

A brief introduction to Assembly Programming for the PIC18 microcontroller

Objectives: To provide a pre-Lab introduction to the PIC18's Assembly Language programming and the procedures involved in writing and assembling an assembly program. We will also look at the addressing modes of the PIC18 and examine some of the most commonly used Mnemonics with examples.

To introduce the student to the Assembler Directives. This will aid in the understanding in the PIC18 experimental work that follows.

An introduction to the Assembler

The only language that the computer understands is **machine code**, which can be considered as a pattern of bits, or as list of numbers, in its memory. A program in this form is hard for a human either to understand or write. Consequently, programs are more usually written in **Assembly language**, which directly corresponds to the machine code, but makes use of **Mnemonic** names for instructions and registers. It also allows the programmer to use symbolic names for addresses within the program, and for other values. A program called an **Assembler** is used to translate from the assembly language into machine language. A program is composed of a series of steps called **Instructions**. Each instruction is written as one line of assembly language. The instruction itself has a mnemonic name of three or more letters, and for some instructions the name is all that need be written on the line. For example,

NOP

is an instruction that does nothing at all! (Such an instruction is not completely pointless: it can be used for debugging purposes or when very short delays are required).

For most instructions the name alone is not sufficient, as we must also specify which registers are to be used or where in memory are the data on which they are to operate. This is done by putting the **operand** after the name (with one or more spaces in between) e.g.

MOVLW F2h ; Load hex value F2 into the Working Register

Anything else used on the line after the instruction and it's operands is ignored by the assembler and is normally preceded by a semicolon (;) This allows the use of comments in the program in order to make it easily understood.

e.g.:

MOVLW F2h ; Load hex value F2 into Working Register (highlighted text is the comment)

If a line starts with a semicolon (;), the whole line is treated as a comment e.g.:

MOVLW F2h ; Load hex value F2 into the Working register
** This is a comment line*

In the previous examples instruction names are written indented from the left-hand margin. If a line of assembly does not start with a space, the first item is taken to be a **label**, which is a symbolic name for the memory address at that location. The name of the label must be a legal symbolic name consisting of from one to nine upper or lower case characters, decimal digits, dollar, period or underline characters. The assembler remembers the label and the address which it refers to, and this label may be used somewhere else in the program to refer to that address. This is particularly useful with jump and branch instructions:

```
START  MOVLW      F2h          ; Load hex value F2 into the Working register
;This is a comment line
      GOTO        START        ; Go to location START
```

The GOTO START instruction will instruct the processor to go back and execute the MOVLW 0xF2 instruction. By using the label we do not have to know the actual address of this instruction. If modifications are made to the program we will not have to modify the GOTO instruction every time.

The Microchip MPASM Assembler

The MPASM assembler is a free PC based application that provides a development platform for assembly language coding for the PIC family of micros. It provides a universal solution for developing assembly code for the complete range of Microchip devices. Overall its features are as follows:

- Support for all PIC micro instruction sets
- Command Line Interface
- Command Shell Interfaces
- Rich Directive Language
- Flexible Macro Language
- MPLAB compatibility

Two versions are available. The PC Windows based version **MPASMWIN** and command line version **MPASM**. The MPASMWIN version as the name shows is for the Windows platform while MPASM is the command line version. Please note that the MPLAB integrates the assembler in the IDE so there is no need to run the assembler as a stand-alone application (see later).

The assembler translates assembly language source programs to PIC machine code (**Object code**) which can be subsequently downloaded into the Target board. It also generates an error file (.err) that show the various syntax errors if any.

Writing and testing of assembly language programs using the assembler involves a basic edit- assemble - test cycle. A summary of the steps involved in the assembly language development process is as follows:

1. Create a source program file using a text editor.
2. Run the assembler to translate the source file to a machine code (object .o) module.
3. Use the text editor to correct any errors reported by the assembler
4. Download the object code in the target board and program the device.
5. Run and test the program on the target system or simulate the program using the MPLAB IDE
6. If the program has 'bugs', use the text editor to correct the errors in the source file then go to step 2.

Steps 1-4 can be integrated in one software package such as MPLAB (see later).

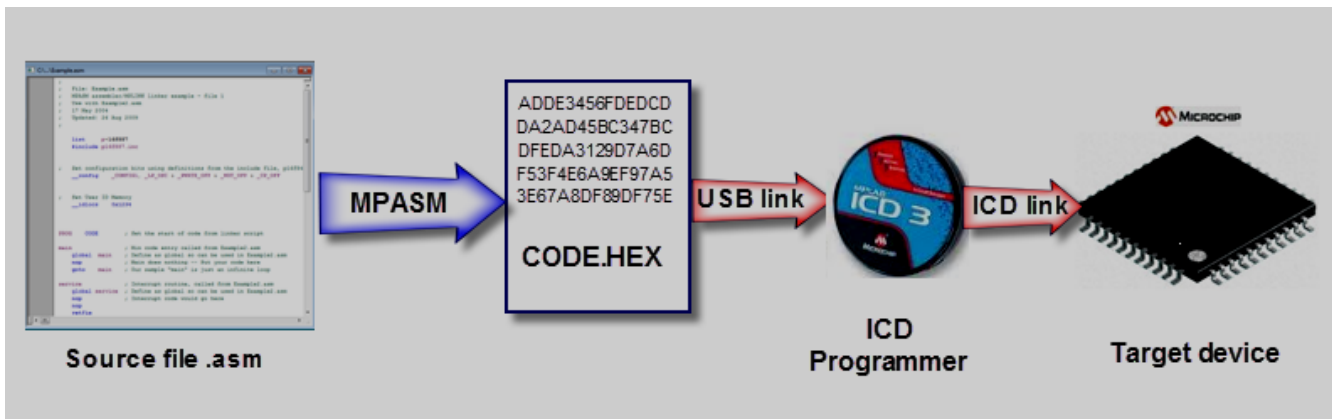


Fig.1 Development environment

Features of the Assembler

Assembly language instructions are divided into a number of fields as shown in the table below:

1. A label field
2. An operation field or Mnemonic field
3. An operand field
4. A comment field

The **Label field** begins in the first character position of the line. Some assembler directives (see below) require a label field. Label must start with an alpha character or underscore. Labels may be up to 32 characters long. By default they are case sensitive

The first character of the source line must be a space or tab if the line does not contain a label.

The **Operation field** or **Mnemonic** field specifies the machine language instruction or Assembler directive (see below). It immediately follows the label field. It is separated by the previous field by one or more spaces.

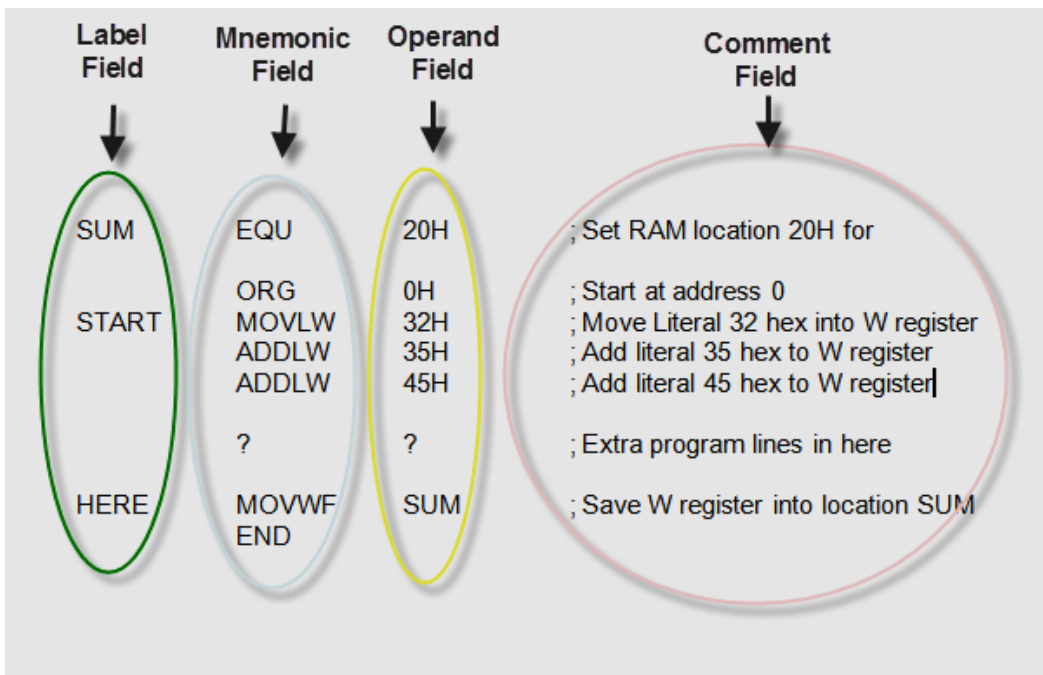
Instruction mnemonics can be in either uppercase or lowercase characters.

The **Operand field** follows the instruction field. They must be separated by at least one space or tab character. Multiple operands must be separated by commas. Some instructions don't use operands.

The **Comment field** is optional and is used to facilitate the readability of the program.

The assembler program listing below shows a typical assembly language format.

SUM	EQU	20H	; Set RAM location 20H for storage
	ORG	0H	; Start at address 0
START	MOVLW	32H	; Move Literal 32 hex into W register
	ADDLW	35H	; Add literal 35 hex to W register
	ADDLW	45H	; Add literal 45 hex to W register
	?	?	; Extra program lines in here
HERE	MOVWF	SUM	; Save W register into location SUM
	END		; End of Assembly code



Assembler Directives

Some assembly language instructions are not directly translated into machine language instructions. These instructions are directives to the assembler: they assign the program to certain areas in memory, define symbols, designate areas of memory for data storage, place tables or fixed data in memory, allow reference to other programs etc.

MPASM provide five types of directives:

- **Control directives.** These permit sections of conditionally assembled code.
- **Data directives:** these control the allocation of memory and allows for the data to be represented symbolically.
- **Listing directives:** these control the assembler's listing format such as titles, pagination etc.
- **Macro directives :** these control the execution and data allocation within the macro body definition
- **Object directives:** used when creating an object file.

The Include File: An assembler include, or **header file** is a file that contains valid assembly code. This file usually contains device-specific register assignment. It could be included in the code and could be re-used by many programs. As an example we could include the standard header file for the PIC18F4520 micro by using the following include statement:

```
#include p18f4520.inc
```

The following Table 1 shows some of the most common assembler directives.

Assembler Directive	Action Summary
list	Select listing option (also radix and processor type)
#include	Include additional source file
org	Set the program origin
equ	Assign a value to a label
#pragma config	Defines states of PIC18 configuration bits
_config	Configuration of various fuse settings (see later)
cblock	Define a Block Constant
end	End of program block

Table 1 some of the most common directives

A full table of the MPASM are given in the appendix.

The number representation in MPASM

MPASM can accept the full range of different number bases such as binary, hexadecimal, decimal and octal as well as ASCII values. The user can select a default setting for example hex and the numbers entered can be interpreted as such. Hexadecimal is the default radix, therefore the number base does not need to be explicitly specified. However it makes good programming sense to use the appropriate radix indicators on your code. Table 2 below shows the number presentation used in MPASM.

Radix	Number Representation example
Hex	H56, or 56H or 0x56
Decimal	D `86`
Binary	B `01010110`
Octal	O `126`
ASCII	`C` or A `C`

Table 2 MPASM Radix representation

NOTE : If the hex number used starts with A,B, C,D,E or F then the number must be preceded with a zero otherwise the hex number might be interpreted as a label. For example the number C3H must be entered as 0xC3.

Data Addressing Modes

While the program memory could be addressed in only one way by using the program counter, information stored in the data memory space can be addressed in various ways. These are as specified by the addressing modes.

Some of the instructions use only one addressing mode which is fixed. However other instructions may use up to three addressing modes depending on the operands used and whether the extended instruction set is enabled. The addressing mode for the PIC18 are :

- Inherent
- Literal
- Direct
- Indirect

The additional ***Indexed Literal Offset*** addressing mode is available when the extended instruction set is enabled.

Inherent and Literal Addressing

Some of the PIC18 instructions do not need any arguments. This addressing mode is known as ***Inherent*** addressing. Examples of this addressing mode include NOP, SLEEP, RESET etc.

Other instructions work in a similar way to the above but they also need an additional argument in the code. This is known as ***Literal*** addressing mode because they require an argument which is a literal value. Examples of this addressing mode include ADDLW and MOVLW. The ADDLW adds a literal value supplied as the argument into the W register (working register). The second example moves/loads the literal value specified into the W register.

Examples of literal addressing:

MOVLW 0x3F ;Load **L**iteral value 3F hex into W register

As already mentioned the PIC18 has a 16-bit instruction data bus a graphical representation of the above instruction is as shown in Fig. 3 below:

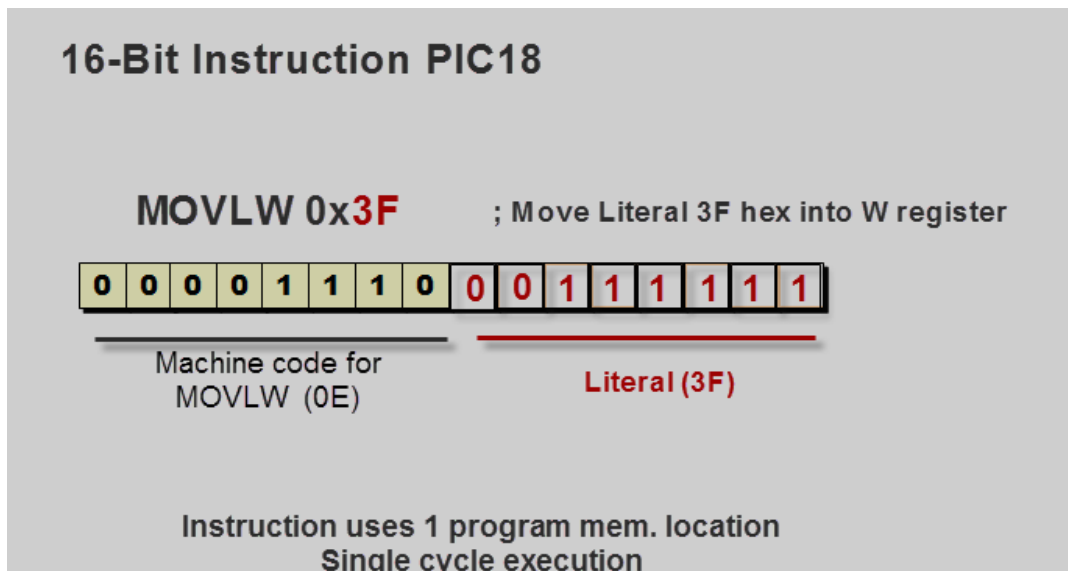


Fig.3 Representation of Literal addressing

Direct Addressing

In the direct addressing mode the operand data is in a RAM memory location of a known address. This address is given as part of the instruction. Contrast this Literal addressing mode whereby the operand data itself is provided with the instruction as shown in Fig.3 above. In the example shown below the letter L in the literal addressing mode has been replaced by 'F' in the instruction. This signifies the address of the File Register location.

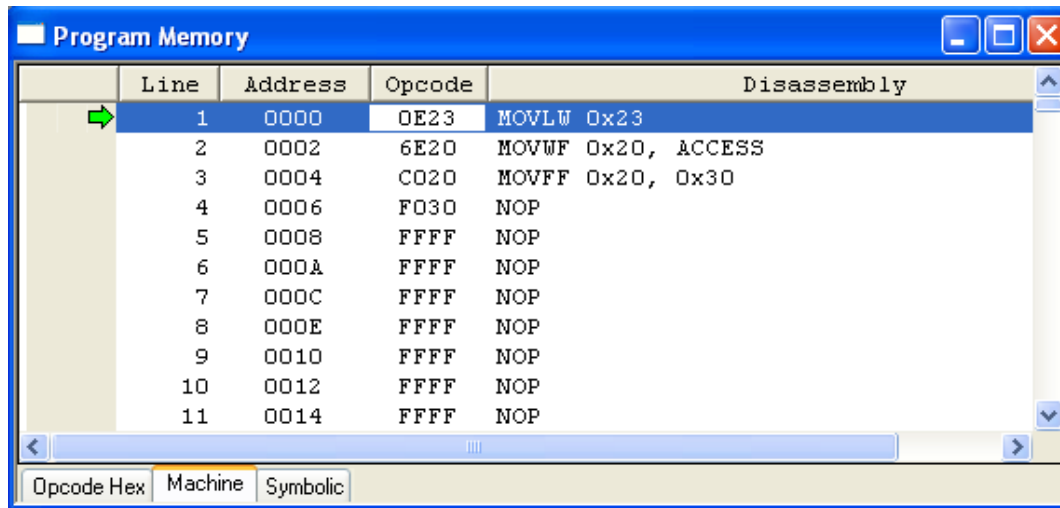
Example: Move contents of source file register Fs to destination register Fd

MOVFF Fs,Fd ;means Fd ← [Fs]

Instruction 1	MOVLW	0x23	; copy literal hex 23 into W register
Instruction 2	MOVWF	0x20	; copy W reg into File Register RAM hex loc 20
Instruction 3	MOVFF	0x20, 0x30	; copy data from location 20 (hex) to hex loc 30

The first instruction uses literal addressing mode while the last two instructions use direct addressing mode.

If we now probe further into the Opcode for the above instructions we have the following:
Looking at the program memory where the code resides we see the following hex data.

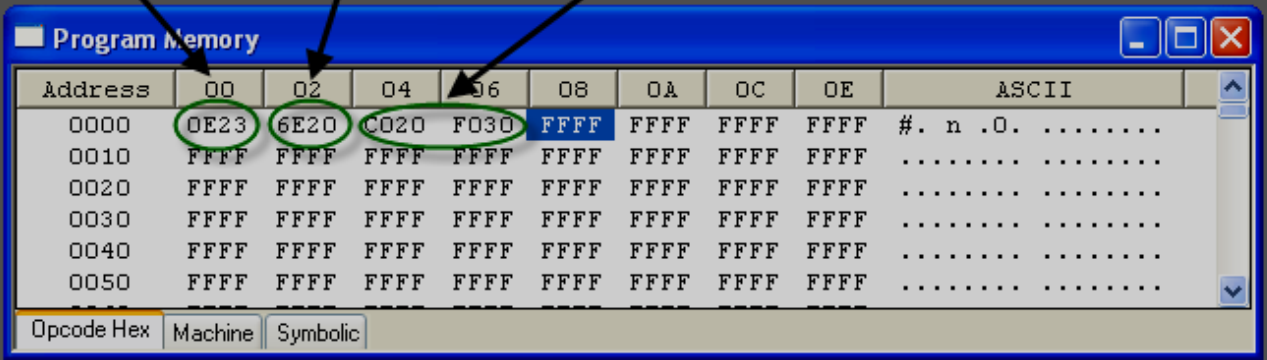


Line	Address	Opcode	Disassembly
1	0000	0E23	MOVLW 0x23
2	0002	6E20	MOVWF 0x20, ACCESS
3	0004	C020	MOVFF 0x20, 0x30
4	0006	F030	NOP
5	0008	FFFF	NOP
6	000A	FFFF	NOP
7	000C	FFFF	NOP
8	000E	FFFF	NOP
9	0010	FFFF	NOP
10	0012	FFFF	NOP
11	0014	FFFF	NOP

Opcode Hex Machine Symbolic

Fig.4 Program memory showing disassembly of previous example

MOVLW 0x23
MOVWF 0x20
MOVFF 0x20,0x30



Address	00	02	04	06	08	0A	0C	0E	ASCII
0000	0E23	6E20	C020	F030	FFFF	FFFF	FFFF	FFFF	#. n .0.
0010	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0020	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0030	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0040	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0050	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF

Opcode Hex Machine Symbolic

Program memory shows opcode for previous assembly listing

Fig.5 Hex display of Opcode

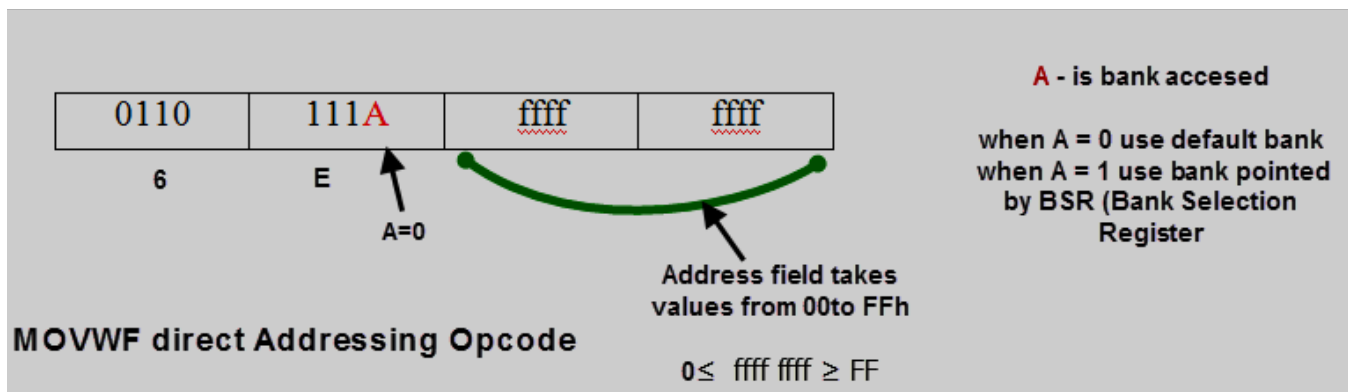


Fig.6 Direct Addressing Opcode for the MOVWF instruction

Indirect Addressing

In the indirect addressing mode a register is used as a pointer to the data RAM location. The registers used for this purpose are the FSRs (File Select Registers) 0, 1 and 2. The FSRs are a 12-bit registers that allow access to the entire 4k bytes of the data RAM space of the PIC18. To do this we use the LFSR (load FSR) instruction to load the RAM address. In other words we use the FSR registers as pointers. An example is as shown below:

LSFR 0, 0x40 ; load File Select Register 0 with hex 40 i.e. load the pointer SFR0 with 40 hex

FSRs are very useful in implementing data structures such as arrays and tables in data memory. As already mentioned the FSR registers are 12-bit registers and as such they cannot fit into the SFR address space unless they are split into 8-bit chunks. This is done by having the FSRs split into a low byte and a high byte parts called FSRxL and FSRxH where x is 0-2. Note that FSRxH is only a 4-bit register and that the 4 most significant bits are not used. Another register associated with the indirect addressing mode is the **Indirect Register INDF**. Each of the FSRs has an INDF register associated with it. Therefore we have INDF0, INDF1, and INDF2. When we move data into INDFs we are moving data into RAM location pointed by the FSR. Similarly when we are reading data from the INDFs we are reading data from RAM locations pointed to by the FSR.

The Linker MPLINK

MPLINK is integrated with MPLAB. It is a linker for the relocatable assembler MPASM. It is also used with the Microchip MCC18 compiler (see later).

It performs the following functions:

- **Locates Code and Data:** takes as input relocatable source files and by using the linker script it decides on the location of the code in program memory and the location of variables in RAM.
- **Resolves addresses:** external references in the source file(s) generate relocation entries in the object file. It uses relocation information to update all external references with actual addresses.
- **Generates an Executable:** it produces the .HEX that can be used to program the micro.
- **Identifies Address conflicts:** ensures that allocated areas of program and data occupy space that has not been already assigned.
- **Configures Stack Size and Location:** allocates RAM space for dynamic stack usage.
- **Provides Symbolic Debug Information:** creates a file the MPLAB uses in order to track address labels, variable locations, and line/file information for debugging at source level.

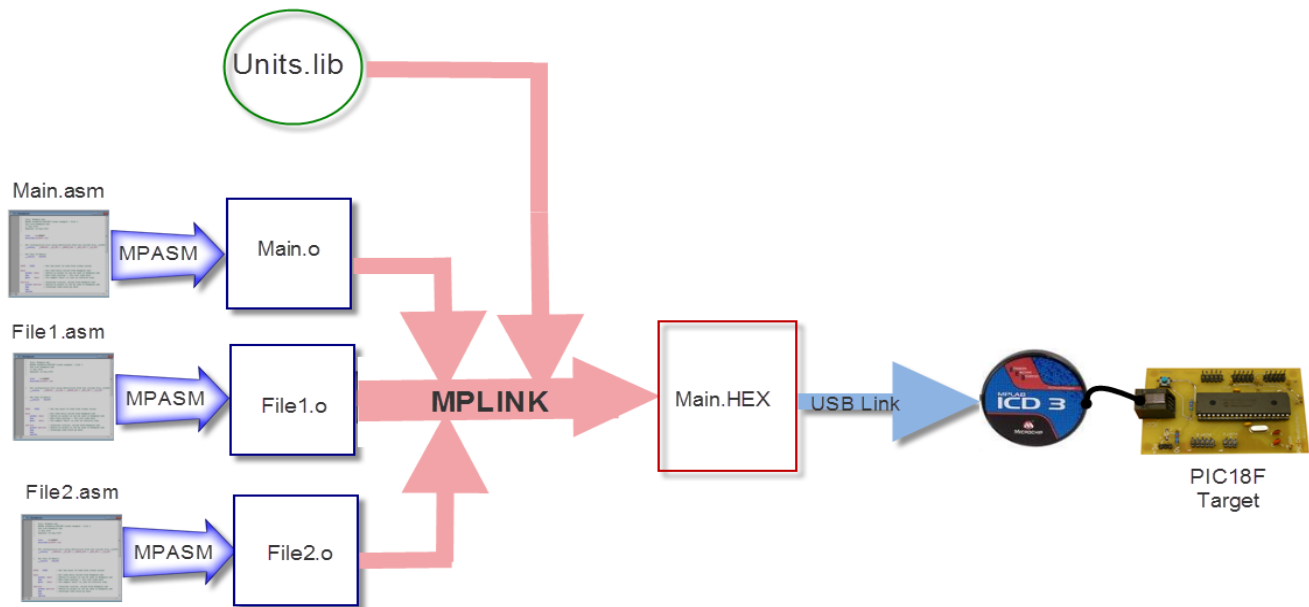


Fig.7 Assembler-Linker code development process

Linker Input/Output Files

As already mentioned the linker combines multiple object files (.o) into one executable hex file (.hex). Input file types for the linker are:

- Object files (.o) : Relocatable code that has been produce from the source file (.asm)
- Library files (.lib) : a collection of object files that have been grouped together and form a library
- Linker Script files (.lkr) : is a description of a memory resource layout for a particular micro/project

The linker produces the following output files:

- COFF file (.out, .cof) : are intermediate files used by the linker for producing the code file, hex file and listing file.
- Code file (.cod) : is the Debug file used by MPLAB
- HEX file (.hex) : is the binary executable file used for programming the micro.
- Listing file (.lst) : is a combination of the original source file together with the binary code produced
- Map file (.map) : it shows the memory layout after linking.

Configuration Fuses

The PIC micros have a unique way that specify some operational aspects of the chip when is powered up. These bits reside in an address that is outside the execution address and they are known as **configuration fuses**.

Operational options selected by these configuration fuses are the CPU oscillator settings, the reset operation, code protection, the watchdog timer, Program Code protection etc. Without the appropriate settings of these configuration fuses the micro will fail to operate properly.

In MPASM there is an assembler directive that specifies the configuration fuses. This is as follows:

CONFIG followed by the selected parameters

OSC =	Oscillator selection bits (LP,XT,HS,RC etc)
BOREN =	ON/OFF
WDT =	ON/OFF
MCLRE =	ON/OFF
LVP =	ON/OFF
DEBUG =	ON/OFF
CP =	ON/OFF
MCLRE =	ON/OFF
PWRT =	ON/OFF

The various configuration options are selected by using the ON or OFF argument that enables or disables the operational configuration of a fuse.

When you experiment with the PIC18 using our target boards it is recommended that you use the following standard fuse settings:

- BOREN = OFF Brown-out detect off (brown out detection is power supply voltage monitoring)
- LVP = OFF LVP is the low voltage programming feature. It is disabled as it uses some IO pins
- OSC = HS High Speed Oscillator, external crystal (4-20MHz)
- PWRT = ON Power-Up timer, gives a 70ms start-up timer to ensure proper clock operation
- WDT = OFF Watchdog Timer
- DEBUG = OFF Controls the In- Circuit Debugger mode
- MCLRE = ON MCLR pin enabled; RE3 input pin disabled

The following table gives the configuration fuses and labels for the PIC18F45xx

CONFIGURATION BITS AND DEVICE IDs									
File Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Default/ Unprogrammed Value
300001h CONFIG1H	IESO	FCMEN	—	—	FOSC3	FOSC2	FOSC1	FOSC0	00-- 0111
300002h CONFIG2L	—	—	—	BORV1	BORV0	BOREN1	BOREN0	PWRTEN	---1 1111
300003h CONFIG2H	—	—	—	WDTPS3	WDTPS2	WDTPS1	WDTPS0	WDTEN	---1 1111
300005h CONFIG3H	MCLRE	—	—	—	—	LPT1OSC	PBADEN	CCP2MX	1--- -011
300006h CONFIG4L	DEBUG	XINST	—	—	—	LVP	—	STVREN	10-- -1-1
300008h CONFIG5L	—	—	—	—	CP3 ⁽¹⁾	CP2 ⁽¹⁾	CP1	CP0	---- 1111
300009h CONFIG5H	CPD	CPB	—	—	—	—	—	—	11-- ----
30000Ah CONFIG6L	—	—	—	—	WRT3 ⁽¹⁾	WRT2 ⁽¹⁾	WRT1	WRT0	---- 1111
30000Bh CONFIG6H	WRTD	WRTB	WRTC	—	—	—	—	—	111- ----
30000Ch CONFIG7L	—	—	—	—	EBTR3 ⁽¹⁾	EBTR2 ⁽¹⁾	EBTR1	EBTR0	---- 1111
30000Dh CONFIG7H	—	EBTRB	—	—	—	—	—	—	-1-- ----
3FFFFEh DEVID1 ⁽¹⁾	DEV2	DEV1	DEV0	REV4	REV3	REV2	REV1	REV0	xxxx xxxx ⁽²⁾
3FFFFFh DEVID2 ⁽¹⁾	DEV10	DEV9	DEV8	DEV7	DEV6	DEV5	DEV4	DEV3	0000 1100

Legend: x = unknown, u = unchanged, — = unimplemented, q = value depends on condition.
Shaded cells are unimplemented, read as '0'.

Note 1: Unimplemented in PIC18F2420/4420 devices; maintain this bit set.

2: See Register 23-14 for DEVID1 values. DEVID registers are read-only and cannot be programmed by the user.

© 2004 Microchip Technology Inc. Preliminary DS39631A-page 249

Table 3 Configuration bits

Full Data Sheets can be found :

<http://datainfo.coventry.ac.uk/Panos/Info/Microcontrollers/PIC/PIC18F/PIC18F4520.pdf>

<http://datainfo.coventry.ac.uk/Panos/Info/Microcontrollers/PIC/PIC%20Fuse%20configuration%20Bits.pdf>

Sample Assembly Program Template

MPLAB provides a selection of assembler templates for the complete family of the PIC processors. These are located in your Program files /Microchip/MPASM Suite/Template folder.

A simplified version of this is as follows:

This could also be downloaded from the following URL:

<http://datainfo.coventry.ac.uk/Panos/Info/Experiments/Year%201/102SE/Assembler%20Templates/>

The template for the PIC18F4520 processor that we will be using is as shown below:

```
*****
;
; This file is a basic template for assembly code for a PIC18F4520. Copy
; this file into your project directory and modify or add to it as needed.
;
; Refer to the MPASM User's Guide for additional information on the
; features of the assembler.
;
; Refer to the PIC18Fx420/x520 Data Sheet for additional
; information on the architecture and instruction set.
;
*****
;
; Filename:
; Date:
; File Version:
; Author:
; Company:
;
*****
;
; Files Required: P18F4520.INC
;
*****

LIST P=18F4520          ; directive to define processor
#include <P18F4520.INC>  ; processor specific variable definitions

*****
; CONFIGURATION BITS
; Oscillator Selection and other fuse settings:
;
CONFIG   OSC = HS        ; High Speed clock
CONFIG   MCLRE = ON      ; MCLR enabled
CONFIG   DEBUG = OFF     ; Background debugger disabled, RB6 and RB7 configured as general IO
CONFIG   LVP = OFF       ; Low Voltage Programming OFF
CONFIG   WDT = OFF       ; WDT disabled
;
*****
; RESET VECTOR
; This code will start executing when a reset occurs.
```

ORG 0x0000

goto Main ; go to start of main code

```
*****  
;  
; Start of main program  
; THE MAIN PROGRAM CODE IS PLACED HERE.
```

Main:

```
;  
; *** main code goes here ***
```

```
*****  
;  
; End of program
```

END

From the previous assembler template we can create our example assembler file below:
Delete some of the options that are not required.

```
*****  
;  
; Filename: Assembler Example1.asm  
; Date: DD/MM/YY  
; File Version: 1.0  
; Author: Your Name Here  
; Company: Coventry University  
;  
; Program Function: Simple Port B output counting hex values, no delay routines  
*****  
; Files Required: P18F4520.INC  
*****  
; LIST P=18F4520 ; directive to define processor  
; #include <P18F4520.INC> ; processor specific variable definitions  
*****  
; CONFIGURATION BITS  
  
; Oscillator Selection and other fuse settings:  
  
CONFIG OSC = HS ;High Speed clock  
CONFIG MCLRE = ON ;MCLR enabled  
CONFIG DEBUG = OFF ;Background debugger disabled, RB6 and RB7 configured as general IO  
CONFIG LVP = OFF ; Low Voltage Programming OFF  
CONFIG WDT = OFF ; WDT disabled  
  
*****  
; RESET VECTOR  
; This code will start executing when a reset occurs.  
  
ORG 0x0000 ; ORG Directive  
  
GOTO Main ; go to start of main code
```

Main:	MOVLW	0x00	; move literal 00 into W register
	MOVWF	TRISD	; copy content of W into register TRISD
			; above instruction sets PORTD as Output
Loop:	MOVLW	0x55	; Move Literal hex 55 into W
	MOVWF	PORTD	; Move contents of W into PortD
	NOP		; No Operation
	MOVLW	0xAA	; Move Literal hex AA into W
	MOVWF	PORTD	; Move contents of W into PortD
	NOP		; No Operation
	GOTO	Loop	; Go to location Loop and repeat process
; End of program			
	END		; End Directive

Exercise 1: Add comments to the following assembly code program. Describe its function.

SUM	EQU	10H	;
*			
	ORG	0H	;
	MOVLW	25H	;
	ADDLW	0x34	;
	ADDLW	11H	;
	ADDLW	D'18'	;
	ADDLW	1CH	;
	ADDLW	B'00000110'	;
	MOVWF	SUM	;
HERE	GOTO	HERE	;
	END		

Exercise 2: Write an assembly program using the template that starts with the Binary value of 10101000 in the W register and stores incrementing values into 5 sequential memory locations Mem1 to Mem5.

Reading material :

<http://datainfo.coventry.ac.uk/Panos/Info/Microcontrollers/PIC/PIC18F/PIC18%20Assembly%20ch02.pdf>

APPENDIX

MPASM User's Guide with MPLINK and MPLIB

Table 5.1: Directive Summary

Directive	Description	Syntax
<code>--_BADRAM</code>	Specify invalid RAM locations	<code>--_badram <expr>[-<expr>][, <expr>[-<expr>]]</code>
<code>BANKISEL</code>	Generate RAM bank selecting code for indirect addressing	<code>bankisel <label></code>
<code>BANKSEL</code>	Generate RAM bank selecting code	<code>banksel <label></code>
<code>CBLOCK</code>	Define a Block of Constants	<code>cblock [<expr>]</code>
<code>CODE</code>	Begins executable code section	<code>[<name>] code [<address>]</code>
<code>--_CONFIG</code>	Specify configuration bits	<code>--_config <expr> OR --_config <addr>, <expr></code>
<code>CONSTANT</code>	Declare Symbol Constant	<code>constant <label>[=<expr>, ..., <label>[=<expr>]]</code>
<code>DA</code>	Store Strings in Program Memory	<code>[<label>] da <expr> [, <expr2>, ..., <exprn>]</code>
<code>DATA</code>	Create Numeric and Text Data	<code>[<label>] data <expr>[, <expr>, ..., <expr>] [<label>] data "<text_string>"[, "<text_string>"], ...]</code>
<code>DB</code>	Declare Data of One Byte	<code>[<label>] db <expr>[, <expr>, ..., <expr>] [<label>] db "<text_string>"[, "<text_string>"], ...]</code>
<code>DE</code>	Define EEPROM Data	<code>[<label>] de <expr>[, <expr>, ..., <expr>] [<label>] de "<text_string>"[, "<text_string>"], ...]</code>
<code>#DEFINE</code>	Define a Text Substitution Label	<code>define <name> [<value>] define <name> [<arg>, ..., <arg>] <value></code>
<code>DT</code>	Define Table	<code>[<label>] dt <expr>[, <expr>, ..., <expr>] [<label>] dt "<text_string>"[, "<text_string>"], ...]</code>
<code>DW</code>	Declare Data of One Word	<code>[<label>] dw <expr>[, <expr>, ..., <expr>] [<label>] dw "<text_string>"[, "<text_string>"], ...]</code>
<code>ELSE</code>	Begin Alternative Assembly Block to IF	<code>else</code>
<code>END</code>	End Program Block	<code>end</code>
<code>ENDC</code>	End an Automatic Constant Block	<code>endc</code>
<code>ENDIF</code>	End conditional Assembly Block	<code>endif</code>
<code>ENDM</code>	End a Macro Definition	<code>endm</code>
<code>ENDW</code>	End a While Loop	<code>endw</code>
<code>EQU</code>	Define an Assembly Constant	<code><label> equ <expr></code>
<code>ERROR</code>	Issue an Error Message	<code>error "<text_string>"</code>
<code>ERRORLEVEL</code>	Set Error Level	<code>errorlevel 0 1 2 <+ -><message number></code>

Directive Summary (Continued)

Directive	Description	Syntax
EXITM	Exit from a Macro	exitm
EXPAND	Expand Macro Listing	expand
EXTERN	Declares an external label	extern <label>[,<label>]
FILL	Fill Memory	[<label>] fill <expr>, <count>
GLOBAL	Exports a defined label	global <label>[,<label>]
IDATA	Begins initialized data section	[<name>] idata [<address>]
__IDLOCS	Specify ID locations	__idlocs <expr>
IF	Begin Conditionally Assembled Code Block	if <expr>
IFDEF	Execute If Symbol has Been Defined	ifdef <label>
IFNDEF	Execute If Symbol has not Been Defined	ifndef <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>[,<label>]
MACRO	Declare Macro Definition	<label> macro [<arg>,...,<arg>]
__MAXRAM	Specify maximum RAM address	__maxram <expr>
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
PAGESEL	Generate ROM page selecting code	pagesel <label>
PROCESSOR	Set Processor Type	processor <processsor_type>
RADIX	Specify Default Radix	radix <default_radix>
RES	Reserve Memory	[<label>] res <mem_units>
SET	Define an Assembler Variable	<label> set <expr>
SPACE	Insert Blank Listing Lines	space <expr>
SUBTITLE	Specify Program Subtitle	subtitle "<sub_text>"

MPASM User's Guide with MPLINK and MPLIB

Directive Summary (Continued)

Directive	Description	Syntax
TITLE	Specify Program Title	title "<title_text>"
UDATA	Begins uninitialized data section	[<name>] udata [<address>]
UDATA_ACS	Begins access uninitialized data section	[<name>] udata_acs [<address>]
UDATA_OVR	Begins overlayed uninitialized data section	[<name>] udata_ovr [<address>]
UDATA_SHR	Begins shared uninitialized data section	[<name>] udata_shr [<address>]
#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>

PIC18F2420/2520/4420/4520

23.0 SPECIAL FEATURES OF THE CPU

PIC18F2420/2520/4420/4520 devices include several features intended to maximize reliability and minimize cost through elimination of external components. These are:

- Oscillator Selection
- Resets:
 - Power-on Reset (POR)
 - Power-up Timer (PWRT)
 - Oscillator Start-up Timer (OST)
 - Brown-out Reset (BOR)
- Interrupts
- Watchdog Timer (WDT)
- Fail-Safe Clock Monitor
- Two-Speed Start-up
- Code Protection
- ID Locations
- In-Circuit Serial Programming

The oscillator can be configured for the application depending on frequency, power, accuracy and cost. All of the options are discussed in detail in **Section 2.0 "Oscillator Configurations"**.

A complete discussion of device Resets and interrupts is available in previous sections of this data sheet.

In addition to their Power-up and Oscillator Start-up Timers provided for Resets, PIC18F2420/2520/4420/4520 devices have a Watchdog Timer, which is either permanently enabled via the configuration bits or software controlled (if configured as disabled).

The inclusion of an internal RC oscillator also provides the additional benefits of a Fail-Safe Clock Monitor (FSCM) and Two-Speed Start-up. FSCM provides for background monitoring of the peripheral clock and automatic switchover in the event of its failure. Two-Speed Start-up enables code to be executed almost immediately on start-up, while the primary clock source completes its start-up delays.

All of these features are enabled and configured by setting the appropriate configuration register bits.

23.1 Configuration Bits

The configuration bits can be programmed (read as '0') or left unprogrammed (read as '1') to select various device configurations. These bits are mapped starting at program memory location 300000h.

The user will note that address 300000h is beyond the user program memory space. In fact, it belongs to the configuration memory space (300000h-3FFFFh), which can only be accessed using table reads and table writes.

Programming the configuration registers is done in a manner similar to programming the Flash memory. The WR bit in the EECON1 register starts a self-timed write to the configuration register. In normal operation mode, a TBLWT instruction with the TBLPTR pointing to the configuration register sets up the address and the data for the configuration register write. Setting the WR bit starts a long write to the configuration register. The configuration registers are written a byte at a time. To write or erase a configuration cell, a TBLWT instruction can write a '1' or a '0' into the cell. For additional details on Flash programming, refer to **Section 6.5 "Writing to Flash Program Memory"**.

TABLE 23-1: CONFIGURATION BITS AND DEVICE IDs

File Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Default/ Unprogrammed Value
300001h CONFIG1H	IESO	FCMEN	—	—	FOSC3	FOSC2	FOSC1	FOSC0	00-- 0111
300002h CONFIG2L	—	—	—	BORV1	BORV0	BOREN1	BOREN0	PWRTEN	---1 1111
300003h CONFIG2H	—	—	—	WDTPS3	WDTPS2	WDTPS1	WDTPS0	WDTEN	---1 1111
300005h CONFIG3H	MCLRE	—	—	—	—	LPT1OSC	PBADEN	CCP2MX	1--- -011
300006h CONFIG4L	DEBUG	XINST	—	—	—	LVP	—	STVREN	10-- -1-1
300008h CONFIG5L	—	—	—	—	CP3 ⁽¹⁾	CP2 ⁽¹⁾	CP1	CP0	---- 1111
300009h CONFIG5H	CPD	CPB	—	—	—	—	—	—	11-- ----
30000Ah CONFIG6L	—	—	—	—	WRT3 ⁽¹⁾	WRT2 ⁽¹⁾	WRT1	WRT0	---- 1111
30000Bh CONFIG6H	WRTD	WRTB	WRTC	—	—	—	—	—	111- ----
30000Ch CONFIG7L	—	—	—	—	EBTR3 ⁽¹⁾	EBTR2 ⁽¹⁾	EBTR1	EBTR0	---- 1111
30000Dh CONFIG7H	—	EBTRB	—	—	—	—	—	—	-1-- ----
3FFFFEh DEVID1 ⁽¹⁾	DEV2	DEV1	DEV0	REV4	REV3	REV2	REV1	REV0	xxxx xxxx ⁽²⁾
3FFFFFh DEVID2 ⁽¹⁾	DEV10	DEV9	DEV8	DEV7	DEV6	DEV5	DEV4	DEV3	0000 1100

Legend: x = unknown, u = unchanged, — = unimplemented, q = value depends on condition.
Shaded cells are unimplemented, read as '0'.

Note 1: Unimplemented in PIC18F2420/4420 devices; maintain this bit set.

Note 2: See Register 23-14 for DEVID1 values. DEVID registers are read-only and cannot be programmed by the user.

24.0 INSTRUCTION SET SUMMARY

PIC18F2420/2520/4420/4520 devices incorporate the standard set of 75 PIC18 core instructions, as well as an extended set of 8 new instructions, for the optimization of code that is recursive or that utilizes a software stack. The extended set is discussed later in this section.

24.1 Standard Instruction Set

The standard PIC18 instruction set adds many enhancements to the previous PICmicro® instruction sets, while maintaining an easy migration from these PICmicro instruction sets. Most instructions are a single program memory word (16 bits), but there are four instructions that require two program memory locations.

Each single-word instruction is a 16-bit word divided into an opcode, which specifies the instruction type and one or more operands, which further specify the operation of the instruction.

The instruction set is highly orthogonal and is grouped into four basic categories:

- **Byte-oriented** operations
- **Bit-oriented** operations
- **Literal** operations
- **Control** operations

The PIC18 instruction set summary in Table 24-2 lists **byte-oriented**, **bit-oriented**, **literal** and **control** operations. Table 24-1 shows the opcode field descriptions.

Most **byte-oriented** instructions have three operands:

1. The file register (specified by 'f')
2. The destination of the result (specified by 'd')
3. The accessed memory (specified by 'a')

The file register designator 'f' specifies which file register is to be used by the instruction. The destination designator 'd' specifies where the result of the operation is to be placed. If 'd' is zero, the result is placed in the WREG register. If 'd' is one, the result is placed in the file register specified in the instruction.

All **bit-oriented** instructions have three operands:

1. The file register (specified by 'f')
2. The bit in the file register (specified by 'b')
3. The accessed memory (specified by 'a')

The bit field designator 'b' selects the number of the bit affected by the operation, while the file register designator 'f' represents the number of the file in which the bit is located.

The **literal** instructions may use some of the following operands:

- A literal value to be loaded into a file register (specified by 'k')
- The desired FSR register to load the literal value into (specified by 'f')
- No operand required (specified by '—')

The **control** instructions may use some of the following operands:

- A program memory address (specified by 'n')
- The mode of the CALL or RETURN instructions (specified by 's')
- The mode of the table read and table write instructions (specified by 'm')
- No operand required (specified by '—')

All instructions are a single word, except for four double-word instructions. These instructions were made double-word to contain the required information in 32 bits. In the second word, the 4 MSBs are '1's. If this second word is executed as an instruction (by itself), it will execute as a NOP.

All single-word instructions are executed in a single instruction cycle, unless a conditional test is true or the program counter is changed as a result of the instruction. In these cases, the execution takes two instruction cycles, with the additional instruction cycle(s) executed as a NOP.

The double-word instructions execute in two instruction cycles.

One instruction cycle consists of four oscillator periods. Thus, for an oscillator frequency of 4 MHz, the normal instruction execution time is 1 μ s. If a conditional test is true, or the program counter is changed as a result of an instruction, the instruction execution time is 2 μ s. Two-word branch instructions (if true) would take 3 μ s.

Figure 24-1 shows the general formats that the instructions can have. All examples use the convention 'nnh' to represent a hexadecimal number.

The Instruction Set Summary, shown in Table 24-2, lists the standard instructions recognized by the Microchip Assembler (MPASM™).

Section 24.1.1 "Standard Instruction Set" provides a description of each instruction.

PIC18F2420/2520/4420/4520

OPCODE FIELD DESCRIPTIONS

Field	Description
a	RAM access bit a = 0: RAM location in Access RAM (BSR register is ignored) a = 1: RAM bank is specified by BSR register
bbb	Bit address within an 8-bit file register (0 to 7).
BSR	Bank Select Register. Used to select the current RAM bank.
C, DC, Z, OV, N	ALU Status bits: Carry, Digit Carry, Zero, Overflow, Negative.
d	Destination select bit d = 0: store result in WREG d = 1: store result in file register f
dest	Destination: either the WREG register or the specified register file location.
f	8-bit Register file address (00h to FFh) or 2-bit FSR designator (0h to 3h).
f _s	12-bit Register file address (000h to FFFh). This is the source address.
f _d	12-bit Register file address (000h to FFFh). This is the destination address.
GIE	Global Interrupt Enable bit.
k	Literal field, constant data or label (may be either an 8-bit, 12-bit or a 20-bit value).
label	Label name.
mm	The mode of the TBLPTR register for the table read and table write instructions. Only used with table read and table write instructions:
*	No change to register (such as TBLPTR with table reads and writes)
*+	Post-Increment register (such as TBLPTR with table reads and writes)
*-	Post-Decrement register (such as TBLPTR with table reads and writes)
++	Pre-Increment register (such as TBLPTR with table reads and writes)
n	The relative address (2's complement number) for relative branch instructions or the direct address for Call/Branch and Return instructions.
PC	Program Counter.
PCL	Program Counter Low Byte.
PCH	Program Counter High Byte.
PCLATH	Program Counter High Byte Latch.
PCLATU	Program Counter Upper Byte Latch.
PD	Power-down bit.
PRODH	Product of Multiply High Byte.
PRODL	Product of Multiply Low Byte.
s	Fast Call/Return mode select bit s = 0: do not update into/from shadow registers s = 1: certain registers loaded into/from shadow registers (Fast mode)
TBLPTR	21-bit Table Pointer (points to a Program Memory location).
TABLAT	8-bit Table Latch.
TO	Time-out bit.
TOS	Top-of-Stack.
u	Unused or unchanged.
WDT	Watchdog Timer.
WREG	Working register (accumulator).
x	Don't care ('0' or '1'). The assembler will generate code with x = 0. It is the recommended form of use for compatibility with all Microchip software tools.
z _s	7-bit offset value for indirect addressing of register files (source).
z _d	7-bit offset value for indirect addressing of register files (destination).
{ }	Optional argument.
[text]	Indicates an indexed address.
(text)	The contents of text.
[expr] <n>	Specifies bit n of the register indicated by the pointer expr.
→	Assigned to.
< >	Register bit field.
∈	In the set of.
<i>italics</i>	User defined term (font is Courier).

PIC18F2420/2520/4420/4520

GENERAL FORMAT FOR INSTRUCTIONS

Byte-oriented file register operations

15	10	9	8	7	0
OPCODE		d	a	f (FILE #)	

d = 0 for result destination to be WREG register
d = 1 for result destination to be file register (f)
a = 0 to force Access Bank
a = 1 for BSR to select bank
f = 8-bit file register address

Example Instruction

ADDWF MYREG, W, B

Byte to Byte move operations (2-word)

15	12	11	0
OPCODE		f (Source FILE #)	

MOVFF MYREG1, MYREG2

15	12	11	0
1111		f (Destination FILE #)	

f = 12-bit file register address

Bit-oriented file register operations

15	12	11	9	8	7	0
OPCODE		b (BIT #)	a	f (FILE #)		

BSF MYREG, bit, B

b = 3-bit position of bit in file register (f)
a = 0 to force Access Bank
a = 1 for BSR to select bank
f = 8-bit file register address

Literal operations

15	8	7	0
OPCODE		k (literal)	

MOVLW 7Fh

k = 8-bit immediate value

Control operations

CALL, GOTO and Branch operations

15	8	7	0
OPCODE		n<7:0> (literal)	

GOTO Label

15	12	11	0
1111		n<19:8> (literal)	

n = 20-bit immediate value

15	8	7	0
OPCODE		S	n<7:0> (literal)

CALL MYFUNC

15	12	11	0
1111		n<19:8> (literal)	

S = Fast bit

15	11	10	0
OPCODE		n<10:0> (literal)	

BRA MYFUNC

15	8	7	0
OPCODE		n<7:0> (literal)	

BC MYFUNC

PIC18F2420/2520/4420/4520

PIC18FXXX INSTRUCTION SET

Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word				Status Affected	Notes
				MSb		LSb			
BYTE-ORIENTED OPERATIONS									
ADDWF	f, d, a	Add WREG and f	1	0010	01da0	ffff	ffff	C, DC, Z, OV, N	1, 2
ADDWFC	f, d, a	Add WREG and CARRY bit to f	1	0010	0da	ffff	ffff	C, DC, Z, OV, N	1, 2
ANDWF	f, d, a	AND WREG with f	1	0001	01da	ffff	ffff	Z, N	1, 2
CLRF	f, a	Clear f	1	0110	101a	ffff	ffff	Z	2
COMF	f, d, a	Complement f	1	0001	11da	ffff	ffff	Z, N	1, 2
CPFSEQ	f, a	Compare f with WREG, skip =	1 (2 or 3)	0110	001a	ffff	ffff	None	4
CPFSGT	f, a	Compare f with WREG, skip >	1 (2 or 3)	0110	010a	ffff	ffff	None	4
CPFSLT	f, a	Compare f with WREG, skip <	1 (2 or 3)	0110	000a	ffff	ffff	None	1, 2
DECf	f, d, a	Decrement f	1	0000	01da	ffff	ffff	C, DC, Z, OV, N	1, 2, 3, 4
DECFSZ	f, d, a	Decrement f, Skip if 0	1 (2 or 3)	0010	11da	ffff	ffff	None	1, 2, 3, 4
DCFSNZ	f, d, a	Decrement f, Skip if Not 0	1 (2 or 3)	0100	11da	ffff	ffff	None	1, 2
INCF	f, d, a	Increment f	1	0010	10da	ffff	ffff	C, DC, Z, OV, N	1, 2, 3, 4
INCFSZ	f, d, a	Increment f, Skip if 0	1 (2 or 3)	0011	11da	ffff	ffff	None	4
INFSNZ	f, d, a	Increment f, Skip if Not 0	1 (2 or 3)	0100	10da	ffff	ffff	None	1, 2
IORWF	f, d, a	Inclusive OR WREG with f	1	0001	00da	ffff	ffff	Z, N	1, 2
MOVF	f, d, a	Move f	1	0101	00da	ffff	ffff	Z, N	1
MOVFF	f _s , f _d	Move f _s (source) to f _d (destination)	2	1100	ffff	ffff	ffff	None	
MOVWF	f, a	Move WREG to f	1	0110	111a	ffff	ffff	None	
MULWF	f, a	Multiply WREG with f	1	0000	001a	ffff	ffff	None	1, 2
NEGF	f, a	Negate f	1	0110	110a	ffff	ffff	C, DC, Z, OV, N	
RLCF	f, d, a	Rotate Left f through Carry	1	0011	01da	ffff	ffff	C, Z, N	1, 2
RLNCF	f, d, a	Rotate Left f (No Carry)	1	0100	01da	ffff	ffff	Z, N	
RRCF	f, d, a	Rotate Right f through Carry	1	0011	00da	ffff	ffff	C, Z, N	
RRNCF	f, d, a	Rotate Right f (No Carry)	1	0100	00da	ffff	ffff	Z, N	
SETF	f, a	Set f	1	0110	100a	ffff	ffff	None	1, 2
SUBFWB	f, d, a	Subtract f from WREG with borrow	1	0101	01da	ffff	ffff	C, DC, Z, OV, N	
SUBWF	f, d, a	Subtract WREG from f	1	0101	11da	ffff	ffff	C, DC, Z, OV, N	1, 2
SUBWFB	f, d, a	Subtract WREG from f with borrow	1	0101	10da	ffff	ffff	C, DC, Z, OV, N	
SWAPF	f, d, a	Swap nibbles in f	1	0011	10da	ffff	ffff	None	4
TSTFSZ	f, a	Test f, skip if 0	1 (2 or 3)	0110	011a	ffff	ffff	None	1, 2
XORWF	f, d, a	Exclusive OR WREG with f	1	0001	10da	ffff	ffff	Z, N	

- Note 1:** When a Port register is modified as a function of itself (e.g., `MOVF PORTB, 1, 0`), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.
- 2:** If this instruction is executed on the TMR0 register (and where applicable, 'd' = 1), the prescaler will be cleared if assigned.
- 3:** If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP.
- 4:** Some instructions are two-word instructions. The second word of these instructions will be executed as a NOP unless the first word of the instruction retrieves the information embedded in these 16 bits. This ensures that all program memory locations have a valid instruction.

PIC18F2420/2520/4420/4520

PIC18FXXX INSTRUCTION SET (CONTINUED)

Mnemonic, Operands	Description	Cycles	16-Bit Instruction Word				Status Affected	Notes	
			MSb		LSb				
BIT-ORIENTED OPERATIONS									
BCF	f, b, a	Bit Clear f	1	1001	bbba	ffff	ffff	None	1, 2
BSF	f, b, a	Bit Set f	1	1000	bbba	ffff	ffff	None	1, 2
BTFSC	f, b, a	Bit Test f, Skip if Clear	1 (2 or 3)	1011	bbba	ffff	ffff	None	3, 4
BTFSS	f, b, a	Bit Test f, Skip if Set	1 (2 or 3)	1010	bbba	ffff	ffff	None	3, 4
BTG	f, d, a	Bit Toggle f	1	0111	bbba	ffff	ffff	None	1, 2
CONTROL OPERATIONS									
BC	n	Branch if Carry	1 (2)	1110	0010	nnnn	nnnn	None	4
BN	n	Branch if Negative	1 (2)	1110	0110	nnnn	nnnn	None	
BNC	n	Branch if Not Carry	1 (2)	1110	0011	nnnn	nnnn	None	
BNN	n	Branch if Not Negative	1 (2)	1110	0111	nnnn	nnnn	None	
BN OV	n	Branch if Not Overflow	1 (2)	1110	0101	nnnn	nnnn	None	
BNZ	n	Branch if Not Zero	1 (2)	1110	0001	nnnn	nnnn	None	
BOV	n	Branch if Overflow	1 (2)	1110	0100	nnnn	nnnn	None	
BRA	n	Branch Unconditionally	2	1101	0nnn	nnnn	nnnn	None	
BZ	n	Branch if Zero	1 (2)	1110	0000	nnnn	nnnn	None	
CALL	n, s	Call subroutine 1st word	2	1110	110s	kkkk	kkkk	None	
		2nd word		1111	kkkk	kkkk	kkkk		
CLRWD T	—	Clear Watchdog Timer	1	0000	0000	0000	0100	\overline{TO} , \overline{PD}	
DAW	—	Decimal Adjust WREG	1	0000	0000	0000	0111	C	
GOTO	n	Go to address 1st word	2	1110	1111	kkkk	kkkk	None	
		2nd word		1111	kkkk	kkkk	kkkk		
NOP	—	No Operation	1	0000	0000	0000	0000	None	
NOP	—	No Operation	1	1111	xxxx	xxxx	xxxx	None	
POP	—	Pop top of return stack (TOS)	1	0000	0000	0000	0110	None	
PUSH	—	Push top of return stack (TOS)	1	0000	0000	0000	0101	None	
RCALL	n	Relative Call	2	1101	1nnn	nnnn	nnnn	None	
RESET		Software device Reset	1	0000	0000	1111	1111	All	
RETFIE	s	Return from interrupt enable	2	0000	0000	0001	000s	GIE/GIEH, PEIE/GIEL	
RETLW	k	Return with literal in WREG	2	0000	1100	kkkk	kkkk	None	
RETURN	s	Return from Subroutine	2	0000	0000	0001	001s	None	
SLEEP	—	Go into Standby mode	1	0000	0000	0000	0011	\overline{TO} , \overline{PD}	

- Note 1:** When a Port register is modified as a function of itself (e.g., MOVF PORTB, 1, 0), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.
- 2:** If this instruction is executed on the TMR0 register (and where applicable, 'd' = 1), the prescaler will be cleared if assigned.
- 3:** If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP.
- 4:** Some instructions are two-word instructions. The second word of these instructions will be executed as a NOP unless the first word of the instruction retrieves the information embedded in these 16 bits. This ensures that all program memory locations have a valid instruction.

PIC18F2420/2520/4420/4520

TABLE 24-2: PIC18FXXX INSTRUCTION SET (CONTINUED)

Mnemonic, Operands	Description	Cycles	16-Bit Instruction Word				Status Affected	Notes	
			MSb		LSb				
LITERAL OPERATIONS									
ADDLW	k	Add literal and WREG	1	0000	1111	kkkk	kkkk	C, DC, Z, OV, N	
ANDLW	k	AND literal with WREG	1	0000	1011	kkkk	kkkk	Z, N	
IORLW	k	Inclusive OR literal with WREG	1	0000	1001	kkkk	kkkk	Z, N	
LFSR	f, k	Move literal (12-bit) 2nd word to FSR(f) 1st word	2	1110	1110	00ff	kkkk	None	
MOVLB	k	Move literal to BSR<3:0>	1	0000	0001	0000	kkkk	None	
MOVLW	k	Move literal to WREG	1	0000	1110	kkkk	kkkk	None	
MULLW	k	Multiply literal with WREG	1	0000	1101	kkkk	kkkk	None	
RETLW	k	Return with literal in WREG	2	0000	1100	kkkk	kkkk	None	
SUBLW	k	Subtract WREG from literal	1	0000	1000	kkkk	kkkk	C, DC, Z, OV, N	
XORLW	k	Exclusive OR literal with WREG	1	0000	1010	kkkk	kkkk	Z, N	
DATA MEMORY ↔ PROGRAM MEMORY OPERATIONS									
TBLRD*	Table Read	2	0000	0000	0000	1000	None		
TBLRD*+	Table Read with post-increment		0000	0000	0000	1001	None		
TBLRD*-	Table Read with post-decrement		0000	0000	0000	1010	None		
TBLRD*+	Table Read with pre-increment		0000	0000	0000	1011	None		
TBLWT*	Table Write	2	0000	0000	0000	1100	None		
TBLWT*+	Table Write with post-increment		0000	0000	0000	1101	None		
TBLWT*-	Table Write with post-decrement		0000	0000	0000	1110	None		
TBLWT*+	Table Write with pre-increment		0000	0000	0000	1111	None		

- Note** 1: When a Port register is modified as a function of itself (e.g., `MOVF PORTB, 1, 0`), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.
- 2: If this instruction is executed on the TMR0 register (and where applicable, 'd' = 1), the prescaler will be cleared if assigned.
- 3: If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP.
- 4: Some instructions are two-word instructions. The second word of these instructions will be executed as a NOP unless the first word of the instruction retrieves the information embedded in these 16 bits. This ensures that all program memory locations have a valid instruction.