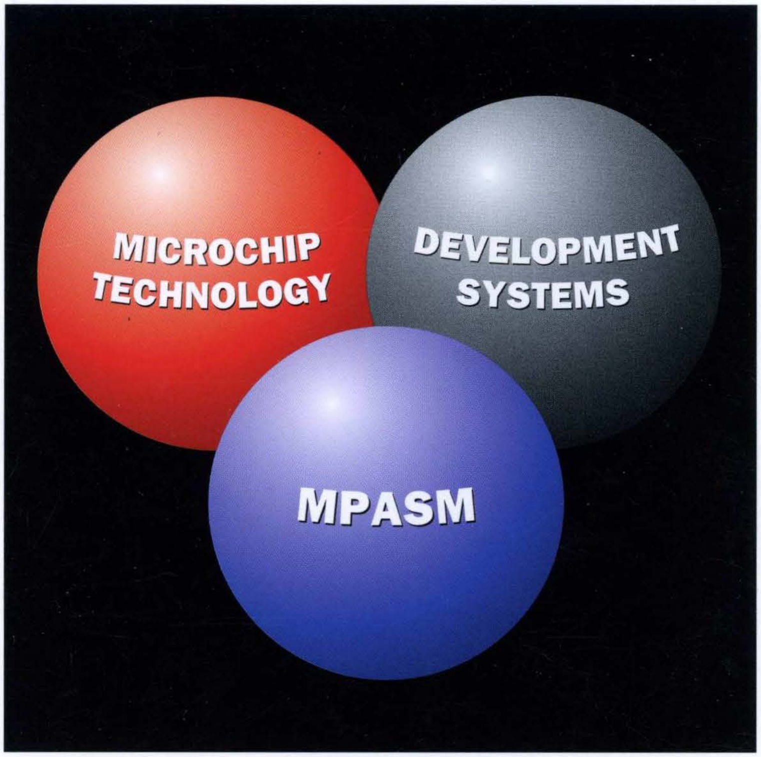
MICROCHIP MPASM USER'S GUIDE

MPASM
ASSEMBLER



USER'S GUIDE

MICROCHIP

"Information contained in this publication regarding device applications and the like is intended by way of suggestion only. No representation or warranty is given and no liability is assumed by Microchip Technology Incorporated with respect to the accuracy or use of such information. Use of Microchip's products as critical components in life support systems is not authorized except with express written approval by Microchip. The Microchip logo and name are trademarks of Microchip Technology Incorporated. All rights reserved. All other trademarks mentioned herein are the property of their respective companies".

Table of Contents

Table of Contents

Table of Contents

TABLE OF EXAMPLES

TABLE OF TABLES

Preface

Welcome

Microchip Technology Incorporated is committed to providing useful and innovative solutions to your microcontroller designs. MPASM is the first Universal Assembler available for Microchip's entire product line of microcontrollers. MPASM will generate solid code with a directive language rich in potential.

Feature List and Product Information

MPASM provides a universal solution for developing assembly code for the PIC16C5X, PIC16CXX, PIC17CXX, and future microcontroller offerings. Notable features include:

- PIC16C5X, PIC16CXX, and PIC17CXX Instruction Set
- Command Line Interface
- Command Shell Interface
- Rich Directive Language
- Flexible Macro Language
- PICMASTER™ Compatibility
- MPSIM Compatibility

Use of the Microchip MPASM Universal Assembler requires an IBM PC/AT® or compatible computer, running MS-DOS® V4.1 or greater.

Migration Path

Users of MPALC and ASM17 will find that much of their existing code will assemble with MPASM with little or no editing. This provides a simple migration path. Appendices C and D describe some of the simple changes you may need to implement to assemble existing code.

But more importantly, because the program is universal, an application developed for the PIC16C54 can be easily translated into a program for the PIC16C71. This would simply require changing the instruction mnemonics that are not the same between the machines (assuming that register and peripheral usage were similar). The rest of the directive and macro language will be the same.

MPASM was developed in conjunction with Byte Craft Limited, recognized as a world leader in microcontroller language tools.

Chapter 1. Introduction

Product Definition

MPASM is a DOS based PC application that provides a platform for developing assembly language code for Microchip microcontrollers including the PIC16C5X, PIC16CXX and PIC17CXX families. Generically, MPASM will refer to the entire development platform including the macro assembler and utility functions. Specifically:

MPASM - refers to the macro assembler that generates relocatable object code from assembly source code.

MPLINK - refers to the linker that translates relocatable objects to executable binary code at absolute memory locations.

MPLIB - refers to the librarian utility that allows relocatable objects to be grouped together in one file, or library, for convenience. These libraries can be referenced via MPLINK as an object file output from MPASM.

Documentation Layout

The documentation is intended to describe how to use the assembler, and its environment. It also provides some basic information about specific Microchip microcontrollers and their instruction sets, but detailed discussion of these issues is deferred to the data sheets for specific microcontrollers. In particular:

Chapter 1: Introduction - Introduces the user to MPASM. It describes the User's Guide layout, general conventions and terms, as well as a brief discussion of installation, and platform requirements.

Chapter 2: Environment and Usage - This chapter describes the assembler's Command Line Interface (CLI), and shell interface. Also discussed here are the files used by MPASM, both input and output, including object file formats.

Chapter 3: Directive Language - This chapter describes native directive language of MPASM. This language should be familiar to previous users of either Microchip or Byte Craft development system.

Chapter 4: Macro Language - This chapter describes the macro language of MPASM. Macros are best learned by example; several will be offered for consideration.

Chapter 5: Expression Syntax and Operation - This chapter describes the expression syntax of MPASM, including operator precedence, radix override notation, examples and discussion.

Chapter 6: MPLIB - MPASM Librarian - This chapter describes the purpose, use and CLI of the librarian utility provided as part of the MPASM development environment. It will also provide examples and discussion.

Chapter 7: MPLINK - MPASM Linker - This chapter describes the purpose, use and CLI of the linker utility provided with MPASM. It also describes the script language provided to specify where relocated objects should be placed in memory.

This document offers the following General Reference sections:

Appendix A: Object code formats, a brief overview.

Appendix B: Customer Support - Provides information about accessing the Microchip Bulletin Board for the latest revisions of products, user forums and non-urgent questions about applying Microchip products.

Appendix C: MPALC Conversion Guide. A short description designed to assist users of MPALC to move their code to MPASM.

Appendix D: ASM17 Conversion Guide. A short description designed to assist users of ASM17 to move their code to MPASM.

Appendix E: Error Messages. A list of the error messages generated by MPASM, with descriptions.

Index: A keyword cross reference to important topics and keywords.

Quick Reference Guide: This section provides a quick reference to the instruction set for each family of microprocessors as well as quick references to the directive and macro language, possibly for "tear-out."

TABLE 1: DOCUMENTATION CONVENTIONS

Character	Represents
Square Brackets ([])	Optional Arguments
Angle Brackets (<>)	Delimiters for special keys: <TAB>, <ESC>, or additional options.
Pipe Character (I)	Choice of mutually exclusive arguments; an OR selection.
Lowercase characters	Type of data
Italic characters	A variable argument; it can be either a type of data (in lowercase characters or a specific example (in uppercase characters)
`Courier Font`	User entered code or sample code.

Terms

In order to provide a common frame of reference, the following terms are defined:

PIC16/17

PIC16/17 refers to any Microchip microcontroller, including the representatives of the PIC16C5X, PIC16CXX, and PIC17CXX families.

Source Code

This is the file of PIC16/17 instructions and MPASM directives and macros that will be translated into executable code. This code is suitable for use by a PIC16/17 or Microchip development system product like an emulator, a simulator or a programmer. It is an ASCII file that can be created using any ASCII text editor.

Assemble

The act of executing the MPASM macro assembler to translate source code to relocatable object code.

Mnemonics

These are instructions that are translated directly into machine code. These are used to perform arithmetic and logical operations on data residing program or data memory of a PIC16/17. They also have the ability to move data in and out of registers and memory as well as conditionally branch to specified program addresses.

Directives

Directives provide control of the assembler's operation by telling MPASM how to treat mnemonics, data references and format the listing file. Directives make coding easier and provide custom output according to specific needs.

Macro

A macro consists of a sequence of assembler commands. Passing arguments to a macro allows for flexible functionality.

Relocatable Object

A unit of intermediate code that may not have an absolute base address in PIC16/17 memory. This base address may be assigned at link time.

Linking

Linking, or to link, refers to the translation of relocatable objects to machine code suitable for execution by a PIC16/17. Absolute addresses may be assigned to relocated objects at this time.

Listing

A listing is an ASCII text file that shows the machine code generated for each assembly instruction, MPASM directive, or macro encountered in a source file. An absolute listing file shows the collection of relocated objects, together with their absolute addresses in PIC16/17 memory (relative addresses will be shown in listings output directly from the macro assembler).

PC

Any IBM or compatible Personal Computer.

DOS

Disk Operating System that provides the basis for most applications that run on PCs.

Chapter 1: Introduction

Recommended Reading

This manual is intended to provide a reference to using the MPASM development environment. It is not intended to replace reference material regarding specific PIC16/17 microcontrollers. Therefore, you are urged to read the Data Sheets for the PIC16/17 specified by your application.

If this is your first microcontroller application, you are encouraged to review the Microchip "Embedded Control Handbook." You will find a wealth of information about applying PIC16/17s. The application notes described within are available from the Microchip BBS (see Appendix B).

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

System Requirements

MPASM will run on any PC/AT or compatible computer, running DOS V4.1 or greater. The distribution is provided on 3.5", double density (720k) floppy diskettes.

No special display or ancillary devices are required.

Warranty Registration

NOTE: Upon receiving the diskette you should complete and return the Warranty Registration Card enclosed with the disk, and mail it promptly. Doing so will help to ensure that you receive product updates and notification of interim releases that become available.

Installation

Never use the original diskette as your working copy. Make a backup copy of the MPASM distribution disk using the DOS "DISKCOPY" program, then label the new copy and store the original in a safe place.

It is recommended that you execute MPASM from your hard disk. To do this, create a new directory (MKDIR) for the assembler and copy all files from the backup distribution diskette to that directory (MPASM, and its accompanying utility programs and source examples are distributed at the root level of the distribution diskette).

If you want to be able to run MPASM from any directory (without fully qualifying the path to the executable program), you must add the new directory to the DOS PATH environment variable.

For information on using DISKCOPY or any DOS command, and DOS environment variables, refer to your IBM DOS User's Guide.

Compatibility Issues

MPASM is compatible with all Microchip development systems currently in production. This includes MPSIM (PIC16/17 software core simulator), PICMASTER, PRO MATE™ (the Microchip Universal Programmer), and the Microchip low-cost development programmer.

It is not compatible with certain older Microchip In-Circuit Emulators.

Microchip Technology is sensitive to your investment in PIC16/17 firmware. Whenever possible, we endeavor to protect that investment and remain backward compatible as new products are developed and released.

MPASM is intended to be backward compatible with source code developed with MPALC and ASM17. In order to provide the largest coverage in backward compatibility, you may encounter small discrepancies in the directive and macro syntax. Whenever practical, MPASM will attempt to make a rational decision as to your coding intent, and flag older syntax as warnings. Unfortunately, this is not always possible.

Specifically, MPASM no longer supports the "." (dot) directives within the macro language. This is offset by the fact that almost all directives can be executed within or without a macro.

As with any software product, users may have questions about the MPASM Assembler. These questions can be posed to Microchip in the following ways:

- Contact your regional sales office. The locations, phone and fax numbers are listed at the end of this manual.
- Contact your local distributor or representative.
- Connect worldwide to the Microchip BBS using the CompuServe® communications network. In most cases a local call is your only expense. The Microchip BBS connection does not use CompuServe membership services, therefore **you do not need CompuServe membership to use the Microchip BBS.**

  The procedure to connect will vary slightly from country to country. Please check with your local CompuServe agent for details if you have a problem. CompuServe services allow multiple users at baud rates up to 9600.
- Contact the factory Applications Group.

Chapter 2. Environment and Usage

Introduction

MPASM provides a universal platform for developing code for PIC16/17s. The product is represented by several programs: MPASM, MPLINK, and MPLIB. Each of these programs has its own Command Line Interface; the former two can be accessed through the MPASM shell while MPLIB can only be accessed through its CLI. This chapter is dedicated to describing the MPASM CLI and the MPASM shell.

Highlights

The points that will be highlighted in this chapter are:

- MPASM Command Line Interface
- MPASM Shell Interface
- MPASM Input Files
- MPASM and Associated Output Files

Terms

Command Line Interface

Command Line Interface or CLI refers to executing a program with options. In the case of MPASM, executing MPASM with any command line options or just the file name will invoke the assembler. In the absence of any command line options, a prompted input interface (shell) will be executed.

Shell

The MPASM shell is a prompted input interface to the macro assembler and linker. It is a DOS Text Graphics screen where the user fills in the appropriate assembly and linker options.

Alpha Character

Alpha characters are those characters, regardless of case, that are normally contained in the alphabet: (a, b, ..., z, A, B, ..., Z).

Alpha Numeric

Alpha Numeric characters include Alpha characters and numbers: (0, 1, ..., 9).

Command Line Interface

MPASM can be invoked through the CLI as follows:

```
MPASM [/<Option>[,/<Option>...]] <file_name>
```

Where

`/<Option>` - refers to one of the command line options

`<file_name>` - is the file being assembled

For example, if test.asm exists in the current directory, it can be assembled with following command:

```
MPASM /e /l test
```

The assembler defaults (noted in Table 2) can be overriden with options supplied to the CLI:

- / <option>   enables the option
- /<option>+   enables the option
- /<option>-   disables the option

TABLE 2: ASSEMBLER COMMAND LINE OPTIONS

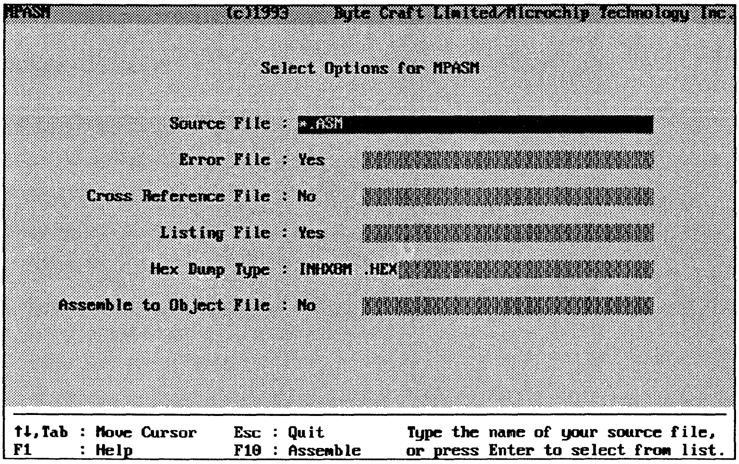
Option	Default	Description
?	N/A	Displays the MPASM Help Panel
c	On	Enables/Disables case sensitivity
e	On	Enable/Disable Error File
h	N/A	Displays the MPASM Help Panel
l	On	Enables/Disables the listing file generated from the macro assembler. This would include relative addresses in the case of relocatable objects.
m	Off	Enables/Disable macro expansion

o	N/A	Sets the path for object files
		`/o<path>\object.file`
		where `<path>` describes the output directory, and `object.file` to be created. For example:
		`/Oc:\temp\file.obj`

p	None	Set the processor type: `/p<processor_type>` Where `<processor_type>` is one of [PIC16C54 I PIC16C55 I PIC16C56 I PIC16C57 I PIC16C71 I PIC16C84 I PIC17C42 I PIC16C58 I PIC16C64] .
q	Off	Enable/Disable quiet mode (suppress screen output)
r	Hex	Defines default radix: `/r<radix>` where `<radix>` is one of `[ HEX | DEC | OCT ]`
x	Off	Enable/Disable cross reference in listing file.
a	INHX8M	Generate absolute .COD and hex output directly from assembler: `/a<hex-format>` where `<hex-format>` is one of [INHX8M I INHX8S I INHX32]

Shell Interface

The MPASM Shell interface displays a screen in Text Graphics mode. On this screen, you can fill in the name of the source file you want to assemble and other information.



```
MPASM              (c)1993     Byte Craft Limited/Microchip Technology Inc.

                        Select Options for MPASM

                  Source File : *.ASM
                   Error File : Yes
         Cross Reference File : No
                 Listing File : Yes
                Hex Dump Type : INHX8M  .HEX
      Assemble to Object File : No

↑↓,Tab : Move Cursor     Esc : Quit        Type the name of your source file,
F1     : Help            F10 : Assemble    or press Enter to select from list.
```

Source File

Type the name of your source file. The name can include a DOS path and wild cards. If you use wild cards (one of * or ?), a list of all matching files is displayed for you to select from. A binary code file (<sourcename>.COD) is automatically created.

Error File

An error file (<sourcename>.ERR) is created by default. To tum the error field off, use the <↓> to move to the YES and press **<RET>** to change it to NO. The error file name can be changed by pressing the **<TAB>** key to move to the shaded area and typing a new name. Wild cards are not allowed.

Cross Reference File

Modify this field as for the Error File. It is used to optionally create a cross reference file (<sourcename>.XRF). The name may be modified as for Error File and again, wild cards are not allowed.

Listing File

Modify this field as for the Error File. It is used to optionally disable the listing file. This may be a relative or absolute listing file, depending on whether or not the Linker is invoked. The output file name may be modified as for the Error file.

HEX Dump Type

Set this value to generate the desired output format from the Linker. Changing this value is accomplished by moving to the field with the <↓> key and pressing the **<RET>** key to scroll through the available options. To change the HEX file name press the <**TAB**> key to move the shaded area, and type in the new name.

Assemble to Object File

Changing this option will generate the relocatable object code that can be input to the linker. It is modified as for the Error File. Turning it off will have the effect of generating no object file at all.

Source Code Formats

Code written for previous Microchip assemblers (MPALC and ASM17) need not be rewritten according to these standards.

The source code file is created using any ASCII Text File editor (the editor included with the PICMASTER Source Level Debugger was designed for this purpose). It should conform to the following basic guidelines.

Each line of the source file may contain up to four types of information:

- labels
- mnenonics
- operands
- comments

The order and position of these are important. Labels must start in column one. Mnemonics may start in column two or beyond. Operands follow the mnemonic. Comments may follow the operands, mnenonics or labels, or can start in any column if the first non space character is either an asterisk (*) or a semi-colon (;). The maximum column width is 255 characters.

One or more spaces must separate the label and the mnemonic, or the mnemonic and the operand(s). Operands may be separated by a comma. For example:

EXAMPLE 1: SAMPLE MPASM SOURCE CODE

```
;
; Sample MPASM Source Code.  It is for illustration only.
;
        list    p=16C54,r=HEX

        org     0x1ff               ; Reset Vector
        goto    Start               ; Go back to the begin-
ning

        org     0x000               ; The main line code
starts here

Start
        movlw   0x0a        ; Perform some PIC16/17 code
        movlw   0x0b        ;
        goto    Start               ; do it forever...

        end
```

Labels

All labels must start in column 1. It may be followed by a colon (:), space, tab or the end of line. Comments may also start in column 1 if one of the valid comment denotations is used.

Labels must begin with an alpha character or an under bar (_) and may thereafter contain alpha numeric characters and the under bar and the question mark.

Labels may be up to 31 characters long. By default they are case sensitive, but case sensitivity may be overridden by command line or directive options. If a colon is used when defining a label it is treated as a label operator and not part of the label itself.

Mnemonics

Assembler instruction mnemonics, assembler directives and macro calls must begin in at least column 2. If there is a label on the same line, they must be separated from that label by a colon or by one or more spaces or tabs.

Operands

Operands must be separated from mnemonics by one or more spaces or tabs. Operand lists must be separated by commas. If the operand requires a fixed number of operands, anything on the line after the operands is ignored. Comments are allowed at the end of the line. If the mnemonics permits a variable number of operands, the end of the operand list is determined by the end of the line or the comment.

Comments

Comments which are on a line by themselves must start with either of the comment characters (* or ;). Comments at the end of a source line must be separated from the rest of the line by one or more spaces or tabs. Anything encountered on the line following the comment character is ignored until the end of line.

Files Used by MPASM and Utility Functions

There are a number of default file extensions used by MPASM and the associated utility functions.

TABLE 3: MPASM DEFAULT FILE EXTENSIONS

Extension	Purpose
.ASM	Default source code file extension input to MPASM: `<source_name>.ASM`
.OBJ	Default output extension for relocatable objects from MPASM: `<source_name>.OBJ`
.LST	Default output extension for listing files generated from either the assembler or MPASM or MPLINK: `<source_name>.LST`
.ERR	Default output extension from MPASM for specific error files: `<source_name>.ERR`
.MAP	Default output extension from MPLINK for map output: `<source_name>.MAP`
.HEX	Default output extension from MPASM or MPLINK for Intel Hex object code (see Appendix A) `<source_name).HEX`
.HXL/.HXH	Default output extensions from MPASM or MPLINK for separate low byte and high byte Intel Hex format files: `<source_name>.HXL, <source_name>.HXH`
.LIB	Default extension for library files created by MPLIB, and referenced by MPLINK: `<source_name>.LIB`
.LNK	Default extension for linker script files: `<source_name>.LNK`
.COD	Default output extension for the symbol and debug file. This file may be output from MPASM or MPLINK: `<source_name>.COD`

Object Code Formats

MPLINK and MPASM (with absolute addresses) are capable of producing a number of different output formats. See Appendix A.

Listing File Format

Sample MPASM Listing File (.LST)

```
MPASM 00.00.64 Beta                        10-22-1993  13:21:21            PAGE  1


LOC  OBJECT CODE      LINE SOURCE TEXT

                      0001 ;
                      0002 ; Sample MPASM Source Code.  It is for illustration only.
                      0003 ;
                      0004           list      p=16c54,r=HEX
                      0005           org       0x1ff          ; Reset Vector
01FF 0A00             0006           goto      Start          ; Go back to the beginning
                      0007
                      0008           org       0x000          ; The main line code starts
                      0009
0000 0C0A             0010 Start     movlw     0x0a           ; Perform some PIC16/17 code
0001 0C0B             0011           movlw     0x0b           ;
0002 0A00             0012           goto      Start          ; do it forever...
                      0013
                      0014
                      0015           end
                      0016

MPASM 00.00.64 Beta                        10-22-1993  13:21:21            PAGE  2
SYMBOL TABLE
LABEL                     VALUE
Start                     0000
MEMORY USAGE MAP ('X' = Used,  '-' = Unused)

0000 : XXX------------- ---------------- ---------------- ----------------

0040 : ---------------- ---------------- ---------------- ----------------

0180 : ---------------- ---------------- ---------------- ----------------

01C0 : ---------------- ---------------- ---------------- ---------------X

All other memory blocks unused.


Errors   :     0
Warnings :     0
```

The listing file format produced by MPASM is straight forward:

The product name and version, the assembly date and time, and the page number appear at the top of every page.

The first column of numbers, four characters wide, contains the base address in memory where the code will be placed. The second column, also four characters wide, is reserved for the machine instruction. This is the code that will be executed by the PIC16/17. The third column lists the associated source file line number for this line. The remainder of the line is reserved for the source code line that generated the machine code.

The symbol table lists all symbols in the program, and where they are defined. The memory usage map gives a graphical representation of memory usage. 'X' marks a used location and '-' marks memory that is not used by this object.

Error File Format (.ERR)

MPASM can generate an error file by supplying the /e option. This file can be used to provide useful information when debugging your code. The file name is followed by the line number of the offending line. A description of the error encountered follows. (The PICMASTER Source Level Debugger will automatically open this file in the case of an error). The error file looks like this:

```
Error EXAMPLE.ASM 7 :Undefined argument (start in get arg)
```

Appendix E alphabetically describes the error messages generated by MPASM.

Chapter 3. Directive Language

Introduction

This chapter describes the MPASM directive language.

Directives are assembler commands that appear in the source code but are not translated directly into opcodes. They are used to control the assembler: its input, output, and data allocation.

Many of the assembler directives have alternate names and formats. These may exist to provide *backward* compatibility with previous assemblers from Microchip and to be compatible with individual programming practices. If portable code is desired, it is recommended that programs be written using the specifications contained within this document.

There are four basic types of directives provided by MPASM.

Highlights

The points that will be highlighted in this chapter are:

- Data Directives
- Listing Directives
- Control Directives
- Macro Directives

Terms

Data Directives

Data Directives are those that control the allocation of memory and provide a way to refer to data items symbolically, that is, by meaningful names.

Listing Directives

Listing Directives are those directives that control the MPASM listing file and format. They allow the specification of titles, page ejects and other listing control.

Control Directives

Control directives permit sections of conditionally assembled code.

Macro Directives

These directives control the execution and data allocation within macro body definitions.

TABLE 4: DIRECTIVE SUMMARY

Directive	Description	Syntax
CBANK	Future Feature	
CBLOCK	Define a Block of Constants	cblock [<expr>]
CONSTANT	Declare Symbol Constant	constant <label>[=<expr>, ...,<label>[=<expr>]]
DATA	Create Numeric and Text Data	data <expr>,[,<expr>,...,<expr>] DATA "text_string>[,"<text_string>",...]
DB	Declare Data of One Byte	db <expr>[,<expr>,...,<expr>]
#DEFINE	Define a Text Substitution Label	define <name> [<value>} define <name> [<arg>,...,<arg>] <value>
DW	Declare Data of One Word	dw <expr>[,<expr>,...,<expr>]
ELSE	Begin Alternative Assembly Block to IF	else
END	End Program Block	end
ENDC	End an Automatic Constant Block	endc
ENDIF	End conditional Assembly Block	endif
ENDM	End a Macro Definition	endm
ENDW	End a While Loop	endw
EQU	Define an Assembly Constant	<label> equ <expr>
ERROR	Issue an Error Message	error "<text_string>"
EXITM	Exit from a Macro	exitm
EXPAND	Expand Macro Listing	expand
FILL	Specify Memory Fill Value	fill <expr>
IF	Begin ConditionallyAssembled Code Block	if <expr>
IFDEF	Execute If Symbol has Been Defined	ifdef <label>
IFNDEF	Execute If Symbol has not Been Defined	ifdef <label>
INCLUDE	Include Additional Source File	include <<include_file>> <include_file>"
LIST	Listing Options	list [<list_option>,...,<list_option>]
LOCAL	\Declare Local Macro Variable	local <label>[,<local>]
MACRO	Declare Macro Definition	label macro [<arg>,...,<arg>]
MESSG	Create User Defined Message	messg "<message_text>
NOEXPAND	Turn off Macro Expansion	noexpand

(Cont.)

NOLIST	Turn off Listing Output	nolist
ORG	Set Program Origin	<label> org <expr>
PAGE	Insert Listing Page Eject	page
PROCESSOR	Set Processor Type	processor <processsor_type>
RADIX	Specify Default Radix	radix <default_radix>
RES	Reserve Memory	res <mem_units>
SET	Define an Assembler Variable	<label> set <expr>
SPACE	Insert Blank Listing Lines	space <expr>
SUBTITLE	Specify Program Subtitle	subtitl "<sub_text>"
TITLE	Specify Program Title	title "<title_text>"
#UNDEFINE	Delete a Substitution Label	#undefine <label>
VARIABLE	Declare Symbol Variable	variable <label>[=<expr>,...,<label>[=<expr>]]
WHILE	Perform Loop While Condition is True	while <expr> . . . endw

Directive Details

The remainder of this chapter is dedicated to providing a detailed description of the directives supported by MPASM. Each definition will show:

- Syntax
- Description
- Example

A table of the MPASM directives is provided as a quick reference at the end of this document.

CBANK - Future Feature

Syntax

Description

Example

See Also

CBLOCK - Define a Block of Constants

Syntax

```
cblock   [<expr>]
```

Description

Define a list of named constants. Each is assigned a value of one higher than the last one. The purpose of this directive is to assign address offsets to many labels. The list of names end when and ENDC directive is encountered.

<expr> indicates the starting value for the first name in the block. If no expression is found, the first name will receive a value one higher than the final name in the previous CBLOCK or the current program counter.

Multiple names may be given on a line, separated by commas.

Example

```
cblock   0x20                  ; name_1 will be
                               ; assigned 20
         name_1, name_2        ; name_2, 21 and so on
         name_3, name_4        ; name_4 is assigned 23.
 endc
```

See Also

ENDC

CONSTANT - Declare Symbol Constant

Syntax

```
constant <label>[=<expr>,

...,<label>[=<expr>] ]
```

Description

<label> is a valid MPASM label, and <expr> is a valid MPASM expression. The expression must be fully resolvable at the time of the assignment.

The CONSTANT directive creates symbols for use in MPASM expressions. Contants may not be reset after having once been initialized. This is the principal difference between symbols declared as CONSTANT and those declared as VARIABLE, or created by the SET directive. Otherwise, constants and variables may be used interchangeably in expressions.

Example

```
variable RecLength=64              ; Set Default
                                     RecLength
constant BufLength=512, MaxMem     ; Init BufLength
         .                         ; RecLength may
         .                         ; be reset later
         .                         ; in RecLength=128
         .                         ;
         .                         ;
MaxMem=RecLength+BufLength         ;CalcMaxMem
```

See Also

SET

VARIABLE

DATA - Create Numeric and Text Data

Syntax

```
data     <expr>,[,<expr>,...,<expr>]
data     "<text_string>[,"<text_string>",...]
```

Description

Initialize one or more words of program memory with data. The data may be in the form of constants, relocatable or external labels or expressions of any of the above.

The data may also consist of ASCII character strings, <text_string>, enclosed in single quotes for one character, or double quotes for strings. Single character items are placed right justified into a whole word, while strings are packed two to a word with the first character in the most significant byte of the word. If an odd number of characters are given in a string, the final byte is zero filled.

All of the ANSI escape characters may be used in either of the latter two data formats.

Example

```
data     reloc_label+10     ; constants
data     1,2,ext_label      ; constants, externals
data     "testing 1,2,3"    ; text string
data     'N'                ; single character
data     start_of_program   ; relocatable label
```

See Also

DW **DB**

DB - Declare Data of One Byte

Syntax

```
db <expr>[,<expr>,...,<expr>]
```

Description

Reserve memory bytes, 8-bits of value expression. Multiple expressions continue to fill bytes consecutively until the end of expressions. Should there be an odd number of expressions, the last byte will be null filled.

Example

```
db      't', 0x0f, 'e', 0x0f, 's', 0x0f, 't', '\n'
```

See Also

DATA **DW**

#DEFINE - Define a Text Substitution Label

Syntax

```
#define <name> [<string>]
```

Description

This directive defines a text substitution string. Wherever `<name>` is encountered in the assembly code, `<string>` will be substituted and evaluated if possible.

Using the directive with no `<value>` causes a definition of `<name>` to be noted internally and may be tested for using the #IFDEF directive.

This directive emulates the ANSI 'C' standard for #define. Symbols defined with this method are not available for viewing using the PICMASTER or MPSIM.

Example

```
#define  length      20
#define  control     0x19,7
#define  position    (X,Y,Z)     (y-(2 * Z +X))
    .
    .
    .
test_label     dw    position(1, length, 512)
               bsf   control     ; set bit 7 in f19
```

See Also

IFDEF
IFNDEF
#UNDEFINE

DW - Declare Data of One Word

Syntax

```
dw <expr>[,<expr>,...,<expr>]
```

Description

Reserve memory words for data, initializing that space to specific values. <expr> is a variable number of valid MPASM expressions. Values are stored into successive memory locations and the location counter is incremented by one. Expressions may be literal strings and are stored as described in the DATA directive.

Example

```
dw      39, "diagnostic 39", (d_list*2+d_offset)
dw      diagbase-1
```

See Also

DATA **DB**

ELSE - Begin Alternative Assembly Block to IF

Syntax

```
else
```

Description

Used in conjunction with an IF directive to provide an alternative path of assembly code should the IF evaluate to false. ELSE may be used inside a regular program block or macro.

Example

```
speed             macro rate
   if     rate < 50
   dw     slow
   else
   dw     fast
   endif
   endm
```

See Also

IF **ENDIF**

END - End Program Block

Syntax

```
end
```

Description

Indicates the end of the program. After program termination, the symbol table is dumped to the listing file.

Example

```
start
   .                      ; executable code
   .                      ;
   .                      ;
end                       ; end of instructions
```

See Also

N/A

ENDC - End an Automatic Constant Block

Syntax

```
endc
```

Description

`ENDC` terminates the end of a `CBLOCK` list. It must be supplied to terminate the list.

See Also

CBLOCK

ENDIF - End Conditional Assembly Block

Syntax

```
endif
```

Description

This directive marks the end of a conditional assembly block. `ENDIF` may be used inside a regular program block or macro.

See Also

IF **ELSE**

ENDM - End a Macro Definition

Syntax

```
endm
```

Description

Macro definitions begin with a `MACRO` directive, and are terminated by the `ENDM` directive.

Example

```
make_table                macro       arg1, arg2
dw        "arg1", 0       ; null terminate table name
resv      arg2            ; reserve storage
   endm
```

See Also

MACRO **EXITM**

ENDW - End a While Loop

Syntax

```
endw
```

Description

ENDW terminates a WHILE loop. As long as the condition specified by the WHILE directive remains true, the source code between the WHILE directive and the ENDW directive will be repeatedly expanded in the assembly source code stream. This directive may be used inside a regular program block or macro.

Example

```
See the example for while
```

See Also

WHILE

EQU - Define an Assembler Constant

Syntax

```
<label>  equ   <expr>
```

Description

<expr> is a valid MPASM expression. The value of the expression is assigned to <label>.

Example

```
four     equ   4     ; assigned the numeric value of
                     ; to label four
```

See Also

SET **#DEFINE**

ERROR - Issue an Error Message

Syntax

```
error    "<text_string>"
```

Description

When conditions dictate that the MPASM assembler encounters an ERROR directive, the `<text_string>` is printed in a format identical to any MPASM error message. `<text_string>` may be from one to eighty characters.

Example

```
error_checking       macro       arg1
   if    arg1   >=    55    ; if arg is out of range
         error "error_checking-01 arg out of range"
   endif
endm
```

See Also

MESSG

EXITM - Exit from a Macro

Syntax

```
exitm
```

Description

Forces immediate retum from macro expansion during assembly. The effect is the same as if an `ENDM` directive had been encountered.

Example

```
test           macro fileReg
   if    filereg == 1      ; check for valid file
         exitm
   else
         error "bad file assignment"
endm
```

See Also

MACRO　　　　**ENDM**

EXPAND - Expand Macro Listing

Syntax

```
expand
```

Description

Causes all macros to be fully expanded in the listing file. This directive is roughly equivalent to the `/m` MPASM command line option, but may be limited in scope by the occurrence of a subsequent `NOEXPAND`.

See Also

MACRO **NOEXPAND**

FILL - Specify Memory Fill Value

Syntax

```
fill     <expr>
```

Description

The purpose of the `FILL` directive is to control the value placed in unused code locations of PROMs and ROMs. The `FILL` directive enables the fill function and specifies the fill value. This means that the linker output code files will contain record values for these locations. Unused code gaps are created by specifying address advances with `ORG` and `RES` directives.

The fill directive can be invoked multiple times to cause different values to be used. If no `FILL` directives are encountered, then no data records are generated for the unused code locations.

Example

```
fill            0x1009      ; fill with a constant
```

See Also

DW **ORG** **RES**

IF - Begin Conditionally Assembled Code Block

Syntax

```
if      <expr>
```

Description

Begin execution of a conditional assembly block. If <expr> evaluates to true, the code immediately following the if will assemble. Otherwise, subsequent code is skipped until an ELSE directive or an ENDIF directive is encountered.

Other conditions that may be checked:

- IFABS - If <label> is absolute
- IFNDEF - If <label> is not defined

An expression that evaluates to zero is considered logically FALSE. An expression that evaluates to any other value is considered logically TRUE. The IF and WHILE directives operate on the logical value of an expression. A relational TRUE expression is guaranteed to return a value of one; FALSE a value of zero.

Example

```
if version == 100; check current version
   movlw      0x0a
   movwf      io_1
else
   movlw      0x01a
   movwf      io_2
endif
```

See Also

ELSE   ENDIF

IFDEF - Execute If Symbol has Been Defined

Syntax

```
ifdef    <label>
```

Description

`<label>` is a valid MPASM label. If the label has been previously defined, usually by issuing a `#DEFINE` directive or by setting the value on the MPASM command line, the conditional path is taken. Assembly will continue until a matching `ELSE` or `ENDIF` directive is encountered.

Example(s)

```
#define  testing     1      ; set testing "on"
           .
           .
           .
ifdef    testing
   <execute test code>      ; this path would
endif                       ; be executed.
```

See Also

#DEFINE ELSE ENDIF

IFNDEF #UNDEFINE

IFNDEF - Execute If Symbol has not Been Defined

Syntax

```
ifdef    <label>
```

Description

<label> is a valid MPASM label. If the label has not been previously defined, or has been undefined by issuing an #UNDEFINE directive, then the code following the directive will be assembled. Assembly will be enabled or disabled until the next matching ELSE or ENDIF directive is encountered.

Example

```
#define   testing1     ; set testing on
   .
   .
   .
#undefine testing1     ; set testing off
ifndef    testing1     ; if not in testing mode
   .                   ; execute
   .                   ; this path
   .                   ;
endif                  ;
                       ;
   end                 ; end of source
```

See Also

#DEFINE ELSE
IFDEF #UNDEFINE
ENDIF

INCLUDE - Include Additional Source File

Syntax

```
include  <<include_file>>
include  "<include_file>"
```

Description

The specified file is read in as source code. Upon end-of-file, source code assembly will resume from the original source file. Up to six levels of nesting is permitted. `<include_file>` may be enclosed in quotes or angle brackets. In either case, only the current working directory will be searched, unless a fully qualified path is specified.

Example

```
include  "c:\sys\sysdefs.inc"     ; system defs
include  <regs.h>                 ; register defs
```

See Also

N/A

LIST - Listing Options

Syntax

```
list     [<list_option>, ..., <list_option>]
```

Description

Occurring on a line by itself, the `<list>` directive has the effect of turning listing output on, if it had been previously turned off. Otherwise, one of the following list options can be supplied to control the assembly process:

TABLE 5: LIST DIRECTIVE OPTIONS

Option	Default	Description
`c=nnn`	`80`	Set column width.
`n=nnn`	`59`	Set lines per page.
`t=ON|OFF`	`OFF`	Truncate lines of listing (otherwise wrap).
`p=<type>`	`None`	Set processor type: PIC16C54, PIC16C55, PIC16C56, PIC16C57, PIC16C71, PIC16C84, PIC17C42, PIC16C58, PIC16C64.
`r=<radix>`	`hex`	Set default radix: hex, dec, oct.
`x=ON|OFF`	`Off`	Turn macro expansion on or off.

See Also

NOLIST

LOCAL - Declare Local Macro Variable

Syntax

```
local    <label>[,<local>]
```

Description

<label> is a valid MPASM label. It may be a label that exists outside the context of the macro definition. The directive declares that the specified data elements are to be considered in local context to the macro.

If the macro is called recursively, each invocation will have its own local copy.

Example

```
<main code segment>
   .
   .
   .
len     equ   10        ; global version
size    equ   20        ; note that a local variable
                        ; may now be created and modi-
fied
test    macro size      ;
  local len,  label     ; local len and label
  len   set   size      ; modify local len
label   res   len       ; reserve buffer
  len   set   len-20    ;
  endm                  ; end macro
```

See Also

MACRO **ENDM**

MACRO - Declare Macro Definition

Syntax

```
label       macro    [<arg>, ..., <arg>]
```

Description

A macro is a sequence of instructions that can be inserted in the assembly source code by using a single macro call. The macro must first be defined, then it can be referred to in subsequent source code.

A macro can call another macro, or may call itself recursively.

Please refer to the chapter “Macro Language” for more information.

Example

```
Read     macro device, buffer, count
         movlw device
         movwf ram   20

         movlw buffer       ; buffer address
         movwf ram   21
         movlw count        ; byte count

         callsys_21         ; read file call

         endm
```

See Also

ENDM
IF
ENDIF
LOCAL
ELSE
EXITM

MESSG - Create User Defined Message

Syntax

```
messg    "<message_text>"
```

Description

Causes an informational message to be printed in the listing file. The message text can be up to 255 characters. Issuing a `MESSG` directive does not set any error return codes.

Example

```
macro    mssg_macro
   messg "mssg_macro-001 invoked without argument"
endm
```

See Also

ERROR

NOEXPAND - Turn off Macro Expansion

Syntax

```
noexpand
```

Description

Turns off macro expansion.

See Also

EXPAND

NOLIST - Turn off Listing Output

Syntax

```
NOLIST
```

Description

Turn off listing file output.

See Also

LIST

ORG - Set Program Origin

Syntax

```
<label>  org   <expr>
```

Description

Set the program origin for subsequent code at the address defined in <expr>. MPASM outputs relocatable object code, while MPLINK will place the code at the specified address. If <label> is specified, it will be given the value of the <expr>. The default origin is zero.

Example

```
int_1    org   0x20
         ; Vector 20 code goes here

int_2    org   int_1+0x10
         ; Vector 30 code goes here
```

See Also

RES

FILL

PAGE - Insert Listing Page Eject

Syntax

```
page
```

Description

Inserts a page eject into the listing file.

See Also

LIST   **TITLE**

PROCESSOR - Set Processor Type

Syntax

```
processor             <processor_type>
```

Description

Set the processor type to `<processor_type>`:
`[16C54 | 16C55 | 16C56 | 16C57 | 16C71 | 16C84 | 17C42]`

Example

```
processor      16C54
```

See Also

LIST

RADIX - Specify Default Radix

Syntax

```
radix          <default_radix>
```

Description

Sets the default radix for data expressions. The default radix is `hex`. Valid radix are: `hex, dec,` or `oct.`

Example

```
radix          dec
```

See Also

LIST

RES - Reserve Memory

Syntax

```
res       <mem_units>
```

Description

The RES directive is a relative `org` command. The command causes the program counter to be advanced from its current location by the value specified in `<mem_units>`.

Example

```
buffer   res   64    ; reserve 64 words of storage
```

See Also

ORG   **FILL**

SET - Define an Assembler Variable

Syntax

```
<label>  set   <expr>
```

Description

`<label>` assumes the value of the valid MPASM expression specified by `<expr>`. The `SET` directive is functionally equivalent to the `EQU` directive except that `SET` values may be altered by `SET` directives.

Example

```
area    set     0
width   set     0x12
length  set     0x14
Area    set     length * width
length  set     length + 1
```

See Also

EQU

SPACE - Insert Blank Listing Lines

Syntax

```
space          <expr>
```

Description

Insert <expr> number of blank lines into the listing file.

Example

```
space    3      ;Inserts three blank lines
```

See Also

LIST

SUBTITLE - Specify Program Subtitle

Syntax

```
subtitl  "<sub_text>"
```

Description

<sub_text> is an ASCII string enclosed in double quotes, 60 characters or less in length. This directive establishes a second program header line for use as a subtitle in the listing output.

Example

```
subtitle "diagnostic section"
```

See Also

TITLE

TITLE - Specify Program Title

Syntax

```
title          "<title_text>"
```

Description

`<title_text>` is a printable ASCII string enclosed in double quotes. It must be 60 characters or less in length. This directive establishes the text to be used in the top line of the listing page.

Example

```
title          "operational code, rev 5.0"
```

See Also

LIST **SUBTITL**

#UNDEFINE - Delete a Substitution Label

Syntax

```
#undefine      <label>
```

Description

`<label>` is an identifier previously defined with the `#DEFINE` directive. It must be a valid MPASM label. The symbol named is removed from the symbol table.

Example

```
#define        length      20
    .
    .
    .
#undefine      length
```

See Also

#DEFINE	**IFDEF**
INCLUDE	**IFNDEF**

VARIABLE - Declare Symbol Variable

Syntax

```
variable <label>[=<expr>,...,<label>[=<expr>] ]
```

Description

<label> is a valid MPASM label, and <expr> is a valid MPASM expression. The expression must be fully resolvable at the time of the assignment.

The VARIABLE directive creates symbols for use in MPASM expressions. Variables differ from constants may be used interchangeably in expressions.

The VARIABLE directive creates a symbol that is functionally equivalent to those created by the SET directive. The difference being that the VARIABLE directive does not require that symbols be initailized when they are declared.

Example

Please refer to the CONSTANT example.

See Also

SET **CONSTANT**

WHILE - Perform Loop While Condition is True

Syntax

```
while <expr>
    .
    .
    .
endw
```

Description

<expr> is a valid MPASM expression that controls the number of times the loop is performed. An expression that evaluates to zero is considered logically FALSE. An expression that evaluates to any other value is considered logically TRUE. The IF and WHILE directives operate on the logical value of an expression. A relational TRUE expression is guaranteed to return a value of one; FALSE a value of zero.

Example

```
test mac macro count
         variable i
         i = 0
         while  i  < count
         movlw  i
         i += 1
         endw
         endm
start
         test mac  5
         end
```

See Also

ENDW

IF

Chapter 4. Macro Language

Introduction

Macros are user defined sets of instructions and directives that will be included in-line with the assembler source code whenever the macro is invoked.

Macros consist of sequences of assembler instructions and directives. They can be written to accept arguments, making them flexible. Their advantages are:

- Higher levels of abstraction, improving readability and reliability.
- Consistent solutions to frequently performed functions.
- Simplified changes.
- Improved testability.

Applications might include, creating complex tables, frequently used code and complex operations.

Highlights

The points that will be highlighed in this chapter are:

- Macro Syntax
- Text Substitution
- Local Symbols
- Recursive Macros
- Macro Usage
- Examples

Terms

Macro

As define before, a macro is a collection of assembler instructions that are included in the assembly code when the macro is invoked by the source code. Macros must be defined before their first invocation; forward references to macros are **not** allowed.

All statements following the MACRO directive (see Chapter 5) are part of the macro definition. Lines consisting of a comment only are not saved in the macro definition. Labels used within the macro must be local to the macro so the macro can be called repetitively.

Local Label

A local label is one that is defined with the LOCAL directive (see Chapter 5). These labels are particular to a given instance of the macro's instantiation. In other words, the symbols and labels that are declared as local are purged from the symbol table when the ENDM macro is encountered.

Recursion

This is the concept that a 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.

Macro Syntax

MPASM macros are defined according to the following syntax:

```
<label>  macro        [<arg>, ..., <arg>]
          .
          .
          .
         endm
```

Where <label> is a valid MPASM label and <arg> are any number of optional arguments supplied to the macro. The values assigned to these arguments at the time the macro is invoked will be substituted wherever the argument name occurs in the body of the macro.

The body of a macro may be comprised of MPASM directives, PIC16/17 assembly instructions, or MPASM Macro Directives (LOCAL for example). Refer back to Chapter 5. MPASM continues to process the body of the macro until a EXITM or ENDM directive is encountered

NOTE: Once again, forward references to macros are not permitted.

Macro Directives

As noted in Chapter 5, there are a few directives that are unique to macro definitions. They make no sense out of the macro context (refer to Chapter 5 for details concerning these directives):

- `MACRO`
- `LOCAL`
- `EXITM`
- `ENDM`

When writing macros, you can use any of these directives **PLUS** any other directives supported by MPASM.

> **NOTE:** The previous syntax of the "dot" format for macro specific directives is no longer supported. For compatibility reasons, old ASM17 code that uses this format will assemble by MPASM, but as mentioned before, you are encouraged to write new code based on the constructs defined within this document to ensure upward compatibility with MPASM.

Text Substitution

A variety of string replacement and parsing patterns may appear within the body of a macro. They may be used only within the body of a macro.

Command	Description
`<arg>`	Substitute the argument text supplied as part of the macro invocation.
`#v(<label>)`	Returns the integer value of the simple `<label>`. Typically, used to create unique variable names with common prefixes or suffixes.

Arguments may be used anywhere within the body of the macro, **except** as part of normal expression. For example, the following macro:

```
define_table                macro
        variable            a = 0

        while               a < 3

            entry#v( a )    dw    0

            a += 1

        endw

endm
```

would generate:

```
        entry0      dw      0
        entry1      dw      0
        entry2      dw      0
        entry3      dw      0
```

when invoked.

Recursive Macros

Macros may invoke themselves. This is known as recursion. Care should be exercised, as in all cases of recursion, to avoid infinite loops. Macros called recursively will generate their own local variables if the LOCAL directive is used (see Chapter 3).

Macro Usage

Once the macro has been defined, it can be invoked at any point within the source module by using a macro call, as described below.

```
<macro_name>    [<arg>, ..., <arg>]
```

Where <macro_name> is the name of a previously defined macro, and arguments are supplied, as required.

The macro call itself will not occupy any locations in memory. However the macro expansion will begin at the current memory location. Commas may be used to reserve an argument position. In this case, the argument will be NULL. The argument list is terminated by white space or a semicolon-colon.

The EXITM directive (see Chapter 3) provides an alternate method for terminating a macro expansion. During a macro expansion, this directive causes expansion of the current macro to stop and all code between the EXITM and the ENDM directives for this macro to be ignored. If macros are nested, EXITM causes code generation to return to the previously level of macro expansion.

Examples

Eight by Eight Multiply

```
subtitl  "macro definitions"
page
;
; multiply - eight by eight multiply macro, executing
; in program memory.  optimized for speed, straight
; line code.
;
; written for the PIC17C42.
;
multiply macro  arg1, arg2, dest_hi, dest_lo
                        ;
   local i              ; establish local index variable
   variable i = 0       ; and initialize it.
                        ;
   movlw arg1           ; setup multiplier
   movwf mulplr         ;
                        ;
   movlw arg2           ; setup multiplicand in w reg
                        ;
   clrf  dest_hi        ; clear the destination regs
   clrf  dest_lo        ;
                        ;
   bcf   _carry         ; clear carry for test
                        ;
   while i < 8          ; do all eight bits
   addwf dest_hi        ; then add multiplicand
   rrcf  dest_hi        ; shift right through carry
   rrcf  dest_lo        ; shift right again, snag carry
                        ; if set by previous rotate
   i += 1               ; increment loop counter
   endw                 ; break after eight iterations
endm                    ; end of macro.
```

The macro declares all of the required arguments. In this case, there are four. The LOCAL directive then establishes a local variable "i" that will be used as an index counter. It is initialized to zero.

A number of assembler instructions are then included. When the macro is executed, these instructions will be written in line with the rest of the assembler source code.

The macro writes the multiplication code using an algorithm that uses right shifts and adds for each bit set in the eight bits of the multiplier. The WHILE directive is used for this function, continuing the loop until 'i' is greater than or equal to eight.

The end of the loop is noted by the ENDW directive. Execution continues with the statement immediately following the ENDW when the WHILE condition becomes TRUE. The entire macro is terminated by the ENDM directive.

Constant Compare

For further example, if the following macro were written:

```
include  "16cxx.reg"
;
; compare file to constant and jump if file
; >= constant.
;
cfl_jge  macro file, con, jump_to
         movlw con & 0Xff
         subwf file, w
         btfsc status, carry
         goto  jump_to
         endm
```

and invoked by:

```
cfl_jge  switch_val, max_switch, switch_on
```

it would produce:

```
   movlw max_switch & 0xff
   subwf switch_val, w
   btfsc status, carr
   goto  switch_on
```

Chapter 5. Expression Syntax and Operation

Introduction

This chapter describes various expression formats, syntax, and operations used by MPASM.

Highlights

The points that will be highlighted in this chapter are:

- Text Strings
- Numeric Constants and Radix
- Arithmetic Operators and Precedence
- High / Low Operators

Terms

Expressions

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

> **NOTE:** Expressions are evaluated in 32 bit integer math (floating point is not currently supported).

Operators

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

Precedence

Precedence is the concept that some elements of an expression get evaluated before others. In general, precedence is established from left to right, and expressions within parentheses are always evaluated first.

Radix

Radix is the base numbering system that the assembler is supposed to use when evaluating expressions. The default radix is hexadecimal (base 16). You can change the default radix (See Chapter 5) and override the default with certain radix override operators. These are described in this chapter.

Text Strings

A “string” is a sequence of any valid ASCII character (of the decimal range of 0 to 127) enclosed by double quotes.

Strings may be of any length that will fit within a 132 column source line. If a matching quote mark is found, the string ends. If none is found before the end of the line, the string will end at the end of the line. While there is no direct provision for continuation onto a second line, it is generally no problem to use a second DW directive for the next line.

The DW directive will store the entire string into successive words. If a string has an odd number of characters (bytes), the DW and DATA directives will pad the end of the string with one byte of zeros (00).

If a string is used as a literal operand, it must be exactly one character long, or an error will occur.

See the examples below for the object code generated by different statements involving strings.

```
7465 7374 696E      dw              "testing output string
one\n"
6720 6F75 7470
7574 2073 7472
696E 6720 6F6E
650A

                    #define str     "testing output string
two"

B061                movlw           "a"

7465 7374 696E      data            "testing first output
string"
6720 6669 7273
7420 6F75 7470
7574 2073 7472
696E 6700
```

The assembler accepts the ANSI ‘C’ escape sequences to represent certain special control characters:

TABLE 6: ANSI 'C' ESCAPE SEQUENCES

Escape Character	Description
\a	Bell (alert) character
\b	Backspace character
\f	Form feed character
\n	New line character
\r	Carriage return character
\t	Horizontal tab character
\v	Vertical tab character
\\	Backslash
\?	Question mark character
\'	Single quote (apostrophe)
\”	Double quote character
\0OO	Octal number (zero, Octal digit, Octal digit)
\xHH	Hexadecimal number

Numeric Constants and Radix

MPASM supports the following radix forms: hexadecimal, decimal, octal, binary, and character. The default radix is hexadecimal; the default radix determines what value will be assigned to constants in the object file when they are not explicitly specified by a base descriptor.

Constants can be optionally preceded by a plus or minus sign. If unsigned, the value is assumed to be positive.

> **NOTE:** Intermediate values in constant expressions are treated as 32-bit unsigned integers. Whenever an attempt is made to place a constant in a field for which it is too large, a truncation warning will be issued.

The following table presents the various radix specifications:

TABLE 7: RADIX SPECIFICATIONS

Type	Syntax	Example
Decimal	`D'<digits>'`	`D'100'`
Hexadecimal	`H'<hex_digits>'`	`H'9f`
Octal	`O'<octal_digits>'`	`O'777'`
Binary	`B'<binary_digits>'`	`B'00111001'`
Character	`'<character>'`	`'C'`
	`A'<Character>'`	`A'C'`

TABLE 8: ARITHMETIC OPERATORS AND PRECEDENCE

Operator		Example
(	Left Parenthesis	`1 + ( d * 4 )`
)	Right Parenthesis	`( Length + 1 ) * 256`
!	Item NOT (logical complement)	`if ! ( a - b )`
-	Negation (2's complement)	`-1 * Length`
high	Return high byte	`movlw high CTR Table`
low	Return low byte	`movlw low CTR Table`
*	Multiply	`a = b * c`
/	Divide	`a = b / c`
%	Modulus	`entry_len = tot_len % 16`
+	Add	`tot_len = entry_len * 8 + 1`
-	Subtract	`entry_len = ( tot - 1 ) / 8`
<<	Left shift	`<< flags`
>>	Right shift	`>> flags`
>=	Greater or equal	`if entry_idx >= num_entries`
>	Greater than	`if entry_idx > num_entries`
<	Less than	`if entry_idx < num_entries`
<=	Less or equal	`if entry_idx <= num_entries`
==	Equal to	`if entry_idx == num_entries`
!=	Not equal to	`if entry_idx != num_entries`
&	Bitwise AND	`flags = flags & ERROR_BIT`
^	Bitwise exclusive OR	`flags = flags ^ ERROR_BIT`
|	Bitwise inclusive OR	`flags = flags | ERROR_BIT`
~	Complement	`flags = flags ~ ERROR_BIT`
&&	Logical AND	`if ( len == 512 ) && b == c`
||	Logical OR	`if ( len == 512 ) || b == c`
=	Set equal to	`entry_index = 0`
+=	Add to, set equal	`entry_index += 1`
-=	Subtract, set equal	`entry_index -= 1`
*=	Multiply, set equal	`entry_index *= entry_length`
/=	Divide, set equal	`entry_total /= entry_length`
%=	Modulus, set equal	`entry_index %= 8`
<<=	Left shift, set equal	`flags <<= 3`
>>=	Right shift, set equal	`flags >>= 3`
&=	AND, set equal	`flags &= ERROR_FLAG`
|=	Inclusive OR, set equal	`flags |= ERROR_FLAG`
^=	Exclusive OR, set equal	`flags ^= ERROR_FLAG`
$	Return program counter	`goto $ + 3`

High / Low

Syntax

```
<instruction>  high  <operand>
<instruction>  low   <operand>
```

Description

Where `<instruction>` is an appropriate MPASM assembler instruction and `<operand>` is an appropriate argument list for that instruction.

The high operators are used to return the high byte or the low byte of a 16-bit label value. This is done to handle dynamic pointer calculations as might be used with table read and write instructions.

Example

```
movlw    low   size        ; handle the lsb's
movpf    wreg, low size_lo
movlw    high  size        ; handle the msb's
movpf    wreg, high size_hi
```

Chapter 6. MPLIB - MPASM Librarian

Introduction

A librarian is a tool that allows several different objects to be grouped into one logical collection, or library. MPLIB combines several object modules, created with MPASM, into a single file. When a library file is linked with other modules, only those library functions that are referenced by the other modules are linked into the final binary code.

Object modules can be added, deleted, or replaced from the MPLIB Command Line Interface.

Chapter 7. MPLINK - MPASM Linker

Introduction

MPLINK is the MPASM relocatable object linker. It joins any number of object files, together with any library modules, into an executable binary file that is fixed in the PIC16/17's memory. MPLINK can also generate an absolute listing file of the PIC16/17 application. This file can be invaluable when debugging your design; it is in the same general format as the listing file generated for relocatable objects by MPASM.

MPLINK accepts options from either its Command Line Interface or a script file.

Appendix A. Object Code Formats

Introduction

MPASM and MPLINK are capable of outputting several different object file formats, suitable for a variety of programmer and emulator applications.

Highlights

- Intel® HEX Format (INHX8M)
- Intel Split HEX Format (INHX8S)
- Intel HEX 32 Format (INHX32)

Object Code Formats

MPLINK and MPASM (with absolute addresses) are capable of producing a number of different output formats.

Intel HEX Format (.HEX)

This format produces one 8-bit HEX file with a low byte, high byte combination. Since each address can only contain 8 bits in this format, all addresses are doubled. This file format is useful for transferring PIC16/17 series code to third party EPROM programmers (for example, Data I/O® Unisite™ and Logical Devices ALLPRO™).

Each data record begins with a 9 character prefix and ends with a 2 character checksum. Each record has the following format:

```
:BBAAAATTHHHH....HHHCC
```

where

BB - is a two digit hexadecimal byte count representing the number of data bytes that will appear on the line.

AAAA - is a four digit hexadecimal address representing the starting address of the data record.

TT - is a two digit record type record type that will always be ‘00’ except for the end-of-file record, which will be ‘01’.

HH - is a two digit hexadecimal data word, presented in low byte, high byte combinations.

CC - is a two digit hexadecimal checksum that is the two's compliment of the sum of all preceding bytes in the record including the prefix.

Example

<file_name>.HEX

```
:1000000000000000000000000000000000000000F0
:0400100000000000EC
:100032000000280040006800A800E800C80028016D
:100042006801A9018901EA01280208026A02BF02C5
:10005200E002E80228036803BF03E803C8030804B8
:1000620008040804030443050306E807E807FF0839
:06007200FF08FF08190A57
:00000001FF
```

8-Bit Split Format (.HXL/.HXH)

The Intellec split 8-bit file format produces two output files: .HXL and .HXH. The format is the same as the normal 8-bit format, except that the low bytes of the data word are stored in the .HXL file, and the high bytes of the data word are stored in the .HXH file.

Example

<file_name>.HXL

```
:0A00000000000000000000000000F6
:1000190000284068A8E8C82868A989EA28086ABFAA
:10002900E0E82868BFE8C8080808034303E8E8FFD0
:03003900FFFF19AD
:00000001FF
```

<file_name>.HXH

```
:0A00000000000000000000000000F6
:1000190000000000000000010101010102020202CA
:100029000202030303030304040404050607070883
:0300390008080AAA
:00000001FF
```

32-Bit Hex Format (.HEX)

The extended 32-bit address HEX format is similar to the Hex 8 format described above, except that the Intel extended linear address record is output also to establish the upper 16 bits of the data address.

Each data record begins with a 9 character prefix and ends with a 2 character checksum. Each record has the following format:

```
:BBAAAATTHHHH....HHHCC
```

where

BB - is a two digit hexadecimal byte count representing the number of data bytes that will appear on the line.

AAAA - is a four digit hexadecimal address representing the starting address of the data record.

TT - is a two digit record type record type:

00 - Data record
01 - End of File record
02 - Segment address record
04 - Linear address record

HH - is a two digit hexadecimal data word.

CC - is a two digit hexadecimal checksum that is the two's compliment of the sum of all preceding bytes in the record including the prefix.

Appendix B. Customer Support

Keeping Current with Microchip Systems

Microchip Technology endeavors at all times to provide the best service and responsiveness possible to it users. The Microchip Technology Systems BBS is one mechanism to facilitate this process.

The BBS is supported as a service to its customers. This is where all of the most recent information regarding systems products can be found. The BBS is monitored several times a week for questions. Truly urgent issues should not be left with the BBS, but referred to your local distributor, or Microchip sales office.

The BBS is an evolving product. Details of its operation will not be found here. This chapter provides a brief discussion of the general services available.

This chapter also describes the Microchip Systems software numbering scheme.

Highlights

The points that will be highlighted in this chapter are:

- Access to the BBS
- Special Interest Groups
- Files
- Mail
- Software Releases

Bulletin Board Access

Access to the bulletin board is 24 hours per day, barring technical or mechanical difficulties. Access is gained by calling your local CompuServe access number. Your modem should be set to 8-bits, No parity, 1 stop bit (8-1-N). The service supports baud rates from 300 to 9600 baud. To access the BBS, follow these steps:

1. Dial your local CompuServe access number.
2. Press `<ret>` and a garbage string will appear.
3. Enter `+<ret>` and `Host Name`: will appear.
4. Enter `mchipbbs<ret>` and you will be connected to the Microchip BBS.

There is **no charge** for connecting to the BBS. There is **no charge** to dial the CompuServe access number. You do **not** need to be a CompuServe member to take advantage of this connection (you never actually log in to CompuServe).

Bulletin Board Usage

The bulletin board is a multifaceted tool. It can provide you with information on a number of different topics.

- Special Interest Groups
- Files
- Mail
- Bug Lists
- Technical Assistance

Special Interest Groups

Special Interest Groups, or SIGs as they are commonly referred to, provide you with the opportunity to discuss issues and topics of interest with others that share your interest or questions. They may be able to provide you with information not available by any other method because of the broad background of the PIC16/17 user community.

There are SIGs for most Microchip systems, including:

- PRO MATE™
- PICMASTER™
- MPASM
- Utilities
- Bugs

These groups are monitored by the Microchip staff.

Files

The Microchip BBS is used regularly to distribute technical information, Application Notes' source code, errata sheets, bug reports, and interim patches for Microchip systems software products. Users can contribute files for distribution on the BBS. These files will be monitored, scanned and approved or disapproved by the moderator of the SIG to which the file is submitted. No executable files are accepted from the user community in general to limit the spread of computer viruses.

Mail

The BBS can be used to distribute mail to other users of the service. This is one way to get answers to your questions and problems from the Microchip staff, as well as to keep in touch with fellow Microchip users worldwide.

Consider mailing the moderator of your SIG, or SYSOP, if you have ideas or questions about Microchip products, or the operation of the BBS. Be aware, though, that the SIGs are moderated only about once per day. Truly urgent questions should be referred to your local distributor, sales representative, or FAE. They are your first line of defense.

Software Revisions

Software products released by Microchip are referred to by version numbers. Version numbers use the form:

```
xx.yy.zz <status>
```

Where `xx` is the major release number, `yy` is the minor number, and `ZZ` is the intermediate number. The `status` field displays one of the following categories:

- Alpha
- Intermediate
- Beta
- Released

Production releases are numbered with major, and minor version numbers like:

```
3.04  Released
```

Alpha, Beta and Intermediate releases are numbered with the major, minor and intermediate numbers:

```
3.04.01  Alpha
```

Alpha Release

Alpha designated software is engineering software that has not been submitted to any quality assurance testing. In general, this grade of software is intended for software development team access only, but may be sent to selected individuals for conceptual evaluation. Once Alpha grade software has passed quality assurance testing, it may be upgraded to Beta or Intermediate status.

Intermediate Release

Intermediate released software represents changes to a releasd software system and is designated as such by adding an intermediate number to the version number. Intermediate changes are represented by:

- Bug Fixes
- Special Releases
- Feature Experiments

Intermediate released software does not represent our most tested and stable software. Typically, it will not have been subject to a thorough and rigorous test suite, unlike production released versions. Therefore, users should use these versions with care, and only in cases where the features provided by an intermediate release are required.

Intermediate releases are primarily available through the BBS.

Beta Release

Preproduction software is designated as Beta. Beta software is sent to Applications Engineers and Consultants, FAEs, and select customers. The Beta Test period is limited to a few weeks. Software that passes Beta testing without having significant flaws, will be production released. Flawed software will be evaluated, repaired, and updated with a new revision number for a subsequent Beta trial.

Production Release

Production released software is software shipped with tool products. Example products are PRO MATE™, PICSTART™, and PICMASTER™. The Major number is advanced when significant feature enhancements are made to the product. The minor version number is advanced for maintenance fixes and minor enhancements. Production released software reresents Microchip's most stable and thoroughly tested software.

There will always be a period of time when the Production Released software is not reflected by products being shipped until stocks are rotated. You should always check the BBS for the current production release.

Appendix C. MPALC Conversion Guide

Introduction

MPASM attempts to be backward compatible with MPALC.

It is very possible that your source code will assemble as is. There are, however, a number of inconsistencies between the two assemblers that require some simple changes. In addition, MPASM attempts to provide a clean and simple assembler solution for the future. To that end, there are also a number of changes that are recommended to encourage compatibility going forward.

Highlights

- Required Source Code Updates
- Recommended Source Code Updates

Required Source Code Updates

- Specify processor type at the very top of the first source file in programs assembled with MPASM using either the PROCESSOR or LIST directives, or specify the processor on the command line. MPASM will not assemble your source without this information.

Recommended Source Code Updates

- You are encouraged to move all of your labels to column one, and assembler directives (like LIST and TITLE) to at least column two. An example would be:

```
            list        p=16c54, r=hex
            title       Sample Code
#include    "c:\tools\regs.h"

#define     zero        0
#define     one         1

alabel      set         zero
blabel      set         one

            org         0x00
            goto        Start

            org         0x28

Start
            goto        Start
            end
```

  and so on.

- Specify the radix you are assuming using either the RADIX or LIST directives, or the command line option.

- Change all radix overrides currently included in your code to one of those specified in the MPASM User's Guide. For example:

  Change: `movlw 5D`

  to: `movlw D'5'`

- Fully qualify moves of label addresses by using the HIGH or LOW directives.

Appendix D. ASM17 Conversion Guide

Introduction

MPASM attempts to be backward compatible with ASM17.

It is very possible that your source code will assemble as is. There are, however, a number of inconsistencies between the two assemblers that require some simple changes. In addition, MPASM attempts to provide a clean and simple assembler solution for the future. To that end, there are also a number of changes that are recommended to encourage compatibility going forward.

Highlights

- Required Source Code Updates
- Recommended Source Code Updates

Required Source Code Updates

- Specify processor type at the very top of the first source file in programs assembled with MPASM using either the PROCESSOR or LIST directives, or specify the processor on the command line. MPASM will not assemble your source without this information.
- Change DOS paths for ASM17 source code from using two back slashes to one.

Example:	Change:	`#include`	`c:\\tools\\regs.h`
	to:	`#include`	`c:\tools\regs.h`

- Recode any macros that use a variable number of arguments to call out specific arguments. This would only be appropriate for ASM17 code, and is a feature that will be included at some point in the future.
- If you are using the ASM17 HALT directive, recode this as a macro or remove it. This feature will be included at some point in the future.
- If you are using the ASM17 FILL directive, temporarily remove it. This feature will be included at some point in the future.

Recommended Source Code Updates

- You are encouraged to move all of your labels to column one, and assembler directives (like LIST and TITLE) to at least column two. An example would be:

```
            list        p=17c42, r=dec
            title       Sample Code
#include    "c:\tools\regs.h"

#define     zero        0
#define     one         1

alabel      set         zero
blabel      set         one

            org         0x00
            goto        Start

            org         0x28

Start
            goto        Start
            end
```

  and so on.

- Specify the radix you are assuming using either the RADIX or LIST directives, or the command line option.

- Change all radix overrides currently included in your code to one of those specified in the MPASM User's Guide. For example:

  Change: `movlw 5D`

  to: `movlw D'5'`

- Fully qualify moves of label addresses by using the HIGH or LOW directives.

Appendix E. Error Messages

The following error and warning messages are produced by MPASM. These messages always appear in the listing file directly above each line in which the error occurred.

The error and warning messages are stored in the error file (.ERR) if no MPASM options are specified. If the /e- option is used (turns error file off), then the messages will appear on the screen. If the /q (quiet mode) option is used with the /e-, then the messages will not display on the screen or in an error file. The messages will still appear in the listing file.

Error Messages

Address exceeds maximum limit available

You are trying to access memory that is not supported. The current program counter is greater than the maximum program memory limit for the specified processor type. Please refer to the data sheet for this processor to find the valid memory range.

Attempt to redefine reserved word

The words "END", "ERROR", "HIGH", "LOW" and PAGE are reserved words in MPASM. You must not use these words as labels or symbols. Remove or rename any occurances of these words and then reassemble.

NOTE: It is too costly with regard to both time and space to treat all directives, opcodes and operators as reserved words. MPASM reserves only the above minimal list of words to avoid the most common misuses of directives.

Branch or jump out of range

A branch or jump statement is addressing the last half of a program memory page. This is not allowed. Any instruction which writes to the Program Counter (CALL, JUMP, BRANCH or GOTO) is limited to the first 256 locations of any program memory page.

Call or jump not allowed at this address

A call or computed jump statement is addressing the last half of a program memory page. This is not allowed. Any Instruction which writes to the Program Counter (CALL, JUMP, BRANCH or GOTO) is limited to the first 256 locations of any program memory page.

Couldn't open . . .

MPASM couldn't open the specified object file, memory map file, code file, error file, listing file, or cross-reference file. Either the file already exists and is read/write protected, or there is not enough disk space to create or write to the file.

Couldn't open source file . . .

The source file specified on the command line, or through the interactive menu, does not exist. Check the current directory for the desired file, and verify the spelling of the source file name.

Duplicate label or redefining symbol that cannot be redefined

You have either used the same label name twice in your program, or a constant, a #DEFINE'd symbol, or an EQU'd symbol has been used on the left-hand side of an equation. MPASM does not know which definition to use. This ERROR message appears before BOTH definitions of the symbol, when appropriate.

Error in parameter

One of the options used on the MPASM command line was not a valid option, or an option was incorrectly formatted. Type mpasm /h or mpasm /? at the DOS prompt to see a usage message showing the valid command line options.

Expected . . .

The syntax of the source line is incorrect. MPASM expected to see one thing, but got something different. Check the syntax of the directive or opcode in error in the MPASM User's Guide.

File not found

The file specified in the shell screen's source file field does not exist in the current directory. This error appears when the shell interface is used to invoke MPASM, rather than the command line interface. Check the spelling of the file name, and verify that you are in the desired directory. Press any key to continue.

Illegal argument

The radix specified with either a LIST directive or the RADIX directive is not one of the valid radix choices. Change the radix to: DEC for decimal, OCT for octal, or HEX for hexadecimal radix.

Illegal condition

An IF statement is using an illegal comparison operator. The valid conditions which can be checked by an IF statement are:

== (equal to)

!= (not equal to)

> (greater than)

< (less than)

>= (greater than or equal to)

<= (less than or equal to).

Illegal condition, EOF encountered before END or conditional end directive

The END directive is missing, or a CBLOCK, an IF, a WHILE or a MACRO statement is missing an ENDC, ENDIF, ENDW or ENDM respectively.

Illegal conditional compile

There is a problem with the construction of the indicated IF / ELSE / ENDIF statements.

Illegal character . . . in label . . .

The specified label contains an illegal character. Legal characters are: underscore (_), period (.), capital letters (A through Z), lower case letters (a through z), or decimal digits (0 through 9).

Illegal digit

The specified digit is illegal in the context used. The digit is either incorrect for the radix specified in the source file, or is an unsupported ANSI escape sequence. Check the LIST directive used in the source code to verify the specified radix. See Chapter 5: Expression Syntax and Operation in the MPASM User's Guide for valid radix specifications and ANSI 'C' escape sequences.

Illegal opcode

The indicated opcode or directive is not recognized by MPASM. The opcode may be misspelled, or is no longer supported. Or, an otherwise legal opcode may be used in an illegal context. For example, a valid directive, such as LIST, prepended with a pound sign (#LIST) will generate this "Illegal opcode" error message. Other examples are: using an ELSE without an associated IF, or using an INCLUDE directive inside a macro definition.

Include file not found

The file to be included does not exist in the current directory. Check the spelling of the include file name, and verify that you are in the desired directory. If necessary, specify the complete DOS path (for example:

C:\SOURCE\INCLUDE\FILENAME.H).

Include files nested too deep

The current include file cannot include another file, because you have reached the maximum level of include file nesting. The maximum number of include files nested within each other is five (5).

Macro name missing

The term "macro" has been encountered without an associated name for the macro. Macro names can be any legal, unique MPASM label.

Macros nested too deep

The current macro definition cannot call another macro, because you have reached the maximum level of macro nesting. The maximum number of macros nested within each other is eight (8).

Missing argument(s)

This opcode, directive, or macro call requires at least one more operand (argument) than is provided. Check the instruction set for the proper syntax. This error can also occur if a #DEFINE'd label is missing a value. In this case, the error message won't appear until the label is used as an operand.

Missing terminator

There is an open parenthesis, curly bracket or square bracket without amatching closed parenthesis, curly bracket or square bracket, respectively. This error message can also occur when a comma or blank is expected, but not found.

Nested forward reference not allowed

The indicated label has not been defined yet, and is not allowed to be used before it is defined. Specifically, forward references to macros are not permitted. If a macro call is generating the error, move the call to a point in the code below the macro definition.

This error message also occurs when MPASM cannot tell the type of a given label: variable, constant, address, local variable, or reserved word. The label may be defined more than once.

Out of memory

All the PC's available memory has been used to create code segments, macros and forward references. Try reducing the number of macros in your source file(s). Also, close out of any terminate-and-stay-resident programs (TSR's) and close any applications, then try assembling the file again. If you are running Windows and assembling from a DOS prompt, try exiting Windows and assembling directly from DOS.

Overwriting previous address contents

The location for which MPASM is trying to generate object code has already been used by this program. Usually, an ORG directive for this address occurs prior to the source line that produces this error message. The only time MPASM will allow you to overwrite a previously-used address is if it was reserved with an RES directive.

Processor type is undefined

No processor type has been specified. Use the LIST or PROCESSOR directive in your source code, or use the /p option on the command line, to define a processor type (PIC16C54, PIC16C55, PIC16C56, PIC16C57, PIC16C58A, PIC16C64, PIC16C71, PIC16C84, or PIC17C42).

Processor type previously defined

A processor type has already been specified. You cannot change processor types in the middle of a program. Check the LIST or PROCESSOR directive in your source code to see which processor type is defined. If you specify the same processor type two or more times in the same program, no error will occur.

Symbol table full

All the PC's available memory has been used to create symbols. MPASM requires more memory to create all the symbols defined in this source file. Eliminate any TSR's, and close any applications, then try assembling the file again. If you are running Windows and assembling from a DOS prompt, try exiting Windows and assembling directly from DOS.

You may get the qualifier "Out of macro space (#define)" appended to this message. If this is the case, eliminate #DEFINE symbols in your code by hardcoding as many values as possible.

Or, you may see this error message qualified with "more than 8 locals in a macro." The maximum number of local variables allowed per macro is eight (8). Eliminate local variables from the macro definition.

Temp file creation error

A temporary file could not be created as needed. MPASM uses temporary files when building the symbol table. This error could be caused by a DOS disk write error, if the disk is full.

Too many arguments

An opcode has been given too many operands, or a macro has been invoked with too many arguments. Check the processor's instruction set for the proper opcode syntax, or verify the number of arguments in the macro definition.

Undefined argument

A label is being used that has not yet been defined. The label might be used as an operand or as a macro argument. If there is a mistake in the argument list of a macro definition, this error message will appear when the macro is invoked, since the arguments were never properly defined.

Unknown error

An error has occurred which MPASM cannot understand. It is not any of the errors described in this appendix. Contact your Microchip Field Application Engineer (FAE) if you cannot debug this unknown error.

WHILE failed to terminate within 256 iterations

The end condition of a WHILE loop was never met. This is flagged as an "Unknown error" because MPASM doesn't know why the WHILE loop didn't terminate. Check your condition statement for proper syntax and logic.

Warning Messages

Addresses above 32K not currently supported. Using MaxRom.

MPASM does not currently allow you to access memory above 0x8000 (32K). Eventually, addresses up to 64K will be supported, when the linker and librarian are implemented.

Argument out of range, least significant bits used

The operand is not between the maximum and minimum values allowed for this opcode in this processor family. Most "Argument out of range" errors are WARNING messages. The out-of-range argument is truncated to the maximum value allowed. However, any argument that can produce unexpected object code (for example, TRIS 0 would evaluate to a NOP) generates an ERROR message rather than a WARNING.

Crossing page boundary — ensure page bits are set

MPASM is informing you that a page boundary has been crossed, and is recommending that you check to see that you have properly set the page bits. Please refer to the data sheet for your specific processor to find its memory boundaries.

... Is not currently supported

Any directive that is not currently supported generates this WARNING. Please look in the User's Guide for alternative directives, or contact your Microchip Field Applications Engineer for options.

LCALL should only be used for multi-paged program memory

You are using the LCALL opcode, when you should be using CALL instead. LCALL only applies for processors that have more than one page of program memory (such as the PIC16C57). LCALL uses 4 execution cycles, while CALL only uses 2. This is only a WARNING, and correct code is generated.

... May not be handled as preprocessor directive

The #DEFINE and #UNDEFINE directives generate this WARNING to inform you that these directives do not function exactly as you would expect them to operate in the C language. Please refer to the User's Guide descriptions for these directives to determine how they will behave.

... Not a single byte quantity

You have specified a literal value that is larger than 8-bits. This WARNING is produced when you use an opcode that requires a single-byte value (such as MOVLW). MPASM truncates to the lower 8 bits of the value. You may be more specific by preceding the value with a HIGH or LOW operator.

This number is being treated as a binary representation

The specified number is ambiguous, and could be interpreted as either a binary or hexadecimal number. MPASM is assuming that it is binary. Example: b0101.

MPASM expects hexadecimal numbers to be represented as 0xb0101 or as H'b0101'.

Quick Reference Guide

Quick Reference Guide

This quick reference guide is supplied to give you all of the instructions for the Microchip family of microcontrollers, including their description, function and status bits modified.

If more information is required, please refer to the data sheets for the PIC16/17 in question.

Highlights

- Directive Summary
- PIC16C5X Instruction Set and Notes
- PIC16CXX Instruction Set and Notes
- PIC17CXX Instruction Set and Notes

Terms

PIC16C5X

Microchip's low-end 8-bit microcontroller with 12-bit wide instruction set currently including: PIC16C54, PIC16C55, PIC16C56, PIC16C57, and PIC16C58.

PIC16CXX

Microchip's mid-range 8-bit microcontroller with 14-bit wide instruction set currently including: PIC16C71, PIC16C64, and PIC16C84.

PIC17CXX

Microchip's high-end 8-bit microcontroller family with 16-bit wide instruction set currently including PIC17C42.

CUT HERE

TABLE 9: DIRECTIVE SUMMARY

Directive	Description	Syntax
CBANK		Future Feature
CBLOCK	Define a Block of Constants	cblock [<expr>]
CONSTANT	Declare Symbol Constant	constant <label>[=<expr>, ...,<label>[=<expr>]]
DATA	Create Numeric and Text Data	data <expr>,[,<expr>,...,<expr>]DATA "text_string>[,"<text_string>",...]
DB	Declare Data of One Byte	db <expr>[,<expr>,...,<expr>]
#DEFINE	Define a Text Substitution Label	define <name> [<value>} define <name> [<arg>,...,<arg>] <value>
DW	Declare Data of One Word	dw <expr>[,<expr>,...,<expr>]
ELSE	Begin Alternative Assembly Block to IF	else
END	End Program Block	end
ENDC	End an Automatic Constant Block	endc
ENDIF	End conditional Assembly Block	endif
ENDM	End a Macro Definition	endm
ENDW	End a While Loop	endw
EQU	Define an Assembly Constant	<label> equ <expr>
ERROR	Issue an Error Message	error "<text_string>"
EXITM	Exit from a Macro	exitm
EXPAND	Expand Macro Listing	expand
FILL	Specify Memory Fill Value	fill <expr>
IF	Begin ConditionallyAssembled Code Block	if <expr>
IFDEF	Execute If Symbol has Been Defined	ifdef <label>
IFNDEF	Execute If Symbol has not Been Defined	ifdef <label>
INCLUDE	Include Additional Source File	include <<include_file>> "<include_file>"
LIST	Listing Options	list [<list_option>,...,<list_option>]
LOCAL	Declare Local Macro Variable	local <label>[,<local>]
MACRO	Declare Macro Definition	label macro [<arg>,...,<arg>]
MESSG	Create User Defined Message	messg "<message_text>
NOEXPAND	Turn off Macro Expansion	noexpand
NOLIST	Turn off Listing Output	nolist
ORG	Set Program Origin	<label> org <expr>
PAGE	Insert Listing Page Eject	page
PROCESSOR	Set Processor Type	processor <processsor_type>
RADIX	Specify Default Radix	radix <default_radix>
RES	Reserve Memory	res <mem_units>
SET	Define an Assembler Variable	<label> set <expr>
SPACE	Insert Blank Listing Lines	space <expr>
SUBTITLE	Specify Program Subtitle	subtitl "<sub_text>"
TITLE	Specify Program Title	title "<title_text>"
#UNDEFINE	Delete a Substitution Label	#undefine <label>
VARIABLE	Declare Symbol Variable	variable <label>[=<expr>,...,<label>[=<expr>]]

WHILE	Perform Loop While Condition is True	while <expr>
		⋮
		endw

PIC16C5X Instruction Set

All instructions execute in a single instruction cycle unless otherwise noted. Any unused opcode is executed as a NOP. The instruction set is highly orthogonal and is grouped into three basic catagories:

- Byte Oriented operations
- Bit Oriented Operations
- Literal and Control Operations

The following tables list the instructions recognized by the MPASM assembler, where:

TABLE 10: PIC16C5X OPERAND CODES

Field	Description
f	Register file address (0x00 to 0xFF)
W	Working register (accumulator)
b	Bit address within an 8 bit file register
k	Literal field, constant data or label.
x	Don't care location.
d	Destination select; d = 0: store result in W (f0A), d = 1: store result in file register f. Default is d = 1.

TABLE 11: PIC16C5X BYTE ORIENTED FILE REGISTER OPERATIONS

Binary	Hex	Mnemonic		Description	Function	Bits	Notes
0001 11df ffff	1Cf	ADDWF	f,d	Add W and f	W+f→d	c dc z	1,2,4
0001 01df ffff	14f	ANDWF	f,d	AND W and f	W.AND.f →d	z	2,4
0000 011f ffff	06f	CLRF	f	Clear f	0 → f	z	4
0000 0110 0000	040	CLRW		Clear W	0 → W	z	
0010 01df ffff	24f	COMF	f,d	Complement f	f → d	z	2,-1
0000 11df ffff	0Cf	DECF	f,d	Decrement f	$\overline{f}$ - 1 → d	z	2,4
0010 11df ffff	2Cf	DECFSZ	f,d	Decrement f, skip if zero	f - 1 → d,skip if zero	None	2,4
0010 10df ffff	28f	INCF	f,d	Increment f	f + 1 → d	z	2,4
0011 11df ffff	3Cf	INCFSZ	f,d	Increment f, skip if zero	f + 1 → d,skip if zero	None	2,4
0001 00df ffff	10f	IORWF	f,d	Inclusive OR W and f	Wvf → d	z	2,4
0010 00df ffff	20f	MOVF	f,d	Move f	f → d	z	2,4
0000 001f ffff	02f	MOVWF	f	Move W to f	W → f	None	1,4
0000 0000 0000	000	NOP		No operation		None	
0011 01df ffff	34f	RLF	f,d	Rotate left f	f<n> → d<n+1>, C → d<0>, f<7> → C	c	2,4
0011 00df ffff	30f	RRF	f,d	Rotate right f	f<n> →d<n-1>,C →d<7>, f<0> → C	c	2,4
0000 10df ffff	08f	SUBWF	f,d	Subtract W from f	f - W →d[f+ $\overline{w}$ +1→d]	c dc z	1,2,4
0011 10df ffff	38f	SWAPF	f,d	Swap halves f	f<0:3> ↔ f<4:7> →d	None	2,4
0001 10df ffff	18f	XORWF	f,d	Exclusive OR W and f	W ⊕ f → d	z	2,4

TABLE 12: PIC16C5X BIT ORIENTED FILE REGISTER OPERATIONS

Binary	Hex	Mnemonic		Description	Function	Bits	Notes
0110 bbbf ffff	4bf	BCF	f,b	Bit clear f	0 →f(b)	None	2,4
0101 bbbf ffff	5bf	BSF	f,b	Bit set f	1 →f(b)	None	2,4
0110 bbbf ffff	6bf	BTFSC	f,b	Bit test, skip if clear	skip if f(b) = 0	None	
0111 bbbf ffff	8bf	BTFSS	f,b	Bit test, skip if set	skip if f(b) = 1	None	

TABLE 13: PIC16C5X LITERAL AND CONTROL OPERATIONS

Binary	Hex	Mnemonic	Description	Function	Bits	Notes
1110 kkkk kkkk	Ekk	ANDLW k	And literal and W	k & W → W	z	
1001 kkkk kkkk	9kk	CALL k	Call subroutine	PC + 1→TOS, k →PC	None	1
0000 0000 0100	004	CLRWDT	Clear watch dog timer	0 → WDT (and Prescaler if assigned)	to pd	
101k kkkk kkkk	Akk	GOTO k	Goto address (k is nine bits)	k→PC(9 bits)	None	
1101 kkkk kkkk	Dkk	IORLW k	Incl. OR literal and W	kvW → W	z	
1100 kkkk kkkk	Ckk	MOVLW k	Move Literal to W	k → W	None	
0000 0000 0010	002	OPTION	Load OPTION register	W → OPTION Register	None	
1000 kkkk kkkk	8kk	RETLW k	Return with literal in W	k → W, TOS → PC	None	
0000 0000 0011	003	SLEEP	Go into stand by mode	0 → WDT, stop oscillator	to pd	
0000 0000 0fff	00f	TRIS f	Tristate port f	W → I/O control reg f	None	3
1111 kkkk kkkk	Fkk	XORLW k	Exclusive OR literal and W	k ⊕ W → W	z	

PIC16C5X Notes

1. If the destination of any instruction is the program counter (register file 2), the 8-bit destination value will be loaded into the lower 8-bits of the program counter (PC) and the 9th bit of the PC will be cleared. For the PIC16C56 and PIC16C57, the upper 3 bits of the status register (register file 3), PA2:PA0, are loaded into the most significant 3 bits of the PC(11:9). In case of the GOTO instruction, the lower 9 bits of the PC are loaded with the destination address, and the 3 most significant bits of the PC(11:9) are loaded with PA2:PA0 from the status register.

2. When an I/O register is modified as a function of itself (i.e. `MOVF 6,1`) the value used will be the value present on the pins themselves. For example, a tristated pin with data latch "1" but is driven low by an external device will be relatched in the low state.

3. The instruction "`TRIS f`", where `f = 5, 6`, or `7` causes the contents of the `W` register to be written to the tristate latches of the specified file (port). A one forces the pin to a high impedance state and disables the output buffers.

4. If this instruction is executed on file register `f1` (and, where applicable `d=1`), the prescaler will be cleared if assigned to the `RTCC`.

PIC16CXX Instruction Set

The PIC16CXX instruction set consists of 36 instructions, each a single 14-bit wide word. Most instructions operate on a file register, `f`, and the working register, `W` (accumulator). The result can be directed either to the file register or the `W` register or to both in the case of some instructions. A few instructions operate solely on a file register (`BSF` for example).

All instructions execute in a single instruction cycle unless otherwise noted. Any unused opcode is executed as a NOP. The instruction set is highly orthogonal and is grouped into three basic catagories:

- Byte Oriented operations
- Bit Oriented Operations
- Literal and Control Operations

The following tables list the instructions recognized by the MPASM assembler.

TABLE 14: PIC16CXX BYTE ORIENTED FILE REGISTER OPERATIONS

Binary	Hex	Mnemonic	Description	Function	Bits	Notes
00 0111 dfff ffff	07ff	ADDWF f,d	Add W and f	W+f→ d	c dc z	2,3
00 0101 dfff ffff	05ff	ANDWF f,d	AND W and f	W.AND.f →d	z	2,3
00 0001 1fff ffff	018f	CLRF f	Clear f	0 → f	z	3
00 0001 0xxx xxxx	0100	CLRW	Clear W	0 → W	z	
00 1001 dfff ffff	09ff	COMF f,d	Complement f	$\overline{\text{f}}$ → d	z	2,3
00 0011 dfff ffff	03ff	DECF f,d	Decrement f	f - 1 → d	z	2,3
00 1011 dfff ffff	0Bff	DECFSZ f,d	Decrement f, skip if zero	f - 1 → d,skip if zero	None	2,3
00 1010 dfff ffff	0Aff	INCF f,d	Increment f	f + 1 → d	z	2,3
00 1111 dfff ffff	0Fff	INCFSZ f,d	Increment f, skip if zero	f + 1 → d,skip if zero	None	2,3
00 0100 dfff ffff	04ff	IORWF f,d	Inclusive OR W and f	Wvf → d	z	2,3
00 1000 dfff ffff	08ff	MOVF f,d	Move f	f → d	z	2,3
00 0000 1fff ffff	008f	MOVW f	Move W to f	W → f	None	3
00 0000 0xx0 0000	0000	NOP	No operation		None	
00 1101 dfff ffff	0Dff	RLF f,d	Rotate left f	f<n>→d<n+1>, C→d<0>, f<7>→C	c	2,3
00 1100 dfff ffff	0Cff	RRF f,d	Rotate right f	f<n>→d<n-1>, C→d<7>, f<0>→C	c	2,3
00 0110 dfff ffff	02ff	SUBWF f,d	Sutract W from f	f - W →d[f+ $\overline{\text{W}}$ +1→d]	c dc z	2,3
00 1110 dfff ffff	0Eff	SWAPF f,d	Swap haves f	f<0:3> ↔ f<4:7>→d	None	2,3
00 0110 dfff ffff	06ff	XORWF f,d	Exclusive OR W and f	W ⊕ f -> d	z	2,3

TABLE 15: PIC16CXX BIT ORIENTED FILE REGISTER OPERATIONS

Binary	Hex	Mnemonic	Description	Function	Bits	Notes
01 00bb bfff ffff	1bff	BCF f,b	Bit clear f	0 → f(b)	None	2,3
01 01bb bfff ffff	1bff	BSF f,b	Bit set f	1 → f(b)	None	2,3
01 10bb bfff ffff	1bff	BTFSC f,b	Bit test, skip if clear	skip if f(b) = 0	None	
01 11bb bfff ffff	1bff	BTFSS f,b	Bit test, skip if set	skip if f(b) = 1	None	

TABLE 16: PIC16CXX LITERAL AND CONTROL OPERATIONS

Binary	Hex	Mnemonic	Description	Function	Bits	Notes
11 111x kkkk kkkk	3Ekk	ADDLW k	Add literal to W	k +W → W	c dc z	
11 1001 kkkk kkkk	39kk	ANDLW k	And literal and W	k & W → W	z	
10 0kkk kkkk kkkk	2kkk	CALL k	Call subroutine	PC + 1→ TOS, k→ PC	None	
00 0000 0110 0100	0064	CLRW T	Clear watch dog timer	0 → WDT (and Prescaler if assigned)	to pd	
10 1kkk kkkk kkkk	2kkk	GOTO k	Goto address (k is nine bits)	k→ PC(9 bits)	None	
11 1000 kkkk kkkk	38kk	IORLW k	Incl. OR literal and W	kvW → W	z	
11 00xx kkkk kkkk	30kk	MOVLW k	Move Literal to W	k → W	None	
00 0000 0110 0010	0062	OPTION	Load OPTION register	W → OPTION Register	None	1
00 0000 0000 1001	0009	RETFIE	Return from Interrupt	TOS →PC, 1 → GIE	None	
11 01xx kkkk kkkk	34kk	RETLW k	Return with literal in W	k → W, TOS → PC	None	
00 0000 0000 1000	0008	RETURN	Return from subroutine	TOS → PC	None	
00 0000 0110 0011	0063	SLEEP	Go into stand by mode	0 → WDT, stop oscillator	to pd	
11 110x kkkk kkkk	3Ckk	SUBLW k	Subtract W from literal	k - W → W	c dc z	
00 0000 0110 0fff	006f	TRIS f	Tristate port f	W → I/O control reg f	None	1
11 1010 kkkk kkkk	3Akk	XORLW k	Exclusive OR literal and W	k ⊕ W → W	z	

TABLE 17: PIC16CXX SPECIAL INSTRUCTION MNEMONICS

Name	Mnemonic	Equivalent Operation(s)		Status
Clear Carry	CLRC	BCF	3,0	-
Set Carry	SETC	BSF	3.0	-
Clear Digit Carry	CLRDC	BCF	3,1	-
Set Digit Carry	SETDC	BSF	3,1	-
Clear Zero	CLRZ	BCF	3,2	-
Set Zero	SETZ	BSF	3,2	-
Skip on Carry	SKPC	BTFSS	3,0	-
Skip on No Carry	SKPNC	BTFSC	3,0	-
Skip on Digit Carry	SKPDC	BTFSS	3,1	-
Skip on No Digit Carry	SKPNDC	BTFSC	3,1	-
Skip on Zero	SKPZ	BTFSS	3,2	-
Skip on Non Zero	SKPNZ	BTFSC	3,2	-
Test File	TSTF f	MOVF	f,1	Z
Move File to W	MOVFW f	MOVF	f,0	Z
Negate File	NEGF f,d	COMF INCF	f,1 f,d	Z
Add Carry to File	ADDCF f,d	BTFSC INCF	3,0 f,d	Z
Subtract Carry from File	SUBCF f,d	BTFSC DECF	3,0 f,d	Z
Add Digit Carry to File	ADDDCF f,d	BTFSC INCF	3,1 f,d	Z
Subtract Digit Carry from File	SUBDCF f,d	BTFSC DECF	3,1 f,d	Z
Branch	B k	GOTO	k	-
Branch on Carry	BC k	BTFSC GOTO	3,0 k	-
Branch on No Carry	BNC k	BTFSS GOTO	3,0 k	-
Branch on Digit Carry	BDC k	BTFSC GOTO	3,1 k	-
Branch on No Digit Carry	BNDC k	BTFSS GOTO	3,1 k	-
Branch on Zero	BZ k	BTFSC GOTO	3,2 k	-

TABLE 17: PIC16CXX SPECIAL INSTRUCTION MNEMONICS (CONT)

Name	Mnemonic	Equivalent Operation(s)	Status
Branch on Non Zero	BNZ k	BTFSS 3,2 GOTO k	
Call across page boundary	LCALL k	BCF 3,5 or BSF 3,5 BCF 3,6 or BSF 3,6 CALL k	

PIC16CXX Notes

1. TRIS and OPTION instructions are included in the instruction set for upward compatability with the PIC16C5X products. Microchip strongly recommends not using these instructions for new code development. Instead of using these instructions, directly address the TRIS and OPTION registers to obtain equivalent control. These instructions may not be supported in future PIC16CXX products.
2. When an I/O register is modified as a function of itself (i.e. `MOVF 6,1`) the value used will be the value present on the pins themselves. For example, a tristated pin with data latch "1" but is driven low by an external device will be relatched in the low state.
3. If this instruction is executed on file register f1 (and, where applicable d=1), the prescaler will be cleared if assigned to the RTCC.

PIC17C42 Instruction Set

The PIC17C42 instruction set consists of 55 instructions, each a single 16-bit wide word. Most instructions operate on a file register, f, and the working register, W (accumulator). The result can be directed either to the file register or the W register or to both in the case of some instructions. A few instructions operate solely on a file register (BSF for example).

All instructions execute in a single instruction cycle unless otherwise noted. Any unused opcode is executed as a NOP. The instruction set is highly orthogonal and is grouped into four basic catagories:

- Data Move Operations
- Arithmetic and Logical Operations
- Bit Manipulation Operations
- Special Control Operations

The following tables list the instructions recognized by the MPASM assembler, where:

TABLE 18: PIC17C42 OPERAND CODES

Field	Description
f	Register file address (0x00 to 0xFF)
p	Peripheral register file address (0x00 to 0x1f)
b	Bit address within an 8 bit file register
i	Table pointer control; i = 0: do not change, i = 1: increment after instruction execution.
t	Table byte select; t = 0: perform operation on lower byte, t = 1: peform operation on upper byte.
k	Literal field, constant data or label.
x	Don't care location.
d	Destination select; d = 0: store result in W (f0A), d = 1: store result in file register f. Default is d = 1.

TABLE 19: PIC17C42 DATA MOVE INSTRUCTIONS

Binary	Hex	Mnemonic	Description	Function	Bits	Notes
`011p pppp ffff ffff`	6pff	`MOVFP f,p`	Move f to p	f → p	None	4
`1011 1000 kkkk kkkk`	B8kk	`MOVLB k`	Move literal to BSR	k → BSR	None	
`010p pppp ffff ffff`	4pff	`MOVPF p,f`	Move p to f	p → w	Z	4
`0000 0001 ffff ffff`	01ff	`MOVWF f`	Move W to F	w → f	None	

`1010 10ti ffff ffff`	a8ff	`TABLRD t,i,f`	Read data from table latch into file f, then update table latch with 16-bit contents of memory location addressed by table pointer	TBLATH → f if t = 1,
				TBLATL → f if t = 0;
				ProgMem(TBLPTR)→TBLAT;
				TBLPTR+1→TBLPTR if i=1
					None	8,10

`11ti ffff ffff`	acff	`TABLWT t,i,f`	Write data from file f to table latch and then write 16-bit table latch to program memory location addressed by table pointer	f→TBLATH if t = 1,
				f→TBLATL if t = 0;
				TBLAT→ProgMem(TBLPTR);
				TBLPTR+1→TBLPTR if i=1
					None	6

`1010 00tx ffff ffff`	a0ff	`TLRD t,f`	Read data from table latch into file f (table latch unchanged)	TBLATH → f if t = 1 TBLATL → f if t = 0	None	
`1010 01tx ffff ffff`	a4ff	`TLWT t,f`	Write data from file f into table latch	f → TBLATH if t = 1 f → TBLATL if t = 0	None	

TABLE 20: PIC17C42 ARITHMETIC AND LOGICAL INSTRUCTIONS

Binary	Hex	Mnemonic	Description	Function	Bits	Notes
1011 0001 kkkk kkkk	b1kk	ADDLW k	Add literal to W	(W+k_ → W	ov c dc z	
0000 111d ffff ffff	0eff	ADDWF f,d	Add W to F	(W+f) → d	ov c dc z	
0001 000d ffff ffff	10ff	ADDWFC f,d	Add W and Carry to f	(W+f+C) → d	ov c dc z	
1011 0101 kkkk kkkk	b5kk	ANDLW k	AND Literal and W	(W.AND.k) → W	z	
0000 101d ffff ffff	0aff	ANDWF f,d	AND W with f	(W.AND.f) → d	z	
0010 100d ffff ffff	28ff	CLRF f,d	Clear f and Clear d	0x00 → f,0x00 → d	None	3
0001 001d ffff ffff	12ff	COMF f,d	Complement f	→ d	z	
0010 111d ffff ffff	2eff	DAW f,d	Dec. adjust W, store in f,d	W adjusted →f and d	c	
0000 011d ffff ffff	06ff	DECF f,d	Decrement f	(f - 1)→f and d	ov c dc z	
0001 010d ffff ffff	14ff	INCF f,d	Increment f	(f + 1)→f and d	ov c dc z	
1011 0011 kkkk kkkk	b3kk	IORLW k	Inclusive OR literal with W	(W.OR.k)→W	z	
0000 100d ffff ffff	08ff	IORWF f,d	Inclusive or W with f	(w.OR.f)→d	z	
1011 0000 kkkk kkkk	b0kk	MOVLW k	Move literal to W	k → W	None	
0010 110d ffff ffff	2cff	NEGW f,d	Negate W, store in f and d	(w + 1)→f,(w + 1)→d	ov c dc z	1,3
0001 101d ffff ffff	1aff	RLCF f,d	Rotate left through carry	f<n>→d<n+1>,f<7>→C, C→d<0>	c	
0010 001d ffff ffff	22ff	RLNCF f,d	Rotate left (no carry)	f<n>→d<n+1>, f<7>→d<0>	None	
0001 100d ffff ffff	18ff	RRCF f,d	Rotate right through carry	f<n>→d<n-1>,f<0>→c, c→d<7>	None	
0010 000d ffff ffff	20ff	RRNCF f,d	Rotate right (no carry)	f<n> →d<n-1>, f<0> →d<7>	None	
0010 101d ffff ffff	2aff	SETF f,d	Set f and Set d	0xff→f,0xff→d	None	3
1011 0010 kkkk kkkk	b2kk	SUBLW k	Subtract W from literal	(k-w)→w	ov c dc z	
0000 010d ffff ffff	04ff	SUBWF f,d	Subtract W from f	(f-w)→d	ov c dc z	1
0000 001d ffff ffff	02ff	SUBWFB f,d	Subtract from f with borrow	(f-w-c)→d	ov c dc z	1
0001 110d ffff ffff	1cff	SWAPF f,d	Swap f	(f<0:3> → d<4:7>, f<4:7> → d<0:3>	None	
1011 0100 kkkk kkkk	b4kk	XORLW k	Exclusive OR literal with W	(W.XOR.k)→w	z	
0000 110d ffff ffff	0cff	XORWF f,d	Exclusive OR W with f	(W.XOR.f)→d	z	

TABLE 21: PIC17C42 PROGRAM CONTROL INSTRUCTIONS

Binary	Hex	Mnemonic	Description	Function	Bits	Notes
111k kkkk kkkk kkkk	ekkk	CALL k	Subroutine call (within 8k page)	PC+1 → TOS,	None	8
0011 0001 ffff ffff	31ff	CPFSEQ f	Compare f/w, skip if f = w	f-W, skip if f = W	None	7
0011 0010 ffff ffff	32ff	CPFSGT f	Compare f/w, skip if f > w	f-W, skip if f > W	None	2,7
0011 0000 ffff ffff	30ff	CPFSLT f	Compare f/w, skip if f< w	f-W, skip if f < W	None	2,7
0001 011d ffff ffff	16ff	DECFSZ f,d	Decrement f, skip if 0	(f-1)→ d, skip if 0	None	7
0010 011d ffff ffff	26ff	DCFSNZ f,d	Decrement f, skip if not 0	(f-1)→d,skip if not 0	None	7
110k kkkk kkkk kkkk	ckkk	GOTO k	Unconditional branch (within 8k)	k → PC<12:0> k<12:8> → f3<4:0>, PC<15:13> → f3<7:5>	None	8
0001 111d ffff ffff	1eff	INCFSZ f,d	Increment f, skip if zero	(f+1)→d,skip if 0	None	7
0010 010d ffff ffff	24ff	INFSNZ f,d	Increment f, skip if not zero	(f+1)→d,skip if not 0	None	7
1011 0111 kkkk kkkk	b7kk	LCALL k	Long Call (within 64k)	(PC+1)→ TOS	None	5,8
0000 0000 0000 0101	0005	RETFIE	Return from interrupt, enable interrupt	(f3) → PCH:k → PCL "0" → GLINTD	GLINTD	8
1011 0110 kkkk kkkk	b6kk	RETLW k	Return with literal in W	k → W, TOS → PC, (f3 unchanged)	None	8
0000 0000 0000 0010	0002	RETURN	Return from subroutine	TOS → PC(f3 unchanged)	None	8
0011 0011 ffff ffff	33ff	TSTFSZ f	Test f, skip if zero	skip if f = 0	None	7

TABLE 22: PIC17C42 BIT HANDLING INSTRUCTIONS

Binary	Hex	Mnemonic	Description	Function	Bits	Notes
1000 1bbb ffff ffff	8bff	BCF f,b	Bit clear f	0 → f(b)	None	4
1000 0bbb ffff ffff	8bff	BSF f,b	Bit set f	1 → f(b)	None	4
1001 1bbb ffff ffff	9bff	BTFSC f,b	Bit test, skip if clear	skip if f(b) = 0	None	4,7
1001 0bbb ffff ffff	9bff	BTFSS f,b	Bit test, skip if set	skip if f(b) = 1	None	4,7
0011 1bbb ffff ffff	3bff	BTG f,b	Bit toggle f	f($\overline{b}$) → f(b)	None	4

TABLE 23: PIC17C42 SPECIAL CONTROL INSTRUCTIONS

Binary	Hex	Mnemonic	Description	Function	Bits	Notes
0000 0000 0000 0100	0004	CLRWT	Clear watch dog timer	0→WDT,0→WDT prescaler, 1→$\overline{\text{PD}}$, 1 →$\overline{\text{TO}}$	$\overline{\text{PD}}$, $\overline{\text{TO}}$	
0000 0000 0000 0100	0000	NOP	No operation	None	None	
0000 0000 0000 0011	0003	SLEEP	Enter sleep mode	Stop oscillator, power down, 0->WDT, 0->WDTPrescaler 1->$\overline{\text{PD}}$, 1-> $\overline{\text{TO}}$	$\overline{\text{PD}}$, $\overline{\text{TO}}$	

PIC17C42 Notes

1. 2's complement arithmetic
2. Unsigned arithmetic
3. If d=1, only the file is affected; if d=0, both W and the file are affected; if only W is required to be affected, then f=0ah (File 0ah) must be defined.
4. The hex representation is not accurate. The value of the bit to be modified has to be incorporated into the third digit.
5. During an LCALL, the contents of File 03h are loaded into the MSB of the PC and kkkk kkkk is loaded into File 02h, the LSB of the PC.
6. Multiple cycle instructions for EPROM programming when table pointer selects internal EPROM. The instruction is terminated by an interrupt event. When writing to external program memory, it is a two cycle instruction.
7. Two cycle instructions when condition is TRUE, else single cycle instruction.
8. Two cycle instruction, except for TABLRD to File 02h (Program Counter low byte) in which case it takes 3 cycles.
9. A skip means that instructions fetched during execution of current instruction are not executed. Instead, a NOP is executed.
10. Any instruction that writes to PCL (File 02h) is a two cycle instruction, except for TABLRD to File 02h, which is a three cycle instruction.

NOTES:

Index

Index

It is our intention to provide you with the best documentation possible to ensure successful use of your Microchip product. If you wish to provide your comments on organization, clarity, subject matter, and ways in which our documentation can better serve you, please FAX your comments to the Technical Publications Manager at (602) 786-7578.

Please list the following information, and use this outline to provide us with your comments about this Data Sheet.

To: Technical Publications Manager ______ Total pages sent ______

RE: Reader Response

From: Name ______

Company ______

Address ______

City / State / ZIP / Country ______

Telephone: (____) ______ - ______ FAX: (____) ______ - ______

Application (optional): ______ Would you like a reply? ___Y ___ N

Product: MPASM User's Guide Literature Number: DS33014C

Questions:

1. What are the best features of this document? ______

2. How does this document meet your hardware and software development needs? ______

3. Do you find the organization of this document easy to follow? If not, why? ______

4. What additions to the document do you think would enhance the structure and subject matter?

5. What deletions from the document could be made without affecting the overall usefulness?

6. Is there any incorrect or misleading information (what and where)? ______

7. How would you improve this document? ______

8. How would you improve our software, systems, and silicon products? ______

WORLDWIDE SALES & SERVICE

AMERICAS

Corporate Office
Microchip Technology Inc.
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 602 786-7200 Fax: 602 899-9210

Atlanta
Microchip Technology Inc.
1521 Johnson Ferry Road NE, Suite 170
Marietta, GA 30062
Tel: 404 509-8800 Fax: 404 509-8600

Boston
Microchip Technology Inc.
Five The Mountain Road, Suite 120
Framingham, MA 01701
Tel: 508 820-3334 Fax: 508 820-4326

Chicago
Microchip Technology Inc.
665 Tollgate Road, Unit C
Elgin, IL 60123-9312
Tel: 708 741-0171 Fax: 708 741-0638

Dallas
Microchip Technology Inc.
14651 Dallas Parkway, Suite 816
Dallas, TX 75240-8809
Tel: 214 991-7177 Fax: 214 991-8588

AMERICAS (continued)

Los Angeles
Microchip Technology Inc.
18201 Von Karman, Suite 455
Irvine, CA 92715
Tel: 714 263-1888 Fax: 714 263-1338

New York
Microchip Technology Inc.
150 Motor Parkway, Suite 416
Hauppauge, NY 11788
Tel: 516 273-5305 Fax: 516 273-5335

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

ASIA/PACIFIC
Microchip Technology Inc.
Unit No. 3002-3004, Tower 1
Metroplaza
223 Hing Fong Road
Kwai Fong, N.T. Hong Kong
Tel: 852 401 1200 Fax: 852 401 3431

EUROPE

United Kingdom
Arizona Microchip Technology Ltd.
Unit 6, The Courtyard
Meadow Bank, Furlong Road
Bourne End, Buckinghamshire SL8 5AJ
Tel: 44 0628 851 077 Fax: 44 0628 850 259

Germany
Arizona Microchip Technology GmbH
Gustav-Heinemann-Ring 125
81739 Muenchen, Germany
Tel: 49 089 627144 0 Fax: 49 089 627144 44

France
Arizona Microchip Technology SARL
2, Rue Du Buisson aux Fraises
F-91300 Massy, France
Tel: 33 01 6930 9090 Fax: 33 01 6930 9079

Italy
Arizona Microchip Technology SRL
Centro Direzionale Colleoni
Palazzo Pegaso Ingresso No. 2
Via Paracelso 23, 20041 Agrate Brianza (MI) Italy
Tel: 39 039 68 99 939 Fax: 39 039 68 99 883

JAPAN
Microchip Technology International Inc.
Shinyokohama Gotoh Bldg. 8F, 3-22-4
Shinyokohama, Kohoku-Ku, Yokohama-Shi
Kanagawa 222 Japan
Tel: 81 45 471 6166 Fax: 81 45 471 6122

MICROCHIP

7/01/94

*Information contained in this publication regarding device applications and the like is intended by way of suggestion only. No representation or warranty is given and no liability is assumed by Microchip Technology Inc. with respect to the accuracy or use of such information. Use of Microchip's products as critical components in life support systems is not authorized except with express written approval by Microchip. The Microchip logo and name are trademarks of Microchip Technology Incorporated. All rights reserved. All other trademarks mentioned herein are the property of their respective companies.

MICROCHIP

Microchip Technology Inc.
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 602.786.7200 Fax: 602.899.9210