions, I/O status, timers, or other modes or peripherals.

Stack - Hardware

An area in PICmicro MCU memory where function arguments, return values, local variables and return addresses are stored; (i.e., a “Push-Down” list of calling routines). Each time a PICmicro MCU executes a `CALL` or responds to an interrupt, the software pushes the return address to the stack. A return command pops the address from the stack and puts it in the program counter.

The PIC18CXXX family also has a hardware stack to store register values for “fast” interrupts.

Stack - Software

The compiler uses a software stack for storing local variables and for passing arguments to and returning values from functions.

Static RAM or SRAM

Static Random Access Memory. Program memory you can Read/Write on the target board that does not need refreshing frequently.

Status Bar (MPLAB IDE v5.xx/v6.xx)

The Status Bar is located on the bottom of the MPLAB IDE window and indicates such current information as cursor position, development mode and device and active tool bar.

Step Into (MPLAB IDE v5.xx/v6.xx)

This command is the same as Single Step. Step Into (as opposed to Step Over) follows a CALL instruction into a subroutine.

Step Over (MPLAB IDE v5.xx/v6.xx)

Step Over allows you to debug code without stepping into subroutines. When stepping over a CALL instruction, the next breakpoint will be set at the instruction after the CALL. If for some reason the subroutine gets into an endless loop or does not return properly, the next breakpoint will never be reached.

The Step Over command is the same as Single Step except for its handling of CALL instructions.

Stimulus (Simulators)

Input to the simulator, i.e., data generated to exercise the response of simulation to external signals. Often the data is put into the form of a list of actions in a text file. Stimulus may be asynchronous, synchronous (pin), clocked and register.

Stopwatch (Simulators)

A counter for measuring execution cycles.

Symbol (MPLAB IDE v5.xx/v6.xx)

A symbol is a general purpose mechanism for describing the various pieces which comprise a program. These pieces include function names, variable names, section names, file names, struct/enum/union tag names, etc.

Symbols in MPLAB IDE refer mainly to variable names, function names and assembly labels.

System Button

The system button is another name for the system window control. Clicking on the system button pops up the system menu.

System Window Control

The system window control is located in the upper left corner of windows and some dialogs. Clicking on this control usually pops up a menu that has the items “Minimize,” “Maximize,” and “Close.” In some MPLAB IDE windows, additional modes or functions can be found.

T

Target (MPLAB ICE 2000, MPLAB ICD, MPLAB ICD 2)

Refers to user hardware.

Target Application (MPLAB ICE 2000, MPLAB ICD, MPLAB ICD 2)

Firmware residing on the target board.

Target Board (MPLAB ICE 2000, MPLAB ICD, MPLAB ICD 2)

The circuitry and programmable device that makes up the target application.

Target Processor (MPLAB ICE 2000, MPLAB ICD, MPLAB ICD 2)

The microcontroller device on the target application board.

Template (Editor)

Lines of text that you build for inserting into your files at a later time. The MPLAB Editor stores templates in template files.

Tool Bar (MPLAB IDE v5.xx/v6.xx)

A row or column of icons that you can click on to execute MPLAB IDE functions.

Trace (Debuggers)

An emulator or simulator function that logs program execution. The emulator logs program execution into its trace buffer which is uploaded to MPLAB IDE's trace window.

Trace Memory (Debuggers)

Trace memory contained within the emulator. Trace memory is sometimes called the trace buffer.

Trigger Output (MPLAB ICE 2000)

Trigger output refers to an emulator output signal that can be generated at any address or address range, and is independent of the trace and breakpoint settings. Any number of trigger output points can be set.

Trigraphs (Compilers)

These are three-character sequences, all starting with `??`, that are defined by ISO C to stand for single characters

The nine trigraphs and their replacements are

Trigraph:	??(	??)	??<	??>	??=	??/	??'	??!	??-
Replacement:	[	]	{	}	#	\	^	|	~

U

Unassigned Section (MPLINK Linker)

A section which has not been assigned to a specific target memory block in the linker command file. The linker must find a target memory block in which to allocate an unassigned section.

Uninitialized Data

Data which is defined without an initial value. In C,

```
int myVar;
```

defines a variable which will reside in an uninitialized data section.

Upload

The Upload function transfers data from a tool, such as an emulator or programmer, to the host PC or from the target board to the emulator.

W

Warning

An alert that is provided to warn you of a situation that would cause physical damage to a device, software file, or equipment.

Watchdog Timer (WDT)

A timer on a PICmicro microcontroller that resets the processor after a selectable length of time. The WDT is enabled or disabled and set up using configuration bits.

Watch Variable (MPLAB IDE v5.xx/v6.xx)

A variable that you may monitor during a debugging session in a watch window.

Watch Window (MPLAB IDE v5.xx/v6.xx)

Watch windows contain a list of watch variables that are updated at each breakpoint.

Index

NOTES:

NOTES:

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200 Fax: 480-792-7277
Technical Support: 480-792-7627
Web Address: http://www.microchip.com

Rocky Mountain
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7966 Fax: 480-792-4338

Atlanta
3780 Mansell Road, Suite 130
Alpharetta, GA 30022
Tel: 770-640-0034 Fax: 770-640-0307

Boston
2 Lan Drive, Suite 120
Westford, MA 01886
Tel: 978-692-3848 Fax: 978-692-3821

Chicago
333 Pierce Road, Suite 180
Itasca, IL 60143
Tel: 630-285-0071 Fax: 630-285-0075

Dallas
4570 Westgrove Drive, Suite 160
Addison, TX 75001
Tel: 972-818-7423 Fax: 972-818-2924

Detroit
Tri-Atria Office Building
32255 Northwestern Highway, Suite 190
Farmington Hills, MI 48334
Tel: 248-538-2250 Fax: 248-538-2260

Kokomo
2767 S. Albright Road
Kokomo, Indiana 46902
Tel: 765-864-8360 Fax: 765-864-8387

Los Angeles
18201 Von Karman, Suite 1090
Irvine, CA 92612
Tel: 949-263-1888 Fax: 949-263-1338

San Jose
Microchip Technology Inc.
2107 North First Street, Suite 590
San Jose, CA 95131
Tel: 408-436-7950 Fax: 408-436-7955

Toronto
6285 Northam Drive, Suite 108
Mississauga, Ontario L4V 1X5, Canada
Tel: 905-673-0699 Fax: 905-673-6509

ASIA/PACIFIC

Australia
Microchip Technology Australia Pty Ltd
Suite 22, 41 Rawson Street
Epping 2121, NSW
Australia
Tel: 61-2-9868-6733 Fax: 61-2-9868-6755

China - Beijing
Microchip Technology Consulting (Shanghai)
Co., Ltd., Beijing Liaison Office
Unit 915
Bei Hai Wan Tai Bldg.
No. 6 Chaoyangmen Beidajie
Beijing, 100027, No. China
Tel: 86-10-85282100 Fax: 86-10-85282104

China - Chengdu
Microchip Technology Consulting (Shanghai)
Co., Ltd., Chengdu Liaison Office
Rm. 2401-2402, 24th Floor,
Ming Xing Financial Tower
No. 88 TIDU Street
Chengdu 610016, China
Tel: 86-28-86766200 Fax: 86-28-86766599

China - Fuzhou
Microchip Technology Consulting (Shanghai)
Co., Ltd., Fuzhou Liaison Office
Unit 28F, World Trade Plaza
No. 71 Wusi Road
Fuzhou 350001, China
Tel: 86-591-7503506 Fax: 86-591-7503521

China - Hong Kong SAR
Microchip Technology Hongkong Ltd.
Unit 901-6, Tower 2, Metroplaza
223 Hing Fong Road
Kwai Fong, N.T., Hong Kong
Tel: 852-2401-1200 Fax: 852-2401-3431

China - Shanghai
Microchip Technology Consulting (Shanghai)
Co., Ltd.
Room 701, Bldg. B
Far East International Plaza
No. 317 Xian Xia Road
Shanghai, 200051
Tel: 86-21-6275-5700 Fax: 86-21-6275-5060

China - Shenzhen
Microchip Technology Consulting (Shanghai)
Co., Ltd., Shenzhen Liaison Office
Rm. 1812, 18/F, Building A, United Plaza
No. 5022 Binhe Road, Futian District
Shenzhen 518033, China
Tel: 86-755-82901380 Fax: 86-755-82966626

China - Qingdao
Rm. B503, Fullhope Plaza,
No. 12 Hong Kong Central Rd.
Qingdao 266071, China
Tel: 86-532-5027355 Fax: 86-532-5027205

India
Microchip Technology Inc.
India Liaison Office
Divyasree Chambers
1 Floor, Wing A (A3/A4)
No. 11, O'Shaugnessey Road
Bangalore, 560 025, India
Tel: 91-80-2290061 Fax: 91-80-2290062

Japan
Microchip Technology Japan K.K.
Benex S-1 6F
3-18-20, Shinyokohama
Kohoku-Ku, Yokohama-shi
Kanagawa, 222-0033, 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 135-882
Tel: 82-2-554-7200 Fax: 82-2-558-5934

Singapore
Microchip Technology Singapore Pte Ltd.
200 Middle Road
#07-02 Prime Centre
Singapore, 188980
Tel: 65-6334-8870 Fax: 65-6334-8850

Taiwan
Microchip Technology (Barbados) Inc.,
Taiwan Branch
11F-3, No. 207
Tung Hua North Road
Taipei, 105, Taiwan
Tel: 886-2-2717-7175 Fax: 886-2-2545-0139

EUROPE

Austria
Microchip Technology Austria GmbH
Durisolstrasse 2
A-4600 Wels
Austria
Tel: 43-7242-2244-399
Fax: 43-7242-2244-393

Denmark
Microchip Technology Nordic ApS
Regus Business Centre
Lautrup hoj 1-3
Ballerup DK-2750 Denmark
Tel: 45 4420 9895 Fax: 45 4420 9910

France
Microchip Technology SARL
Parc d'Activite du Moulin de Massy
43 Rue du Saule Trapu
Batiment A - ler Etage
91300 Massy, France
Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79

Germany
Microchip Technology GmbH
Steinheilstrasse 10
D-85737 Ismaning, Germany
Tel: 49-89-627-144 0 Fax: 49-89-627-144-44

Italy
Microchip Technology SRL
Centro Direzionale Colleoni
Palazzo Taurus 1 V. Le Colleoni 1
20041 Agrate Brianza
Milan, Italy
Tel: 39-039-65791-1 Fax: 39-039-6899883

United Kingdom
Microchip Ltd.
505 Eskdale Road
Winnersh Triangle
Wokingham
Berkshire, England RG41 5TU
Tel: 44 118 921 5869 Fax: 44-118 921-5820

12/05/02