Introduction

Thank you for purchasing Parallax PIC development tools. We've done our best to ensure that our tools are easy-to-use and complete.

As of January, 1995, we offer six products for PIC development:

- PIC16Cxx Assembler
- PIC16Cxx Software Simulator
- PIC16Cxx Programmer
- ClearView '5x In-Circuit Emulator
- ClearView 'xx In-Circuit Emulator
- BackDraft '17 Programmer

The **PIC16Cxx Assembler** is used to convert assembly language source code into object code, which is then used by the simulators, programmers, and emulators. Our assembler accepts programs written using our 8051-like instruction set, as well as the original Microchip instruction set. If you plan to use source code in the Microchip format, please note that while our assembler will accept their instructions, it will not accept their syntax. For instance, the Microchip assembler expects a binary number to be written as b'00100111'; our assembler expects the same number to be written in the Intel format, as 00100111b. If you plan to write code using Microchip instructions, you can certainly do so, as long as the syntax is in the Intel format. However, if you plan to use Microchip code samples verbatim, you will need to change the syntax of the code. Or, you may want to use Microchip's assembler.

The **PIC16Cxx Simulator** is a piece of software that simulates the execution of PIC programs on the PC's screen. Execution can be single-stepped, breakpoints can be set, and registers can be watched. Many customers find the simulator useful for preliminary testing and debugging. And for hardware testing, the simulator can be used with our I/O simulator and in-circuit emulator products. In both cases, the simulator can exercise a real circuit designed to accept a PIC chip.

The **PIC16Cxx Programmer** allows you to program, read, and verify PICs. The programmer's single-screen software makes programming a snap. And inexpensive upgrades and adapters keep the programmer current. As of January, 1995, the programmer supports the PIC16C5x, 16C64 (with adapter), 16C71, 16C74 (with adapter), and 16C84. Optional adapters are available with ZIF, SOIC, and SSOP sockets.

Introduction

The programmer is available in two configurations:

Complete Package	With cables, power supply, and printed documentation.
Hobbyist Pack	With user-supplied cables and power supply, and documentation on disk.

Please see page 69 for cable information (see page 65 in the PDF version).

ClearView '5x and ClearView 'xx are in-circuit emulators. They plug in place of a PIC in a target circuit, and allow you to run your code in-circuit at full hardware speeds (32 kHz - 20 MHz). They also support full debugging features, such as stepping, breakpoints, and register modification. The main difference between the two units is in which PICs they support: the '5x unit only supports the 16C5x parts, and the 'xx unit only supports the newer 16Cxx parts (16C64, 16C71, 16C84,...). These emulators provide the most comprehensive help to test and debug code in-circuit.

BackDraft '17 is our first product for the PIC17C42. It allows you to program, read, and verify 17C42s. The unit has a 40-pin ZIF socket for DIP parts, and an optional adapter is available for PLCC parts. For an assembler, we provide Microchip's assembler and documentation.

Important Information

Warranty

Parallax warrants its products against defects in materials and workmanship for a period of 90 days.

If you discover a defect, Parallax will, at its option, repair, replace, or refund the purchase price. Simply return the product with a description of the problem and a copy of your invoice (if you do not have your invoice, please include you name and telephone number). We will return your product, or its replacement, using the same shipping method used to ship the product to Parallax (for instance, if you ship your product via overnight express, we will do the same).

This warranty does not apply if the product has been modified or damaged by accident, abuse, or misuse.

14-Day Money-Back Guarantee

If, within 14 days of having received your product, you find that it does not suit your needs, you may return it for a refund. Parallax will refund the purchase price of the product, excluding shipping/handling costs. If the product has been altered or damaged, a partial refund will be given.

Copyrights and Trademarks

Disclaimer of Liability

Parallax, Inc. is not responsible for special, incidental, or consequential damages resulting from any breach of warranty, or under any legal theory, including lost profits, downtime, goodwill, damage to or replacement of equipment or property, and any costs or recovering, reprogramming, or reproducing any data stored in or used with Parallax products.

Parallax BBS and Internet Access

Parallax has a 24-hour BBS for your convenience. Customers call our BBS to obtain the latest versions of Parallax software or to try software before purchasing the complete product. In addition, some customers use the BBS as a forum to communicate their ideas with other customers.

The BBS telephone number is (916) 624-7101.
Settings are: 300-14400 baud, 8 data bits, 1 stop bit, no parity.

You can now access Parallax via Internet. Many BBS files are available on our ftp site, and you can send email to our sales and technical staff. The Internet address is *parallaxinc.com.*

Important Information

Using Parallax PIC Tools with Non-Parallax Tools

If you want to use Parallax development tools with tools from other companies (such as using our assembler with one of Microchip's programmers), please be aware that you may experience some difficulties. These difficulties are usually the result of incompatible file formats.

Some of our customers have gone so far as to write custom software to make various tools work together. However, this much effort is usually not required. If you'll be mixing brands, just keep the following notes in mind:

1) Our assemblers produce files in Microchip's *Intel Hex 8-bit Merged* format (referred to as "INHX8M"). If you're using a simulator, programmer, or emulator from another company, find out if it supports this format.

2) Many of our hardware tools expect hex files in the above mentioned "INHX8M" format. If you're using an assembler from another company, find out if it supports this format. *The Microchip assembler supports this format, but does not default to it.*

3) Our products produce and expect certain device data in hex files (device data indicates which PIC is being used, which oscillator, etc.). Development tools from other companies may simply ignore this data, in which case you will need to restate the settings. Some tools, however, may refuse to load files with extra data.

   Microchip's PICSTART-16B programmer is one of the more notable development tools that doesn't accept files with our device data.

   In such cases, the Parallax assemblers can be instructed to leave out the device data.

4) When using Microchip's MPASM assembler or Byte Craft's C compiler with our simulator, you need to use a ".cod" file, instead of the usual ".lst" file used with our assemblers.

PIC16C54 Examples

This section contains two example projects that use the PIC16C54. Schematics and commented source code are given. It's assumed that you're familiar with digital circuits and assembly language programming. And, although the source code for these projects is included on your PIC diskette, you may want to type in the programs. The programs are very short and typing them will give you a better feel for the PIC.

You may wish to read the PIC16Cxx assembler and programmer chapters before continuing. However, these examples have been written to work with no knowledge of those chapters. So, if you really want to try a simple PIC project right away, you shouldn't have any problems.

Each project uses a PIC16C54-RC/P, which is the least expensive one-time-programmable PIC. Since you might make mistakes, you should have a small supply of PIC's (five or ten, perhaps). Or, if you'd rather deal with just one part, the PIC16C54/JW (erasable) will do the job.

Aside from the above, the examples assume two more things: 1) that you have a text editor capable of saving ASCII files, and 2) that you are in the disk/directory which contains the PIC16C5x software from Parallax.

The first project is a variation on the perennial button-and-LED circuit; the PIC flashes an LED at one speed and then, if the button is pressed, speeds up the flashing. This project is useful as a very basic introduction to the PIC (and microcontrollers in general).

The second project is only slightly more complicated; it incorporates a 7-segment display, which increments whenever the button is pressed. In addition to the basics, this project shows how to debounce a button and how to make a lookup table on the PIC.

For a broader selection of PIC applications, please refer to the PIC16Cxx Applications Handbook. It contains a collection of application notes on various topics, such as interfacing to LCD displays, reading keypads, sending serial data, and controlling servos.

BLANK PAGE

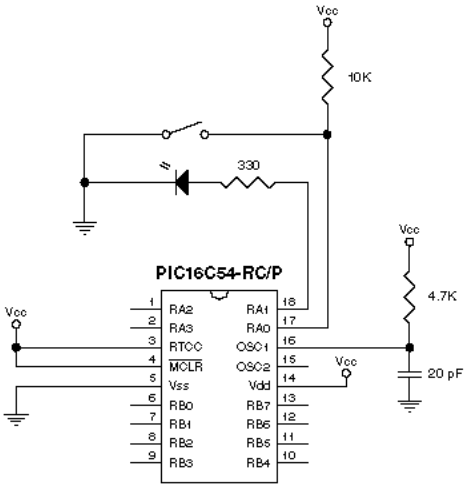
PIC16C54 Examples

PIC Project #1: "Simple"

As mentioned at the beginning of this chapter, this project gives you a very simple introduction to the PIC. The PIC flashes an LED at one speed and then, if the button is pressed, speeds up the flashing.

The PIC itself requires only two parts to operate: a resistor and capacitor to form its RC oscillator. The remaining four parts are the button, the LED, and their resistors.

Before continuing, build the circuit shown below. Only seven parts are necessary, so the task shouldn't be too laborious. If you don't have exactly what's called for, you can substitute a close match (the 4.7K resistor and 20 pF capacitor should not be substituted with smaller values, however).



PIC16C54 Examples

PIC Project #1: "Simple" *(continued)*

The following listing is the source code for the project:

```
; PIC EXAMPLE PROGRAM: "SIMPLE"
; June 8, 1992
;
; This 16-word program is a very simple PIC application.  Its only purpose
; is to flash an LED at one of two rates.  Normally, the LED flashes slowly.
; However, if bit 0 of port A (RA0) is grounded, the LED flashes roughly twice
; as fast.
;
; As the program is written, it expects to run on a PIC16C54-RC/P.  A push
; button connects RA0 to ground (don't forget a pull-up resistor from RA0 to
; Vdd, perhaps 10K).  A current-limited LED is connected from RA1 to ground
; (330 ohms work well).  For an oscillator, a 20 pF capacitor to ground and
; a 4.7K resistor to Vcc are connected to OSC1 (OSC2 is left open).  RTCC and
; MCLR should be tied high.  Lastly, power and ground are connected to Vdd
; and Vss, respectively.
;
; If you're looking for a simple introduction to the PIC, this program should
; help.  Among other things, it shows the following basic concepts:
;
;    * Setting device options
;    * Setting the reset vector
;    * Setting I/O pins as inputs or outputs
;    * Using global labels
;    * Using local labels (see ":Loop" below)
;
; If you build the circuit described above using a PIC programmed with the
; following program, the LED should start flashing as soon as you apply power.

; START OF PROGRAM

; Set the device type, oscillator type, watchdog timer status, and code
; protect status

            DEVICE  PIC16C54,RC_OSC,WDT_OFF,PROTECT_OFF

            RESET   Start             ;Set reset vector to address at Start
                                      ;(PIC will jump to this when reset)

Count0      equ     10h               ;Assign labels to registers 10h & 11h
Count1      equ     11h

            clr     Count0
            clr     Count1
            clr     RA                ;Clear port before setting direction register

Start       mov     !RA,#00000001b    ;Set data direction register for port A
                                      ;(make bit 0 an input)
;
; Loop 65536 times, then invert the LED
;

:Loop       djnz    Count0,:Loop      ;Decrement Count0 until it reaches zero
            djnz    Count1,:Loop      ;Decrement Count1.  If it's not zero,
                                      ;jump back to :Loop

            xor     RA,#00000010b     ;Invert the LED (bit 1 of port A)

;
; Check button status.  If it's pressed, skip the additional delay and jump
; back to the first loop
;
```

PIC Project #1: "Simple" *(continued)*

```
ChkBtn      jnb     RA.0,Start:Loop   ;Jump to 1st loop if button is pressed
                                      ;(button is low when pressed)

:Loop       djnz    Count0,:Loop      ;Decrement Count0 until it reaches zero
            djnz    Count1,:Loop      ;Decrement Count1.  If it's not zero,
                                      ;jump back to :Loop

            jmp     Start:Loop        ;Jump back to first loop
```

Once you've typed in the program, save it as SIMPLE.SRC and exit your text editor. The next step is to assemble the source code into object code, which is then used by the programmer.

To assemble the program, type the following command from the DOS prompt:

PASM SIMPLE (the extension is only necessary if it's *not* ".SRC")

If the assembler finds any errors, it will give you the line number and description of each error. Before continuing, you'll need to correct any errors and re-assemble the program.

If the program assembles correctly, the assembler will produce an object file, called SIMPLE.OBJ.

To start the PIC16Cxx Programmer software, type this command from the DOS prompt:

PEP SIMPLE (here, the extension is only necessary is it's *not* ".OBJ")

After a second or two, you'll see the screen shown on the next page.

PIC Project #1: "Simple" *(continued)*

```
     DEVICE: 54 55 56 57 58  PIC16C5x-PGM/EMU
  OSCILLATOR: RC HS XT LP     (C)1994 Parallax
    WATCHDOG: ON OFF
CODE PROTECT: OFF ON      ID: FFFF CHECKSUM

000- FFF FFF FFF FFF FFF FFF FFF FFF  .......
008- FFF FFF FFF FFF FFF FFF FFF FFF  .......
010- FFF FFF FFF FFF FFF FFF FFF FFF  .......
018- FFF FFF FFF FFF FFF FFF FFF FFF  .......
020- FFF FFF FFF FFF FFF FFF FFF FFF  .......
028- FFF FFF FFF FFF FFF FFF FFF FFF  .......
030- FFF FFF FFF FFF FFF FFF FFF FFF  .......
038- FFF FFF FFF FFF FFF FFF FFF FFF  .......

  HEX ENTRY      ASCII ENTRY      FILL (   )

PROGRAM           EMULATE           ESC EXIT
VERIFY
READ              LOAD: SIMPLE.OBJ
BLANK CHECK       SAVE:
```

The screen should appear exactly as above, except that your object code should fill the center of the screen (in place of the FFFs).

If everything checks out, then it's time to program a PIC. To do so, follow these steps:

1) Make sure that your PIC Programmer is properly connected to your PC (see the Programmer chapter, if you have any doubts).
2) Place an erased PIC16C54-RC / P into the programmer's 18-pin LIF socket.
3) Press 'B' (*blank check*) on your keyboard. The screen will display a message indicating whether or not the PIC is blank. If the PIC is not blank, try another one.

PIC Project #1: "Simple" *(continued)*

4) Once you have a blank PIC in the programmer, press 'P' (*program*). Programming and verification should take a second or two. Then, a message will be displayed, indicating the results of the verification.

   If, for some reason, the PIC did not program and verify, try another PIC before continuing.

Now you're ready to try the programmed PIC in your circuit. Make sure the power to your circuit is off, and then plug in the PIC.

When you turn the power on, the LED should be flashing. If you push the button, the LED should blink faster.

If your project appears totally "dead", then something is very wrong. Double-check the circuit (power, RC oscillator, etc.) and source code. If it appears to work, but not quite right, then the problem is probably a software bug.

Hopefully, this project worked out smoothly. Although a very simple project, it shows a number of basic concepts. If you're not yet comfortable with the PIC, read the source code carefully. The comments will help you understand each instruction.

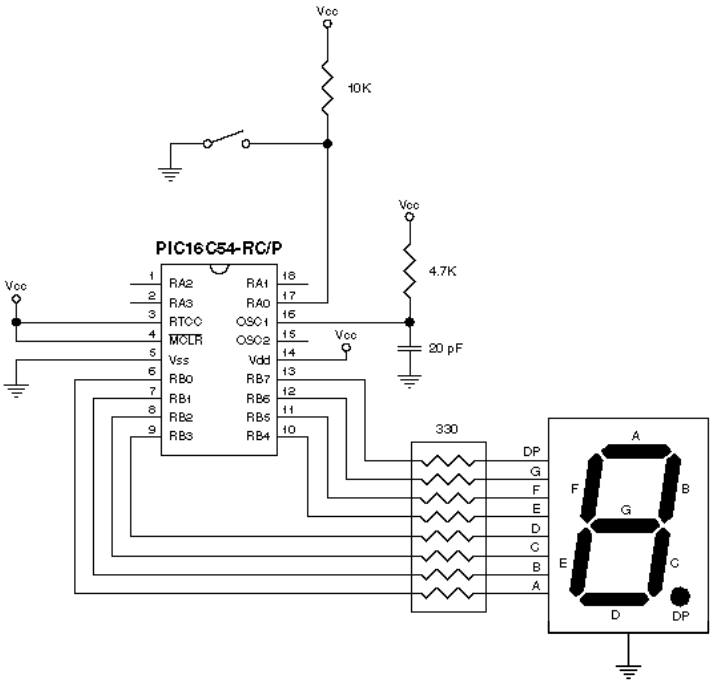
BLANK PAGE

PIC Project #2: "Table"

Like the first project, this project gives you a simple introduction to the PIC. But, it also demonstrates the useful topics of how to debounce push button inputs and how to make a lookup table.

The schematic below shows the circuit. Whenever the button is pressed, the 7-segment display increments.

The PIC itself requires only two parts to operate: a resistor and capacitor to form its RC oscillator. The remaining four parts are the button, the 7-segment display (common cathode), a 10K resistor, and a resistor pack (330-ohm, 8 isolated resistors).



PIC Project #2: "Table" *(continued)*

The following listing is the source code for the project:

```
; PIC EXAMPLE PROJECT: "TABLE"
; August 19, 1992
;
; This 44-word program is a simple PIC application that shows some useful
; routines.  Its purpose is to monitor a button and display a digit on a
; 7-segment display.  When the button is pressed, the digit increments.
;
; As the program is written, it expects to run on a PIC16C54-RC/P.  A push
; button connects RA0 to ground (don't forget a pull-up resistor from RA0 to
; Vcc, perhaps 10K).  A 7-segment display (common cathode) is connected to
; port B (segment A to bit 0, segment B to bit 1, etc.).  For an oscillator,
; a 20 pF capacitor to ground and a 4.7K resistor to Vcc are connected to
; OSC1 (OSC2 is left open).  RTCC and MCLR should be tied high.  Lastly,
; power and ground are connected to Vdd and Vss, respectively.
;
; Among other things, this program shows the following basic concepts:
;
;    * Setting device options
;    * Setting the reset vector
;    * Setting I/O pins as inputs or outputs
;    * Using labels
;    * Debouncing a push button input
;    * Reading from a lookup table
;
; If you build the circuit described above using a PIC programmed with the
; following program, the display should show a "0" when you apply power.

; START OF PROGRAM

; Set the device type, oscillator type, watchdog timer status, and code
; protect status

            DEVICE  PIC16C54,RC_OSC,WDT_OFF,PROTECT_OFF

            RESET   Start             ;Set reset vector to address at Start
                                      ;(PIC will jump to this when reset)

Count0      equ     10h               ;Register labels
Count1      equ     11h
Number      equ     12h

Flag        equ     13h.0             ;Bit label (bit 0 of register 13h)

Button      equ     RA.0              ;Port labels (bit 0 of port A)
Display     equ     RB                ;(entire port)

Start       clr     Number
            clr     Flag
            clr     RA
            clr     RB

            mov     !RA,#00000001b    ;Set data direction register for port A
                                      ;(make bit 0 an input)

            mov     !RB,#00000000b    ;Set data direction register for port B
                                      ;(make all bits outputs)

; Main loop

Digit       mov     W,Number          ;Move Number into W
```

PIC Project #2: "Table" *(continued)*

```
            call    GetPattern      ;Get bit pattern for digit
            mov     Display,W       ;Show digit

            jnb     Flag,UpLoop     ;If flag is clear, check for 2048 reads
                                    ;of button not pressed

            jb      Button,Digit    ;Jump if flag set, but button not
                                    ;pressed

; If button pressed after not being pressed for 2048 reads, then increment number

            inc     Number          ;Increment number
            cjne    Number,#10,Clear

            clr     Number          ;If Number reached 10, reset to 0

Clear       clr     Flag
            jmp     Digit

; Set flag if button not pressed for 2048 reads.  Faster oscillator speeds
; may require more than 2048 reads to properly debounce button inputs.

UpLoop      clr     Count0
            mov     Count1,#08

Loop        jnb     Button,Digit    ;If button pressed before 2048 reads,
            djnz    Count0,Loop     ;jump back to main loop
            djnz    Count1,Loop

            setb    Flag            ;Set flag if button not pressed
            jmp     Digit

; 7-segment lookup table

GetPattern  jmp     PC+W            ;Jump to appropriate bit pattern,
                                    ;load W with pattern, and then return
                                    ;to calling routine.

            retw    3Fh,06h,5Bh     ;Bit patterns for digits 0-9
            retw    4Fh,66h,6Dh
            retw    7Dh,07h,7Fh
            retw    6Fh


;           retw    'This is text'  ;Sample text lookup table (not used by program)
```

Once you've typed in the program, save it as TABLE.SRC and exit your text editor. The next step is to assemble the source code into object code, which is then used by the programmer.

To assemble the program, type the following command from the DOS prompt:

PASM TABLE (the extension is only necessary if it's *not* ".SRC")

PIC Project #2: "Table" *(continued)*

If the assembler finds any errors, it will give you the line number and description of each error. Before continuing, you'll need to correct any errors and re-assemble the program.

If the program assembles correctly, the assembler will produce an object file, called TABLE.OBJ.

To start the PIC16Cxx Programmer software, type this command from the DOS prompt:

PEP TABLE (here, the extension is only necessary is it's *not* ".OBJ")

After a second or two, you'll see the screen shown below:

```
     DEVICE: 54 55 56 57 58  PIC16C5x-PGM/EMU
  OSCILLATOR: RC HS XT LP     (C)1994 Parallax
    WATCHDOG: ON OFF
CODE PROTECT: OFF ON      ID: FFFF CHECKSUM

000- FFF FFF FFF FFF FFF FFF FFF FFF  .......
008- FFF FFF FFF FFF FFF FFF FFF FFF  .......
010- FFF FFF FFF FFF FFF FFF FFF FFF  .......
018- FFF FFF FFF FFF FFF FFF FFF FFF  .......
020- FFF FFF FFF FFF FFF FFF FFF FFF  .......
028- FFF FFF FFF FFF FFF FFF FFF FFF  .......
030- FFF FFF FFF FFF FFF FFF FFF FFF  .......
038- FFF FFF FFF FFF FFF FFF FFF FFF  .......

  HEX ENTRY      ASCII ENTRY      FILL (   )

PROGRAM           EMULATE           ESC EXIT
VERIFY
READ              LOAD: TABLE.OBJ
BLANK CHECK       SAVE:
```

PIC Project #2: "Table" *(continued)*

The screen should appear exactly as on the previous page, except that your object code should fill the center of the screen (in place of the FFF's).

If everything checks out, then it's time to program a PIC. To do so, follow these steps:

1) Make sure that your PIC Programmer is properly connected to your PC (see the Programmer chapter, if you have any doubts).
2) Place an erased PIC16C54-RC/P into the programmer's 18-pin LIF socket.
3) Press 'B' (*blank check*) on your keyboard. The screen will display a message indicating whether or not the PIC is blank. If the PIC is not blank, try another one.
4) Once you have a blank PIC in the programmer, press 'P' (*program*). Programming and verification should take a second or two. Then, a message will be displayed, indicating the results of the verification.

   If, for some reason, the PIC did not program and verify, try another PIC before continuing.

Now you're ready to try the programmed PIC in your circuit. Make sure the power to your circuit is off, then plug in the PIC.

When you turn the power on, the display should show "0". If you push the button, the display should increment.

If your project appears totally "dead", then something is very wrong. Double-check the circuit (power, RC oscillator, etc.) and source code. If it appears to work, but not quite right, then the problem is probably a software bug.

Hopefully, this project worked out smoothly. If you're not yet comfortable with the PIC, read the source code carefully.

BLANK PAGE

PIC16Cxx Assemblers

When you write programs for PIC microcontrollers, you'll use a text editor to create source code. Source code is the format that you're accustomed to looking at; it contains English-like labels, instructions, and data.

Before your source code can be used by a programmer, downloader, or other development tool, it must be converted into object code. Object code is the "machine-readable" version of source code; it contains instructions and data in the form of hexadecimal data, which can be executed by the PIC.

An assembler is a piece of software that converts source code into object code. For instance, this line of source code:

```
CALL    SENDBYTE            ;Call send routine
```

assembles into just two bytes of PIC object code:

```
2420h
```

It's possible to write programs directly in object code, using the individual machine codes that make up each instruction. However, most people find it preferable to use an assembler.

To support the family of PIC16Cxx chips, we provide two assemblers. The first assembler, PASM, assembles programs for PIC16C5x chips. And the second assembler, PASMX, assembles programs for the newer chips (16C64, '71, '84,...).

BLANK PAGE

PIC16Cxx Assemblers

Assembler Basics

The purpose of both assemblers is to convert assembly language source code into object code.

The assembler accomplishes its task in two passes:

Pass 0 - The source code is scanned in an attempt to resolve all symbols. This is possible if all *origin*, *define space*, and *equate* directives can be resolved (*equated* symbols may be referenced by *origin* or *define space*). All other symbols can be resolved by byte-offsets which are determined by the mnemonic/operand combinations. If Pass 0 is successful, the assembler will advance to Pass 1. If Pass 0 is unsuccessful, a list of errors will be shown and assembly will be aborted.

Pass 1 - The source code is scanned once more in order to assemble the object code. Since all symbols were resolved in Pass 0, all instructions and miscellaneous directives can be fully resolved in Pass 1. If Pass 1 is successful, an object file containing the assembled code will be created. If Pass 1 is not successful, a list of errors will be shown and assembly will be aborted.

In addition to an object file, the assembler can be used to generate an assembly listing. An assembly listing shows line numbers, equated values, addresses, data, and original source code *(for more information, see the section Generating Assembly Listings later in this chapter).*

Addressing Definitions

Throughout your programs, you'll refer to bits and bytes by their address. Depending upon the instruction being used and the item being referred to, the address will be given in one of the following forms:

addr9	A 9-bit address (within the current 512-word page).
addr11	An 11-bit address (anywhere in program memory).
bit	An address for bitwise operations *Example: PortC.3 = bit 3 of port C*

fr	A file register (RAM) address.
rel	A relative address ranging from -7Fh to +80h.
literal	An immediate 8-bit value.

Data Types

Eight data types are allowed in the assembler. These data types are:

Symbol/label	Binary value
Local symbol/label	ASCII value
Decimal value	Assembly address (origin)
Hex value	EEPROM address (origin)

The examples below demonstrate various data types:

100	Decimal value *100*
18h	Hex value *18*
0A7h	Hex value *A7 (hex values with A-F as the first digit must have a leading zero)*
1011b	Binary value *1011*
‘A’	ASCII value for the letter *A* (65 decimal)
Start	Label *Start*
:loop	Local label *:loop*
$	Current assembly address
%	Current EEPROM address

In addition to single-character text, entire strings can be generated using the RETW instruction:

retw	‘The fox jumped over the lazy dog’

Expressions

Mathematical expressions are used in many instructions. These expressions may be created using the following operators:

&	*Logical AND*	/	*Divide*
|	*Logical OR*	<<	*Shift left*
^	*Exclusive OR*	>>	*Shift right*
+	*Add*	<	*High byte*
-	*Subtract*	>	*Low byte*
*	*Multiply*	.	*Bit address*

Some example expressions:

setb	PortA.0	*Set bit 0 on port A.*
mov	Count+3,#88h	*Store 88h in location Count+3.*
ds	N*2	*Define empty space of Nx2 bytes.*

All expressions are resolved strictly from *left to right*. Please make note of this, since it may affect the result of expressions. For instance, the expression 5+2*4 would normally be resolved as (2*4)+5, for a result of 13. Since our assemblers resolve expressions strictly from left to right, however, the example would be resolved as (5+2)*4, for a result of 28.

Symbols & Labels

Symbols are used to name locations and values within your program. Many people refer to address symbols as “labels”, but both have the same effect. For instance, by assigning a symbol to the start of an important routine, you can later call that routine by its name, rather than its address. And by giving a symbol to a common value, you can refer to it by its name. Rather than typing “3.1415” many times in your program, you can type PI = 3.1415 at the beginning of your program and then use PI wherever you need it.

Symbols may be up to 32 character long. They must begin with a letter or underscore (_) and must contain only letters, numbers, underscores, and colons. Further, if you're labeling an address (such as the start of a routine), the label must start at the beginning of the line.

Here are some exmples of valid symbols:

```
min_count       =       20h
maximum_count   =       21h

begin           mov     min_count,#05h
```

Local Labels

Like normal global lables, local labels may be used to name a location in your program. However, unlike a global label, a local label may be used to name more than one location. This is done by separating occurrences of the same local label with at least one global label.

Local labels have the same syntax rules as global labels, except they must begin with a colon (:), and must be referred to with a colon. Local labels can be referred to from beyond the global label boundary by referring to global label name: local label name.

The following code demonstrates how to use the local label :loop for common looping purposes. In the code below, Routine1, Routine2, and Routine3 are global labels; :loop is a local label.

```
Routine1        mov     count,#100
:loop           call    send_a
                djnz    count,:loop
                ret

Routine2        mov     count,#200
:loop           call    send_b
                djnz    count,:loop
                ret

Routine3        mov     count,#250
                jmp     Routine2:loop
```

Default Symbol Tables

When the assembler is started, its symbol table is initialized with various PIC symbols, such as C for the Carry register, RA for Port A, etc. This saves you from having to define every register and bit in the PIC.

Since the PIC16C5x parts are very similar to each other, the '5x assembler (PASM) uses a set symbol table. The assembler for newer PICs (PASMX), however, changes its symbol table depending upon which PIC it's assembling for. The device directive (explained later) tells the assembler which PIC is being used.

If you'd like to see the symbol table for a particular PIC, please refer to the Assembler Symbol Tables at the end of this manual.

Comments

Comments can be placed anywhere in your source code, beginning at any point on a line and continuing to the end. A semicolon initiates a comment. The following are example comments:

```
;
;Move literal value 61h into w
;

Input       =       10h
Output      =       11h

            mov     Input,#61h      ;Load 61h into Input
            mov     Output,#10h     ;Load 10h into Output

            call    talk_host       ;Call routine to
                                    ;communicate with
                                    ;host
```

Blank lines can be used to provide space between lines and make the code more readable.

Assembler Directives

Assembler directives are instructions that direct the assembler to do something. Directives do many things; some tell the assembler to set aside space for variables, others tell the assembler to include additional source files, and others establish the start address for your program. The directives available in both PIC assemblers are given below:

= Assigns a value to a symbol (same as EQU)

EQU Assigns a value to a symbol (same as =)

ORG Sets the current origin to a new value. This is used to set the program or register address during assembly. For example, the directive ORG 0100h tells the assembler to assemble all subsequent code starting at address 0100h.

DS Defines an amount of free space. No code is generated. This is sometimes used for allocating variable space.

ID Sets the PIC's identification.

PIC16C5x chips have two ID bytes, which can be set to a 2-byte value or checksum. If the ID is CHECKSUM, the programmer software will compute a checksum.

Newer PICs have four 7-bit ID locations, which can be filled with a 4-character text string

INCLUDE Loads another source file during assembly. This allows you to insert an additional source file in your code during assembly. Included source files usually contain common routines or data. By using an INCLUDE directive at the beginning of your program, you can avoid re-typing common information.

Included files may not contain other included files.

DEVICE Establishes the device type, oscillator type, watchdog status, and code-protect status. The following options are used with DEVICE to define the PIC being used:

PIC16C54	*Select PIC16C54*
PIC16C55	*Select PIC16C55*
PIC16C56	*Select PIC16C56*
PIC16C57	*Select PIC16C57*
PIC16C58	*Select PIC16C58*
PIC16C64	*Select PIC16C64*
PIC16C71	*Select PIC16C71*
PIC16C84	*Select PIC16C84*
RC_OSC	*Use RC oscillator*
XT_OSC	*Use XT oscillator*
HS_OSC	*Use HS oscillator*
LP_OSC	*Use LP oscillator*
WDT_ON	*Turn watchdog timer on*
WDT_OFF	*Turn watchdog timer off*
PROTECT_ON	*Turn code-protect on*
PROTECT_OFF	*Turn code-protect off*
PWRT_ON *	*Turn power-up timer on*
PWRT_OFF *	*Turn power-up timer off*

** These options are not available for PIC16C5x chips.*

In programs written for older PICs (PIC16C5x), the DEVICE directive is optional. However, it must be present at the beginning of programs written for newer PICs (’64, ’71, ’84,...). This is because the assembler for the newer PICs (PASMX) needs to make various changes to its internal symbol table.

In addition to giving the assembler information, the DEVICE directive passes useful data to other tools down the line. For instance, the programmer software reads this data to preset options on the screen.

RESET Sets the reset start address. This address is where program execution will start following a reset.

A jump to the given address is inserted at the last location in memory. After the PIC is reset, it starts executing code at the last location, which holds the jump to the given address.

RESET is only available for PIC16C5x chips.

EEORG Sets the current data EEPROM origin to a new value. This is used to set the data EEPROM address during assembly. This directive usually precedes EEDATA.

EEORG is only available for PICs that have EEPROM memory (currently, the '84).

EEDATA Loads data EEPROM with given values. This provides a means of automatically storing values in the data EEPROM when the PIC is programmed. This is handy for storing configuration or start-up information.

EEDATA is only available for PICs that have EEPROM memory (currently, the '84).

Assembler Directive Examples

The following examples demonstrate the directives given above:

```
        DEVICE  PIC16C54,RC_OSC,WDT_OFF,PROTECT_OFF
        DEVICE  PIC16C64,PWRT_OFF,PROTECT_ON
        DEVICE  PIC16C71,XT_OSC,PWRT_ON
        DEVICE  PIC16C84,PROTECT_ON

Digit   =       43h             ;Assign value 43h to Digit
Max     EQU     1Ah             ;Assign value 1Ah to Max

        ORG     10h             ;Set assembly address to 10h
```

```
Count   DS      2           ;Define 2 bytes at 10h & 11h
                            ;Bytes can be referred to
                            ;later as Count and Count+1

        ID      1234h       ;Set PIC16C5x ID to 1234h
        ID      Checksum    ;Set PIC16C5x ID to checksum

        ID      'AMMO'      ;Set new PIC ID to 'AMMO'

        INCLUDE 'KEYS.SRC'  ;Include KEYS.SRC file at
                            ;point of insertion

        RESET   XXYY        ;Set reset jump to location
                            ;XXYY

XXYY    mov     Count,#00   ;This code will be executed
        mov     Count+1,#02 ;whenever the PIC is reset


        EEORG   10h         ;Set EEPROM address to 10h
        EEDATA  02h,88h,34h ;Store 3 bytes in EEPROM
```

Source Code Formatting

We recommend that you format your source code with evenly spaced tabs, preferably 8 spaces each, since this will lend consistency to assembly listings. The assemblers are not case sensitive (except in the instance of strings), so you may follow your own convention for using upper and lower case.

Running the Assembler

To assemble your source code into object code, type the following command at the DOS prompt:

PASM *filename*	Assembles text file *filename.src* into hex file *filename.obj*. This example uses the PIC16C5x assembler.
PASMX *filename*	Assembles text file *filename.src* into hex file *filename.obj*. This example uses the PIC16Cxx assembler.

If you're assembling a program for any of the newer PICs ('64, '71, '84, etc.), you must use PASMX. This assembler operates much like PASM, but has slight internal differences to accomodate the newer PICs.

As shown above, only the name of the source file is given. The assembler uses the same name for the object file, but replaces the extension with ".obj". If you do not specify an extension as part of the source file name, the assembler will assume that the extension is ".src".

As an example, to assemble a file called EXAMPLE.SRC written for PIC16C5x devices, you would type:

```
PASM EXAMPLE
```

The assembler would produce an object file called EXAMPLE.OBJ.

Generating Assembly Listings

It's possible to have the assembler create an assembly listing of your program. An assembly listing is a duplicate of the source code, but with object code information (line number, address, opcode, & data) preceding each original line. To have the assembler create a listing file, simply add "/L" after the filename:

PASM *filename* /L	Assembles text file *filename.src* into hex file *filename.obj* and creates a listing file called *filename.lst*.

Command-Line Options

The PIC assemblers have several options which can be invoked when they are run. These command-line options are shown below:

PASM *filename*	Assembles text file *filename.src* into hex file *filename.obj*. PASM assembler is used for PIC16C5x devices.
PASMX *filename*	Assembles text file *filename.src* into hex file *filename.obj*. PASMX assembler is used for newer PICs ('64, '71).
PASM *filename.xxx*	Assembles text file *filename.xxx* into hex file *filename.obj*.
PASM *filename* /L	Assembles text file *filename.src* into hex file *filename.obj* and creates a listing file called *filename.lst*.
PASM *filename* /S	Assembles text file *filename.src* into hex file *filename.obj*, but suppresses Parallax device data normally included in the file. This may be useful with tools that don't accept files with extra data, such as the PICSTART-16B from Microchip.

All options are available with both PASM and PASMX.

Error Messages

During assembly, any errors will be brought to your attention. You may pause an error list by typing CTRL-S. Or, press ESC or CTRL-C to abort.

The following error messages may occur during assembly. They will always be preceded by "**ERROR IN** xxx**:**", where xxx is the line number where the error occurred. In the case of an include file, the line number will be shown, with the included file's line number in parenthesis.

Address limit of xxxxh was exceeded: Data was assembled at an address which exceeded the limit for the given PIC.

Attempt to divide by 0: An attempt was made to divide a quantity by zero.

Bit number must be from 0 to 7: A bit-address expression attempted to use a bit number greater than 7.

Data was already entered at location xxxxh: An object code location which had already been assigned data, was written to again.

Equate directive must be preceded by a symbol: An "EQU" or "=" directive was not preceded by a necessary symbol.

Illegal mnemonic: The assembler encountered an unknown directive or instruction.

Include files cannot be nested: An INCLUDE directive was found in an included file.

Syntax error in operand: The operand contained an expression which did not follow proper syntax.

Invalid filename for include file: An invalid filename was given in an INCLUDE directive.

Line cannot exceed 256 characters: Line length exceeded the line limit of 256 characters.

Mnemonic field cannot exceed 7 characters: More than 7 characters were in the mnemonic (instruction) field.

Illegal mnemonic/operand combination: The operand structure did not match the instruction's or directive's possibilities.

Literal value must be from 0 to 0FFh: A value which needed to be from 0 to 255, was greater than 255.

Operand field cannot exceed 256 characters: More than 256 characters were in the operand field after expansion by the assembler.

Redefinition of symbol xxxx: An attempt was made to redefine a symbol that was already defined.

Symbol field cannot exceed 32 characters: More than 32 characters were in the symbol field.

(operand) symbol must contain only letters, numbers, '_', and ':': A symbol contained illegal characters.

(operand) symbol is a reserved word: A symbol was identical to a reserved word.

(operand) symbol is too long: A symbol referenced in the operand exceeded 32 characters.

(operand) symbol must begin with a letter or '_': A symbol started with an illegal character.

Symbol table full: The symbol table's 16K limit was exceeded.

Use of unknown symbol xxxx: A symbol was referenced in the operand, which was never declared.

BLANK PAGE

PIC16Cxx Simulator

The PIC simulator is a piece of software that simulates the execution of PIC programs on the PC. Using a listing file (.lst) or code file (.cod) from an assembler or C compiler, the simulator "runs" PIC code on the screen. A single screen shows all of the PIC's registers, as well as the source code given in the input file. Execution can be single-stepped, breakpoints can be set, and registers can be modified.

Many customers find the simulator very useful for preliminary testing and debugging.

Although not discussed in this chapter, the simulator can also be used with our in-circuit emulator products. In this case, the simulator can exercise a real circuit designed to accept a PIC chip. For more information on these, please refer to the ClearView chapters.

BLANK PAGE

PIC16Cxx Simulator

Running the Simulator

To run the simulator, type the following command at the DOS prompt:

PSIM *filename*	Runs the simulator and loads *filename.lst*.

The simulator uses the listing file that is optionally produced by the Parallax assemblers. The extension ".lst" can be omitted, since it's assumed by default. If, however, an extension is given, it will be used.

The simulator can also be used with Microchip's MPASM assembler and Byte Craft's MPC C compiler. In both cases, you must use the code file (.cod) produced by these products (do not use .lst files).

The simulator loads the file and displays the source code in the lower half of the screen. The PIC's registers are displayed in the upper half of the screen.

Scrolling Through Source Code

You can scroll through the source code displayed in the lower half of the screen by using the up and down arrow keys, as well as PGUP, PGDN, HOME and END. The HOME key brings you to the beginning of the address space, while the END key brings you to the end.

Two highlighted lines are always present in the source code display. The "current line" indicates the current line being executed, while the "marker line" is used for special functions, such as setting breakpoints.

The current line is indicated by blue text on a grey background. The current line moves from line to line as your program runs. Any change to the Program Counter (PC register) will move the current line.

The marker line is indicated by yellow text on a black background. The marker line does not move; instead, it remains in the middle of the source code display. By scrolling the source code up and down, you can position a particular line of code in the marker line. When a line is in the marker line, special functions can be performed on the line, such as setting a breakpoint.

Device Type

The simulator determines the device type being simulated by a variety of methods. The recommended method is to include the DEVICE directive in your source code. The simulator locates the directive in the listing file and sets the device type accordingly. If the directive is not present in the file, the simulator determines the device type by the type of listing file and the amount of program space used by the source code. If there are no instructions above address 1FFh hex, the PIC16C54 is assumed. If there are instructions higher than 1FFh, but not higher than 3FFh, the PIC16C56 is assumed. If there are instructions above 3FFh, then the PIC16C57 is assumed.

Another method of selecting the device type is to use the command-line option "/D=". Examples are as follows (they are all equivalent):

PSIM *filename* /D=PIC16C54
PSIM *filename* /D=PIC54
PSIM *filename* /D=54

The final method of selecting the device type is to use the Alt-D command from within the simulator. Pressing Alt-D will pop up a menu where you can use the up and down arrow keys to select the desired device. Press RETURN when the desired device is highlighted.

When loading files into the simulator, keep in mind that the PIC16C5X and new PICs ('71, '84,...) do not have compatible object code. If the device type is set to PIC16C71 while simulating PIC16C5X code, unexpexted results will occur.

Crystal Frequency

The simulator uses a simulated crystal value to calculate the instruction cycle time. The crystal value can be entered by one of two methods. The first method is to enter it in the command-line after the filename using the "/X=" option. Possible suffixes are "Mhz", "khz", and "hz". If no suffix is entered, the frequency is assumed to be in hertz (cycles/second). Some examples are as follows:

PSIM *filename* /X=4Mhz

PSIM *filename* /X=32.768khz
PSIM *filename* /X=32768hz
PSIM *filename* /X=32768

The second method of setting the crystal value is to press Alt-C from within the simulator and then type in a new value. See the Alt-C later in this chapter for further details. If no crystal frequency is entered, the simulator will default to 8 Mhz upon startup.

Setting Breakpoints

As mentioned at the beginning of this chapter, the highlighted line in the middle of the source code display is the marker line. To set a breakpoint, scroll through the code until the desired line is highlighted by the marker, and then press F2. The line will be highlighted in red, which indicates a breakpoint. To clear the breakpoint, press F2 again.

There is no limit to the number of breakpoints you can set.

Modifying Registers

To modify a register during simulation, you may use one of two methods. If your computer has a mouse, you can move the mouse cursor until the desired register is highlighted, then press the left mouse button to increment the register contents or the right mouse button to decrement it. The upper and lower nibbles of a register can be incremented or decremented separately for registers that are displayed in hex. For registers that are displayed in binary, you can change each bit separately. Some of the registers cannot be changed. For instance, you can't change the indirect address register (00h). The indirect address register is not physically implemented in the PIC, so altering it would have no effect. Also, if the selected device is a PIC16C54 or '55, you can't change the upper two bits of the program counter, since these devices do not implement these bits. For the '56, you can't alter the upper bit of the program counter.

If you do not have a mouse, you can select a register by pressing and holding the CTRL key while using the arrow keys to move the cursor. The cursor will move from register to register as you press the arrow keys. When the cursor is on the desired register, release the CTRL key

and type the desired value for the register. If you are modifying a register that is displayed in hex, you can type any valid hex character. If you are modifying a register that is displayed in binary, you can only type in 0's and 1's. You must type in all the numbers that are highlighted before the new value will be accepted. The cursor will disappear when you resume simulation.

Watchdog Timer

The watchdog timer may be enabled or disabled using one of two methods. The first is to include the WDT_ON or WDT_OFF option in the DEVICE directive in your source code. The second method is to press Alt-W from within the simulator. A menu will pop up, allowing you to select "enable" or "disable" with the up and down arrow keys. Press RETURN when the proper selection is highlighted.

Exiting the Simulator

To exit the simulator, press escape ESC or Alt-X. You will be asked to verify your decision. Press "Y" to quit or "N" to return to the simulator.

Function Keys

The use of each function key is described in the following text:

F1 **Help menu.** This displays the help menu. Use the PGDN key to display the second page of the help menu.

F2 **Toggle breakpoint.** Toggle breakpoint at the marker line. To insert a breakpoint at a specific line, use the cursor keys to scroll the program display up and down until the desired line is highlighted by the marker, and then press F2. The line will then turn red to indicate that a breakpoint is set. Pressing F2 again on the same line will clear the breakpoint.

F3 **Clear all breakpoints.**

F4 **Execute to marker.** Execute code until the marker line is reached. Pressing F4 will simulate the program line-

by-line until the marker is reached. “Running...” will appear at the bottom of the screen. When the line is reached, the screen will be updated and the registers that were altered will be highlighted. To stop the simulation while it’s running, press any key.

F5 **Reset time.** Reset the real-time display to zero. This is useful for timing code execution.

F6 **Execute code (with update).** Pressing F6 causes the simulator to start executing code. It will not stop until a key is pressed or a breakpoint is reached. *The screen is updated and changes are highlighted after each line is executed.*

F7 **Step line-by-line.** This causes one line to be executed. Changes in the registers are highlighted after the line is executed. Pressing the space bar also executes one line.

SPACE **Step line-by-line.** Same as F7.

F8 **Execute to next line.** Pressing F8 causes the simulator to execute code until the next line is reached or a breakpoint is reached. This is useful for executing through subroutine calls. If F8 is pressed when the cursor is on a subroutine call, the simulator will execute the code until the line after the call is reached. The user can stop execution at any time by pressing any key. If the F8 key is pressed when the highlighted line is at the end of the address space (1FFh for the PIC16C54, etc.), the simulator will never reach the next line, since it is out of code space. In this case, the simulator will continue running until a key is pressed.

F9 **Execute code (without update).** Pressing F9 causes the simulator to start executing code. It will not stop until a key is pressed or a breakpoint is reached. *The screen is not updated until execution is stopped.*

F10 **Reset PIC.** Pressing F10 simulates a hardware reset.

Alt-C **Set crystal frequency.** The crystal frequency defaults to 8 Mhz, which is used to calculate execution time. To enter a new frequency, press Alt-C. A window will pop up, allowing you to enter the desired value. You can enter the value in hertz (hz), kilohertz (khz), or megahertz (Mhz) by typing the value and the appropriate suffix. If no suffix is entered, the frequency is assumed to be in hertz. The suffix can be upper or lower case. Some examples are: 32768, 32768hz, 32.768khz, 1Mhz, 3.57Mhz.

Alt-D **Select device type.** To select a different device type, press Alt-D. A menu will appear, allowing you to select the desired device. Press ENTER when the desired PIC is highlighted.

You must be careful when using this method. Any code above the address limit of the newly selected device will be lost. Also, keep in mind that the PIC16C5x and new PICs ('71, '84,...) are not object code compatible. If the device is set to PIC16C71 while simulating PIC16C5x code, unexpexted results will occur.

To exit from the device menu without changing the device type, press ESC.

Alt-E **Display EEPROM data.** If the simulated PIC has EEPROM (currently, the '84), you can view the contents of the EEPROM data memory by pressing Alt-E. This is not available when any other PIC is selected.

Alt-F **Load file.** To load a different listing file, press Alt-F. A file menu will appear, allowing you to select a file to load. Listing files are displayed in yellow and directories are in red (only files with a '.lst' extension are shown). To load a file, use the cursor keys to highlight the desired file and then press ENTER. If you need to load a file from a different directory, move the cursor to the desired directory and then press ENTER. The display will list the files in the selected directory for you to choose from. To quit the file menu without loading in a new file, press ESC.

Alt-S **Display stack.** The first two stack locations are displayed on the screen at all times. If the PIC being simulated is a new PIC ('71, '84,...), you can display all stack locations in a pop up window by pressing Alt-S.

Alt-W **Watchdog timer.** To enable / disable the watchdog timer, press Alt-W. A menu will pop up, allowing you to select "enable" or "disable" with the up and down arrow keys. Press RETURN when the proper selection is highlighted.

Alt-F4 **Jump to marker.** Pressing Alt-F4 causes the simulator to jump to the marker line, without executing any instructions. This is useful for jumping directly to code segments that you wish to debug, without having to run through the normal code execution to get there. It's also useful for jumping over long delay loops in your code.

Alt-F7 **Back step.** Pressing Alt-F7 steps backward one line in your code. The entire state of the PIC is stored in a 100-step history buffer. By recalling the history buffer contents, you can step back through your code (up to 100 steps).

Alt-X **Exit simulator.** Pressing Alt-X exits the simulator and returns to DOS. A pop-up prompt will allow you to confirm the exit command.

Advanced Features

Journal Files

All keystrokes entered during simulation are automatically stored in a journal file called PSIM.JRN. This file can be "played back" to re-create a simulator session.

The journal file is automatically created during each simulator session (unless disabled with the "/J-" option when starting the simulator). When the simulator is started, any old journal file will be overwritten. If you want to retain a journal file for later use, rename it or copy it to another directory.

To re-run a journal file, start the simulator by typing the following:

PSIM *filename* /J=*journalfile*

where *journalfile* is the name of the journal file you want to use. This will load *filename* and then execute the keystrokes read from the journal file.

The journal file feature is useful for configuring your simulation session upon startup. For instance, you can initialize an I/O port to a certain value or set breakpoints in your code. This setup can then be executed upon startup of later simulation sessions.

Note that the journal file only records keystrokes – mouse actions are not stored in the file.

Input Stimulus Files

The stimulus file feature allows you to schedule bit changes in I/O port pins at specified times during simulation. This scheduling is controlled via a text file called a *stimulus file*. The "/I=filename" command-line option is used to read the stimulus file.

The stimulus file is an ASCII text file that contains instructions for applying stimuli to the I/O pins during simulation. All commands must be comma separated. White spaces and tabs are ignored. Any line beginning with a semicolon is a comment line.

There are five schedule commands that tell the simulator when a given stimulus should be applied. The five commands are:

t = real	Provide stimulus when execution time reaches real.
c = int	Provide stimulus when cycle counter reaches int.
dt = real	Provide stimulus when delta time reaches real.
dc = int	Provide stimulus when delta cycles reaches int.
pc = int	Provide stimulus when program counter reaches int.

After the schedule command comes the stimulus command. Stimulus commands can be applied to an individual I/O pin or to a full port. They can also be applied to the RTCC pin or to any analog inputs (when simulating a PIC with A/D).

Up to eight bit changes or three port changes can be present on a single line in the stimulus file. The following are valid stimulus commands:

;This is a comment line. It has no effect on the simulator.

pc=8, ra=2Dh, rb=FEh	When the program counter reaches 8, port A will be set to 2Dh and port B will be set to FEh.
t=0.00002, ra0=1, ra3=0	When execution time reaches 0.00002 seconds, bit 0 of port A will be set and bit 3 will be cleared.
c=50, rb0=1	When the cycle counter reaches 50, bit 0 of port B will be set.
dt=0.00001, rb7=1	This sets bit 7 of port B when dt=0.00001 seconds. In this example, it will occur 0.00001 seconds after the last stimulus.
dc=12, rtcc=1	Twelve cycles after the last stimulus, this sets the RTCC pin.
dc=5, ain2=4.80v	Five cycles after the last stimulus, this sets analog input #2 to 4.8 volts.

Command-line Options

The simulator has various options that can be invoked when it's started. These options are shown below:

PSIM *filename*	Runs the simulator and loads *filename.lst*.
PSIM *filename.xxx*	Runs the simulator and loads *filename.xxx*.
PSIM *filename* /D=*device*	Sets device type to *device*, where *device* is "54", "55", "16C71", etc.
PSIM *filename* /E-	Disables serial port checking for ClearView emulators.
PSIM *filename* /E=*port*	Causes the simulator to only look for an emulator on the serial port specified, where *port* is a number from 1 to 4 (note: 1 & 2 work best). Normally, the simulator checks all ports for the presence of emulators. This may disturb other serial devices, such as mice. If you have such problems, try specifying the emulator port, as described above.
PSIM *filename* /F-	Disables serial port checking for Reflection.
PSIM *filename* /F=*port*	Causes the simulator to only look for Reflection on the serial port specified, where *port* is a number from 1 to 4 (note: 1 & 2 work best). Normally, the simulator checks all ports for the presence of Reflection. This may disturb other serial

devices, such as mice. If you have such problems, try specifying the Reflection port, as described above.

PSIM *filename* /I=*stimfile*

Runs the simulator and loads *filename.lst*. Then loads and runs the stimulus file called *stimfile*.

PSIM *filename* /J-

Disables the saving of keystrokes in a journal file.

PSIM *filename* /J=*journalfile*

Runs the simulator and loads *filename.lst*. Then executes keystrokes from *journalfile*.

PSIM *filename* /L

Causes the simulator to run ClearView '5x or ClearView 'xx in "low speed" mode.

To avoid errors, this should be used if you're running the emulator at speeds below 500 kHz.

PSIM *filename* /O=osc

Configures the ClearView hardware to work with the specified oscillator.

Osc may be specified as RC, XT, HS, or LP.

This option may be necessary if the oscillator type is not specified in your source code.

PSIM *filename* /P=port,int

Causes the simulator to use the specified serial port and interrupt for communication with Reflection or ClearView.

Port is the base address (in hex) of the serial port, and *int* is the

	associated interrupt. An example would be "/P=3E8,9".
	This option may be necessary if the serial port uses a non-standard base address and interrupt.
PSIM *filename* /R	Causes the simulator to start with registers in a random state, which more accurately simulates a real PIC.
PSIM *filename* /S	Disables normal stack overflow and underflow error detection.
PSIM *filename* /X=xtal	Sets crystal frequency to *xtal*, where *xtal* is "4mhz", "3.12khz", "1000hz", etc.
PSIM *filename* /?	Display help menu which shows valid command-line options.
	These options may have changed since the printing of this manual.

PIC16Cxx Programmer

The PIC16Cxx Programmer allows you to program, read, and verify PICs. The programmer's single-screen software makes programming a snap. And inexpensive upgrades and adapters keep the programmer current with the growing family of PIC16Cxx chips.

As of January, 1995, the programmer supports the PIC16C5x, 16C64 (with adapter), 16C71, 16C74 (with adapter), and 16C84. Adapters are available with ZIF, SOIC, and SSOP sockets.

BLANK PAGE

PIC16Cxx Programmer

System Requirements

To use the PIC16Cxx Programmer, you will need the following items:

- IBM PC or compatible computer
- 3.5-inch disk drive
- Parallel port
- 128K of RAM
- MS-DOS 2.0 or greater

If you plan to write your own PIC programs, you will also need the following software:

- Text editor or word processor capable of saving ASCII files

Packing List

The programmer package should contain the following items. If any are missing, please let us know.

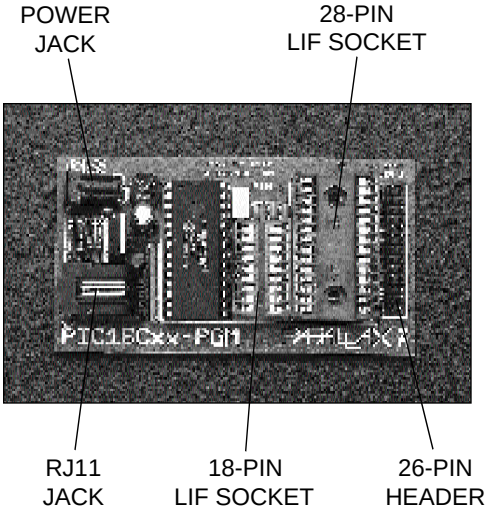
- Programmer PC board
- Power supply*
- DB25-to-RJ11 adapter
- 4-conductor telephone cable (7 feet)
- PIC Tools diskette

* Power supplies are only shipped with orders to the United States and Canada. If your order was shipped to another country, you will need to obtain power supplies with the proper output voltage and current:

12 VAC or 16 VDC, 250 mA

PIC16Cxx Programmer

Hardware Features



Power Jack: Accepts power from external power supply. A "wall pack" power supply is included with orders shipped to the United States and Canada.

RJ11 Jack: Connects to PC parallel port via 4-conductor telephone cable.

18- and 28-pin LIF Sockets: Accept PICs for programming and reading. Voltage is only applied to these sockets when the programmer is programming or reading a device.

26-pin Header: Used for connecting optional socket adapters. At this time, we offer adapters with ZIF, SOIC, and SSOP sockets.

PIC16Cxx Programmer

Programmer Installation

To install your programmer, follow these steps:

1) Plug the power supply into an AC outlet.
2) Plug the power supply cord into the programmer's power jack.
3) Plug one end of the 7-foot telephone cable into the DB25-to-RJ11 adapter.
4) Plug the DB25-to-RJ11 adapter into an available parallel port on your PC.
5) Plug the other end of the telephone cable into the programmer's RJ11 jack.
6) If you have an optional ZIF or surface-mount programming adapter, attach it to the dual-row header on the right side of the programmer. A short 1-inch ribbon cable (supplied with the adapter) makes attachment simple.

PIC16Cxx Programmer

Running the Software

To run the programmer software, make sure you're in the PIC16C5x or PIC16Cxx directory and then type

PEP (16C5x) or PEPX (new PICs)

from the DOS prompt. After several seconds, you'll see the screen shown below *(the PEPX screen is slightly different):*

```
     DEVICE: 54 55 56 57 58  PIC16C5x-PGM/EMU
  OSCILLATOR: RC HS XT LP     (C)1994 Parallax
    WATCHDOG: ON OFF
CODE PROTECT: OFF ON      ID: FFFF CHECKSUM

000- FFF FFF FFF FFF FFF FFF FFF FFF  .......
008- FFF FFF FFF FFF FFF FFF FFF FFF  .......
010- FFF FFF FFF FFF FFF FFF FFF FFF  .......
018- FFF FFF FFF FFF FFF FFF FFF FFF  .......
020- FFF FFF FFF FFF FFF FFF FFF FFF  .......
028- FFF FFF FFF FFF FFF FFF FFF FFF  .......
030- FFF FFF FFF FFF FFF FFF FFF FFF  .......
038- FFF FFF FFF FFF FFF FFF FFF FFF  .......

  HEX ENTRY      ASCII ENTRY      FILL (   )

PROGRAM           EMULATE           ESC EXIT
VERIFY
READ              LOAD:
BLANK CHECK       SAVE:
```

The software will automatically adjust to the type of display you are using. However, if you are using a laptop computer that has a monochrome display, you may have to tell the software to use monochrome attributes. To do this, type PEP /M from the DOS prompt.

The following pages describe the functions available from this screen. To exit the software, press ESC.

Device-Specific Options

There are several options at the top of the screen that are specific to the device you're using. These options are listed on below:

Device: Pressing 'D' selects the type of device you're using. In PEP, the possible settings are 54, 55, 56, and 57.

In PEPX, the possible settings are 64, 71, 74, and 84. *Microchip is introducing new PICs every year, so there may be more settings in PEPX.*

Oscillator: Pressing 'O' selects the type of oscillator you intend to use with the device selected. Possible settings are RC, HS, XT, and LP.

Watchdog: Pressing 'W' toggles the PIC's watchdog timer. Possible settings are ON and OFF.

Timer: Pressing 'T' toggles the PIC's power-up timer. Possible settings are ON and OFF.
This option only appears in PEPX.

Code Protect: Pressing 'C' toggles the device's code protect bit. Possible settings are ON and OFF. If code protect is on, you will not be able to read the device after programming.

ID: In PEP, pressing 'I' toggles the device ID between a checksum and a 2-byte hex value.

In PEPX, pressing 'I' allows you to enter a 4-character text ID.

Programming a PIC

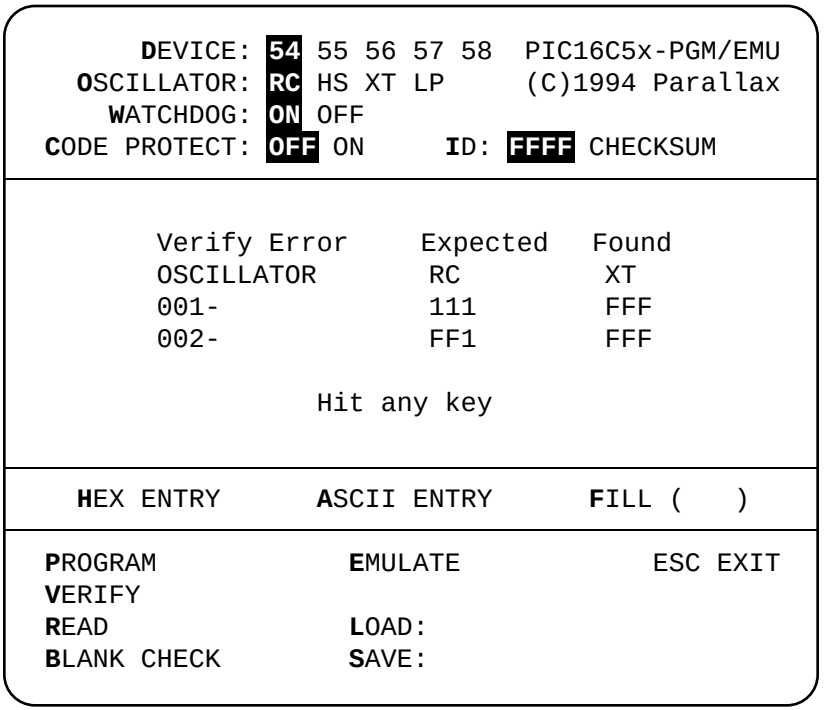
To program a PIC, insert it into the appropriate socket on the programmer and then press 'P'. When programming a part, the software performs three steps: blank check, program, and verify. If any step fails, the software will stop and indicate the error.

If you need to program PICs that are already partically programmed, you can do so by disabling the normal blank-check before programming. To disable blank-check, use the "/d" command-line option when starting PEP or PEPX.

For more information, see the Command-Line Options section at the end of this chapter.

Verifying a Device

To verify that a device equals the data on the screen, press 'V'. The software will compare the following aspects of the device: program memory, oscillator type, watchdog status, power-up timer status (if PEPX), code protect status, and ID. If there are any differences, they will be shown as in the following screen:



Reading a PIC

To read a PIC, insert the device and then press 'R'. The software will read the device's program memory, oscillator type, watchdog status, power-up timer status (if PEPX), code protect status, and ID.

Checking Device Erasure

To find whether or not a device is erased (blank), press 'B'. The software will display the percentage of locations that are erased, as in the following screen:

```
      DEVICE: 54 55 56 57 58  PIC16C5x-PGM/EMU
   OSCILLATOR: RC HS XT LP     (C)1994 Parallax
    WATCHDOG: ON OFF
CODE PROTECT: OFF ON      ID: FFFF CHECKSUM

          ERROR: Device is 79% erased

                 Hit any key

   HEX ENTRY      ASCII ENTRY      FILL (    )

PROGRAM             EMULATE            ESC EXIT
VERIFY
READ                LOAD:
BLANK CHECK         SAVE:
```

If you have an erasable PIC that reports 99% erased, it probably needs further erasing. Sometimes, the oscillator bits are not erased at the factory, so even new chips will give this error.

Loading a File from Disk

To load an object file from disk, press 'L'. Blinking arrows will appear to the right of the word "Load". These arrows indicate that you may enter a filename:

```
      DEVICE: 54 55 56 57 58  PIC16C5x-PGM/EMU
  OSCILLATOR: RC HS XT LP     (C)1994 Parallax
    WATCHDOG: ON OFF
CODE PROTECT: OFF ON      ID: FFFF CHECKSUM

000- FFF FFF FFF FFF FFF FFF FFF FFF  .......
008- FFF FFF FFF FFF FFF FFF FFF FFF  .......
010- FFF FFF FFF FFF FFF FFF FFF FFF  .......
018- FFF FFF FFF FFF FFF FFF FFF FFF  .......
020- FFF FFF FFF FFF FFF FFF FFF FFF  .......
028- FFF FFF FFF FFF FFF FFF FFF FFF  .......
030- FFF FFF FFF FFF FFF FFF FFF FFF  .......
038- FFF FFF FFF FFF FFF FFF FFF FFF  .......

  HEX ENTRY      ASCII ENTRY      FILL (   )

PROGRAM           EMULATE           ESC EXIT
VERIFY
READ              LOAD:>>FILE.OBJ        <<
BLANK CHECK       SAVE:
```

When you have entered the filename, or if the current filename is correct, press RETURN. The software will then attempt to load the specified file. If an error occurs while loading the file, the appropriate message will be displayed.

Saving a File on Disk

To save the buffer data and device options on disk, press 'S'. Blinking arrows will appear to the right of the word "Save". These arrows indicate that you may enter a filename.

When you have entered the filename, or if the current filename is correct, press RETURN. The software will then attempt to save the file. If the file already exists, it will be overwritten. If an error occurs while saving the file, the appropriate message will be displayed.

Saved files contain the buffer data, as well as device type, oscillator type, watchdog status, power-up timer status (if PEPX), code protect status, and device ID.

Moving Around in the Buffer

To move the buffer's cursor, the normal editing keys are used. The effect of each key is shown below:

Left arrow	Moves the cursor 1 location to the left.
Right arrow	Moves the cursor 1 location to the right.
Up arrow	Moves the cursor 1 line up (8 locations).
Down arrow	Moves the cursor 1 line down (8 locations).
PGUP	Moves the cursor 8 lines up (64 locations).
PGDN	Moves the cursor 8 lines down (64 locations).
HOME	Moves the cursor to the first location in the buffer.
END	Moves the cursor to the last location in the buffer.

Editing the Buffer

There are three methods available for editing the buffer. These methods are described on the following page:

Hex entry: Pressing 'H' invokes the hex entry mode. In this mode, any valid hex values that you type are entered into the buffer at the current cursor location. To exit hex entry mode, press RETURN or ESC.

ASCII entry: Pressing 'A' selects the ASCII entry mode. In this mode, any keys that you type are entered into the buffer as 3-digit values. These 3-digit values are comprised of the number '8' and the ASCII value for the key typed. For instance, if you type 'X', the value '878' will be entered into the buffer (78 is the ASCII value for the letter 'X'). The number '8' is the opcode for the "retw" instruction, which loads the subsequent value (78, for instance) into the W register and then returns to the calling routine. This is useful for creating lookup tables.

Fill: Pressing 'F' invokes the fill mode. In this mode, you can fill a portion of the buffer with a specific value.

To fill an area, follow these steps:

1) Place the cursor at the first location within the area to be filled.
2) Press 'F' to invoke the fill mode.
3) Move the cursor to the last location within the area to be filled.
4) Press RETURN. The area from the first location to the last will be filled with the value in the first location.

Command-Line Options

The programmer software has several useful options that can be specified from the DOS command-line. The use of these options is shown below:

PEP	Runs programmer software for PIC16C5x devices.
PEPX	Runs programmer software for newer PICs (16C64, 16C71, 16C74, 16C84,...).
PEP *filename*	Runs software and loads *filename*.
PEP *filename* /D	Disables blank-check function normally used before programming. This must be done if you wish to program PICs that are partially programmed.
PEP *filename* /M	Runs software in monochrome mode.
PEP *filename* /P	Runs software, loads *filename*, and then programs a device. These operations take place entirely from the command-line (the normal display is never seen). This is useful if you want to include the programming process as part of a batch file.

Error Messages

The following list shows the errors that may occur while using the programmer software:

Device is not erased: The PIC you're trying to program is not erased or is not in a socket.

Device is 99% erased: The PIC that you're trying to program is only 99% erased.

Under normal conditions, the PIC must be completely erased before the programmer will program it.

To disable blank-check before programming, use the "/D" option when starting PEP or PEPX.

Also, some erasable PICs may give this error when they're new. This is because the factory does not always erase the oscillator bits. This is easily remedied by erasing the part with a UV eraser.

Verify error: The software found a descrepency between the data on the screen and the data in the PIC.

Difference(s) may be in the program memory, oscillator type, watchdog status, power-up timer status (if PEPX), code protect status, or ID.

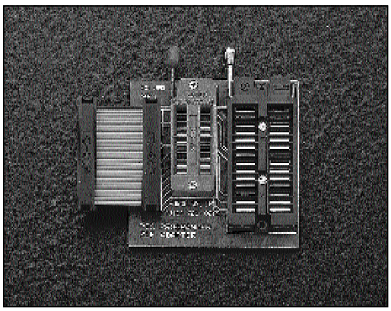
PIC16Cxx-PGM not found: The software could not find the programmer on any parallel port.

Hardware not found: The software could not find the programmer (in PEPX).

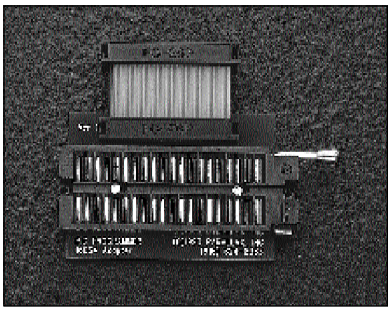
Programming Adapters

On the right side of the programmer, you'll see a dual-row header next to the 28-pin LIF socket. This header allows connection of optional programming adapters.

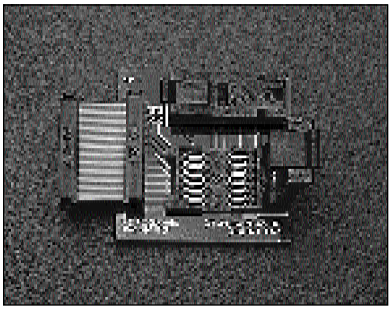
As of November, 1994, four adapters are available:



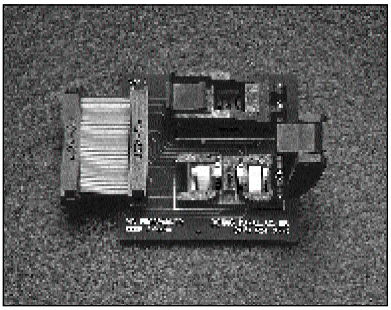
18/28-pin ZIF adapter makes production programming easier.



40-pin ZIF adapter allows programming of larger PICs, such as 16C64 & 16C74.



18/28-pin SOIC adapter supports SOIC surface-mount PICs.

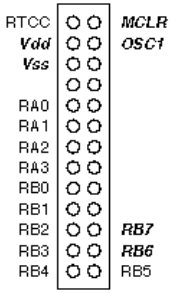


20/28-pin SSOP adapter supports SSOP surface-mount PICs.

Adapter Header Pin-Out

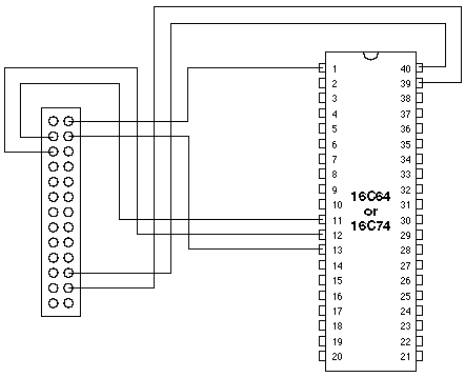
On the previous page, the various programming adapters are shown. As mentioned on that page, the adapters connect to a 26-pin dual-row header, located next to the 28-pin LIF socket.

In some cases, you might prefer to make your own adapters. If so, use the pin-out shown below to connect the header pins to the corresponding pins on the adapter socket(s):



When wiring your adapter, use the following guidelines:

- If the adapter will accept 16C5x devices, you must connect all of the signals shown.
- If the adapter will only accept newer 16Cxx devices (16C64, 16C71, etc.), you only need to connect the six signals shown in ***italic*** print.



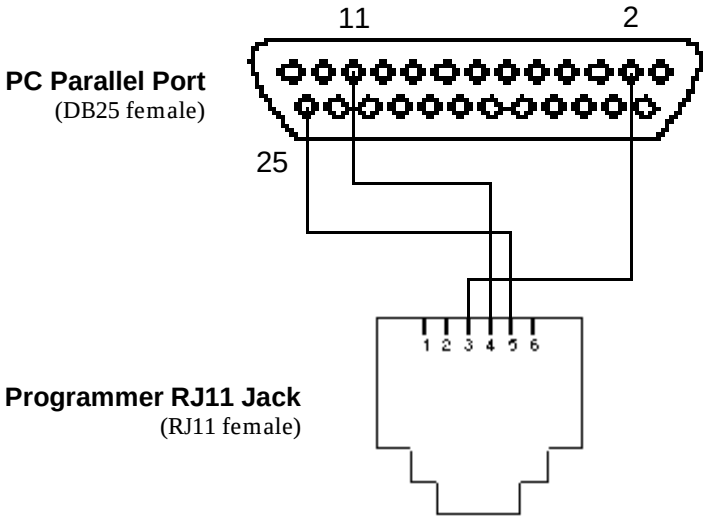
Example circuit for programming the 16C64/74.

Parallel Port Cable Assembly

If you purchased the programmer as part of the Hobbyist Pack, you'll need to make a cable to connect the programmer to your PC.

Most customers use a standard (not flipped) 4-conductor telephone cable; such cables are used with telephones, modems, and other common devices. One end plugs into the programmer's RJ11 jack, and the other end is cut off and replaced with a male DB25 connector.

The cable wiring is shown below:



BLANK PAGE

PIC16Cxx Instruction Set

Parallax Version

BLANK PAGE

PIC16Cxx Instruction Set

Parallax Instruction Set

ADD	fr,#lit	CSNE	fr,#lit	MOVB	bit1,/bit2
ADD	fr1,fr2	CSNE	fr1,fr2	MOVSZ	W,++fr
ADD	fr,W	DEC	fr	MOVSZ	W,--fr
ADD	W,fr	DECSZ	fr	NEG*	fr
ADDB*	fr,bit	DJNZ	fr,addr9	NOP	
AND	fr,#lit	IJNZ	fr,addr9	NOT	fr
AND	fr1,fr2	INC	fr	NOT	W
AND	fr,W	INCSZ	fr	OR	fr,#lit
AND	W,#lit	JB	bit,addr9	OR	fr1,fr2
AND	W,fr	JC	addr9	OR	fr,W
CALL	addr8	JMP	addr9	OR	W,#lit
CJA	fr,#lit,addr9	JMP	PC+W	OR	W,fr
CJA	fr1,fr2,addr9	JMP	W	RET	
CJAE	fr,#lit,addr9	JNB	bit,addr9	RETW	lit,lit,...
CJAE	fr1,fr2,addr9	JNC	addr9	RL	fr
CJB	fr,#lit,addr9	JNZ	addr9	RR	fr
CJB	fr1,fr2,addr9	JZ	addr9	SB	bit
CJBE	fr,#lit,addr9	LCALL*	addr11	SC	
CJBE	fr1,fr2,addr9	LJMP*	addr11	SETB	bit
CJE	fr,#lit,addr9	LSET*	addr11	SKIP	
CJE	fr1,fr2,addr9	MOV	fr,#lit	SLEEP	
CJNE	fr,#lit,addr9	MOV	fr1,fr2	SNB	bit
CJNE	fr1,fr2,addr9	MOV	fr,W	SNC	
CLC		MOV	OPTION,#lit	SNZ	
CLR	fr	MOV	OPTION,fr	STC	
CLR	W	MOV	OPTION,W	STZ	
CLR	WDT	MOV	!port_fr,#lit	SUB	fr,#lit
CLRB	bit	MOV	!port_fr,fr	SUB	fr1,fr2
CLZ		MOV	!port_fr,W	SUB	fr,W
CSA	fr,#lit	MOV	W,#lit	SUBB*	fr,bit
CSA	fr1,fr2	MOV	W,fr	SWAP	fr
CSAE	fr,#lit	MOV	W,/fr	SZ	
CSAE	fr1,fr2	MOV	W,fr-W	TEST	fr
CSB	fr,#lit	MOV	W,++fr	XOR	fr,#lit
CSB	fr1,fr2	MOV	W,--fr	XOR	fr1,fr2
CSBE	fr,#lit	MOV	W,<<fr	XOR	fr,W
CSBE	fr1,fr2	MOV	W,>>fr	XOR	W,#lit
CSE	fr,#lit	MOV	W,<>fr	XOR	W,fr
CSE	fr1,fr2	MOVB	bit1,bit2		

* These instructions are not available in PASMX.

BLANK PAGE

ADD fr,#literal

Add literal into fr

Words: 2 Cycles: 2 Affects: W, C, DC, Z

Operation: Literal is added into fr via W. C will be set if an overflow occurs; otherwise, C will be cleared. DC will be set or cleared depending on whether or not an overflow occurs in the lower nibble. Z will be set if the result is 0; otherwise, Z will be cleared. W is left holding the literal value.

Coding:
```
1100 kkkk kkkk   MOV  W,#lit   (MOVLW  lit)
0001 111f ffff   ADD  fr,W     (ADDWF  fr,1)
```

Example: **Sample** holds 90h.

```
add     sample,#5
```

Sample now holds 95h (90h+5). Both C and DC are *cleared* since no overflow occurred in the byte or in the lower nibble. Z is *cleared* since the result was not 0. W is left holding the literal **5**.

ADD fra,frb

Add frb into fra

Words: 2 Cycles: 2 Affects: W, C, DC, Z

Operation: Frb is added into fra via W. C will be set if an overflow occurs; otherwise, C will be cleared. DC will be set or cleared depending on whether or not an overflow occurs in the lower nibble. Z will be set if the result is 0; otherwise, Z will be cleared. W is left holding the contents of frb.

Coding:
```
0010 000f ffff   MOV  W,frb   (MOVF   frb,0)
0001 111f ffff   ADD  frs,W   (ADDWF  frs,1)
```

Example: **Sum** holds 0F9h and **clicks** holds 10h.

```
add     sum,clicks
```

Sum now holds 9h (the lower byte of 0F9h+10h). C is *set* since an overflow occurred in the byte; however, DC is *cleared* since there was no overflow in the lower nibble. Z is *cleared* since the result was not 0. W is left holding 10h (contents of **clicks**).

ADD fr,W

Add W into fr

Words: 1 Cycles: 1 Affects: C, DC, Z

Operation: W is added into fr. C will be set if an overflow occurrs, otherwise C will be cleared. DC will be set or cleared depending on whether or not an overflow occurs in the lower nibble. Z will be set if the result is 0, otherwise Z will be cleared.

Coding: `0001 111f ffff  ADD fr,W  (ADDWF fr,1)`

Example: **Offset** holds 2Bh and **W** holds 45h.

```
add     offset,w
```

Offset now holds 70h (2Bh+45h). C is *cleared* since no overflow occurred in the byte; however, DC is *set* since there was an overflow in the lower nibble. Z is *cleared* since the result was not 0.

ADD W,fr

Add fr into W

Words: 1 Cycles: 1 Affects: C, DC, Z

Operation: Fr is added into W. C will be set if an overflow occurs, otherwise C will be cleared. DC will be set or cleared depending on whether or not an overflow occurs in the lower nibble. Z will be set if the result is 0, otherwise Z will be cleared.

Coding: `0001 110f ffff  ADD W,fr  (ADDWF fr,0)`

Example: **W** holds 0E8h and **count** holds 18h.

```
add     w,count
```

W now holds 0h (the lower byte of 0E8h+18h). Both C and DC are *set* since an overflow occured in the byte and in the lower nibble. Z is *set* since the result was 0.

ADDB* fr,bit

Add bit into fr

Words: 2 Cycles: 2 Affects: Z

Operation: If bit is set, fr is incremented. If fr is incremented, Z will be set if the result is 0; otherwise, Z will be cleared. This instruction is useful for adding the carry into the upper byte of a double-byte sum after the lower byte has been computed.

Coding:
```
0110 bbbf ffff   SNC  bit   (BTFSC  bit)
0010 101f ffff   INC  fr    (INCF   fr,1)
```

Example: **Sum_high** holds 34h and **C** is set.

```
addb    sum_high,c
```

Sum_high now holds 35h (34h+1b). Z is *cleared* since **sum_high** was incremented and the result was not 0.

* This instruction is not available in PASMX.

AND fr,#literal

AND literal into fr

Words: 2 Cycles: 2 Affects: W, Z

Operation: Literal is AND'd into fr via W. Z will be set if the result is 0; otherwise, Z will be cleared.

Coding:
```
1100 kkkk kkkk   MOV  W,#lit   (MOVLW  lit)
0001 011f ffff   AND  fr,W     (ANDWF  fr,1)
```

Example: **Byte** holds 11011011b.

```
and     byte,#1111b
```

Byte now holds 00001011b (11011011b AND 0000**1111b**). Z is *cleared* since the result was not 0. W is left holding the literal **1111b**.

AND fra,frb

AND frb into fra

Words: 2 Cycles: 2 Affects: W, Z

Operation: Frb is AND'd into fra via W. Z will be set if the result is 0; otherwise, Z will be cleared.

Coding:
```
0010 000f ffff   MOV  W,frb   (MOVF   fr2,0)
0001 011f ffff   AND  fra,W   (ANDWF  fr1,1)
```

Example: **Data** holds 7Ch and **mask** holds 0A5h.

```
and     data,mask
```

Data now holds 24h (7Ch AND 0A5h). Z is *cleared* since the result was not 0. W is left holding 0A5h (**mask**).

AND fr,W

AND W into fr

Words: 1 Cycles: 1 Affects: Z

Operation: W is AND'd into fr. Z will be set if the result is 0; otherwise, Z will be cleared.

Coding:
```
0001 011f ffff   AND  fr,W   (ANDWF  fr,1)
```

Example: **Serial_in** holds 00011010b and **W** holds 11100000b.

```
and     serial_in,w
```

Serial_in now holds 00000000b (00011010b AND 11100000b). Z is *set* since the result was 0.

AND W,#literal

AND literal into W.

Words: 1 Cycles: 1 Affects: Z

Operation: Literal is AND'd into W. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `1110 kkkk kkkk ANDLW literal`

Example: **W** holds 37h (00110111b). The following instruction is executed:

```
and     w,#0F0h
```

W now holds 30h or 00110000b (00110111b AND 11110000b). Z is cleared since the result was not 0.

AND W,fr

AND fr into W.

Words: 1 Cycles: 1 Affects: Z

Operation: Fr is AND'd into W. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `0001 010f ffff ANDWF fr,0`

Example: **W** holds 69h (01101001b) and **filter** holds 96h (10010110b). The following instruction is executed:

```
and     w,filter
```

W now holds 0h or 00000000b (01101001b AND 10010110b). Z is set since the result was 0.

CALL addr8

Call subroutine.

Words: 1 Cycles: 2 Affects: none

Operation: The next instruction address is pushed onto the stack and addr8 is moved to the program counter. The ninth bit of the program counter will be cleared to 0. Therefore, calls are only allowed to the first half of any 512-word page, although the CALL instruction can be anywhere.

Coding:
```
1001 kkkk kkkk CALL    addr8
```

Example: The program counter is at 130h and **send_byte**=29h. The following instruction is executed:

```
call    send_byte
```

The incremented value of the program counter (131h) is pushed onto the stack and a jump to **send_byte** (029h) is executed.

CJA fr,#literal,addr9

Compare fr to literal and jump if above.

Words: 4 Cycles: 4 or 5 (jump) Affects: C, DC, Z

Operation: Fr is compared to literal via W. If fr is greater than literal, a jump to addr9 is executed.

Coding:
```
1100 kkkk kkkk MOVLW   literal^0FFh
0001 110f ffff ADDWF   fr,0
0110 0000 0011 BTFSC   3,0
101k kkkk kkkk GOTO    addr9
```

Example: **X_pos** holds 0CCh, **limit**=0C0h, and **outside**=45h. The following instruction is executed:

```
cja     x_pos,#limit,outside
```

Because **x_pos** (0CCh) is above **limit** (0C0h), a jump to **outside** (45h) is executed. W, C, DC, and Z are scrambled.

CJA fr1,fr2,addr9

Compare fr1 to fr2 and jump if above.

Words: 4 Cycles: 4 or 5 (jump) Affects: W, C, DC, Z

Operation: Fr1 is compared to fr2 via W. If fr1 is greater than fr2, a jump to addr9 is executed.

Coding:
```
0010 000f ffff MOVF   fr1,0
0000 100f ffff SUBWF  fr2,0
0111 0000 0011 BTFSS  3,0
101k kkkk kkkk GOTO   addr9
```

Example: **Input** holds 64h, **max** holds 0ACh, and **update**=168h. The following instruction is executed:

```
cja     input,max,update
```

Because **input** (64h) is not above **max** (0ACh), a jump to **update** (168h) is not executed. W, C, DC, and Z are scrambled.

CJAE fr,#literal,addr9

Compare fr to literal and jump if above or equal.

Words: 4 Cycles: 4 or 5 (jump) Affects: C, DC, Z

Operation: Fr is compared to literal via W. If fr is greater than or equal to literal, a jump to addr9 is executed.

Coding:
```
1100 kkkk kkkk MOVLW  literal
0000 100f ffff SUBWF  fr,0
0110 0000 0011 BTFSC  3,0
101k kkkk kkkk GOTO   addr9
```

Example: **Trigger** holds 10h and **fire**=115h. The following instruction is executed:

```
cjae    trigger,#10h,fire
```

Because **trigger** (10h) is above or equal to 10h, a jump to **fire** (115h) is executed. W, C, DC, and Z are scrambled.

CJAE fr1,fr2,addr9

Compare fr1 to fr2 and jump if above or equal.

Words: 4 Cycles: 4 or 5 (jump) Affects: C, DC, Z

Operation: Fr1 is compared to fr2 via W. If fr1 is greater than or equal to fr2, a jump to addr9 is executed.

Coding:
```
0010 000f ffff MOVF   fr2,0
0000 100f ffff SUBWF  fr1,0
0110 0000 0011 BTFSC  3,0
101k kkkk kkkk GOTO   addr9
```

Example: **Input** holds 64h, **max** holds 0ACh, and **update**=168h. The following instruction is executed:

```
cja     input,max,update
```

Because **input** (64h) is not above **max** (0ACh), a jump to **update** (168h) is not executed. W, C, DC, and Z are scrambled.

CJB fr,#literal,addr9

Compare fr to literal and jump if below.

Words: 4 Cycles: 4 or 5 (jump) Affects: C, DC, Z

Operation: Fr is compared to literal via W. If fr is less than literal, a jump to addr9 is executed.

Coding:
```
1100 kkkk kkkk MOVLW  literal
0000 100f ffff SUBWF  fr,0
0111 0000 0011 BTFSS  3,0
101k kkkk kkkk GOTO   addr9
```

CJB fr1,fr2,addr9

Compare fr1 to fr2 and jump if below.

Words: 4 Cycles: 4 or 5 (jump) Affects: C, DC, Z

Operation: Fr1 is compared to fr2 via W. If fr1 is less than fr2, a jump to addr9 is executed.

Coding:
```
0010 000f ffff MOVF   fr2,0
0000 100f ffff SUBWF  fr1,0
0111 0000 0011 BTFSS  3,0
101k kkkk kkkk GOTO   addr9
```

CJBE fr,#literal,addr9

Compare fr to literal and jump if below or equal.

Words: 4 Cycles: 4 or 5 (jump) Affects: C, DC, Z

Operation: Fr is compared to literal via W. If fr is less than or equal to literal, a jump to addr9 is executed.

Coding:
```
1100 kkkk kkkk MOVLW  /literal
0001 110f ffff ADDWF  fr,0
0111 0000 0011 BTFSS  3,0
101k kkkk kkkk GOTO   addr9
```

CJBE fr1,fr2,addr9

Compare fr1 to fr2 and jump if below or equal.

Words: 4 Cycles: 4 or 5 (jump) Affects: C, DC, Z

Operation: Fr1 is compared to fr2 via W. If fr1 is less than or equal to fr2, a jump to addr9 is executed.

Coding:
```
0010 000f ffff MOVF   fr1,0
0000 100f ffff SUBWF  fr2,0
0110 0000 0011 BTFSS  3,0
101k kkkk kkkk GOTO   addr9
```

CJE fr,#literal,addr9

Compare fr to literal and jump if equal.

Words: 4 Cycles: 4 or 5 (jump) Affects: C, DC, Z

Operation: Fr is compared to literal via W. If fr is equal to literal, a jump to addr9 is executed.

Coding:
```
1100 kkkk kkkk MOVLW  literal
0000 100f ffff SUBWF  fr,0
0110 0100 0011 BTFSC  3,2
101k kkkk kkkk GOTO   addr9
```

CJE fr1,fr2,addr9

Compare fr1 to fr2 and jump if equal.

Words: 4 Cycles: 4 or 5 (jump) Affects: C, DC, Z

Operation: Fr1 is compared to fr2 via W. If fr1 is equal to fr2, a jump to addr9 is executed.

Coding:
```
0010 000f ffff MOVF   fr2,0
0000 100f ffff SUBWF  fr1,0
0110 0100 0011 BTFSC  3,2
101k kkkk kkkk GOTO   addr9
```

CJNE fr,#literal,addr9

Compare fr to literal and jump if not equal.

Words: 4 Cycles: 4 or 5 (jump) Affects: C, DC, Z

Operation: Fr is compared to literal via W. If fr is not equal to literal, a jump to addr9 is executed.

Coding:
```
1100 kkkk kkkk MOVLW  literal
0000 100f ffff SUBWF  fr,0
0111 0100 0011 BTFSS  3,2
101k kkkk kkkk GOTO   addr9
```

CJNE fr1,fr2,addr9

Compare fr1 to fr2 and jump if not equal.

Words: 4 Cycles: 4 or 5 (jump) Affects: C, DC, Z

Operation: Fr1 is compared to fr2 via W. If fr1 is not equal to fr2, a jump to addr9 is executed.

Coding:
```
0010 000f ffff MOVF   fr2,0
0000 100f ffff SUBWF  fr1,0
0111 0100 0011 BTFSS  3,2
101k kkkk kkkk GOTO   addr9
```

CLC

Clear carry.

Words: 1 Cycles: 1 Affects: C

Operation: C is cleared to 0.

Coding:
```
0100 0000 0011 BCF    3,0
```

Example: The carry is set. The following instruction is executed:

clc

The carry is now cleared.

CLR fr

Clear fr.

Words: 1 Cycles: 1 Affects: Z

Operation: Fr is cleared to 0. Z is set to 1.

Coding: `0000 011f ffff CLRF fr`

Example: **Counter** holds 45h. The following instruction is executed:

clr counter

Counter now holds 0h and Z is set.

CLR W

Clear W.

Words: 1 Cycles: 1 Affects: Z

Operation: W is cleared to 0. Z is set to 1.

Coding: `0000 0100 0000 CLRW`

Example: **W** holds 0F3h. The following instruction is executed:

clr w

W now holds 0h and Z is set.

CLR WDT

Clear the watchdog timer.

Words: 1 Cycles: 1 Affects: TO, PD

Operation: The watchdog timer is cleared, along with the prescaler, if assigned. TO and PD are set to 1.

Coding: `0000 0000 0100 CLRWDT`

Example: The prescaler holds 100b and is assigned to the watchdog timer. The following instruction is executed:

```
clr     wdt
```

The watchdog timer is cleared and the prescaler now holds 000b. TO and PD are set.

CLRB bit

Clear bit.

Words: 1 Cycles: 1 Affects: none

Operation: Bit is cleared to 0.

Coding: `0100 bbbf ffff BCF bit`

Example: **Out_bit** is set. The following instruction is executed:

```
clrb    out_bit
```

Out_bit is now cleared.

Note: The Parallax assemblers define a bit as ***port.bitposition***, as in the following examples:

RA.3 = *bit 3 of port A*
PortB.0 = *bit 0 of port B*

CLZ

Clear zero.

Words: 1 Cycles: 1 Affects: Z

Operation: Z is cleared to 0.

Coding:
```
0100 0100 0011 BCF     3,2
```

Example: Z is cleared. The following instruction is executed:

clz

Z remains cleared.

CSA fr,#literal

Compare fr to literal and skip if above.

Words: 3 Cycles: 3 or 4 (skip) Affects: C, DC, Z

Operation: Fr is compared to literal via W. If fr is greater than literal, the following instruction word is skipped.

Coding:
```
1100 kkkk kkkk MOVLW   /literal
0001 110f ffff ADDWF   fr,0
0111 0000 0011 BTFSS   3,0
```

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, CSA may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following CSA is a single-word instruction.**

CSA fr1,fr2

Compare fr1 to fr2 and skip if above.

Words: 3 Cycles: 3 or 4 (skip) Affects: C, DC, Z

Operation: Fr1 is compared to fr2 via W. If fr1 is greater than fr2, the following instruction word is skipped.

Coding:
```
0010 000f ffff MOVF   fr1,0
0000 100f ffff SUBWF  fr2,0
0110 0000 0011 BTFSC  3,0
```

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, CSA may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following CSA is a single-word instruction.**

CSAE fr,#literal

Compare fr to literal and skip if above or equal.

Words: 3 Cycles: 3 or 4 (skip) Affects: C, DC, Z

Operation: Fr is compared to literal via W. If fr is greater than or equal to literal, the following instruction word is skipped.

Coding:
```
1100 kkkk kkkk MOVLW  literal
0000 100f ffff SUBWF  fr,0
0111 0000 0011 BTFSS  3,0
```

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, CSAE may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following CSAE is a single-word instruction.**

CSAE fr1,fr2

Compare fr1 to fr2 and skip if above or equal.

Words: 3 Cycles: 3 or 4 (skip) Affects: C, DC, Z

Operation: Fr1 is compared to fr2 via W. If fr1 is greater than or equal to fr2, the following instruction word is skipped.

Coding:
```
0010 000f ffff MOVF   fr2,0
0000 100f ffff SUBWF  fr1,0
0111 0000 0011 BTFSS  3,0
```

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, CSAE may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following CSAE is a single-word instruction.**

CSB fr,#literal

Compare fr to literal and skip if below.

Words: 3 Cycles: 3 or 4 (skip) Affects: C, DC, Z

Operation: Fr is compared to literal via W. If fr is less than literal, the following instruction word is skipped.

Coding:
```
1100 kkkk kkkk MOVLW  literal
0000 100f ffff SUBWF  fr,0
0110 0000 0011 BTFSC  3,0
```

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, CSB may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following CSB is a single-word instruction.**

CSB fr1,fr2

Compare fr1 to fr2 and skip if below.

Words: 3 Cycles: 3 or 4 (skip) Affects: C, DC, Z

Operation: Fr1 is compared to fr2via W. If fr1 is less than fr2, the following instruction word is skipped.

Coding:
```
0010 000f ffff MOVF   fr2,0
0000 100f ffff SUBWF  fr1,0
0110 0000 0011 BTFSC  3,0
```

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, CSB may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following CSB is a single-word instruction.**

CSBE fr,#literal

Compare fr to literal and skip if below or equal.

Words: 3 Cycles: 3 or 4 (skip) Affects: C, DC, Z

Operation: Fr is compared to literal via W. If fr is less than or equal to literal, the following instruction word is skipped.

Coding:
```
1100 kkkk kkkk MOVLW  /literal
0001 110f ffff ADDWF  fr,0
0110 0000 0011 BTFSC  3,0
```

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, CSBE may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following CSBE is a single-word instruction.**

CSBE fr1,fr2

Compare fr1 to fr2 and skip if below or equal.

Words: 3 Cycles: 3 or 4 (skip) Affects: C, DC, Z

Operation: Fr1 is compared to fr2 via W. If fr1 is less than or equal to fr2, the following instruction word is skipped.

Coding:
```
0010 000f ffff MOVF   fr1,0
0000 100f ffff SUBWF  fr2,0
0111 0000 0011 BTFSS  3,0
```

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, CSBE may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following CSBE is a single-word instruction.**

CSE fr,#literal

Compare fr to literal and skip if equal.

Words: 3 Cycles: 3 or 4 (skip) Affects: C, DC, Z

Operation: Fr is compared to literal via W. If fr is equal to literal, the following instruction word is skipped.

Coding:
```
1100 kkkk kkkk MOVLW  literal
0000 100f ffff SUBWF  fr,0
0111 0100 0011 BTFSS  3,2
```

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, CSE may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following CSE is a single-word instruction.**

CSE fr1,fr2

Compare fr1 to fr2 and skip if equal.

Words: 3 Cycles: 3 or 4 (skip) Affects: C, DC, Z

Operation: Fr1 is compared to fr2 via W. If fr1 is equal to fr2, the following instruction word is skipped.

Coding:
```
0010 000f ffff MOVF   fr2,0
0000 100f ffff SUBWF  fr1,0
0111 0100 0011 BTFSS  3,2
```

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, CSE may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following CSE is a single-word instruction.**

CSNE fr,#literal

Compare fr to literal and skip if not equal.

Words: 3 Cycles: 3 or 4 (skip) Affects: C, DC, Z

Operation: Fr is compared to literal via W. If fr is not equal to literal, the following instruction word is skipped.

Coding:
```
1100 kkkk kkkk MOVLW  literal
0000 100f ffff SUBWF  fr,0
0110 0100 0011 BTFSC  3,2
```

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, CSNE may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following CSNE is a single-word instruction.**

CSNE fr1,fr2

Compare fr1 to fr2 and skip if not equal.

Words: 3 Cycles: 3 or 4 (skip) Affects: C, DC, Z

Operation: Fr1 is compared to fr2 via W. If fr1 is not equal to fr2, the following instruction word is skipped.

Coding:
```
0010 000f ffff MOVF   fr2,0
0000 100f ffff SUBWF  fr1,0
0110 0100 0011 BTFSC  3,2
```

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, CSNE may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following CSNE is a single-word instruction.**

DEC fr

Decrement fr.

Words: 1 Cycles: 1 Affects: Z

Operation: Fr is decremented. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding:
```
0000 111f ffff DECF   fr,1
```

Example: **Points** holds 29h. The following instruction is executed:

```
dec     points
```

Points now holds 28h. Z is cleared since the result was not 0.

DECSZ fr

Decrement fr and skip if zero.

Words: 1 Cycles: 1 or 2 (skip) Affects: none

Operation: Fr is decremented. The next instruction word will be skipped if the result was 0.

Coding:
```
0010 111f ffff DECFSZ  fr,1
```

Example: **Blip** holds 1h. The following instruction is executed:

```
decsz   blip
```

Blip now holds 0h. The following instruction word will be skipped since the result was 0.

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, DECSZ may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following DECSZ is a single-word instruction.**

DJNZ fr,addr9

Decrement fr and jump if not zero.

Words: 2 Cycles: 2 or 3 (jump) Affects: none

Operation: Fr is decremented. If the result is not 0, a jump to addr9 is executed.

Coding:
```
0010 111f ffff DECFSZ  fr,1
101k kkkk kkkk GOTO    addr9
```

Example: **Attempts** holds 41h and **try_again**=37h. The following instruction is executed:

```
djnz    attempts,try_again
```

Attempts now holds 40h. Since the result was not 0, a jump to **try_again** (37h) is executed.

IJNZ fr,addr9

Increment fr and jump if not zero.

Words: 2 Cycles: 2 or 3 (jump) Affects: none

Operation: Fr is incremented. If the result is not 0, a jump to addr9 is executed.

Coding:
```
0011 111f ffff INCFSZ  fr,1
101k kkkk kkkk GOTO    addr9
```

Example: **Counter** holds 0FFh. The following instruction is executed:

```
ijnz    counter,loop
```

Counter now holds 0h. Since the result was 0, execution continues at the next instruction. Had the result been other than 0, a jump to **loop** would have been executed.

INC fr

Increment fr.

Words: 1 Cycles: 1 Affects: Z

Operation: Fr is incremented. Z will be set if the result was 0, otherwise Z will be cleared.

Coding:
```
0010 101f ffff INCF    fr,1
```

Example: **Scan** holds 0B5h. The following instruction is executed:

```
inc     scan
```

Scan now holds 0B6h. Z is cleared since the result was not 0.

INCSZ fr

Increment fr and skip if zero.

Words: 1 Cycles: 1 or 2 (skip) Affects: none

Operation: Fr is incremented. The next instruction word will be skipped if the result was 0.

Coding:
```
0011 111f ffff INCFSZ  fr,1
```

Example: **Turns** holds 3h. The following instruction is executed:

```
incsz   turns
```

Turns now holds 4h. The following instruction word will not be skipped since the result was not 0.

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, INCSZ may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following INCSZ is a single-word instruction.**

JB bit,addr9

Jump if bit.

Words: 2 Cycles: 2 or 3 (jump) Affects: none

Operation: If bit is set, a jump to addr9 is executed.

Coding:
```
0110 bbbf ffff BTFSC   bit
101k kkkk kkkk GOTO    addr9
```

Example: **Flag** is set. The following instruction is executed:

```
jb      flag,process
```

Since **flag** is set, a jump to **process** is executed. Had **flag** been clear, execution would have continued at the next instruction.

JC addr9

Jump if carry.

Words: 2 Cycles: 2 or 3 (jump) Affects: none

Operation: If the carry bit is set, a jump to addr9 is executed.

Coding:
```
0110 0000 0011 BTFSC   3,0
101k kkkk kkkk GOTO    addr9
```

Example: The carry is clear. The following instruction is executed:

```
jc      handle_bit
```

Since the carry is clear, execution continues at the next instruction. Had the carry been set, a jump to **handle_bit** would have been executed.

JMP addr9

Jump to address.

Words: 1 Cycles: 2 Affects: none

Operation: The lower 9-bits of the literal *addr9* is moved into the program counter.

Coding:
```
101k kkkk kkkk GOTO    addr9
```

Example: The following instruction is executed:

```
jmp     main_loop
```

The program counter now holds the value **main_loop**, from which address execution will proceed.

JMP PC+W

Jump to PC+W.

Words: 1 Cycles: 2 Affects: C, DC, Z

Operation: W+1 is added into the program counter. The 9th bit of the program counter is always cleared to 0, so the jump destination will be in the first 256 words of any 512-word page. This instruction is useful for jumping into lookup tables comprised of RETW data, or jumping to particular routines. The flags are set as they would be by an ADD instruction.

Coding: `0001 111f ffff ADDWF 2,1`

Example: The program counter holds 18h and W holds 10h. The following instruction is executed:

```
jmp     pc+w
```

The program counter now holds 29h (18h+10h+1h), from which address execution will proceed.

JMP W

Jump to W.

Words: 1 Cycles: 2 Affects: none

Operation: W is moved into the program counter. The 9th bit of the program counter is always cleared to 0, so the jump destination will be in the first 256 words of any 512-word page. This instruction is useful for jumping into lookup tables comprised of RETW data, or jumping to particular routines.

Coding: **`0000 001f ffff MOVWF 2`**

Example: **W** holds 8h. The following instruction is executed:

```
jmp     w
```

The program counter now holds 8h, from which address execution will proceed.

JNB bit,addr9

Jump if not bit.

Words: 2 Cycles: 2 or 3 (jump) Affects: none

Operation: If bit reads 0, a jump to addr9 is executed.

Coding:
```
0111 bbbf ffff BTFSS   bit
101k kkkk kkkk GOTO    addr9
```

Note: The Parallax assemblers define a bit as ***port.bitposition***, as in the following examples:

RA.3 = *bit 3 of port A*
PortB.0 = *bit 0 of port B*

JNC addr9

Jump if not carry.

Words: 2 Cycles: 2 or 3 (jump) Affects: none

Operation: If C is 0, a jump to addr9 is executed.

Coding:
```
0111 0000 0011 BTFSS   3,0
101k kkkk kkkk GOTO    addr9
```

JNZ addr9

Jump if not zero.

Words: 2 Cycles: 2 or 3 (jump) Affects: none

peration: If Z is 0, a jump to addr9 is executed.

Coding:
```
0111 0100 0011 BTFSS   3,2
101k kkkk kkkk GOTO    addr9
```

JZ addr9

Jump if zero.

Words: 2 Cycles: 2 or 3 (jump) Affects: none

Operation: If Z is 1, a jump to addr9 is executed.

Coding:
```
0110 0100 0011 BTFSC   3,2
101k kkkk kkkk GOTO    addr9
```

LCALL* addr11

Long call.

Words: 1-3 Cycles: 2-4 Affects: none

Operation: Depending on the device size, from zero to two BCF/BSF instructions will be assembled to point the page pre-select bits to addr11's page. The bit set/clear instructions are followed by a CALL to addr11. This instruction is only useful for the PIC16C56 and '57.

Coding:
```
(010x 1xx0 0011        BCF/BSF        3,x)
(010x 1xx0 0011        BCF/BSF        3,x)
1000 kkkk kkkk CALL    addr11
```

Note: Please note that LCALL does not set the page select bits upon return to the calling routine. Therefore, you should set these bits upon returning. This can easily be done using LSET $, which sets the page select bits to the current page.

* This instruction is not available in PASMX.

LJMP* addr11

Long jump.

Words: 1-3 Cycles: 2-4 Affects: none

Operation: Depending on the device size, from zero to two BCF/BSF instructions will be assembled to point the page pre-select bits to addr11's page. The bit set/clear instructions are followed by a jump to addr11. This instruction is only useful for the PIC16C56 and '57.

Coding:
```
(010x 1xx0 0011        BCF/BSF        3,x)
(010x 1xx0 0011        BCF/BSF        3,x)
101k kkkk kkkk GOTO    addr11
```

* This instruction is not available in PASMX.

LSET* addr11

Long set.

Words: 0-2 Cycles: 0-2 Affects: none

Operation: Depending on the device size, from zero to two BCF/BSF instructions will be assembled to point the page pre-select bits to addr11's page. This instruction is only useful for the PIC16C56 and '57.

Coding:
```
(010x 1xx0 0011        BCF/BSF        3,x)
(010x 1xx0 0011        BCF/BSF        3,x)
```

* This instruction is not available in PASMX.

MOV fr,#literal

Move literal into fr.

Words: 2 Cycles: 2 Affects: none

Operation: Literal is moved into fr via W.

Coding:
```
1100 kkkk kkkk MOVLW   literal
0000 001f ffff MOVWF   fr
```

MOV fr1,fr2

Move fr2 into fr1.

Words: 2 Cycles: 2 Affects: Z

Operation: Fr2 is moved into fr1 via W. Z will be set to 1 if the value moved was 0, otherwise Z will be cleared to 0.

Coding:
```
0010 000f ffff MOVF   fr2,0
0000 001f ffff MOVWF  fr1
```

MOV fr,W

Move W into fr.

Words: 1 Cycles: 1 Affects: none

Operation: W is moved into fr.

Coding:
```
0000 001f ffff MOVWF  fr
```

MOV OPTION,#literal

Move literal into OPTION.

Words: 2 Cycles: 2 Affects: none

Operation: Literal is moved into OPTION via W.

Coding:
```
1100 kkkk kkkk MOVLW   literal
0000 0000 0010 OPTION
```

Note: When using any of the newer PICs (16C64, 16C71, 16C84,...), you must be in the proper bank (usually bank 1) for this instruction to function correctly. If your program is not in the proper bank, another register (not OPTION) will be affected.

MOV OPTION,fr

Move fr into OPTION.

Words: 2 Cycles: 2 Affects: Z

Operation: Fr is moved into OPTION via W. Z will be set to 1 if the value moved was 0, otherwise Z will be cleared to 0.

Coding:
```
0010 000f ffff MOVF    fr,0
0000 0000 0010 OPTION
```

Note: When using any of the newer PICs (16C64, 16C71, 16C84,...), you must be in the proper bank (usually bank 1) for this instruction to function correctly. If your program is not in the proper bank, another register (not OPTION) will be affected.

MOV OPTION,W

Move W into OPTION.

Words: 1 Cycles: 1 Affects: none

Operation: W is moved into OPTION.

Coding: `0000 0000 0010 OPTION`

Note: When using any of the newer PICs (16C64, 16C71, 16C84,...), you must be in the proper bank (usually bank 1) for this instruction to function correctly. If your program is not in the proper bank, another register (not OPTION) will be affected.

MOV !port_fr,#literal

Move literal into port_fr's I/O control register.

Words: 2 Cycles: 2 Affects: none

Operation: Literal is moved into the I/O control register of port_fr via W. A "1" bit in W disables the corresponding port pin's output buffer, allowing input use, while a "0" bit enables the output buffer for high or low output. Port_fr must be 5, 6, or 7.

Coding:
```
1100 kkkk kkkk MOVLW   literal
0000 0000 0fff TRIS    port_fr
```

MOV !port_fr,fr

Move fr into port_fr's I/O control register.

Words: 2 Cycles: 2 Affects: Z

Operation: Fr is moved into the I/O control register of port_fr via W. A "1" bit in W disables the corresponding port pin's output buffer, allowing input use, while a "0" bit enables the output buffer for high or low output. Z will be set to 1 if the value moved was 0, otherwise Z will be cleared to 0. Port_fr must be 5, 6, or 7.

Coding:
```
0010 000f ffff MOVF    fr,0
0000 0000 0fff TRIS    port_fr
```

MOV !port_fr,W

Move W into port_fr's I/O control register.

Words: 1 Cycles: 1 Affects: none

Operation: W is moved into the I/O control register of port_fr. A "1" bit in W disables the corresponding port pin's output buffer, allowing input use, while a "0" bit enables the output buffer for high or low output. Port_fr must be 5, 6, or 7.

Coding:
```
0000 0000 0fff TRIS    port_fr
```

MOV W,#literal

Move literal into W.

Words: 1 Cycles: 1 Affects: none

Operation: Literal is moved into W.

Coding: `1100 kkkk kkkk MOVLW literal`

MOV W,fr

Move fr into W.

Words: 1 Cycles: 1 Affects: Z

Operation: Fr is moved into W. Z will be set to 1 if the value moved was 0, otherwise Z will cleared to 0.

Coding: `0010 000f ffff MOVF fr,0`

MOV W,/fr

Move not fr into W.

Words: 1 Cycles: 1 Affects: Z

Operation: The one's complement of fr is moved into W. Z will be set to 1 if the result was 0, otherwise Z will cleared to 0.

Coding: `0010 010f ffff COMF fr,0`

MOV W,fr-W

Move fr-W into W.

Words: 1 Cycles: 1 Affects: C, DC, Z

Operation: W is subtracted from fr and the result is stored in W. C will be cleared to 0 if an underflow occurred, otherwise C will be set to 1. DC will be cleared or set depending on whether or not an underflow occurred in the least-siginificant nibble. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `0000 100f ffff SUBWF fr,0`

MOV W,++fr

Move the incremented value of fr into W.

Words: 1 Cycles: 1 Affects: Z

Operation: The incremented value of fr is moved into W. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `0010 100f ffff INCF fr,0`

MOV W,--fr

Move the decremented value of fr into W.

Words: 1 Cycles: 1 Affects: Z

Operation: The decremented value of fr is moved into W. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `0000 110f ffff DECF fr,0`

MOV W,<<fr

Move the left-rotated value of fr into W.

Words: 1 Cycles: 1 Affects: C

Operation: The left-rotated value of fr is moved into W. On entry, C must hold the value to be shifted into the least-significant bit of the fr value. On exit, C will hold the previous most-significant bit of the fr value.

Coding: `0011 010f ffff RLF fr,0`

MOV W,>>fr

Move the right-rotated value of fr into W.

Words: 1 Cycles: 1 Affects: C

Operation: The right-rotated value of fr is moved into W. On entry, C must hold the value to be shifted into the most-significant bit of the fr value. On exit, C will hold the previous least-significant bit of the fr value.

Coding: `0011 000f ffff RRF fr,0`

MOV W,<>fr

Move the nibble-swapped value of fr into W.

Words: 1 Cycles: 1 Affects: none

Operation: The nibble-swapped value of fr is moved into W.

Coding:
```
0011 100f ffff SWAPF   fr,0
```

MOVB bit1,bit2

Move bit2 to bit1.

Words: 4 Cycles: 4 Affects: none

Operation: Bit2 is moved to bit1.

Coding:
```
0111 bbbf ffff BTFSS   bit2
0100 bbbf ffff BCF     bit1
0110 bbbf ffff BTFSC   bit2
0101 bbbf ffff BSF     bit1
```

Note: The Parallax assemblers define a bit as ***port.bitposition***, as in the following examples:

RA.3 = *bit 3 of port A*
PortB.0 = *bit 0 of port B*

MOVB bit1,/bit2

Move not bit2 to bit1.

Words: 4 Cycles: 4 Affects: none

Operation: The complement of bit2 is moved to bit1.

Coding:
```
0110 bbbf ffff BTFSC   bit2
0100 bbbf ffff BCF     bit1
0111 bbbf ffff BTFSS   bit2
0101 bbbf ffff BSF     bit1
```

Note: The Parallax assemblers define a bit as ***port.bitposition***, as in the following examples:

RA.3 = *bit 3 of port A*
PortB.0 = *bit 0 of port B*

MOVSZ W,++fr

Move the incremented value of fr into W and skip if zero.

Words: 1 Cycles: 1 or 2 (skip) Affects: none

Operation: The incremented value of fr is moved into W. The next instruction word will be skipped if the result was 0.

Coding:
```
0011 110f ffff INCFSZ  fr,0
```

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, MOVSZ may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following MOVSZ is a single-word instruction.**

MOVSZ W,--fr

Move the decremented value of fr into W and skip if zero.

Words: 1 Cycles: 1 or 2 (skip) Affects: none

Operation: The decremented value of fr is moved into W. The next instruction word will be skipped if the result was 0.

Coding:
```
0010 110f ffff DECFSZ  fr,0
```

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, MOVSZ may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following MOVSZ is a single-word instruction.**

NEG fr

Negate fr.

Words: 2 Cycles: 2 Affects: Z

Operation: Fr is converted into its two's complement value. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding:
```
0010 011f ffff COMF    fr,1
0010 101f ffff INCF    fr,1
```

NOP

No operation.

Words: 1 Cycles: 1 Affects: none

Operation: none

Coding: `0000 0000 0000 NOP`

NOT fr

Not fr.

Words: 1 Cycles: 1 Affects: Z

Operation: Fr is converted into its one's complement value. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `0010 011f ffff COMF fr,1`

NOT W

Not W.

Words: 1 Cycles: 1 Affects: Z

Operation: W is converted into its one's complement value. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding:
```
1111 1111 1111 XORLW   0FFh
```

OR fr,#literal

OR literal into fr.

Words: 2 Cycles: 2 Affects: Z

Operation: Literal is OR'd into fr via W. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding:
```
1100 kkkk kkkk MOVLW   literal
0001 001f ffff IORWF   fr,1
```

OR fr1,fr2

OR fr2 into fr1.

Words: 2 Cycles: 2 Affects: Z

Operation: Fr2 is OR'd into fr1 via W. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding:
```
0010 000f ffff MOVF   fr2,0
0001 001f ffff IORWF  fr1,1
```

OR fr,W

OR W into fr.

Words: 1 Cycles: 1 Affects: Z

Operation: W is OR'd into fr. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding:
```
0001 001f ffff IORWF  fr,1
```

OR W,#literal

OR literal into W.

Words: 1 Cycles: 1 Affects: Z

Operation: Literal is OR'd into W. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `1101 kkkk kkkk IORLW literal`

OR W,fr

OR fr into W.

Words: 1 Cycles: 1 Affects: Z

Operation: Fr is OR'd into W. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `0001 000f ffff IORWF fr,0`

RET

Return from subroutine.

Words: 1 Cycles: 2 Affects: none

Operation: The next stack value is moved into the program counter. W is cleared to 0.

Coding:
```
1000 0000 0000 RETLW   0
```

RETW literal1,literal2,...

Assemble RET's which load W with literal data.

Words: ? Cycles: 2 per RETLW Affects: none

Operation: A list of RET's with literal data in the W area is assembled, which can be accessed by JMP PC+W or JMP W instructions. This is useful for lookup tables.

Coding:
```
1000 kkkk kkkk          RETLW  literal1
(1000 kkkk kkkk         RETLW  literal2)
(1000 kkkk kkkk         RETLW  ...)
```

Examples:
```
jmp    pc+w                  ;Jump to byte at
                             ;location pc+w

retw   00100011b             ;Return with w
retw   00h,01h,02h,03h       ;holding
retw   'Enter cycle count'   ;appropriate value
```

RL fr

Rotate left fr.

Words: 1 Cycles: 1 Affects: C

Operation: Fr is rotated left. On entry, C must hold the value to be shifted into the least-significant bit of the fr value. On exit, C will hold the previous most-significant bit of the fr value.

Coding: `0011 011f ffff RLF fr,1`

RR fr

Rotate right fr.

Words: 1 Cycles: 1 Affects: C

Operation: Fr is rotated right. On entry, C must hold the value to be shifted into the most-significant bit of the fr value. On exit, C will hold the previous least-significant bit of the fr value.

Coding: `0011 001f ffff RRF fr,1`

SB bit

Skip if bit.

Words: 1 Cycles: 1 or 2 (skip) Affects: none

Operation: If bit reads 1, the following instruction word is skipped.

Coding: `0111 bbbf ffff BTFSS bit`

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, SB may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following SB is a single-word instruction.**

Note: The Parallax assemblers define a bit as ***port.bitposition***, as in the following examples:

RA.3 = *bit 3 of port A*
PortB.0 = *bit 0 of port B*

SC

Skip if carry.

Words: 1 Cycles: 1 or 2 (skip) Affects: none

Operation: If C is 1, the following instruction word is skipped.

Coding: `0111 0000 0011 BTFSS 3,0`

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, SC may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following SC is a single-word instruction.**

SETB bit

Set bit.

Words: 1 Cycles: 1 Affects: none

Operation: Bit is set to 1.

Coding: `0101 bbbf ffff BSF bit`

Note: The Parallax assemblers define a bit as ***port.bitposition***, as in the following examples:

RA.3 = *bit 3 of port A*
PortB.0 = *bit 0 of port B*

SKIP

Skip.

Words: 1 Cycles: 2 Affects: none

Operation: The following instruction word is skipped.

Coding: `0111 1110 0100 BTFSS 4,7`

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, SKIP may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following SKIP is a single-word instruction.**

SLEEP

Enter sleep mode.

Words: 1 Cycles: 1 Affects: TO, PD

Operation: The watchdog timer is cleared and the oscillator is stopped. TO is set to 1. PD is cleared to 0.

Coding: `0000 0000 0011 SLEEP`

SNB bit

Skip if not bit.

Words: 1 Cycles: 1 or 2 (skip) Affects: none

Operation: If bit reads 0, the following instruction word is skipped.

Coding: `0110 bbbf ffff BTFSC bit`

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, SNB may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following SNB is a single-word instruction.**

Note: The Parallax assemblers define a bit as ***port.bitposition***, as in the following examples:

RA.3 = *bit 3 of port A*
PortB.0 = *bit 0 of port B*

SNC

Skip if not carry.

Words: 1 Cycles: 1 or 2 (skip) Affects: none

Operation: If C is 0, the following instruction word is skipped.

Coding: `0110 0000 0011 BTFSC 3,0`

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, SNC may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following SNC is a single-word instruction.**

SNZ

Skip if not zero.

Words: 1 Cycles: 1 or 2 (skip) Affects: none

Operation: If Z is 0, the following instruction word is skipped.

Coding: `0110 0100 0011 BTFSC 3,2`

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, SNZ may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following SNZ is a single-word instruction.**

STC

Set carry.

Words: 1 Cycles: 1 Affects: C

Operation: C is set to 1.

Coding: `0101 0000 0011 BSF 3,0`

STZ

Set zero.

Words: 1 Cycles: 1 Affects: Z

Operation: Z is set to 1.

Coding: `0101 0100 0011 BSF 3,2`

SUB fr,#literal

Subtract literal from fr.

Words: 2 Cycles: 2 Affects: C, DC, Z

Operation: Literal is subtracted from fr via W. C will be cleared to 0 if an underflow occurred, otherwise C will be set to 1. DC will be cleared or set, depending on whether or not an underflow occurred in the least-siginificant nibble. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding:
```
1100 kkkk kkkk MOVLW   literal
0000 101f ffff SUBWF   fr,1
```

SUB fr1,fr2

Subtract fr2 from fr1.

Words: 2 Cycles: 2 Affects: C, DC, Z

Operation: Fr2 is subtracted from fr1 via W. C will be cleared to 0 if an underflow occurred, otherwise C will be set to 1. DC will be cleared or set, depending on whether or not an underflow occurred in the least-siginificant nibble. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding:
```
0010 000f ffff MOVF    fr2,0
0000 101f ffff SUBWF   fr1,1
```

SUB fr,W

Subtract W from fr.

Words: 1 Cycles: 1 Affects: C, DC, Z

Operation: W is subtracted from fr. C will be cleared to 0 if an underflow occurred, otherwise C will be set to 1. DC will be cleared or set, depending on whether or not an underflow occurred in the least-siginificant nibble. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding:
```
0000 101f ffff SUBWF   fr,1
```

SUBB* fr,bit

Subtract bit from fr.

Words: 2 Cycles: 2 Affects: Z

Operation: If bit reads 0, fr is decremented. If fr was decremented, Z will be set to 1 if the result was 0, else Z will be cleared to 0. This instruction is useful for subtracting the carry from the upper byte of a double-byte value after the lower byte has been subtracted.

Coding:
```
0111 0000 0011 BTFSS   3,0
0000 111f ffff DECF    fr,1
```

Note: The Parallax assemblers define a bit as ***port.bitposition***, as in the following examples:

RA.3 = *bit 3 of port A*
PortB.0 = *bit 0 of port B*

* This instruction is not available in PASMX.

SWAP fr

Swap nibbles in fr.

Words: 1 Cycles: 1 Affects: none

Operation: The high- and low-order nibbles in fr are swapped.

Coding: `0011 101f ffff SWAPF fr,1`

SZ

Skip if zero.

Words: 1 Cycles: 1 or 2 (skip) Affects: none

Operation: If Z is 1, the following instruction word is skipped.

Coding: `0111 0100 0011 BTFSS 3,2`

Note: Only one word is skipped by this instruction. Since some instructions are multi-word, SZ may jump into the middle of them, causing unexpected results. **Please make sure that any instruction following SZ is a single-word instruction.**

TEST fr

Test fr for zero.

Words: 1 Cycles: 1 Affects: Z

Operation: Fr is read and copied back to itself. Z will be set to 1 if the value moved was 0, otherwise Z will be cleared to 0.

Coding: `0010 001f ffff MOVF fr,1`

TEST W

Test W for zero.

Words: 1 Cycles: 1 Affects: Z

Operation: Z will be set to 1 if W is 0, otherwise Z will be cleared to 0.

Coding: `1101 0000 0000 IORLW 0`

XOR fr,#literal

XOR literal into fr.

Words: 2 Cycles: 2 Affects: Z

Operation: Literal is XOR'd into fr via W. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding:
```
1100 kkkk kkkk MOVLW   literal
0001 101f ffff XORWF   fr,1
```

XOR fr1,fr2

XOR fr2 into fr1.

Words: 2 Cycles: 2 Affects: Z

Operation: Fr2 is XOR'd into fr1 via W. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding:
```
0010 000f ffff MOVF    fr2,0
0001 101f ffff XORWF   fr1,1
```

XOR fr,W

XOR W into fr.

Words: 1 Cycles: 1 Affects: Z

Operation: W is XOR'd into fr. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `0001 101f ffff XORWF fr,1`

XOR W,#literal

XOR literal into W.

Words: 1 Cycles: 1 Affects: Z

Operation: Literal is XOR'd into W. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `1111 kkkk kkkk XORLW literal`

XOR W,fr

XOR fr into W.

Words: 1 Cycles: 1 Affects: Z

Operation: Fr is XOR'd into W. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `0001 100f ffff XORWF fr,0`

BLANK PAGE

PIC16Cxx Instruction Set

Microchip Version

BLANK PAGE

PIC16Cxx Instruction Set

Microchip Instruction Set

ADDWF	fr,d
ADDWF	fr
ADDWF	fr,W
ANDLW	lit
ANDWF	fr,d
ANDWF	fr
ANDWF	fr,W
BCF	fr,b
BCF	bit
BSF	fr,b
BSF	bit
BTFSC	fr,b
BTFSC	bit
BTFSS	fr,b
BTFSS	bit
CALL	addr8
CLRF	fr
CLRW	
CLRWDT	
COMF	fr,d
COMF	fr
COMF	fr,W
DECF	fr,d
DECF	fr
DECF	fr,W
DECFSZ	fr,d
DECFSZ	fr
DECFSZ	fr,W
GOTO	addr9
INCF	fr,d
INCF	fr
INCF	fr,W
INCFSZ	fr,d
INCFSZ	fr
INCFSZ	fr,W
IORLW	lit
IORWF	fr,d
IORWF	fr
IORWF	fr,W
MOVF	fr,d
MOVF	fr
MOVF	fr,W
MOVLW	lit
MOVWF	fr
NOP	
OPTION	
RETLW	lit
RETLW	
RLF	fr,d
RLF	fr
RLF	fr,W
RRF	fr,d
RRF	fr
RRF	fr,W
SLEEP	
SUBWF	fr,d
SUBWF	fr
SUBWF	fr,W
SWAPF	fr,d
SWAPF	fr
SWAPF	fr,W
TRIS	port_fr
TRISA	
TRISB	
TRISC	
XORLW	lit
XORWF	fr,d
XORWF	fr
XORWF	fr,W

BLANK PAGE

ADDWF fr,d

Add W and fr.

Cycles: 1 Affects: C, DC, Z

Operation: W is added to fr and the result is stored according to d. If d=0 then then W will hold the result. If d=1 then fr will hold the result. C will be set to 1 if an overflow occurred, otherwise C will be cleared to 0. DC will be set or cleared depending on whether or not an overflow occurred in the least-siginificant nibble. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `0001 11df ffff`

ANDLW literal

AND literal into W.

Cycles: 1 Affects: Z

Operation: Literal is AND'd into W. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `1110 kkkk kkkk`

ANDWF fr,d

AND W and fr.

Cycles: 1 Affects: Z

Operation: W is AND'd with fr and the result is stored according to d. If d=0 then W will hold the result. If d=1 then fr will hold the result. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `0001 01df ffff`

BCF fr,bit

Bit clear fr.

Cycles: 1 Affects: none

Operation: The bit addressed by the operand is cleared to 0.

Coding: `0100 bbbf ffff`

BSF fr,bit

Bit set fr.

Cycles: 1 Affects: none

Operation: The bit addressed by the operand is set to 1.

Coding: `0101 bbbf ffff`

BTFSC fr,bit

Bit test fr and skip if clear.

Cycles: 1 or 2 (skip) Affects: none

Operation: If the bit addressed by the operand reads 0, the next instruction word will be skipped by internally cancelling it into a NOP.

Coding: `0110 bbbf ffff`

BTFSS fr,bit

Bit test fr and skip if set.

Cycles: 1 or 2 (skip) Affects: none

Operation: If the bit addressed by the operand reads 1, the next instruction word will be skipped by internally cancelling it into a NOP.

Coding: `0111 bbbf ffff`

CALL addr8

Call subroutine.

Cycles: 2 Affects: none

Operation: The next instruction address is pushed onto the stack and addr8 is moved to the program counter. The ninth bit of the program counter will be cleared to 0. Therefore, calls are only allowed to the first half of any 512-word page, although the CALL instruction can be anywhere.

Coding: `1001 kkkk kkkk`

CLRF fr

Clear fr.

Cycles: 1 Affects: Z

Operation: Fr is cleared to 0 and Z is set to 1.

Coding: `0000 011f ffff`

CLRW

Clear W.

Cycles: 1 Affects: Z

Operation: W is cleared to 0 and Z is set to 1.

Coding: `0000 0100 0000`

CLRWDT

Clear the watchdog timer.

Cycles: 1 Affects: TO, PD

Operation: The watchdog timer is cleared, along with the prescaler, if assigned. TO and PD are set to 1.

Coding: `0000 0000 0100`

COMF fr,d

Complement fr.

Cycles: 1 Affects: Z

Operation: The one's complement of fr is computed and the result is stored according to d. If d=0 then W will hold the result. If d=1 then fr will hold the result. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `0010 01df ffff`

DECF fr,d

Decrement fr.

Cycles: 1 Affects: Z

Operation: The decremented value of fr is computed and the result is stored according to d. If d=0 then W will hold the result. If d=1 then fr will hold the result. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `0000 11df ffff`

DECFSZ fr,d

Decrement fr and skip if zero.

Cycles: 1 or 2 (skip) Affects: none

Operation: The decremented value of fr is computed and the result is stored according to d. If d=0 then W will hold the result. If d=1 then fr will hold the result. If the result was 0, the next instruction word will be skipped by internally cancelling it into a NOP.

Coding: `0010 11df ffff`

GOTO addr9

Go to address.

Cycles: 2 Affects: none

Operation: Addr9 is moved to the program counter.

Coding: `101k kkkk kkkk`

INCF fr,d

Increment fr.

Cycles: 1 Affects: Z

Operation: The incremented value of fr is computed and the result is stored according to d. If d=0 then W will hold the result. If d=1 then fr will hold the result. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `0010 10df ffff`

INCFSZ fr,d

Increment fr and skip if zero.

Cycles: 1 or 2 (skip) Affects: none

Operation: The incremented value of fr is computed and the result is stored according to d. If d=0 then W will hold the result. If d=1 then fr will hold the result. If the result was 0, the next instruction word will be skipped by internally cancelling it into a NOP.

Coding: `0011 11df ffff`

IORLW literal

OR literal into W.

Cycles: 1 Affects: Z

Operation: Literal is OR'd into W. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `1101 kkkk kkkk`

IORWF fr,d

Inclusive OR W and fr.

Cycles: 1 Affects: Z

Operation: W is OR'd with fr and the result is stored according to d. If d=0 then W will hold the result. If d=1 then fr will hold the result. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `0001 00df ffff`

MOVF fr,d

Move fr.

Cycles: 1 Affects: Z

Operation: If d=0 then fr will be moved toW. If d=1 then fr will be moved to itself. Z will be set to 1 if fr was 0, otherwise Z will be cleared to 0.

Coding: `0010 00df ffff`

MOVLW literal

Move literal into W.

Cycles: 1 Affects: none

Operation: Literal is moved into W.

Coding: `1100 kkkk kkkk`

MOVWF fr

Move W to fr.

Cycles: 1 Affects: none

Operation: W is moved into fr.

Coding: `0000 001f ffff`

NOP

No operation.

Cycles: 1 Affects: none

Operation: none

Coding: `0000 0000 0000`

OPTION

Move W into OPTION.

Cycles: 1 Affects: none

Operation: W is moved into the OPTION register.

Coding: `0000 0000 0010`

RETLW literal

Return from subroutine and move literal into W.

Cycles: 2 Affects: none

Operation: Literal is moved into W and the next stack value is moved into the program counter.

Coding: `1000 kkkk kkkk`

RLF fr,d

Rotate left fr.

Cycles: 1 Affects: C

Operation: The left-rotated value of fr is computed and the result is stored according to d. If d=0 then W will hold the result. If d=1 then fr will hold the result. On entry, C must hold the value to be shifted into the least-significant bit of the fr value. On exit, C will hold the previous most-significant bit of the fr value.

Coding: `0011 01df ffff`

RRF fr,d

Rotate right fr.

Cycles: 1 Affects: C

Operation: The right-rotated value of fr is computed and the result is stored according to d. If d=0 then W will hold the result. If d=1 then fr will hold the result. On entry, C must hold the value to be shifted into the most-significant bit of the fr value. On exit, C will hold the previous least-significant bit of the fr value.

Coding: `0011 00df ffff`

SLEEP

Enter sleep mode.

Cycles: 1 Affects: TO, PD

Operation: The watchdog timer is cleared and the oscillator is stopped. PD is cleared to 0. TO is set to 1.

Coding: `0000 0000 0011`

SUBWF fr,d

Subtract W from fr.

Cycles: 1 Affects: C, DC, Z

Operation: W is subtracted from fr and the result is stored according to d. If d=0 then W will hold the result. If d=1 then fr will hold the result. C will be cleared to 0 if an underflow occurred, otherwise C will be set to 1. DC will be cleared or set depending on whether or not an underflow occurred in the least-siginificant nibble. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `0000 10df ffff`

SWAPF fr,d

Swap fr's high- and low-order nibbles.

Cycles: 1 Affects: none

Operation: The nibble-swapped value of fr is computed and the result is stored according to d. If d=0 then W will hold the result. If d=1 then fr will hold the result.

Coding: `0011 10df ffff`

TRIS port_fr

Move W into port_fr's I/O control register.

Cycles: 1 Affects: none

Operation: W is moved into the I/O control register of port_fr. A "1" bit in W disables the corresponding port pin's output buffer, allowing input use, while a "0" bit enables the output buffer for high or low output. Port_fr must be 5, 6, or 7.

Coding: `0000 0000 0fff`

XORLW literal

XOR literal into W.

Cycles: 1 Affects: Z

Operation: Literal is XOR'd into W. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `1111 kkkk kkkk`

XORWF fr,d

XOR W and fr.

Cycles: 1 Affects: Z

Operation: W is XOR'd with fr and the result is stored according to d. If d=0 then W will hold the result. If d=1 then fr will hold the result. Z will be set to 1 if the result was 0, otherwise Z will be cleared to 0.

Coding: `0001 10df ffff`

BLANK PAGE

PIC16C5x Pre-Defined Symbols

The following text is a copy of the disk file INST.TXT. This file may be found in the PIC16C5x directory on the PIC Tools diskette. The disk version may be more up-to-date than this copy, but this is provided as a quick reference for common symbols used by the PASM assembler.

```
Symbols used with DEVICE directive - choose one from each group - DEVICE
PIC16C54,XT_OSC,WDT_OFF,PROTECT_OFF

PIC16C54        =    0              Devices
PIC16C55        =    1
PIC16C56        =    2
PIC16C57        =    3
PIC16C58        =    4

LP_OSC          =    11000000b      Oscillators
XT_OSC          =    11000001b
HS_OSC          =    11000010b
RC_OSC          =    11000011b

WDT_OFF         =    10110000b      Watchdog status
WDT_ON          =    10110100b

PROTECT_ON      =    01110000b      Code-protect status
PROTECT_OFF     =    01111000b

(Special file register equates)

INDIRECT        =    0              Indirect data addressing
RTCC            =    1              Real time clock/counter register
PC              =    2              Program counter
STATUS          =    3              Status word register
FSR             =    4              File select register
RA              =    5              Port A I/O register
RB              =    6              Port B I/O register
RC              =    7              Port C I/O register

(Status register bit equates)

C               =    STATUS.0       Carry bit
DC              =    STATUS.1       Digit carry bit
Z               =    STATUS.2       Zero bit
PD              =    STATUS.3       Power down bit
TO              =    STATUS.4       Time out bit
PA0             =    STATUS.5       Page address bit 0
PA1             =    STATUS.6       Page address bit 1
PA2             =    STATUS.7       Page address bit 2
```

PIC16Cxx Pre-Defined Symbols

The following text is a copy of the disk file INSTX.TXT. This file may be found in the PIC16Cxx directory on the PIC Tools diskette. The disk version may be more up-to-date than this copy, but this is provided as a quick reference for common symbols used by the PASMX assembler.

```
Dynamic Equates (always reflect current values)

$               =    Current origin
%               =    Current EEPROM origin         (PIC16C84 only)


DEVICE directive equates - establish device and fuse settings

Example:  DEVICE    PIC16C71,XT_OSC,WDT_ON,PWRT_OFF,PROTECT_ON

PIC16C71        =    0000h               Device (select one)
PIC16C84        =    0100h
PIC16C64        =    0200h
PIC16C74        =    0300h

LP_OSC          =    001Ch               Oscillator (select one)
XT_OSC          =    011Ch
HS_OSC          =    021Ch
RC_OSC          =    031Ch

WDT_OFF         =    001Bh               Watchdog timer (select one)
WDT_ON          =    041Bh

PWRT_OFF        =    0017h               Power-up timer (select one)
PWRT_ON         =    0817h

PROTECT_ON      =    000Fh               Code protection (select one)
PROTECT_OFF     =    100Fh

PIC16C71 Equates - enabled by DEVICE PIC16C71

                                 page
INDIRECT        =    00h         0   Indirect addressing register

RTCC            =    01h         0   Real time clock/counter register

PCL             =    02h         0   Program counter low-byte register

STATUS          =    03h         0   Status register
C               =    STATUS.0    0   Carry bit
DC              =    STATUS.1    0   Digit carry bit
Z               =    STATUS.2    0   Zero bit
PD              =    STATUS.3    0   Power down bit
TO              =    STATUS.4    0   Time out bit
RP0             =    STATUS.5    0   Register page bit 0
RP1             =    STATUS.6    0   Register page bit 1
IRP             =    STATUS.7    0   Indirect register page bit

FSR             =    04h         0   File select register

PORTA           =    05h         0   RA i/o register
RA              =    05h         0   RA i/o register

PORTB           =    06h         0   RB i/o register
RB              =    06h         0   RB i/o register

ADCON0          =    08h         0   Analog to Digitial Converter control register 0
ADON            =    ADCON0.0    0   ADC power control bit
ADIF            =    ADCON0.1    0   ADC interrupt flag bit
```

PIC16Cxx Pre-Defined Symbols

```
GO_DONE         =    ADCON0.2    0   ADC go command / done flag bit
CHS0            =    ADCON0.3    0   ADC channel select bit 0
CHS1            =    ADCON0.4    0   ADC channel select bit 1
ADCS0           =    ADCON0.6    0   ADC clock select bit 0
ADCS1           =    ADCON0.7    0   ADC clock select bit 1

ADRES           =    09h         0   Analog to Digital Converter result register

PCLATH          =    0Ah         0   Program counter high-byte register

INTCON          =    0Bh         0   Interrupt control register
RBIF            =    INTCON.0    0   RB4-RB7 change interrupt flag bit
INTF            =    INTCON.1    0   RB0/INT interrupt flag bit
RTIF            =    INTCON.2    0   RTCC overflow interrupt flag bit
RBIE            =    INTCON.3    0   RB4-RB7 change interrupt enable bit
INTE            =    INTCON.4    0   RB0/INT interrupt enable bit
RTIE            =    INTCON.5    0   RTCC overflow interrupt enable bit
ADIE            =    INTCON.6    0   ADC interrupt enable bit
GIE             =    INTCON.7    0   Global interrupt enable bit

OPTION          =    01h         1   OPTION register
PS0             =    OPTION.0    1   Prescaler bit 0
PS1             =    OPTION.1    1   Prescaler bit 1
PS2             =    OPTION.2    1   Prescaler bit 2
PSA             =    OPTION.3    1   Prescaler assignment bit
RTE             =    OPTION.4    1   RTCC signal edge bit
RTS             =    OPTION.5    1   RTCC signal source bit
INTEDG          =    OPTION.6    1   RB0/INT edge select bit
RBPU            =    OPTION.7    1   RB weak pull-up enable bit

TRISA           =    05h         1   RA tristate control register

TRISB           =    06h         1   RB tristate control register

ADCON1          =    08h         1   Analog to Digitial Converter control register 1
PCFG0           =    ADCON1.0    1   RA port configuration bit 0
PCFG1           =    ADCON1.1    1   RA port configuration bit 1


PIC16C84 Equates - enabled by DEVICE PIC16C84

                                 page
IND0            =    00h         0   Indirect addressing register
INDIRECT        =    00h         0   Indirect addressing register

RTCC            =    01h         0   Real time clock/counter register

PCL             =    02h         0   Program counter low-byte register

STATUS          =    03h         0   Status register
C               =    STATUS.0    0   Carry bit
DC              =    STATUS.1    0   Digit carry bit
Z               =    STATUS.2    0   Zero bit
PD              =    STATUS.3    0   Power down bit
TO              =    STATUS.4    0   Time out bit
RP0             =    STATUS.5    0   Register page bit 0
RP1             =    STATUS.6    0   Register page bit 1
IRP             =    STATUS.7    0   Indirect register page bit

FSR             =    04h         0   File select register

PORTA           =    05h         0   RA i/o register
RA              =    05h         0   RA i/o register

PORTB           =    06h         0   RB i/o register
RB              =    06h         0   RB i/o register
```

PIC16Cxx Pre-Defined Symbols

```
EEDATA          =    08h         0   EEPROM data register

EEADR           =    09h         0   EEPROM address register

PCLATH          =    0Ah         0   Program counter high-byte register

INTCON          =    0Bh         0   Interrupt control register
RBIF            =    INTCON.0    0   RB4-RB7 change interrupt flag bit
INTF            =    INTCON.1    0   RB0/INT interrupt flag bit
RTIF            =    INTCON.2    0   RTCC overflow interrupt flag bit
RBIE            =    INTCON.3    0   RB4-RB7 change interrupt enable bit
INTE            =    INTCON.4    0   RB0/INT interrupt enable bit
RTIE            =    INTCON.5    0   RTCC overflow interrupt enable bit
EEIE            =    INTCON.6    0   EEPROM interrupt enable bit
GIE             =    INTCON.7    0   Global interrupt enable bit

OPTION          =    01h         1   OPTION register
PS0             =    OPTION.0    1   Prescaler bit 0
PS1             =    OPTION.1    1   Prescaler bit 1
PS2             =    OPTION.2    1   Prescaler bit 2
PSA             =    OPTION.3    1   Prescaler assignment bit
RTE             =    OPTION.4    1   RTCC signal edge bit
RTS             =    OPTION.5    1   RTCC signal source bit
INTEDG          =    OPTION.6    1   RB0/INT edge select bit
RBPU            =    OPTION.7    1   RB weak pull-up enable bit

TRISA           =    05h         1   RA tristate control register

TRISB           =    06h         1   RB tristate control register

EECON1          =    08h         1   EEPROM control register 1
RD              =    EECON1.0    1   EEPROM read control bit
WR              =    EECON1.1    1   EEPROM write control bit
WREN            =    EECON1.2    1   EEPROM write enable bit
WRERR           =    EECON1.3    1   EEPROM write error flag bit
EEIF            =    EECON1.4    1   EEPROM interrupt flag bit

EECON2          =    09h         1   EEPROM control register 2


PIC16C64 Equates - enabled by DEVICE PIC16C64

                                 page
INDF            =    00h         0   Indirect addressing register
INDIRECT        =    00h         0   Indirect addressing register

TMR0            =    01h         0   Timer0 register

PCL             =    02h         0   Program counter low-byte register

STATUS          =    03h         0   Status register
C               =    STATUS.0    0   Carry bit
DC              =    STATUS.1    0   Digit carry bit
Z               =    STATUS.2    0   Zero bit
PD              =    STATUS.3    0   Power down bit
TO              =    STATUS.4    0   Time out bit
RP0             =    STATUS.5    0   Register page bit 0
RP1             =    STATUS.6    0   Register page bit 1
IRP             =    STATUS.7    0   Indirect register page bit

FSR             =    04h         0   File select register

PORTA           =    05h         0   RA i/o register
RA              =    05h         0   RA i/o register

PORTB           =    06h         0   RB i/o register
RB              =    06h         0   RB i/o register
```

PIC16Cxx Pre-Defined Symbols

```
PORTC           =    07h         0   RC i/o register
RC              =    07h         0   RC i/o register

PORTD           =    08h         0   RD i/o register
RD              =    08h         0   RD i/o register

PORTE           =    09h         0   RE i/o register
RE              =    09h         0   RE i/o register

PCLATH          =    0Ah         0   Program counter high-byte register

INTCON          =    0Bh         0   Interrupt control register
RBIF            =    INTCON.0    0   RB4-RB7 change interrupt flag bit
INTF            =    INTCON.1    0   RB0/INT interrupt flag bit
RTIF            =    INTCON.2    0   RTCC overflow interrupt flag bit
RBIE            =    INTCON.3    0   RB4-RB7 change interrupt enable bit
INTE            =    INTCON.4    0   RB0/INT interrupt enable bit
RTIE            =    INTCON.5    0   RTCC overflow interrupt enable bit
PEIE            =    INTCON.6    0   Perpiheral interrupt enable bit
GIE             =    INTCON.7    0   Global interrupt enable bit

PIR1            =    0Ch         0   Peripheral interrupt flags register
TMR1IF          =    PIR1.0      0   Timer1 interrupt flag bit
TMR2IF          =    PIR1.1      0   Timer2 interrupt flag bit
CCP1IF          =    PIR1.2      0   CCP interrupt flag bit
SSPIF           =    PIR1.3      0   SSP interrupt flag bit
PSPIF           =    PIR1.7      0   PSP interrupt flag bit

TMR1L           =    0Eh         0   Timer1 low byte register

TMR1H           =    0Fh         0   Timer1 high byte register

T1CON           =    10h         0   Timer1 control register
TMR1ON          =    T1CON.0     0   Timer1 enable bit
TMR1CS          =    T1CON.1     0   Timer1 clock select bit
T1INSYNC        =    T1CON.2     0   Timer1 sync enable bit
T1OSCEN         =    T1CON.3     0   Timer1 oscillator enable bit
T1CKPS0         =    T1CON.4     0   Timer1 clock prescaler select bit0
T1CKPS1         =    T1CON.5     0   Timer1 clock prescaler select bit1

TMR2            =    11h         0   Timer2 register

T2CON           =    12h         0   Timer2 control register
T2CKPS0         =    T2CON.0     0   Timer2 clock prescaler select bit0
T2CKPS1         =    T2CON.1     0   Timer2 clock prescaler select bit1
TMR2ON          =    T2CON.2     0   Timer2 enable bit
TOUTPS0         =    T2CON.3     0   Timer2 postscaler select bit0
TOUTPS1         =    T2CON.4     0   Timer2 postscaler select bit1
TOUTPS2         =    T2CON.5     0   Timer2 postscaler select bit2
TOUTPS3         =    T2CON.6     0   Timer2 postscaler select bit3

SSPBUF          =    13h         0   Synchronous Serial Port receive/transmit register

SSPCON          =    14h         0   Synchronous Serial Port control register
SSPM0           =    SSPCON.0    0   SSP mode select bit0
SSPM1           =    SSPCON.1    0   SSP mode select bit1
SSPM2           =    SSPCON.2    0   SSP mode select bit2
SSPM3           =    SSPCON.3    0   SSP mode select bit3
CKP             =    SSPCON.4    0   SSP clock polarity select bit
SSPEN           =    SSPCON.5    0   SSP enable bit
SSPOV           =    SSPCON.6    0   SSP receive overflow flag bit
WCOL            =    SSPCON.7    0   SSP write collision detect bit

CCPR1L          =    15h         0   Capture/Compare/PWM low byte register

CCPR1H          =    16h         0   Capture/Compare/PWM high byte register

CCP1CON         =    17h         0   Capture/Compare/PWM control register
```

PIC16Cxx Pre-Defined Symbols

```
CCP1M0          =    CCP1CON.0   0   CCP mode select bit0
CCP1M1          =    CCP1CON.1   0   CCP mode select bit1
CCP1M2          =    CCP1CON.2   0   CCP mode select bit2
CCP1M3          =    CCP1CON.3   0   CCP mode select bit3
CCP1Y           =    CCP1CON.4   0   CCP 10-bit PWM low order bit0
CCP1X           =    CCP1CON.5   0   CCP 10-bit PWM low order bit1

OPTION          =    01h         1   OPTION register
PS0             =    OPTION.0    1   Prescaler bit 0
PS1             =    OPTION.1    1   Prescaler bit 1
PS2             =    OPTION.2    1   Prescaler bit 2
PSA             =    OPTION.3    1   Prescaler assignment bit
RTE             =    OPTION.4    1   RTCC signal edge bit
RTS             =    OPTION.5    1   RTCC signal source bit
INTEDG          =    OPTION.6    1   RB0/INT edge select bit
RBPU            =    OPTION.7    1   RB weak pull-up enable bit

TRISA           =    05h         1   RA tristate control register

TRISB           =    06h         1   RB tristate control register

TRISC           =    07h         1   RC tristate control register

TRISD           =    08h         1   RD tristate control register

TRISE           =    09h         1   RE tristate control register
TRISE0          =    TRISE.0     1   RE0 tristate control bit
TRISE1          =    TRISE.1     1   RE1 tristate control bit
TRISE2          =    TRISE.2     1   RE2 tristate control bit
PSPMODE         =    TRISE.4     1   PSP mode bit
IBOV            =    TRISE.5     1   PSP input buffer overflow flag
OBF             =    TRISE.6     1   PSP output buffer full flag
IBF             =    TRISE.7     1   PSP input buffer full flag

PIE1            =    0Ch         1   Peripheral interrupt enable register
TMR1IE          =    PIE1.0      1   Timer1 interrupt enable bit
TMR2IE          =    PIE1.1      1   Timer2 interrupt enable bit
CCP1IE          =    PIE1.2      1   CCP interrupt enable bit
SSPIE           =    PIE1.3      1   SSP interrupt enable bit
PSPIE           =    PIE1.7      1   PSP interrupt enable bit

PCON            =    0Eh         1   Power-On-Reset detection register
POR             =    PCON.1      1   Power-On-Reset flag bit

PR2             =    12h         1   Timer2 period register

SSPADD          =    13h         1   Synchronous Serial Port I2C address register

SSPSTAT         =    14h         1   Synchronous Serial Port status register
BF              =    SSPSTAT.0   1   SSP buffer full flag bit
UA              =    SSPSTAT.1   1   SSP 10-bit I2C update address flag bit
R_W             =    SSPSTAT.2   1   SSP I2C read/write status bit
S               =    SSPSTAT.3   1   SSP I2C start flag bit
P               =    SSPSTAT.4   1   SSP I2C stop flag bit
D_A             =    SSPSTAT.5   1   SSP I2C data/address flag bit


PIC16C74 Equates - enabled by DEVICE PIC16C74

                                 page
INDF            =    00h         0   Indirect addressing register
INDIRECT        =    00h         0   Indirect addressing register

TMR0            =    01h         0   Timer0 register

PCL             =    02h         0   Program counter low-byte register
```

PIC16Cxx Pre-Defined Symbols

```
STATUS          =    03h         0   Status register
C               =    STATUS.0    0   Carry bit
DC              =    STATUS.1    0   Digit carry bit
Z               =    STATUS.2    0   Zero bit
PD              =    STATUS.3    0   Power down bit
TO              =    STATUS.4    0   Time out bit
RP0             =    STATUS.5    0   Register page bit 0
RP1             =    STATUS.6    0   Register page bit 1
IRP             =    STATUS.7    0   Indirect register page bit

FSR             =    04h         0   File select register

PORTA           =    05h         0   RA i/o register
RA              =    05h         0   RA i/o register

PORTB           =    06h         0   RB i/o register
RB              =    06h         0   RB i/o register

PORTC           =    07h         0   RC i/o register
RC              =    07h         0   RC i/o register

PORTD           =    08h         0   RD i/o register
RD              =    08h         0   RD i/o register

PORTE           =    09h         0   RE i/o register
RE              =    09h         0   RE i/o register

PCLATH          =    0Ah         0   Program counter high-byte register

INTCON          =    0Bh         0   Interrupt control register
RBIF            =    INTCON.0    0   RB4-RB7 change interrupt flag bit
INTF            =    INTCON.1    0   RB0/INT interrupt flag bit
T0IF            =    INTCON.2    0   RTCC overflow interrupt flag bit
RBIE            =    INTCON.3    0   RB4-RB7 change interrupt enable bit
INTE            =    INTCON.4    0   RB0/INT interrupt enable bit
T0IE            =    INTCON.5    0   RTCC overflow interrupt enable bit
PEIE            =    INTCON.6    0   Perpiheral interrupt enable bit
GIE             =    INTCON.7    0   Global interrupt enable bit

PIR1            =    0Ch         0   Peripheral interrupt flags register
TMR1IF          =    PIR1.0      0   Timer1 interrupt flag bit
TMR2IF          =    PIR1.1      0   Timer2 interrupt flag bit
CCP1IF          =    PIR1.2      0   CCP1 interrupt flag bit
SSPIF           =    PIR1.3      0   SSP interrupt flag bit
TXIF            =    PIR1.4      0   ASP transmit interrupt flag bit
RCIF            =    PIR1.5      0   ASP receive interrupt flag bit
ADIF            =    PIR1.6      0   ADC completion interrupt flag bit
PSPIF           =    PIR1.7      0   PSP interrupt flag bit

PIR2            =    0Dh         0   Peripheral interrupt flags register
CCP2IE          =    PIR2.0      0   CCP2 interrupt enable bit

TMR1L           =    0Eh         0   Timer1 low byte register

TMR1H           =    0Fh         0   Timer1 high byte register

T1CON           =    10h         0   Timer1 control register
TMR1ON          =    T1CON.0     0   Timer1 enable bit
TMR1CS          =    T1CON.1     0   Timer1 clock select bit
T1INSYNC        =    T1CON.2     0   Timer1 sync enable bit
T1OSCEN         =    T1CON.3     0   Timer1 oscillator enable bit
T1CKPS0         =    T1CON.4     0   Timer1 clock prescaler select bit0
T1CKPS1         =    T1CON.5     0   Timer1 clock prescaler select bit1

TMR2            =    11h         0   Timer2 register

T2CON           =    12h         0   Timer2 control register
T2CKPS0         =    T2CON.0     0   Timer2 clock prescaler select bit0
```

PIC16Cxx Pre-Defined Symbols

```
T2CKPS1         =    T2CON.1     0   Timer2 clock prescaler select bit1
TMR2ON          =    T2CON.2     0   Timer2 enable bit
TOUTPS0         =    T2CON.3     0   Timer2 postscaler select bit0
TOUTPS1         =    T2CON.4     0   Timer2 postscaler select bit1
TOUTPS2         =    T2CON.5     0   Timer2 postscaler select bit2
TOUTPS3         =    T2CON.6     0   Timer2 postscaler select bit3

SSPBUF          =    13h         0   Synchronous Serial Port receive/transmit register

SSPCON          =    14h         0   Synchronous Serial Port control register
SSPM0           =    SSPCON.0    0   SSP mode select bit0
SSPM1           =    SSPCON.1    0   SSP mode select bit1
SSPM2           =    SSPCON.2    0   SSP mode select bit2
SSPM3           =    SSPCON.3    0   SSP mode select bit3
CKP             =    SSPCON.4    0   SSP clock polarity select bit
SSPEN           =    SSPCON.5    0   SSP enable bit
SSPOV           =    SSPCON.6    0   SSP receive overflow flag bit
WCOL            =    SSPCON.7    0   SSP write collision detect bit

CCPR1L          =    15h         0   Capture1/Compare1/PWM1 low byte register

CCPR1H          =    16h         0   Capture1/Compare1/PWM1 high byte register

CCP1CON         =    17h         0   Capture1/Compare1/PWM1 control register
CCP1M0          =    CCP1CON.0   0   CCP1 mode select bit0
CCP1M1          =    CCP1CON.1   0   CCP1 mode select bit1
CCP1M2          =    CCP1CON.2   0   CCP1 mode select bit2
CCP1M3          =    CCP1CON.3   0   CCP1 mode select bit3
CCP1Y           =    CCP1CON.4   0   CCP1 10-bit PWM low order bit0
CCP1X           =    CCP1CON.5   0   CCP1 10-bit PWM low order bit1

RCSTA           =    18h         0   Asynchronus Serial Port status and control register
RCD8            =    RCSTA.0     0   ASP 9th/parity bit of received data
OERR            =    RCSTA.1     0   ASP overrun error bit
FERR            =    RCSTA.2     0   ASP framing error bit
CREN            =    RCSTA.4     0   ASP continuous-receive enable bit
SREN            =    RCSTA.5     0   ASP single-receive enable bit
RC8_9           =    RCSTA.6     0   ASP receive data length bit
SPEN            =    RCSTA.7     0   ASP enable bit

TXREG           =    19h         0   Asynchronous serial port transmit register

RCREG           =    1Ah         0   Asynchronous serial port receive register

CCPR2L          =    1Bh         0   Capture2/Compare2/PWM2 low byte register

CCPR2H          =    1Ch         0   Capture2/Compare2/PWM2 high byte register

CCP2CON         =    1Dh         0   Capture2/Compare2/PWM2 control register
CCP2M0          =    CCP2CON.0   0   CCP2 mode select bit0
CCP2M1          =    CCP2CON.1   0   CCP2 mode select bit1
CCP2M2          =    CCP2CON.2   0   CCP2 mode select bit2
CCP2M3          =    CCP2CON.3   0   CCP2 mode select bit3
CCP2Y           =    CCP2CON.4   0   CCP2 10-bit PWM low order bit0
CCP2X           =    CCP2CON.5   0   CCP2 10-bit PWM low order bit1

ADRES           =    1Eh         0   Analog to Digitial Converter result register

ADCON0          =    1Fh         0   Analog to Digitial Converter control register 0
ADON            =    ADCON0.0    0   ADC power control bit
GO_DONE         =    ADCON0.2    0   ADC go command / done flag bit
CHS0            =    ADCON0.3    0   ADC channel select bit 0
CHS1            =    ADCON0.4    0   ADC channel select bit 1
CHS2            =    ADCON0.5    0   ADC channel select bit 2
ADCS0           =    ADCON0.6    0   ADC clock select bit 0
ADCS1           =    ADCON0.7    0   ADC clock select bit 1

OPTION          =    01h         1   OPTION register
```

PIC16Cxx Pre-Defined Symbols

```
PS0             =    OPTION.0    1   Prescaler bit 0
PS1             =    OPTION.1    1   Prescaler bit 1
PS2             =    OPTION.2    1   Prescaler bit 2
PSA             =    OPTION.3    1   Prescaler assignment bit
RTE             =    OPTION.4    1   RTCC signal edge bit
RTS             =    OPTION.5    1   RTCC signal source bit
INTEDG          =    OPTION.6    1   RB0/INT edge select bit
RBPU            =    OPTION.7    1   RB weak pull-up enable bit

TRISA           =    05h         1   RA tristate control register

TRISB           =    06h         1   RB tristate control register

TRISC           =    07h         1   RC tristate control register

TRISD           =    08h         1   RD tristate control register

TRISE           =    09h         1   RE tristate control register
TRISE0          =    TRISE.0     1   RE0 tristate control bit
TRISE1          =    TRISE.1     1   RE1 tristate control bit
TRISE2          =    TRISE.2     1   RE2 tristate control bit
PSPMODE         =    TRISE.4     1   PSP mode bit
IBOV            =    TRISE.5     1   PSP input buffer overflow flag
OBF             =    TRISE.6     1   PSP output buffer full flag
IBF             =    TRISE.7     1   PSP input buffer full flag

PIE1            =    0Ch         1   Peripheral interrupt enable register
TMR1IE          =    PIE1.0      1   Timer1 interrupt enable bit
TMR2IE          =    PIE1.1      1   Timer2 interrupt enable bit
CCP1IE          =    PIE1.2      1   CCP1 interrupt enable bit
SSPIE           =    PIE1.3      1   SSP interrupt enable bit
TXIE            =    PIE1.4      1   ASP transmit interrupt enable bit
RCIE            =    PIE1.5      1   ASP receive interrupt enable bit
ADIE            =    PIE1.6      1   ADC interrupt enable bit
PSPIE           =    PIE1.7      1   PSP interrupt enable bit

PCON            =    0Eh         1   Power-On-Reset detection register
POR             =    PCON.1      1   POR flag bit

PR2             =    12h         1   Timer2 period register

SSPADD          =    13h         1   Synchronous Serial Port I2C address register

SSPSTAT         =    14h         1   Synchronous Serial Port status register
BF              =    SSPSTAT.0   1   SSP buffer full flag bit
UA              =    SSPSTAT.1   1   SSP 10-bit I2C update address flag bit
R_W             =    SSPSTAT.2   1   SSP I2C read/write status bit
S               =    SSPSTAT.3   1   SSP I2C start flag bit
P               =    SSPSTAT.4   1   SSP I2C stop flag bit
D_A             =    SSPSTAT.5   1   SSP I2C data/address flag bit

TXSTA           =    18h         1   Asynchronous Serial Port transmit status and control
TXD8            =    TXSTA.0     1   ASP 9th bit of transmit data
TRMT            =    TXSTA.1     1   ASP transmit shift register empty
BRGH            =    TXSTA.2     1   ASP high baud rate select bit
SYNC            =    TXSTA.4     1   ASP mode bit
TXEN            =    TXSTA.5     1   ASP transmit enable bit
TX8_9           =    TXSTA.6     1   ASP transmit data length bit
CSRC            =    TXSTA.7     1   ASP clock source select bit

SPBRG           =    19h         1   Asynchronous Serial Port baud rate register

ADCON1          =    1Fh         1   Analog to Digitial Converter control register 1
PCFG0           =    ADCON1.0    1   RA port configuration bit 0
PCFG1           =    ADCON1.1    1   RA port configuration bit 1
PCFG2           =    ADCON1.2    1   RA port configuration bit 2
```

BLANK PAGE

Parallax Distributors

Australia

MicroZed Computers	Phone:	int + 61 67 722 777
Armidale	Fax:	int + 61 67 728 987
Technology Affair	Phone:	int + 61 9 246 4810
Carine, Western Australia	Fax:	int + 61 9 246 4809

Austria, Germany, Switzerland

Wilke Technology	Phone:	int + 49 241 15 4071
Aachen	Fax:	int + 49 241 15 8475

Belgium

G.S.E.	Phone:	int + 32 41 77 5151
Soumagne	Fax:	int + 32 41 77 5353

Canada

Aerosystems International	Phone:	(514) 336-9426
St. Laurent, Quebec	Fax:	(514) 336-4383

Czech Republic

MITE	Phone:	int + 42 49 5813 252
Hradec Králové	Fax:	int + 42 49 5813 260

France

Selectronic	Phone:	int + 33 20 52 98 52
Lille	Fax:	int + 33 20 52 12 04

Greece

Peter Caritato & Associates	Phone:	int + 30 1 902 0115
Athens	Fax:	int + 30 1 901 7042

Parallax Distributors

Hungary

HUMANsoft	Phone:	int + 36 1163 2879
Budapest	Fax:	int + 36 1251 3673

India

AL Systems	Phone:	int + 91 422 232 561
Coimbatore	Fax:	int + 91 422 213 849

Israel

Elina Electronic Ltd.	Phone:	int + 972 3 498 543
Tel Aviv	Fax:	int + 972 3 498 745

Japan

Akizuki Denshi Tsusho Ltd.	Phone:	int + 81 3 3251 1779
Tokyo	Fax:	int + 81 3 3432 4492

Netherlands

Antratek	Phone:	int + 31 1803 17666
Nieuwerkerk A/D Ijssel	Fax:	int + 31 1803 16664

South Africa

Jakanaka	Phone:	int + 27 11 907 8475
Alberton	Fax:	int + 27 11 907 9426

South Korea

Prochips	Phone:	int + 82 2 849 8567
Seoul	Fax:	int + 82 2 849 8659

Taiwan

United Tech Electronic Corp.	Phone:	int + 886 2 647 1978
Taipei	Fax:	int + 886 2 648 1895

Parallax Distributors

United Kingdom

Milford Instruments	Phone:	int + 44 977 683 665
South Milford, Leeds	Fax:	int + 44 977 681 465

United States

Digi-Key	Phone:	(800) 344-4539
Thief River Falls, MN	Fax:	(218) 681-3380
Marlin P. Jones & Assoc.	Phone:	(407) 848-8236
Lake Park, FL	Fax:	(407) 844-8764