
Preparing to do a PIC Project

- 1.1 Introduction
- 1.2 Overview of PIC Microcontroller
- 1.3 Basics of PIC Assembly Language
- 1.4 Introduction to C Programming for PIC Microcontroller
- 1.5 MPLAB Integrated Development Environment (IDE)
- 1.6 Advanced Debugger Features – Stimulus

1.1 Introduction

The aim of this chapter is to consider a number of issues that need to be taken into account before doing almost any microcontroller-based project. First, the reader will be introduced to a PIC (programmable interface controller) microcontroller by a brief discussion of one of the models from the PIC microcontroller family – the PIC16F627A. This model is now a common choice for low-cost PIC projects and has practically replaced the very popular PIC16F84 model. The PIC16F627A is therefore a choice for a large number of projects from this book although some other simpler and more complex models are also being used. The rest of the PIC family will be considered briefly, introducing some other models used for the projects in this book. We will then discuss the basics of two programming languages commonly used to develop PIC programs in practice and throughout this book – assembly and C. This will by no means be a detailed discussion of those two languages; a separate book would be needed for that. The aim instead is to provide a short overview of the basic features of both languages and to enable readers to learn the rest of it while doing projects from the other chapters of this book. Material covered in this chapter should therefore be sufficient to allow the reader to start with the first programs and projects from Chapter 2 and gradually build knowledge to do more complex projects from the rest of the book. Finally,

2 PIC Projects: A Practical Approach

we will demonstrate how to develop and test a simple PIC program using the MPLAB[®] -Integrated Development Environment (IDE).

1.2 Overview of PIC Microcontroller

The name PIC denotes several families of microcontrollers manufactured by Microchip Technology. This range is huge and very versatile so discussing even a small number of microcontrollers would be a difficult and time-consuming task. Instead, in this section, we will concentrate on the basic features and layout of one of the most popular members of the mid-range PIC16 family: PIC16F627A. Once the basic features of this device are explained it will be easier to introduce and understand the operations of more complex PICs used in later chapters of this book.

1.2.1 PIC16F627 Building Blocks

Every computer system, however complicated or simple, consists of a number of common building blocks. Those are: the CPU (central processing unit or microprocessor) block, the memory block (RAM and ROM) and the input/output (I/O) block (interface circuitry). The CPU performs all the logic and arithmetic functions; memory is used to store programs and data while the interface provides means of communication and data exchange between the microcomputer system and the external world.

A microcontroller is a stripped-down version of the computer system architecture with one important difference – all of the system blocks are placed on one chip. The microcontroller-based system therefore requires very little additional circuitry for its proper operation. All that is needed in most cases is a clock input to provide timing for the system operation.

The PIC16F27A microcontroller contains all of the previously mentioned blocks. Components of this microcontroller, described in slightly more detail, are:

- *The CPU.* The ‘brain’ of a microcontroller. It is responsible for finding and fetching the right instruction to be executed, for decoding that instruction, and finally for its execution.
- *Memory.* Split into two physically separate blocks – program and data memory. This so-called Harvard architecture is used to speed up the operation of the microcontroller as both data and instructions can be fetched from separate memories using separate buses simultaneously.

- *Program memory.* Used to store a program to be executed in the central processing unit of the microcontroller. It is of flash type so the microcontroller can be programmed many times before a system developer is happy with its performance and the programmed PIC is finally installed into some bigger system. If the power to the microcontroller is switched off, the content of the flash-type memory is not lost. The size of the program memory on the PIC16F27A is 1024 words (1 kwords), where one word holds 14 bits.
- *Data memory.* Used to store microcontroller data. It is further divided into EEPROM and RAM memory: EEPROM memory holds important data that need to be saved when there is no power supply to the microcontroller; RAM is used by a program to store inter-results or temporary data during the program execution. EEPROM contains 128 bytes of data whereas RAM holds 224 bytes (1 byte contains 8 bits so the widths of program and data memories are different).
- *PORTA and PORTB.* Physical connections between the microcontroller and the outside world. Both of those ports have eight pins and those pins are bidirectional – they can be used for input or output of data provided they are properly configured as input or output pins in the program. The exception is pin 5 of port A (RA5), which is an input-only type pin. Two special function registers within PIC, TRISA and TRISB, control the direction of the port pins. Writing ‘1’ in the particular bit of the TRISA register configures the corresponding pin of port A as an input pin; ‘0’ in TRISA makes it an output pin. The same is true for the port B pins and TRISB register. Some of these port pins are multiple-purpose pins and can be used for other peripheral functions of the processor. This will be explained in more detail in Section 1.2.3 where the layout (pin out) of the PIC16F27A chip is discussed.

The PIC blocks mentioned above communicate through a complex system of communication lines called *buses*. The *data buses* are used for the transfer of data through the system and *address buses* communicate addresses of data and program instructions to be accessed during program execution. Various other communication lines exist in the PIC and those are usually referred to as *control bus* lines. Note that since two separate memories exist in the PIC, both data and address bus systems are doubled, i.e. PIC16F27A (Figure 1.1) has a data memory (DM) address bus as well as program memory (PM) address bus. Similarly, this processor also has a DM data bus and PM data bus.

4 PIC Projects: A Practical Approach

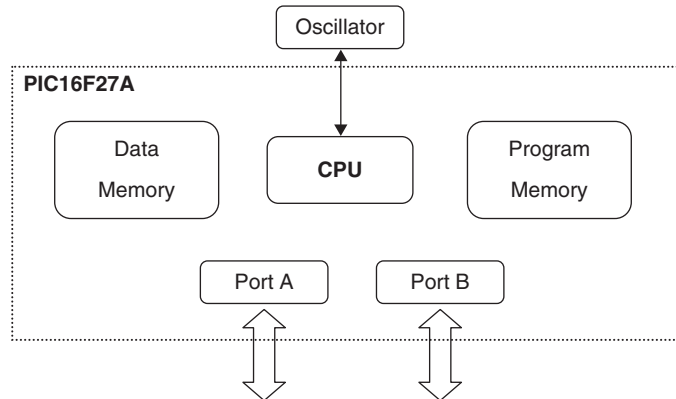


Figure 1.1 Main elements of the PIC16F27A microcontroller

1.2.2 EEPROM and RAM Memories on PIC16F27A

The EEPROM memory on the PIC16F27A holds 128 bytes of non-volatile information. This memory is electronically programmable so it is not a fast RAM-type memory and it can be awkward to access it within the program. It is normally used to store data that is not frequently changed.

The RAM memory on the PIC16F27A is actually split into four memory banks where each bank holds access to 80 memory locations. This, however, does not mean that the total capacity of the RAM memory on the PIC16F27A is 320 bytes. It is more complicated than that. Here is the explanation.

This memory can be considered to consist of two different types of registers – special function registers (SFRs) and general purpose registers (GPRs). The first 32 bytes of each memory bank (00h-1Fh) belong to SFRs. Those registers are used by the CPU to control the desired operation of the device and to record the operating states of the PIC, the I/O port conditions and the other conditions. Not all of those registers are implemented on the PIC16F27A model. There are 21 SFR bytes in bank 0, 18 in bank 1 and seven SFRs in banks 2 and 3. Some of these registers will be used and described in more detail in the remaining chapters of this book.

The GPRs are placed in the first three memory banks – 80 bytes of GPR in banks 0 and 1 and 48 in bank 2. Those registers can be used to store results and conditions temporarily while the program is running. The content of the GPR is lost when the power is switched off. The rest of the available memory space is not used – if it gets accessed, it reads as 0. This memory arrangement is shown schematically in Figure 1.2.

	Bank 0		Bank 1		Bank 2		Bank 3
00h		80h		100h		180h	
01h	TMR0	81h	OPTION	101h	TMR0	181h	OPTION
02h	PCL	82h	PCL	102h	PCL	182h	PCL
03h	STATUS	83h	STATUS	103h	STATUS	183h	STATUS
04h	FSR	84h	FSR	104h	FSR	184h	FSR
05h	PORTA	85h	TRISA	105h		185h	
06h	PORTB	86h	TRISB	106h	PORTB	186h	TRISB
07h		87h		107h		187h	
08h		88h		108h		188h	
09h		89h		109h		189h	
0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah	PCLATH
0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh	INTCON
0Ch	PIR1	8Ch	PIE1	10Ch		18Ch	
0Dh		8Dh		10Dh		18Dh	
0Eh	TMR1L	8Eh	PCON	10Eh		18Eh	
0Fh	TMR1H	8Fh		10Fh		18Fh	
10h	T1CON	90h					
11h	TMR2	91h					
12h	T2CON	92h	PR2				

Figure 1.2 Memory map of the PIC16F27A microcontroller

6 PIC Projects: A Practical Approach

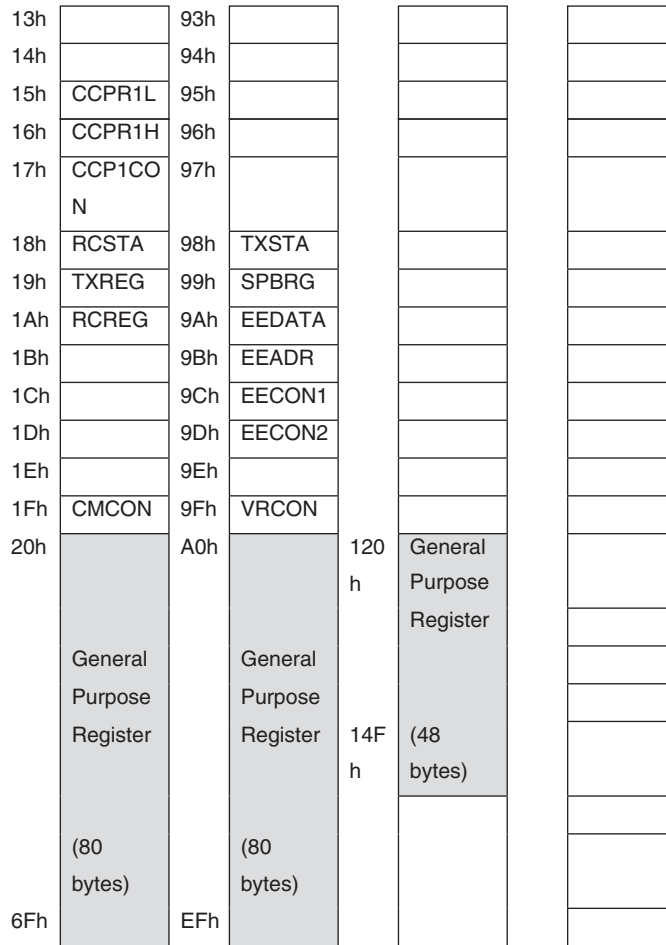


Figure 1.2 (continued)

1.2.3 PIC16F27A Pins

We will complete this section with a short description of all 18 pins of the PIC16F27A microcontroller shown in Figure 1.3.

- RA0 to RA7 are the eight pins of port A. Port A is a bidirectional port, which means it can be configured as an input or an output. The number following RA is the bit number (0 to 7). So, we have one 8-bit directional port where each bit (with the exception of bit 5) can be configured as input or output. As shown in Figure 1.3, all of the port-A pins have alternative functions.

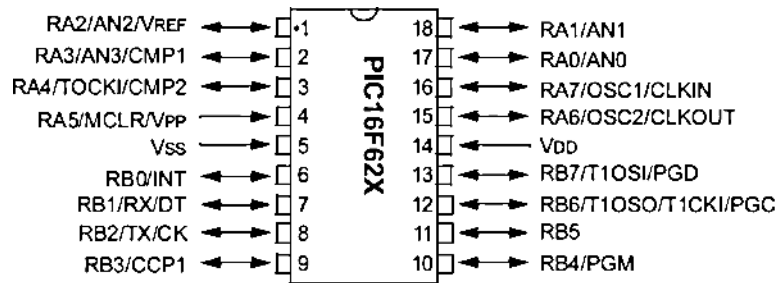


Figure 1.3 Pin-out diagram of the PIC16F27A microcontroller

© Microchip Technology Inc. Reproduced with permission.

- Pins 1, 2, 17 and 18 can be configured and used as analog inputs (AN0 to AN3). Those pins are therefore attached to internal comparators of the PIC. Voltage in the range 0–5 V on those pins is converted into digital form and further processed by the microcontroller.
- Pin 4 can be used as a master clear – reset pin (MCLR). Reset is used for putting the microcontroller into a ‘known’ (default) condition. Upon setting this pin to 0 V, all of its registers will be placed in a starting position. Here we use internal reset circuitry activated when the processor is powered up (power-up reset). This option is normally used when the microcontroller does not behave in an appropriate way due to some undesirable condition.
- Pins 16 and 17 (OSC1/CLKIN and OSC2/CLKOUT) can be used when an external oscillator or crystal/RC timing elements are used to provide timing for the microcontroller.
- Pin 3 (TOCK1) can be used as an input for the Timer 1 module. It operates independently from the main clock.
- RB0 to RB7 are the eight I/O pins of port B. Port B is a second bidirectional port and it behaves in almost the same way as port A. Alternative functions of port pins are different for port A and port B.
- Pin 6 can be configured and used as an external interrupt pin to detect external events. An event is detected when the interrupt pin changes state from ‘1’ to ‘0’ or from ‘0’ to ‘1’ (programmable).
- Pins 7 and 8 can be used for serial communications: TX is the asynchronous serial transmit pin – data is sent from the chip on this pin; RX is the serial receive pin and data is sent to the chip on this pin.
- Pin 9 can be used as a capture/compare pin (CCP1) in order to measure duration of external events (the length of the PWM – pulse width modulated pulse).

8 PIC Projects: A Practical Approach

- Pin 10 is used for a low-voltage programming of the PIC.
- Pins 12 and 13 can be used as an oscillator and as timer inputs for timer 1.
- VSS and VDD are the power supply pins. VDD is the positive supply and VSS is the low supply or 0 V. All PIC family members operate off a 5 V supply and some can use supplies down to 2 V.

1.2.4 More on PIC Architecture

Trying to explain the architecture of PIC microcontrollers in more detail would probably take another book or at least another long and not-very-interesting chapter of this book. Instead, we will provide a very brief insight into some of the PIC features useful for understanding and doing some interesting PIC projects explained in the other chapters.

- *Timing.* An oscillator (internal or external) is generally used to drive the PIC by clocking data and instructions into the processor. The actions of the CPU are caused by every fourth oscillator pulse, which makes the instruction times easy to calculate. Most instructions take one clock cycle (four oscillator pulses), so with a 4 MHz oscillator it will take 1 μ s to execute each of those one-cycle instructions.
- *Program execution control.* The program counter is an internal 13-bit CPU register used to store the current program position. After each loaded instruction the program counter is incremented automatically so that it points to the location of the next instruction in the program memory.
- *CPU status.* Bits of the status register contain the information about the status of the arithmetic and logic unit from the CPU. Those bits are updated after certain instructions that modify the main working register content of the CPU. The PIC16F27A has an 8-bit status register, which contains information about the memory bank currently being accessed, carry, power state of the PIC, borrow or zero results of the executed instruction.
- *Timers.* To provide accurate timing for the microcontroller actions a special function register called a timer can be used. This register is connected to the internal clock and increments at the clock frequency divided by four. When it rolls over from its maximum count (255 for an 8-bit timer, 65535 for a 16-bit timer) to zero, a flag is set to signal that event. It would not take long to count from 0 to the maximum count at 1 MHz, so a programmable prescaler can be used in combination with the timer module. The prescaler can be set to give out

pulses at ratios of 1:2, 1:4 and so forth, up to 1:256, extending the timeout up to tens of milliseconds range for a 4 MHz oscillator used in the system.

- *Interrupts.* An improved solution to exact timing described above is to set up a timer to generate an interrupt. A routine can be set by the programmer to be executed automatically every time the timer overflows (rolls over at the maximum count). This requires some programming skill and will be explained in more detail in Chapter 5. There might be other situations that can cause an interrupt routine to be executed, such as a change of state on the port B pins or some external event sensed by the interrupt pin RB0.
- *Watchdog timer.* To provide a means of recovery from some system problems a watchdog timer is implemented on all PIC microcontrollers. During its execution, the program needs to reset the watchdog timer at predetermined intervals, but if it fails to do so, due to a problem in the execution, the watchdog timer will initiate the reset itself. This can be a useful option in case the program goes into an endless loop or some hardware problem occurs to prevent the program operating correctly.

1.2.5 Brief Overview of the PIC Family

The whole PIC family can be divided in a number of distinct groups:

- *Baseline core devices*, represented by the PIC10 series, as well as some PIC12 and PIC16 devices. Baseline devices are available in six-pin to 40-pin packages.
- *Midrange core devices*, labelled PIC12 and PIC16.
- *PIC17 and PIC18 high-end core devices*, represented by a not-very-popular PIC17 series, produced in packages from 40 to 68 pins and later superseded by the PIC18 architecture.
- *PIC24 and dsPIC microcontrollers* are 16-bit microcontrollers. The PIC24 devices are designed as general-purpose microcontrollers and dsPICs include digital signal processing capabilities.
- *PIC32MX* – these 32-bit microcontrollers are the latest addition to the PIC family introduced in November 2007.

1.3 Basics of PIC Assembly Language

Assembly languages are closer than high-level languages to possessing a one-to-one correspondence between symbolic instructions and executable machine codes. They are also usually more difficult to use. The aim

10 PIC Projects: A Practical Approach

of this section is to give a brief introduction to some aspects of PIC programming using assembler language. Most of the microcontroller programs spend a lot of time on the actual movement of data through the device. This is certainly the case for assembler language-type programs so we will start our introduction to PIC assembler language by looking into some instructions that are commonly used to move data through the microcontroller.

1.3.1 Data Movement, Arithmetic and Logical Operations

The PIC16F27A has a very small set of instructions – there are only 37 of them. Since the width of the program memory is 14 bits and the address of each register can take up to eight bits, two registers cannot be used in the same instruction. Remember that we also need to specify what action we actually want to accomplish with each particular instruction – too much information to fit into just 14 bits!

To move data from one register to the other it is therefore necessary to use intermediate register storage. The working register, labelled *W*, is used to accomplish this task. This is an internal CPU register that does not have a specific address. Thus, movement of data from one register to the other will require two assembly language instructions.

Suppose we want to move the value of variable *MYDATA* to *PORTB* in order to output it to some external device connected to *PORTB*. The following instruction sequence will accomplish this:

```
1 movf      MYDATA,W;  copy MYDATA into working register W
2 movwf    PORTB    ;  copy working register W into PORTB
```

The first instruction is of the form `movf f,d`, which moves the register or memory location *f* to the destination specified with *d*. In the above sequence *d* is specified as our working register *W* so the value of *MYDATA* is copied to the working register *W*. Other possibilities for *d* are 1 and 0. If *d* is 0, the result is the same – *f* is copied over to register *W*. If *d* is 1, the *MYDATA* register will be copied to itself. This option will have no effect on the content of the register because we are effectively overwriting the register *f* with the same value. It does however effect the *Z* flag in the PIC status register and can therefore be a useful option in some situations. We will not discuss this effect further in this section.

The second instruction is of the form `movwf f` and it simply moves whatever is in *W* into the register *f*. Variable *MYDATA* remains

unchanged in the first instruction and *W* remains unchanged in the second. So it is useful to keep in mind that it is a copy rather than a move action we have achieved through this sequence.

Loading a register with a literal (8-bit value) also takes two instructions and involves the use of working register *W*. Here is an example of loading variable *MYDATA* with the literal binary number 11110000 (hexadecimal *F0*, or *F0h*). This effectively clears lower four bits of the working register *W* and sets the upper four bits of the same register:

```
1 movlw 0xF0           ; put the number/literal F0h into W
2 movf MYDATA         ; copy W into MYDATA
```

To summarize here:

- `movf f, d` copies the content of specified register (*f*) to working register *W* or to itself, depending on the destination specification *d* (*W*, 0 or 1)
- `movwf f` copies the content of the working register *W* to specified register *f*
- `movlw l` copies/moves specified 8-bit literal *l* to the working register *W*.

A common mistake by novice PIC programmers is to confuse loading some register with a value from *W* and loading the working register *W* with a value from some other register. Care needs to be taken when using the data movement instructions listed above.

A similar approach must be taken when, instead of copying, we want to apply arithmetic or logical functions (for example, addition, subtraction, logical AND, OR, XOR and others):

```
1 movlw k           ; move number k into W
2 subwf f, d ; subtract W from f and put the result according to d
```

The result of the first line of the above sequence should already be clear to us. Literal *k* is copied to working register *W*. The second line subtracts *W* from *f* and stores the result into the destination specified by *d*; *d* can again be specified using one of three options – *W*, 0 and 1. In case *d* is specified as *W* or 0, the result is stored back to *W*, *f* is not changed and, if *d* is 0, the result is stored in *f* and *W* is unaffected. Of course, in the above sequence the value *k* needs to be replaced with some real value if the program is to work properly (such as *0xF0*).

12 PIC Projects: A Practical Approach

Table 1.1 Common single-operand arithmetic and logic instructions
(*W* is source and destination register)

Syntax	Description
<code>addlw k</code>	put the value of ' $k + W$ ' in <i>W</i>
<code>sublw k</code>	put the value of ' $k - W$ ' in <i>W</i>
<code>andlw k</code>	put the value of ' k and W ' (logical 'and' operation) in <i>W</i>
<code>iorlw k</code>	put the value of ' k ior W ' (logical 'inclusive or' operation) in <i>W</i>
<code>xorlw k</code>	put the value of ' k xor W ' (logical 'exclusive or' operation) in <i>W</i>

Single instructions to accomplish some arithmetic or logical operation will work if the value to be changed is already in *W* and the result destination is again *W*. Examples are given in Table 1.1.

PIC assembly language also has some instructions where there is no choice between the number of operands to be used in the instruction – only one operand can be used. Commonly used single-operand instructions are given in Table 1.2.

We have seen that the move instruction of type `movwf f` does not have a choice of destination – it is always the register *f* specified in the instruction. PIC has more 'WF'-type instructions but the rest of them have a choice of destination – it can be the working register *W* or the other register *f*, depending on the destination specified. These instructions are listed in Table 1.3.

Some other useful instructions are listed in Table 1.4. It is important to remember that the result of those instructions can be stored in the working register *W* (when *d* is specified as *W* or 0) or in the specified file register *f* (when *d* is specified as 1).

1.3.2 Program Flow Control

The instructions listed and discussed above will probably be the ones you will need to use in order to perform some simple arithmetic and logical

Table 1.2 Exclusive single-operand instructions

Syntax	Description
<code>clrf f</code>	set all bits in specified register <i>f</i> to zero (clear register <i>f</i>)
<code>clrw</code>	set all bits in register <i>W</i> to zero (clear register <i>W</i>)
<code>bcf f, b</code>	set bit <i>b</i> in register <i>f</i> to zero (bit clear bit <i>b</i> in <i>f</i>)
<code>bsf f, b</code>	set bit <i>b</i> in register <i>f</i> to one (bit set bit <i>b</i> in <i>f</i>)

Table 1.3 WF-type instructions

Syntax	Description
<code>addwf f, d</code>	put the value of 'f + W' in destination d (either W or f)
<code>subwf f, d</code>	put the value of 'f - W' in destination d (either W or f)
<code>andwf f, d</code>	put the value of 'f and W' (logical 'and' operation) in destination d (either W or f)
<code>Iorwf f, d</code>	put the value of 'f ior W' (logical 'inclusive or' operation) in destination d (either W or f)
<code>xorwf f, d</code>	put the value of 'f xor W' (logical 'exclusive or' operation) in destination d (either W or f)

Table 1.4 Common two-operand type instructions

Syntax	Description
<code>incf f, d</code>	put the value of 'f + 1' in destination (specified with) d
<code>decf f, d</code>	put the value of 'f - 1' in destination d
<code>comf f, d</code>	put the result of toggling all bits from f in d
<code>swapf f, d</code>	put the result of swapping the nibbles in f in destination d
<code>r1f f, d</code>	put the result of rotating f left through carry in destination d
<code>r2f f, d</code>	put the result of rotating f right through carry in destination d

data operations or to move data between the various destinations in the microcontroller. To make your programs really useful and more complex you also need to be able to control the flow of the program. Normal sequential execution of the program assumes that the program starts with the first instruction, then it goes on to the next one in the list and continues in a similar manner. The instruction to disrupt this kind of operation would need to involve an abrupt change in the program counter content. Two instructions from the PIC set can do this – `goto` and `call`. However, those instructions cannot be used interchangeably as there is one important difference in the way they operate. The `goto` instruction will simply make the program execute the instruction at the specified location in the program memory; for example, `goto 0x123` will set the program counter to the value of 123h. This, in turn, will cause the instruction at location 123h to be executed by the CPU next. Instructions following the one at location 123h would be executed and the program would not return to where it left off before being made to execute the instruction at location 123h. However, `call 0x123` would push the address of the next instruction in the program on the stack (dedicated memory space for exactly

14 PIC Projects: A Practical Approach

this kind of situations) and then set the program counter to 123h, thus making the program execute the instruction at that location next. Instructions at the locations following the location 123h will continue to be executed sequentially but only until the return instruction is encountered. This instruction will cause the retrieval of the address most recently stored on the stack into program counter. The program will therefore return to the instruction following the `call 123` instruction and continue execution from there. This is normally used when writing the subroutine part of the program. The main program is made to call the subroutine to execute a specific sequence of instructions and after the execution of the subroutine the program needs to return to where it left off before the call to the subroutine. It is possible to nest a number of subroutines within each other: one subroutine can call another subroutine, which in turn can call another subroutine and so forth. The number of nested subroutines is determined by the size of the stack. In the PIC16F27A, the stack consists of eight words used in circular manner: after the eighth word, it rolls over to the first. It is therefore possible to nest eight subroutine calls. If we try to nest more than eight subroutines the program will become lost while trying to return from the subroutine to a proper instruction in the calling subroutine or the main program. The program cannot find its way back and will act in a strange and unexpected way. The program will continue to execute but the expected results will not be there. This is a stack overflow problem, which is difficult to detect when testing the program.

There are two more versions of the return instruction for PIC16F27A. All three return instructions are listed in Table 1.5.

Another way to control the flow of the program implemented on the PIC microcontroller and a number of other microprocessors is simply to skip the next instructions depending on the result of the current operation. Those instructions are listed in Table 1.6.

1.3.3 Example Assembler Program

We are now ready to write and understand our first PIC assembler language program. It is shown in Listing 1.1.

Table 1.5 Return from subroutine instructions

Syntax	Description
<code>return</code>	return from call, location of the next instruction is retrieved from the stack and copied into the program counter
<code>retlwk</code>	as 'return' but literal k is also placed into W
<code>retfie</code>	return from interrupt

Table 1.6 ‘Skip-the-next-instruction’ type instructions

Syntax	Description
<code>incfsz f, d</code>	put the value of ‘f + 1’ in destination d (either W or f) and skip the next instruction if the result of this increment is zero
<code>decfsz f, d</code>	put the value of ‘f – 1’ in destination d (either W or f) and skip the next instruction if the result of this decrement is zero
<code>btfsz f, b</code>	test bit b (but do not change it) of register f and skip the next instruction if the bit is clear – bit test skip clear
<code>btfss f, b</code>	test bit b (but do not change it) of register f and skip the next instruction if the bit is 1 – bit test skip clear

```

1 ;*****
2 ; this program outputs two binary patterns on port B *
3 ; one after the other - 01010101 and 10101010 *
4 ;*****
5
6 include "p16F627A.inc"
7
8 bsf STATUS, RPO ; set RPO bit = select bank 1
9 movlw 0x00
10 movwf TRISB ; set port B pins to all outputs
11 bcf STATUS, RPO ; clear RPO bit = select bank 0
12
13 loop movlw 0x55
14 movwf PORTB ; PORTB = 0x55 = 0b01010101
15 movlw 0xAA
16 movwf PORTB ; PORTB = 0xAA = 0b10101010
17 goto loop
18 end

```

Listing 1.1 First assembler program

The first line of the program (line 6 in Listing 1.1) contains the `include` statement, which tells the assembler program to include the specified ‘inc’ type file into assembly procedure. While assembling the binary version of our program, assembler looks in the quoted file (`p16F627A.inc`) for any symbols not defined within the program. The `p16F627A.inc` file holds definitions of all SFRs in the PIC together with their addresses. Each device in the PIC family will have its own `.inc` file with definitions specific for that particular device, so care needs to be taken to include the proper file.

16 PIC Projects: A Practical Approach

As long as there is a power supply to the microcontroller, this program outputs two-bit patterns on eight pins of port B – 01010101 (55h) and 10101010 (AAh). The first bit pattern is moved from the working register *W* to port B in line 14 of the program and the second bit pattern is output to port B in line 16. Previously those two patterns were loaded into the working register in lines 13 and 15. Line 17 redirects the program execution back to line 13 and this process is repeated indefinitely.

The first part of the program (lines 8–11) is used to set port B as an output port. This is done by setting corresponding bits (all of them in this case) of register TRISB low. This is done in lines 9 and 10 of the program where 00h is first moved to working register *W* and then from *W* to TRISB register. This register is located in Bank 1 as shown in Figure 1.2 and the default active Bank on PIC is Bank 0. To switch to Bank 1 the RP0 bit of the STATUS register needs to be set to 1. This is done in line 8 of the program. Bank 0 is again activated at line 11 of the program by resetting RP0 bit back to 0.

Descriptions of each program line following the semicolon are just program comments. Everything following the semicolon sign is ignored by assembler and is only used by the programmer to supply some useful documentation to the code. This can be extremely helpful when trying to understand a program that somebody else has written or even a program that you wrote some time ago.

After typing and saving this code we would need to invoke PIC assembler to assemble the file containing the code – to translate it into machine code. Machine code is a pattern of 0s and 1s understandable and executable by the CPU. We would then need to test the program and run it on the PIC16F27A microcontroller. This process will be explained in the last section of this chapter where the MPLAB development environment is considered. The next section will discuss the development of PIC programs using the C programming language.

1.4 Introduction to C Programming for PIC Microcontroller

Sometimes, when more complex algorithms need to be implemented, assembler language programming can lead to cryptic and difficult-to-understand programs. A lot of programmers nowadays tend to use high-level languages, rather than assembler, to program microcontrollers. Although programming in high-level language is generally much easier, there are also disadvantages. Programs written in high-level language are

larger so more program memory might be needed to store them. They can also be less efficient when it comes to speed of execution – a high-level language program is typically slower compared to a similar program written in assembler language. The high-level language most frequently used for microcontroller programming is C. One of the advantages of using C compared to some other high-level languages is that various C compilers (programs that translate C programs into machine code) are usually very efficient. A lot of manufacturers claim their compiler to be 80% efficient. This basically means that the corresponding assembler language program will only be 20% smaller in size on average compared to our program written in C. This tradeoff between the ease of programming and the program size is usually acceptable in most situations, although you might find that this figure of 80% efficiency is difficult to achieve in practice. C language efficiency is usually somewhere in the 50%–70% region.

This section will introduce some basics of C programming for PIC microcontrollers. Even though C is a standard programming language, there are some specific issues to consider when programming any microcontroller in C. For a more general and detailed treatment of C language, the reader is advised to look at some standard C books, while the main aim of this section remains to introduce the basics of C language. Enough information will be given to start with some simple PIC projects explained in Chapter 2 and then build your knowledge further as you attempt more and more complex PIC projects.

1.4.1 Template C Program for PIC16F27A

C is a function-based language and, in fact, any C program is merely a function. Functions are sections of code that perform a single, well-defined task. Functions are the C equivalent of assembly-level sub-routines. A function in C can take any number of parameters and return a maximum of one value. A parameter (or argument) is a value passed to a function that is used to alter its operation or indicate the extent of its operation. A function consists of a name followed by the parentheses enclosing arguments, or a keyword ‘void’ if the function requires no arguments. If there are several arguments to be passed, the arguments are separated by commas. A simple C function is detailed in Figure 1.4.

Every C program must at least have a function named ‘main’, and this function is the one that executes first when the program is run. Here we can write the simplest possible version of the main function that can run

18 PIC Projects: A Practical Approach

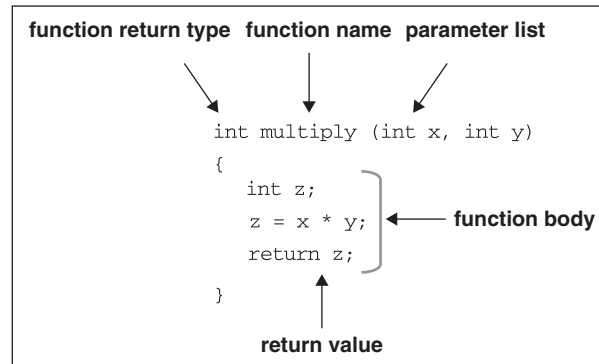


Figure 1.4 Simple C function

```

1 // template C program - does nothing useful - starts and
  stops
2 #include <pic16f2xa.h>
3 void main(void)
4 {
5
6 // body of the program goes here
7
8 }

```

Listing 1.2 Template C program for PIC microcontroller

on the PIC16F27A microcontroller. The program given in Listing 1.2 will do almost nothing – it finishes as soon as it starts. However, it represents a useful template for further development of more complex C programs on our PIC microcontroller.

The second line in Listing 1.2 is the so-called ‘include’ statement. As in assembler language this line tells the C compiler that during the compilation it should look into header file (`pic16f2xa.h`) for any symbols not defined within the program. This file holds definitions of all SFRs in the PIC and their addresses. Different models of PIC microcontroller usually require different `.h` header files with definitions specific for that particular device. Various C compiler manufacturers usually provide their own versions of each header file, specific to and compatible with their own compiler. Care therefore needs to be taken to include a proper `.h` file in your code.

Line 3 from Listing 1.2 declares the main function. The prefix `void` in this line means that this function returns no argument and the second

void means that the function also requires no arguments to be supplied to it. This is almost always the case with PIC C programs, so you will use this construction very often when programming PIC using C language. Line 4 announces the beginning of the main function and line 8 is the end of the function. C functions are always enclosed in curly brackets and the main function is no exception. The fact that there is no real code between those two brackets only means that this C program will do nothing useful – it will start and soon after that it will end. Lines 1 and 6 are program comments.

Everything starting with a double forward slash is a program comment. Like comments in assembler language programs, comments in C programs are ignored by the C compiler and are used to clarify some sections of the code to whoever needs to read, use or modify the program.

The C program in Listing 1.3 is slightly more meaningful.

```
1 // this program sets all 8 pins of port B to high (logic 1)
2 #include <pic1684.h>
3 void main(void)
4 {
5
6     TRISB = 0x00;
7     PORTB = 0xFF;
8
9 }
```

Listing 1.3 Simple C program

This program will set all eight pins of port B to logic 1. Note that the body of the main program consists of just two statements. The statement in line 6 configures all pins of PORTB as outputs by writing logic 0s into the TRISB register and the statement on line 7 sends the number 0xFF to port B causing all pins of that port to go high (logic 1 state). The common way to terminate statements in C code is to use a semicolon at the end of each line, as has been done in the statements in lines 6 and 7 (there are some exceptions not discussed here).

To complete this brief review of C functions we will rewrite the program from Listing 1.3 by introducing a new function called `myfunc`. This function will have a task of sending the number 0xFF to port B. This is, of course, not a clever thing to do – the program itself is so short that there is no point in splitting it even further. We just do it here in order to demonstrate how a C function can be called by the main program, or how to call a function from any other function in a C language program.

20 PIC Projects: A Practical Approach

```
1 // this program sets all 8 pins of port B to logic level 1
2 // it calls a function myfunc to accomplish this task
3 #include <pic1684.h>
4 void main(void)
5 {
6     TRISB = 0x00;
7     myfunc();
8 }
9
10 void myfunc(void)
11 {
12     PORTB = 0xFF;
13 }
```

Listing 1.4 Calling another function from the main function in C

In the program from Listing 1.4, the main function contains two statements. The first statement (line 6) configures PORTB as output and the second statement is a call to the function, `myfunc`, which contains a further, single statement. It copies the hex value of `0xFF` to the PORTB register, thus effectively setting all pins of port B to a logic high state.

1.4.2 Variables

Variables in C are data objects that may change in value during the program execution. Variables must be declared immediately after the curly bracket marking the beginning of a function by specifying the name and data type of the variable. Variable names in C may consist of a single letter or a combination of a number of letters and numbers. Normally up to 52 characters can be used for the variable name. Starting a variable name with a number is not permitted in C and spaces and punctuation are not allowed as part of a variable name. C is a case-sensitive language and two variables 'A' and 'a' are therefore treated as two separate and different variables. In C, a data type defines the way the program stores information in memory. Data types with corresponding specifiers (proper ways of declaring data types in C), allocation sizes and ranges for C18 compiler available from Microchip and designed specifically for PIC18 devices are given in Table 1.7. This is not a universal table as it might change depending on the C compiler.

Defining the type of the variables in a PIC C program is an important factor in the efficiency of a program. The compiler will reserve a memory space sufficient to save a variable according to its type. Larger data types will usually mean longer computation time so if speed is more important than range you should try to make most of the variables in your program

Table 1.7 Basic C data types

INTEGER DATA TYPES					
Type	Size	Minimum	Maximum		
char ^{1,2}	8 bits	-128	127		
signed char	8 bits	-128	127		
unsigned char	8 bits	0	255		
int	16 bits	-32768	32767		
unsigned int	16 bits	0	65535		
short	16 bits	-32768	32767		
unsigned short	16 bits	0	65535		
short long	24 bits	-8,388,608	8,388,607		
unsigned short long	24 bits	0	16,777,215		
long	32 bits	-2,147,483,648	2,147,483,647		
unsigned long	32 bits	0	4,294,967,295		

FLOAT DATA TYPES					
Type	Size	Minimum Exponent	Maximum Exponent	Minimum Normalized	Maximum Normalized
float	32 bits	-126	128	2^{-126} $\approx 1.17549435e$	$2^{128} * (2 - 2^{-15})$ $\approx 6.80564693e$
double	32 bits	-126	128	2^{-126} $\approx 1.17549435e$	$2^{128} * (2 - 2^{-15})$ $\approx 6.80564693e$

© Microchip. Reproduced with permission.

the 'char' type. While doing this you need to be careful and keep in mind the range of the 'char'-type variables. If, for example, the 'unsigned char' variable exceeds the 0–255 range your program will give incorrect results. Since most of the registers on PIC16F27A are 8-bit registers, 'char' is usually sufficient for most variables in the program. To give you an idea how to declare your data in C programs some examples are given below:

```

1 int goals;           // integer variable
2 float x,y,z;         // three float variables
3 char p = 0xFF;       // char variable is declared and
4                       // initialised at the same time
5 double f = 56.3;     // double variable

```

22 PIC Projects: A Practical Approach

1.4.3 Arrays

Arrays in C are specific data structures used to store multiple variables of the same data type. An array of 10 character variables named A can be defined as:

```
char A[10];
```

The above declaration actually declares the array with 10 elements: A[0], A[1], A[2], . . . , A[9], where the value within brackets is called a subscript. In the C language the array subscript starts from 0, so to access all elements of the array A, the subscript needs to take values from 0 to 9. Accessing A[10] is not legal and this can be a source of errors in many programs!

If we want to assign the initial values to elements of an array, we need to enter these values between curly brackets:

```
char days[7] = {1, 2, 3, 4, 5, 6, 7};
```

To access any value from the array days, we need to use [] separators.

```
Tuesday = days[1]; // second value from the array days is
                  // assigned to a variable named Tuesday
Sunday = days[6]; // seventh value from the array days is
                  // assigned to variable named Sunday
Monday = days[7]; // error!!!, days[7] does not exist
                  // some random value will be assigned
                  // to a variable named Monday
```

1.4.4 Constants

C also allows declaration of constants. When you declare a constant, it is rather like a variable declaration except the value cannot be changed later in the program. The attribute 'const' is used to declare constant as shown below:

```
int const a = 1;
const int b = 4;
```

1.4.5 C Operators

C has a full set of arithmetic, relational and logical operators. It also has some useful operators for direct bit-level manipulations on data, which makes it similar to assembler language. Most important operators are listed in Table 1.8. These operators can be used to form expressions. Mathematical operators follow standard precedence rules – multiplication and division will be executed before any addition or subtraction. Sub-expressions in parentheses have the highest precedence and are always evaluated first. Statements may optionally be grouped inside pairs of curly braces { } and as such are called compound statements.

A common mistake in writing the C statements is to use assignment operator '=' instead of equality operator '=='. 'i = j' and 'i == j' are both perfectly legal C statements but the first one will copy the value of variable j into i while the second compares the values of two statements and results in a logical value ('1' if those two variables are equal, '0' if i and j are different).

Table 1.8 Common C operators

	Operator	Action	Example
Assignment operators	=	Assignment	x = y;
Mathematical operators	+	Addition	x = x + y;
	–	Subtraction	x = x – y;
	*	Multiplication	x = x * y;
	/	Division	x = x/y;
	%	Modulus	x = x % y;
Logical operators	&&	Logical AND	x = true && false;
		Logical OR	x = true false;
	&	Bitwise AND	x = x & 0xFF;
		Bitwise OR	x = x 0xFF;
	~	Bitwise NOT	x = ~ x;
	!	Logical NOT	false = !true
	>>	Shift bits right	x = x >> 1;
	<<	Shift bits left	x = x << 2;
Equality operators	==	Equal to	if(x == 10) { . . . }
	!=	Not equal to	if(x != 10) { . . . }
	<	Less than	if(x < 10) { . . . }
	>	Greater than	if(x > 10) { . . . }
	<=	Less than or equal to	if(x <= 10) { . . . }
	>=	Greater than or equal to	if(x >= 10) { . . . }

24 PIC Projects: A Practical Approach

1.4.6 Conditional Statements and Iteration

To control the flow of execution in a C program we usually use a conditional ‘if’ statement as well as ‘for’ and ‘while’ loops.

The ‘if’ statement is used to allow decisions to be made about parts of the program that will be executed depending on some conditions tested by the ‘if’ statement in the program. If the condition given to the ‘if’ statement is true then a section of code is executed as outlined below:

```
if(condition)
{
    execute this code if condition is true
}
```

An example of a simple usage of the ‘if’ statement is given below:

```
if(x<0)
{
    x = -1 * x;
}
```

Sometimes you might want to perform one action when the condition is true and another action when the condition is false. Here, an ‘else’ statement needs to be combined with the ‘if’ statement:

```
if(condition)
{
    execute this code if condition is true
}
else
{
    execute this code if condition is false
}
```

It is easy to chain a lot of ‘if-else’ statements:

```
if(condition 1)
{
    execute this code if condition 1 is true
}
else if(condition 2)
{
    execute this code if condition 2 is true and condition 1
    is false
}
```



```
else // optional
{
    execute this code if condition 1 and condition 2 are false
}
```

It is also possible to nest 'if' statements:

```
if(condition 1)
    if(condition 2)
    {
        execute this code if condition 1 and condition 2 are true
    }
    else
    {
        execute this code if condition 1 or condition 2 are false
    }
```

While the 'if' statement allows branching in the program flow, 'for', 'while' and 'do' statements allow the repeated execution of code in 'loops'. It has been proven that the only loop needed for programming is the 'while' loop but sometimes it is more convenient to use the 'for' loop instead. While the condition specified in the 'while' statement is true a statement or group of statements (compound statements) are executed as outlined below:

```
while(condition)
{
    execute this code while condition is true
}
```

An example of the simple usage of a 'while' statement is given below:

```
while(x>0)
{
    result = result * x;
    x = x - 1;
}
```

The other type of loop is the 'for' loop. It takes three parameters: the starting number, the test condition that determines when the loop stops and the increment expression.

```
for(initial; condition; adjust)
{
    code to be executed while the condition is true
}
```

26 PIC Projects: A Practical Approach

An example of the simple usage of a ‘for’ statement is given below:

```
for(counter = 1; counter <= x; counter = counter +1)
{
    result = result * x;
}
```

1.4.7 Example C Program

To start programming the PIC using C language, we will develop a program very similar to the one designed using the assembler language shown in Listing 1.1. The program will still generate two different bit patterns on port B but will be slightly more complex. In order to ‘slow down’ the program and keep each of two bit patterns on port B for a longer period of time, an additional delay will be generated in the program. This program is given in Listing 1.5.

```
1 // this program outputs two binary patterns on port B
2 // one after the other - 01010101 and 10101010 — with
3 // some delay between them
4
5 #include <pic16f62xa.h>
6
7 void delay(void);
8
9 void main(void)
10 {
11     int j;
12
13     TRISB = 0x00;    // make port B, all bits, outputs
14     while (1)      // do forever, i.e. always true
15     {
16         PORTB = 0x55; // port B = 01010101
17         for(j=0;j<10000;j++) // produce a delay
18         {
19             }
20         PORTB = 0xAA; // port B = 10101010
21         delay();
22     }
23 }
24
25 void delay(void)
26 {
27     int j;
28     for(j=0;j<10000;j++) // produce a delay
29     {
30         }
31 }
```

Listing 1.5 First C PIC program

The ‘main()’ function of this program is defined between lines 9 and 23. After configuring all pins of port B as output pins on line 13, the program enters the ‘while(1)’ loop to execute the sequence of statements in this loop. This type of loop is called an infinite loop. The condition of this loop is always true (it is 1) so the program will stay in this loop ‘forever’, executing statements in the loop as long as there is a power supply to PIC. Unlike conventional C programs, embedded microcontrollers usually run their software as an infinite (do forever) loop. As long as the larger system incorporating the microcontroller operates, it will need the microcontroller to perform its specific task. This task is described by the C sequence enclosed in ‘{}’ following the while(1) statement.

Another way to implement the infinite loop is to use the ‘for’ statement without specifying any condition, i.e. ‘for(;;)’.

Having entered the infinite loop, the program first sends the hexadecimal value 55h to port B. After that it enters the ‘for’ loop at line 17. This loop does nothing particularly useful as the curly brackets following the ‘for’ statement contain no other statements. All it does is waste some time incrementing the loop counter j before the next hexadecimal value (AAh) is output to Port B on line 20. Loop counter j is declared as integer at the beginning of the main program on line 11.

Line 21 calls the function ‘delay()’, which is declared in line 7 and defined between lines 25 and 31. This function, like the ‘for’ loop in the main program, does nothing else but wastes some more time before the program branches back to line 16 to output value 55h to port B again. In fact the main body of this function contains just another ‘for’ loop, identical to the ‘for’ loop already executed in the main program. This is certainly not very good programming practice. A more experienced programmer would probably call the delay() function after line 16 in the program to follow the `PORTB = 0x55` statement and generate the required time delay. The ‘for’ loop used in the main program will then become obsolete. This would reduce the size of the program and make the program somewhat easier to read and understand. We decided to use this rather clumsy approach to demonstrate the use of functions in the C program and to compare the performance of the delay() function with the delay generated by the ‘for’ loop in the main program. With 4 MHz crystal, both time-wasting approaches – the ‘for’ loop in the main program and the delay() function – can be expected to generate a delay of approximately 150 ms. A method to measure the exact delays generated in this program will be explained towards the end of this chapter.

1.5 MPLAB Integrated Development Environment (IDE)

MPLAB is a Windows-based program that makes writing, developing and debugging programs for PIC microcontrollers an easier task. This development environment is produced by Microchip but can incorporate third-party software tools that can also be used when developing an embedded PIC-based application. MPLAB is relatively easy to use, has a friendly graphical interface and can be freely downloaded from the Microchip web site. This free version can later be easily upgraded with more powerful C compilers and other additions for more complex tasks.

Installation of this package is a relatively straightforward procedure. The MPLAB installation program is provided on the web site for this book but can also be downloaded directly from the Microchip web site (<http://www.microchip.com/>, accessed 29 December 2008). The reader should bear in mind that this software is updated frequently by the manufacturer so the most recent version of the MPLAB software will always be available from the Microchip web site and might have a slightly different set of features from those described in this book. MPLAB is, however, backwards compatible, which means that all programs and features described in this book should also be available and accessible in any newer version of MPLAB. In case the reader is unsure and does not want to experiment with newer versions of MPLAB, the best approach is to stick with version 8 of MPLAB, provided with this book, download it from the companion web site and install it on the computer.

The installation procedure is relatively simple. A zip file, 'MPLAB_v8.zip', containing all necessary files for MPLAB installation needs to be extracted, preferably in an empty or newly created directory. Double clicking on the extracted installation file 'Install_MPLAB_v8.exe' will start Installation Shield and bring up the MPLAB installation window.

Nothing special needs to be configured for the MPLAB installation. All that needs to be done is to accept the default settings by choosing the option NEXT several times and accepting the terms of agreement. After several minutes, MPLAB should be installed on your computer without any problems. At the end of the MPLAB installation, you will be offered an option to add the HI-TECH PICC Lite software suite to your MPLAB. This is a free version of C compiler produced by the third-party supplier HI-TECH. Most of the C programs from this book are compiled using this software, so it is advisable to accept this option and install PICC at this point. If you do not do it at this point, the HI-TECH

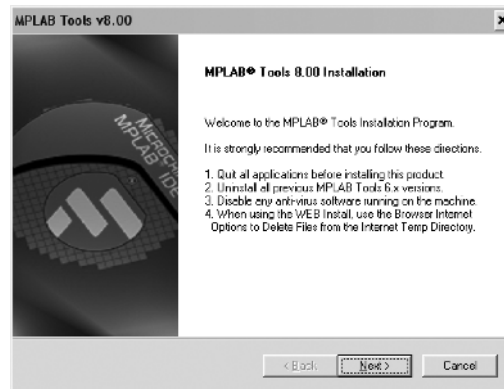


Figure 1.5 Start of the MPLAB installation procedure

© Microchip Technology Inc. Reproduced with permission.

PICC compiler can easily be downloaded from the book web site and installed at some later time

Select NEXT and accept the terms of agreement and select NEXT again. Accept the options offered for the installation and click NEXT again to complete the installation procedure. At this point you will have the option to complete installation and restart the computer. Accept this option if you want to use MPLAB immediately. If not, you will need to restart your computer at some later time before starting to work with

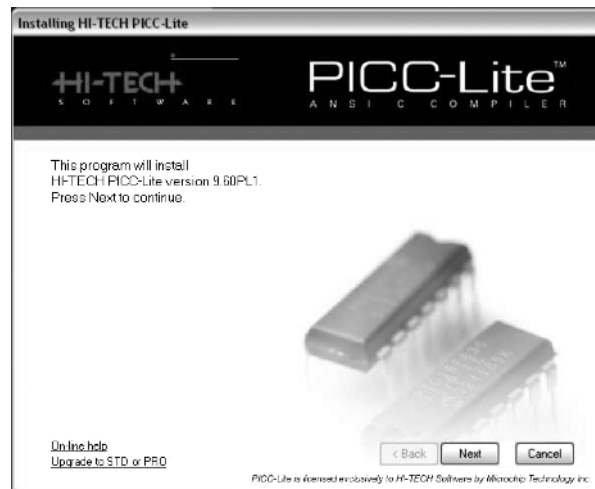


Figure 1.6 Addition of the HI-TECH PICC compiler to the MPLAB suite

© Microchip Technology Inc. Reproduced with permission.

30 PIC Projects: A Practical Approach

MPLAB. To complete our MPLAB development environment it might be a good idea to add one more C compiler to it – C18 C compiler.

The installed HI-TECH C compiler is free and can be used to compile programs for a large number of PIC microcontrollers from the PIC10, PIC12 and PIC16 group. To compile programs for PIC18 devices we need to use the C18 C compiler provided by Microchip. This compiler can also be downloaded from the companion web site for this book or, alternatively, from the manufacturer's (Microchip) web site. It installs easily but some options need to be specified during the installation procedure. After starting the installation procedure by double clicking on the 'MPLAB-C18-Student Edition-v3_16.exe' installation file, click NEXT and accept the licence terms.

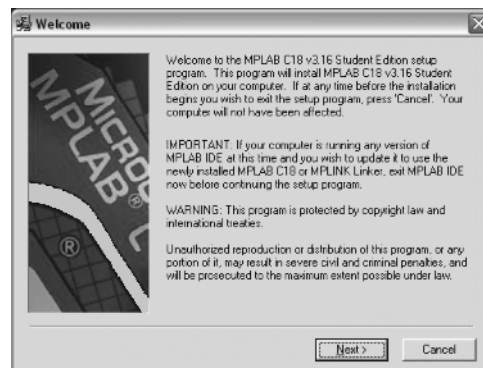


Figure 1.7 Start of the C18 compiler installation procedure

© Microchip Technology Inc. Reproduced with permission.

After selecting the option NEXT, accept the suggested location of the compiler files and select NEXT again. You will be presented with three screens of options for this installation.

We recommend accepting the suggested options from the first window and selecting all options offered (but not selected for you) in the second and third windows. With this selection you will set the environmental variables in the Windows operating system, which makes it easier for the compiler to locate where certain files are on the computer. You will also add this compiler to the MPLAB environment. Clicking NEXT again will start the installation of the C18 compiler on your computer. At the end of the installation, select FINISH to complete it and open the Release Notes for C18, if you want to read through them at this point. Do not forget to restart your computer before starting to use MPLAB for the first time.

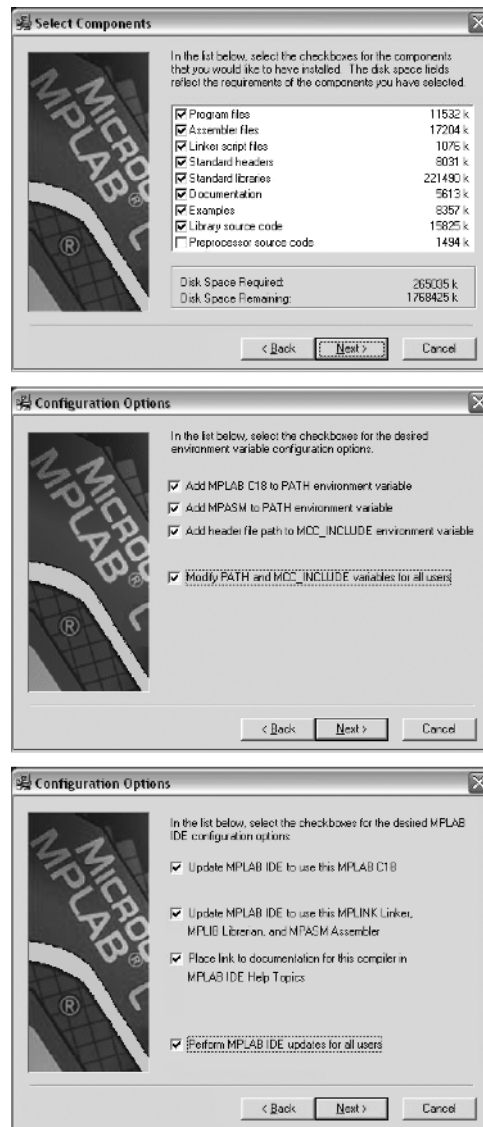


Figure 1.8 Selecting C18 installation options

© Microchip Technology Inc. Reproduced with permission.

After successful installation and restarting the computer, the PIC program can be developed using MPLAB in three main steps – MPLAB project specification, program writing and conversion of the written program into executable binary (1-0 code understandable by the microcontroller).

32 PIC Projects: A Practical Approach

1.5.1 Creating a PIC Project in MPLAB

Once the MPLAB is started, to create a new MPLAB project click on the PROJECT option from the menu and select the Project Wizard, which will open a new MPLAB window, shown in Figure 1.9.



Figure 1.9 Start of the MPLAB Project Wizard procedure

© Microchip Technology Inc. Reproduced with permission.

Click on the NEXT button to continue. We now need to choose the appropriate member of the PIC microcontroller family for this project. Various PIC types will be discussed briefly later in this chapter. For our first PIC project we will select the PIC model discussed in this section: PIC16F627A.

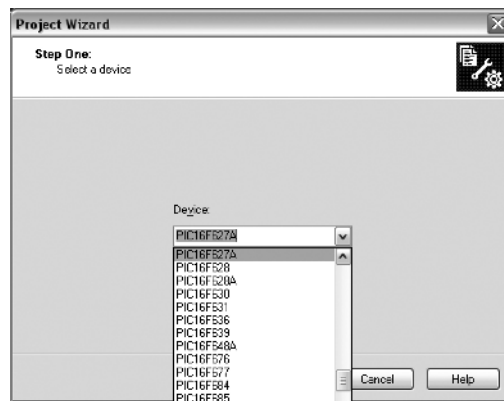


Figure 1.10 Selecting the PIC model during the project definition

© Microchip Technology Inc. Reproduced with permission.

In the next step of the project specification procedure we need to define the programming language to be used in the project. We will use assembler language for our first project, so we need to select the MPASM toolsuite for assembler language software as shown in Figure 1.11.

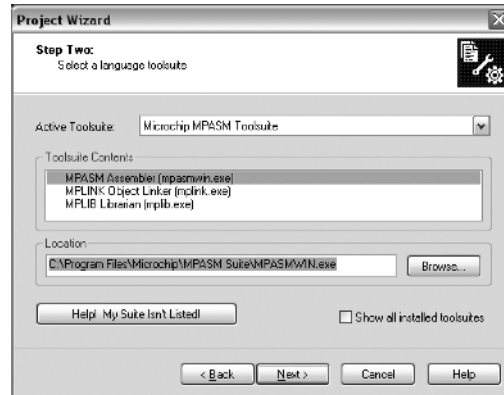


Figure 1.11 Selecting a language and suitable tools for project

© Microchip Technology Inc. Reproduced with permission.

In the final step, we need to select the name and folder for our project. Usually, we select the project name to somehow reflect the nature and purpose of the program. It is usually a good idea to create a new folder for each independent MPLAB project. This is going to be our first PIC program, so we have created a folder 'E:\PICprograms\firstprogram' to hold our new PIC project called 'firstPICprogram'.

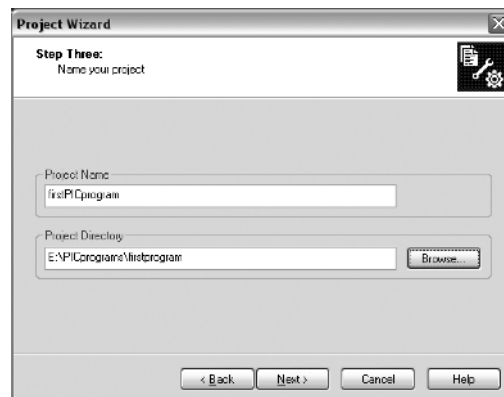


Figure 1.12 Specifying project name and location

© Microchip Technology Inc. Reproduced with permission.

34 PIC Projects: A Practical Approach

We can now click on the NEXT button to finalize the project specification procedure. A new window will offer the possibility of including some existing files in the newly created PIC project. We are going to create a new assembler file for this project so we click NEXT at this stage. This will bring on a new project summary window that contains the summary specification of our project. At any stage of the project specification procedure it is possible to step back by clicking on the BACK button in order to change parameters set in that stage. The same option exists in the project summary window. By clicking on the FINISH button, all project parameters will be selected and the project finally created. This will result in a number of project files being created in the project folder, which will be discussed later in the book, but we will now proceed to the second phase of the PIC project development – program writing.

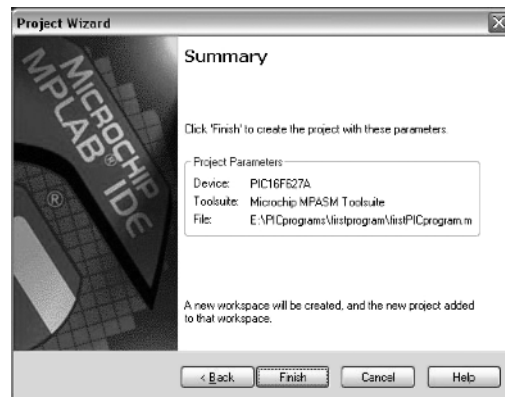


Figure 1.13 Project summary window at the end of the project creation procedure

© Microchip Technology Inc. Reproduced with permission.

1.5.2 Writing a PIC Program

By clicking the FINISH button in the project summary window we have finished the project creation phase and the new screen in MPLAB should now appear to take us through the program-creation phase. At this point your newly created project is open and all of the files created in the previous phase and associated with this project are automatically added to a project.

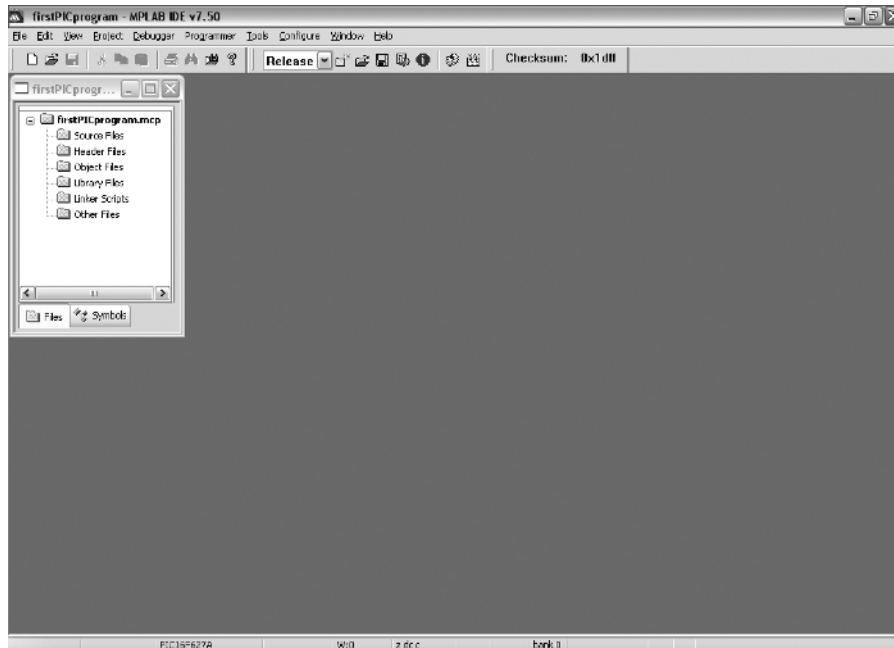


Figure 1.14 Main project window at the start of source file creation

© Microchip Technology Inc. Reproduced with permission.

To create a new file to hold our assembly language program we need to select FILE and then NEW from the MPLAB menu. This will open a new window. In this window we will type assembly code for our PIC program. This code is usually called source code and the file holding the source code is called the source file. Every PIC project needs to contain at least one source file.

We can now start typing the source code into the source file window. The program that we will enter at this stage is a simple program developed in Section 1.3.3 and will only serve the purpose of demonstrating the use of the MPLAB development environment. This program can be entered into this window manually, which is a recommended option for the reader at this stage. Alternatively, all of the programs from this book are also available from this book's companion web site. They can be downloaded, copied and pasted into the source window or once downloaded into an appropriate directory on your computer, included straight into the project. This will be a recommended option for more complex programs used in the later chapters of this book.

36 PIC Projects: A Practical Approach

Readers should now retype the program into the newly opened window. When the new source file is completed, we should save it into our working folder (E:\PICprograms\firstprogram), for example, using some meaningful name for it, to reflect the nature of the program. As this is our first PIC program and the name of the project is ‘firstPICprogram’ we shall name this file ‘firstprogram.asm’. Note the ‘asm’ extension for this file, which indicates the assembler language nature of this source file.

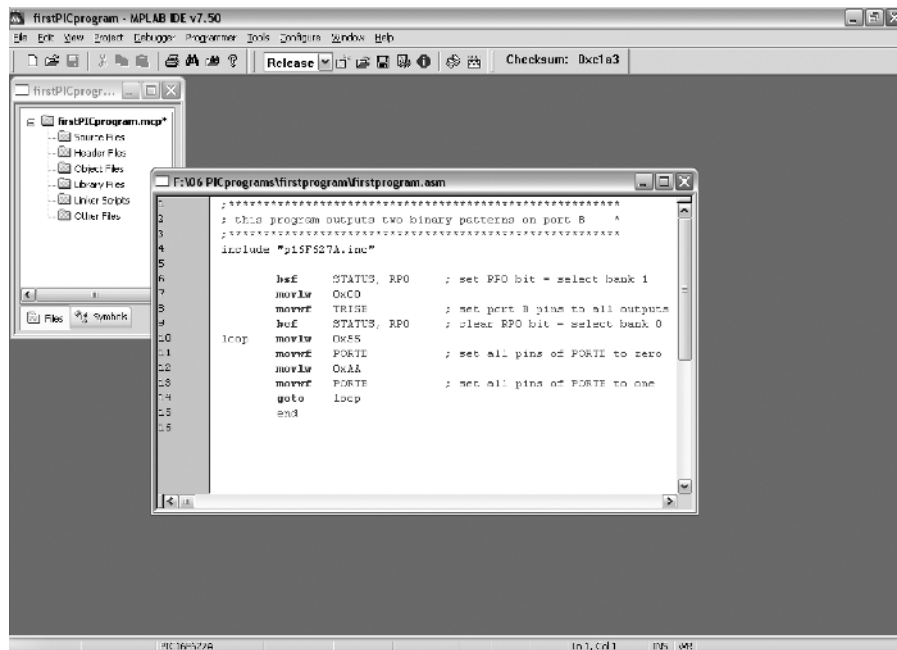


Figure 1.15 Main project window with the newly created source file

© Microchip Technology Inc. Reproduced with permission.

The new file, ‘firstprogram.asm’, should be added to our project. To do this, we need to select the ADD TO THE PROJECT option from the PROJECT menu. A new browser-type window will open. We now need to find our PIC folder and select the file ‘firstprogram.asm’, as shown in Figure 1.16.

The addition of the new source file ‘firstprogram.asm’ to our project will be reflected in the MPLAB project window shown in Figure 1.17. It can be seen that ‘firstprogram.asm’ file now appears in the source files group in that window.

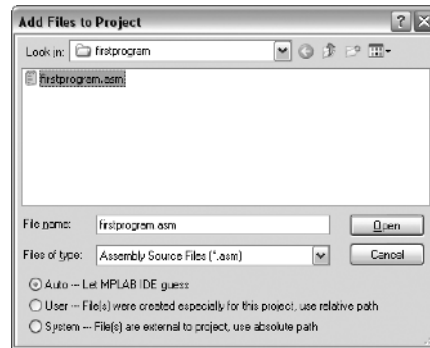


Figure 1.16 Adding source file to a current project
 © Microchip Technology Inc. Reproduced with permission.

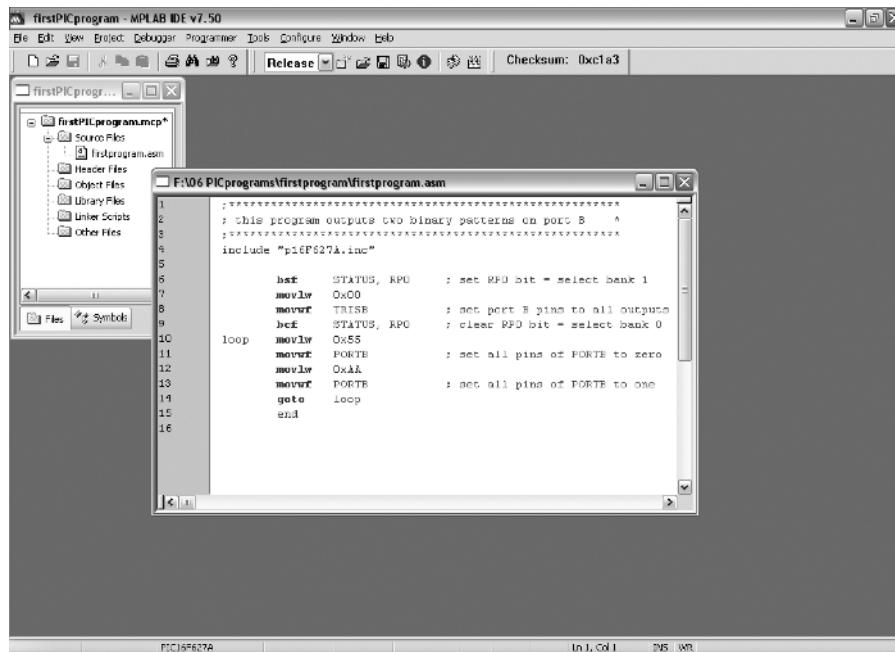


Figure 1.17 Main project window with the newly created source file added to the project

© Microchip Technology Inc. Reproduced with permission.

38 PIC Projects: A Practical Approach

1.5.3 Translating the Program into Executable Code

Once the full program is entered into the source code window we can resave it by choosing the SAVE option from the FILE menu. Alternatively we can translate the program into executable form by selecting the BUILD ALL option from the PROJECT menu or just pressing F10. This will also save the latest version of the program. If the program is written properly, a BUILD SUCCEEDED message should appear in the output window of the MPLAB. This means that our translation was successful and that there were no errors in the source code we typed. This should be the case with your first PIC program if you typed it correctly.

If a syntax error does show up, it needs to be corrected. By double clicking on the relevant error message in the output window you will be transferred to the source code window in the assembler code line where the error was detected. Understanding what the particular error is and how to correct it is one of the main tasks facing the programmer and the rest of this book will hopefully help the reader in understanding some of the issues related to this task more clearly. To be able to do this, the programmer needs to know more about the hardware of the target processor he is programming. He also needs to have a good knowledge of the programming language he is using to program this hardware. The program you have just entered should be simple enough and errors should be easily detected using MPLAB help.

1.5.4 Simulating the Program

We will finish this section by introducing one additional feature of the MPLAB package – the MPSIM simulator. The simulator is part of the MPLAB environment, which provides a better insight into the workings of a microcontroller. Through a simulator, we can monitor current variable values, register values and status of port pins. For a simple program, like the one you have just entered, simulation is not of great importance as the operation of this program is quite straightforward and easy to understand even for the PIC beginner. However, the simulator can be of great help with more complicated programs which include timers, different conditions where something happens and other similar requirements (especially with mathematical operations). Simulation, as the name indicates, ‘simulates the work of a microcontroller’. As the microcontroller executes instructions one by one, the simulator moves through a program step-by-step (line-by-line) and follows what goes on with data

within the microcontroller. When writing is completed, it is advisable to first test the program in a simulator and then run it in a real situation. Unfortunately, as with many other good habits, many programmers tend to avoid this one too, more or less. Reasons for this are partly personality and partly a lack of good simulators. We will now explain how to use the MPLAB simulator to simulate and test our first PIC program.

Once our assembler program is successfully built (assembled) we need to select the MPLAB simulator from the SELECT TOOL option from the DEBUGGER menu. A new set of icons will appear on the menu bar of MPLAB and additional menu items will appear in the debugger menu.

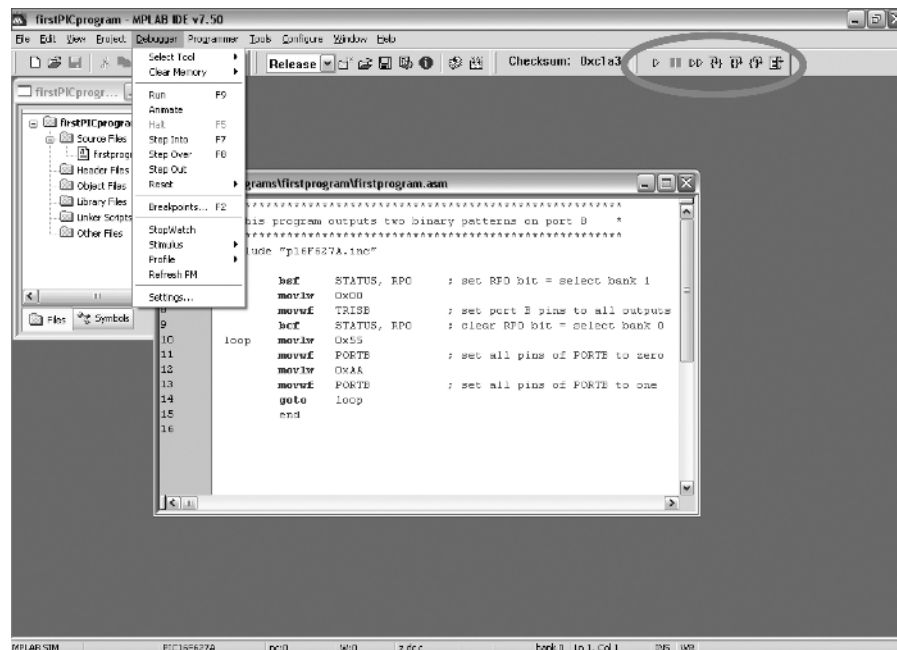


Figure 1.18 Main project window with the source file and invoked simulator

© Microchip Technology Inc. Reproduced with permission.

Our program is now ready to run. It is usually a good idea to reset the program before a fresh run. This can be done by selecting the RESET option from the DEBUGGER menu. A green arrow should appear at the left margin of the source-code window indicating that the first line of the code is to be executed. By selecting RUN from the DEBUGGER menu the program will start running. A text message 'Running . . .' will appear on the status bar. To halt the program execution we need to

40 PIC Projects: A Practical Approach

select HALT from the DEBUGGER menu. The line of code where the application halted will be indicated by the green arrow.

We can also execute a program through single steps – executing each instruction from the program individually. To single-step through the application program, select the STEP INTO option. This will execute the currently indicated line of code and move the arrow to the next line of code to be executed. Options for program execution using MPSIM are:

- *Run* – runs the program at the simulation’s full speed but variables cannot be watched in this mode; this option is normally used with breakpoints set in the program to stop the execution at one or more points in the program.
- *Animate* – runs the program at the speed of only several instructions per second (rate can be further adjusted from the DEBUGGER-SETTINGS menu); the program execution is therefore slowed down so that the effect of each line can be observed.
- *Step into* – executes one line of the program and steps into subroutine if the call to subroutine is encountered.
- *Step over* – executes one line of the program and subroutine in one go if the call to subroutine is encountered (it steps over the subroutine).
- *Step out* – when the program is in a subroutine this option will complete the execution of the subroutine at full speed.

The shortcuts for these commonly used functions in the DEBUGGER toolbar can be used to speed up this procedure. Those shortcuts are shown in Figure 1.19.

Run	Halt	Animate	Step into	Step over	Step out	Reset
						

Figure 1.19 MPSIM shortcuts

© Microchip Technology Inc. Reproduced with permission.

Another feature of MPSIM is the ability to track changes in variable or register values in the program and observe those changes during the program execution. This can be done using the Watch window. A number of Watch windows can be activated from the VIEW menu. In our current application we might want to observe more closely the value of PORTB. This is one of the PIC special function registers, so we can

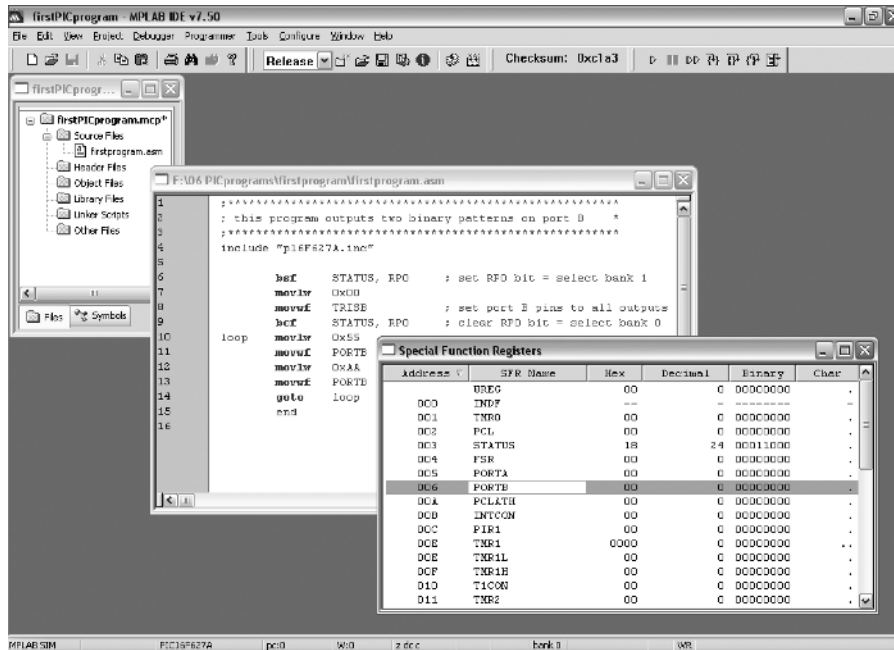


Figure 1.20 Main project window with the SFR Watch window

© Microchip Technology Inc. Reproduced with permission.

select the SPECIAL FUNCTION REGISTERS option from the VIEW menu. A new window with all the files appears in MPLAB.

We can now run the program by animating it or by single-stepping through it. Change on PORTB can be observed clearly in both cases. If we decide just to run the program, those changes cannot be tracked as this option runs the program in ‘near real time’, which is too fast for us to spot anything if the real system is used and too fast for MPLAB to update the Watch window. This feature can, however, be used in combination with breakpoints in the program to reach a certain point in the program quickly, stop the execution of the program and observe the state of the registers or memory at that point in the program. To set the breakpoint in the program you need to double-click on that line of the program in the source window. Let us set the breakpoint on the last line of our program (line 14) by double clicking on that line. A ‘stop’-type breakpoint symbol should appear next to the selected line of code in the left margin of the source window. Now, reset the program and choose the option ‘run’ from the DEBUGGER toolbar. The program will execute quickly and stop at line 14 of our program as indicated by the green arrow over the breakpoint sign in the source window. If we run

42 PIC Projects: A Practical Approach

the program again, it will stop at the same place after one cycle of our endless loop execution has been completed. Several breakpoints can be set in the same program. Breakpoints are a good way to get to a certain point in a long and complicated program quickly. They are not so important for a simple and short program like the one we have in our source window.

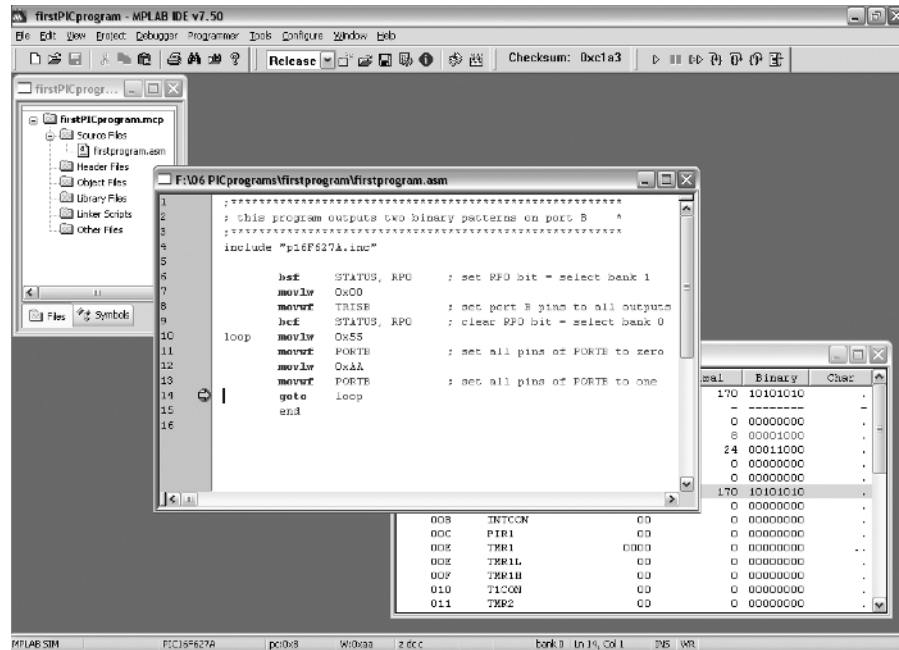


Figure 1.21 Program stop at the breakpoint

© Microchip Technology Inc. Reproduced with permission.

Another useful option in MPSIM is the possibility of measuring the execution time of our program or parts of the program. To do this we first need to set the frequency of the simulator clock. This can be done by selecting the SETTINGS option from the bottom of the DEBUGGER menu. Set the value of the processor frequency to 4 MHz and close this window.

Now select the option STOPWATCH from the same menu to bring up the stopwatch window. Reset the simulation and run the program again. Measurement in the stopwatch window will show the time of the execution of one loop sequence in our program as the program will again stop at the breakpoint we set in the previous phase of program testing. The elapsed time is 8 μ s. This is the correct measurement and agrees with our estimation for the program time execution on PIC made in Section 1.2.4. Our loop consists of eight instructions and each instruction took 1 μ s to

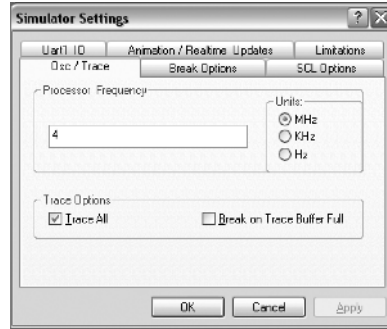


Figure 1.22 Selecting the frequency of the processor for the simulator

© Microchip Technology Inc. Reproduced with permission.

execute. To confirm this, we can reset the program and start stepping through it while observing the stopwatch window. Each step through the program will increment the stopwatch window by 1 μ s. Notice that the only deviation from this rule is the execution of the `goto` instruction in our program. This is a flow-control type instruction and, like the other flow-control instruction (`call`), it takes two clock cycles to execute, which is 2 μ s execution time for the 4 MHz clock.

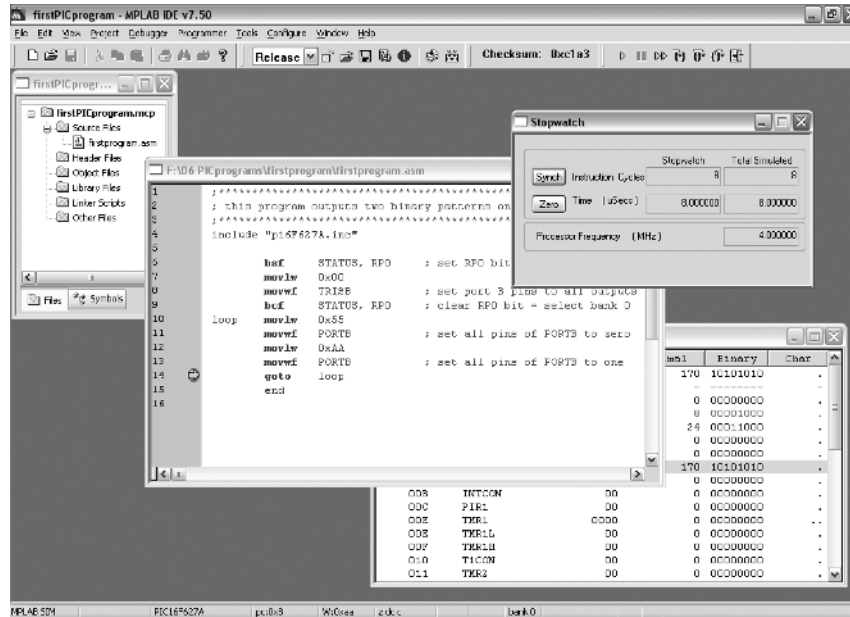


Figure 1.23 Stopwatch window with the indicated time of program execution (up to a breakpoint in the program)

© Microchip Technology Inc. Reproduced with permission.

44 PIC Projects: A Practical Approach

At this point you might want to remove the breakpoint from your program. This can be done by double clicking again on the breakpoint line or break sign in the margin. Alternatively you can choose the BREAKPOINTS option from the DEBUGGER menu and remove the breakpoint by filling in the newly appeared dialogue box. This breakpoints dialogue box can also be used to configure (set new, remove some or all or just disable without permanently removing) breakpoints in your program.

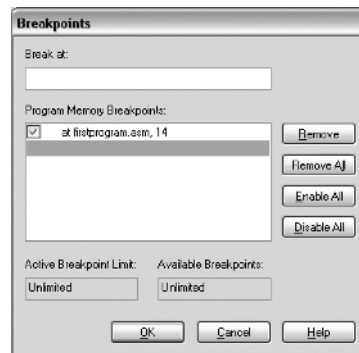


Figure 1.24 Breakpoints control window

© Microchip Technology Inc. Reproduced with permission.

1.5.5 Creating a PIC C Project in MPLAB

MPLAB can be used in a similar way to create and test project for the PIC programs written in C programming language. To create a C-type PIC project after opening MPLAB, start a Project Wizard again and select the PIC device for your project as described in the previous section. The next option in the project creation will require specification of the programming language and compiler tool to be used in the project. Select the 'HI-TECH Universal Toolsuite' and 'HI-TECH C Compiler' on this screen and proceed by selecting NEXT.

On the next screen specify the full path and the name of the new project file as shown in the window below and select NEXT again.

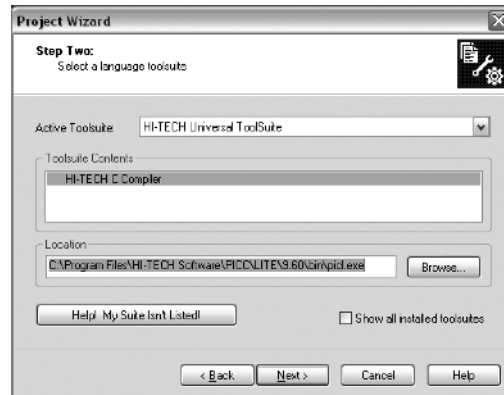


Figure 1.25 Project Wizard window with selected C compiler and specified project location

© Microchip Technology Inc. Reproduced with permission.

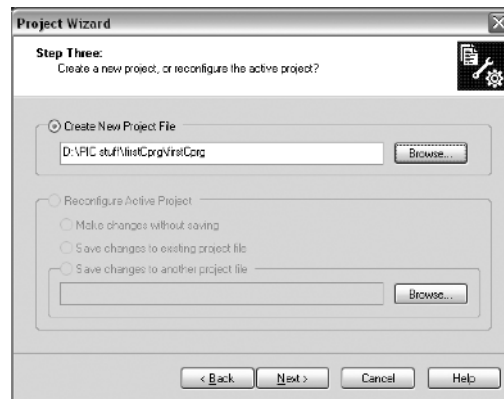


Figure 1.26 Project Wizard ready for the creation of the new project

© Microchip Technology Inc. Reproduced with permission.

On the next screen select NEXT again and FINISH on the project summary window.

Program creation is very similar to the process for the assembler language program explained in Section 1.5.2. This time we need to type the C version of our first PIC program given in Listing 1.5 and save it in the file with a .c extension. The C source file needs to be added to our project and the project needs to be built and tested in a similar way to the assembler MPLAB project. Figure 1.28 shows MPLAB with a project

46 PIC Projects: A Practical Approach

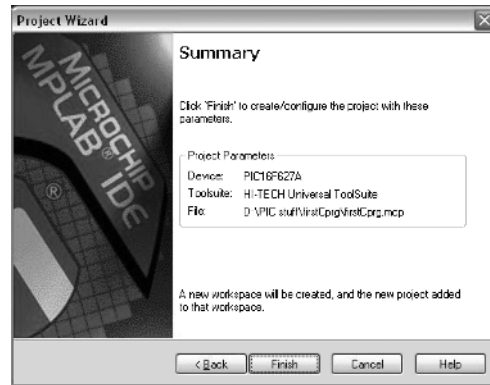


Figure 1.27 Project summary window at the end of the C project creation

© Microchip Technology Inc. Reproduced with permission.

created and built using the C program from Section 1.4.7. The MPLAB SIM tool is activated and PORTB is selected in the Watch window. Two breakpoints are set in the program to stop the execution of the program so that the change of bit pattern on PORTB can be observed.

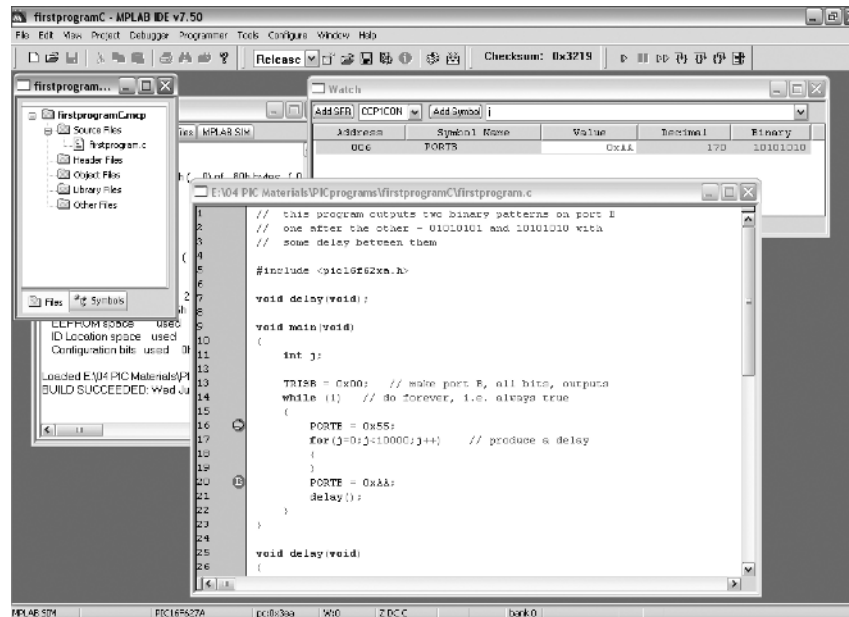


Figure 1.28 First C PIC project

© Microchip Technology Inc. Reproduced with permission.

Time delays generated by the ‘for’ loop and the delay() function can be measured using those breakpoints and a Stopwatch feature from the DEBUGGER menu. To do this, first set the Processor Frequency to 4 MHz by selecting the Settings . . . option from the DEBUGGER menu. In the new Simulator Settings window select Osc/Trace tab and set the Processor Frequency to 4 MHz. The corresponding screenshot of MPLAB is shown in Figure 1.29.

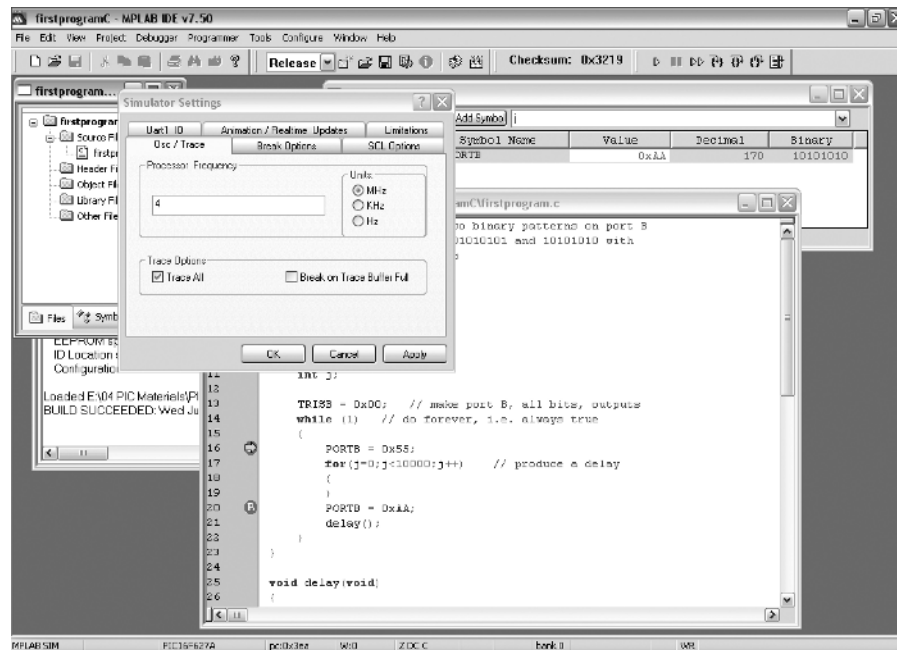


Figure 1.29 Setting the processor frequency

© Microchip Technology Inc. Reproduced with permission.

Confirm your selection by clicking OK to close the Simulator Settings dialogue box and select the Stopwatch option from Debugger menu. Run the program to the first breakpoint and reset the stopwatch time by pressing Zero. Now, run the program again to the second breakpoint and note the elapsed time – 140 ms. Zero and run again. This time the program was delayed by the delay() function and measured delay was slightly longer – 150 ms. Can you think of any reasons for this? The number of instruction cycles executed to the first and second breakpoints, provided by the Stopwatch window, can give you a clue.

48 PIC Projects: A Practical Approach

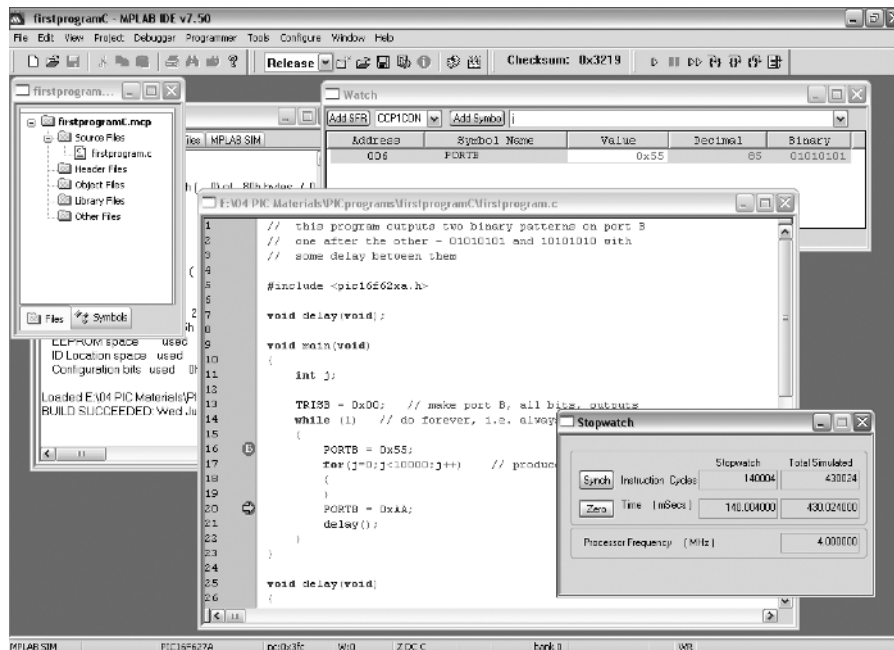


Figure 1.30 Using the Stopwatch feature to measure delays in the program

© Microchip Technology Inc. Reproduced with permission.

1.6 Advanced Debugger Features – Stimulus

We will complete this chapter by demonstrating one more useful feature of the MPLAB SIM tool – simulation of program stimulus from the outside world. To do this we will modify our C program according to Listing 1.6.

The modified program first reads the state of PORTA on line 13 and stores the result in a variable called ‘temp’ – unsigned character. According to the value of this variable (the state of PORTA), one of two binary patterns will be output to PORTB – 55h if all pins of PORTA are set to logic low (0) or AAh if any of PORTA pins is set to logic high (1). The test is implemented using if-else statement between lines 15 and 22. Pins of PORTA are configured as inputs at the beginning of the program, on line 9.


```
1 //this program outputs one of two binary patterns on port B
2 // depending on the state of port A
3
4 #include <pic16f62xa.h>
5
6 void main(void)
7 {
8     unsigned char temp;
9     TRISA = 0xFF;           // make port A, all inputs
10    TRISB = 0x00;          // make port B, all bits, outputs
11    while (1)              // do forever, i.e. always true
12    {
13        temp = PORTA;
14
15        if (temp == 0)
16        {
17            PORTB = 0x55;
18        }
19        else
20        {
21            PORTB = 0xAA;
22        }
23    }
24 }
```

Listing 1.6 Second C PIC program

Create a new PIC project, type in the C program given in Listing 1.6, add it to the project and build the project. Open the Watch window and select PORTB and PORTA to watch during the program animation. In order to test the operation of this project we would need to somehow change the state of the PORTA pins during the program animation. This can be done using the stimulus feature of the MPLAB SIM tool. To open the Stimulus window, select Debugger-Stimulus-New Workbook from the MPLAB menu. The Stimulus window should appear in MPLAB as shown in Figure 1.31.

Three main types of stimulus can be set up in MPLAB using the Stimulus window:

- *Manual triggers* – changes in digital signal levels caused by clicking on a button with a mouse during the program animation or while single-stepping through the program. These allow you to simulate the action of closing a switch, or pulsing a pin.
- *Sequential data* – applied to pins, registers, or bits in registers from a predefined list.

50 PIC Projects: A Practical Approach

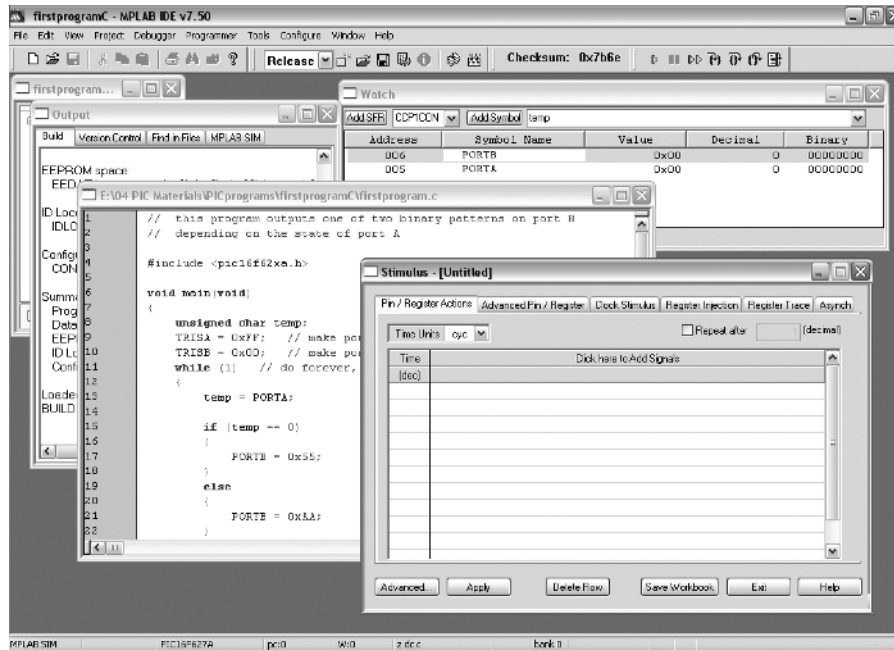


Figure 1.31 Starting Stimulus in MPLAB

© Microchip Technology Inc. Reproduced with permission.

- *Cyclic stimulus* – repeating waveform for a predetermined length of time or continuously.

In this chapter we will show how to set up and use the first two types of stimulus, leaving the cyclic type stimulus for later chapters. We will only demonstrate the basic configuration for those two stimulus types. It is left to the reader to investigate this feature of the MPLAB SIM tool further.

A manually triggered stimulus is an asynchronous type of stimulus, so to configure it we need to select the Asynch tab from the Stimulus window. By clicking on the first line of the Pin/SFR column in the table under this tab we can select the pin or register to be affected by the stimulus. In our example we use PORTA as an input port, select RA0 – first pin of PORTA, to be manually set during the animation of the program. By clicking under the Action field we can select one of five possible types of stimulus for RA0 – ‘Set High’, ‘Set Low’, ‘Toggle’, ‘Pulse High’ or ‘Pulse Low’. Select – ‘Set High’. By pressing the first button in the same row (under the Fire label in the same table) during the program animation we will be able to set RA0 high. Without further action RA0 would remain high until the end of the animation. It might

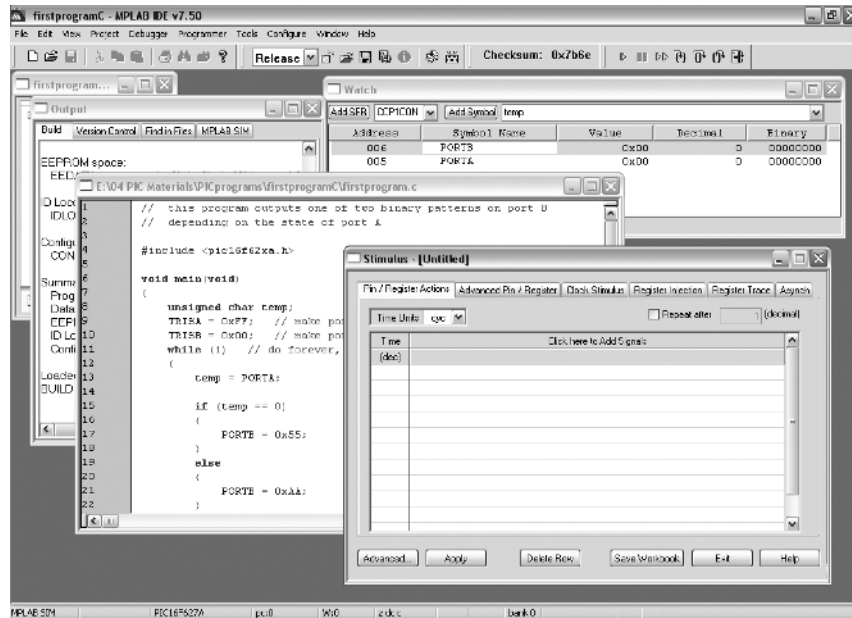



Figure 1.32 Configuration of asynchronous stimulus in MPLAB

© Microchip Technology Inc. Reproduced with permission.

be a good idea to add a second action in order to set this pin low during the same testing sequence. To do this, go to the second row of this table, select RA0 again but choose 'Set Low' as the action in this case. This setup is shown in Figure 1.32. Fire buttons used to set or reset pin RA0 during the program animation are highlighted in this figure.

All that needs to be done now is to animate the program by pressing the  button on the MPLAB shortcut bar or by selecting the Animate option from the DEBUGGER menu. The program will run in the slow mode so its action can be clearly observed. Once the program enters the infinite loop it circulates between lines 11 and 17 – since all of the PORTA pins are initially 0, the condition following the 'if' statement is true and the program continuously sends 55h to PORTB. This can be checked in the Watch window where PORTB = 0x55 while PORTA = 0x00. Firing the first event – setting RA0 high – will change this sequence. PORTA will change its state to 0x01 and the program will send AAh to PORTB instead of 55h. This can again be checked in the Watch window. Firing the second event during the same animation cycle will set RA0 low. The result in the Watch window is: PORTB = 0x55 and PORTA = 0x00 again. More pins of PORTA and different actions can be added to this basic configuration to make it more interesting and versatile.

52 PIC Projects: A Practical Approach

The sequential stimulus can be configured by changing the Pin/Register Actions tab in the Stimulus window. Here, we first need to select between different time units (*cyc* – instruction cycles, *h:m:s* – hours, minutes, seconds for longer simulations, *ms* – milliseconds, μs – microseconds or *ns* – nanoseconds for shorter simulations). Select μs to define a sequence in microseconds. We will define a time interval consisting of four subintervals – each 100 μs long. To do this type 0, 100, 200 and 300 in the first column of the table (Time/dec). To select the register to be affected by the stimulus action click on the ‘Click here to Add Signals’ heading in the table. From the list of registers, select PORTA and Add to move it from the list of Available Signals to the list of Selected Signal(s). No other register needs to be selected for this simple demonstration. After closing the Add/remove Pin/Registers dialogue box a new column with the heading PORTA is added to the table. The first four rows under this heading should be filled with hexadecimal values. Those values will be taken by PORTA at 0, 100, 200 and 300 microseconds after the start of the animation. Fill in the PORTA fields for each of those times with the appropriate values. We have used FF, 00, 10 and 00. The final setup is shown in Figure 1.33. A

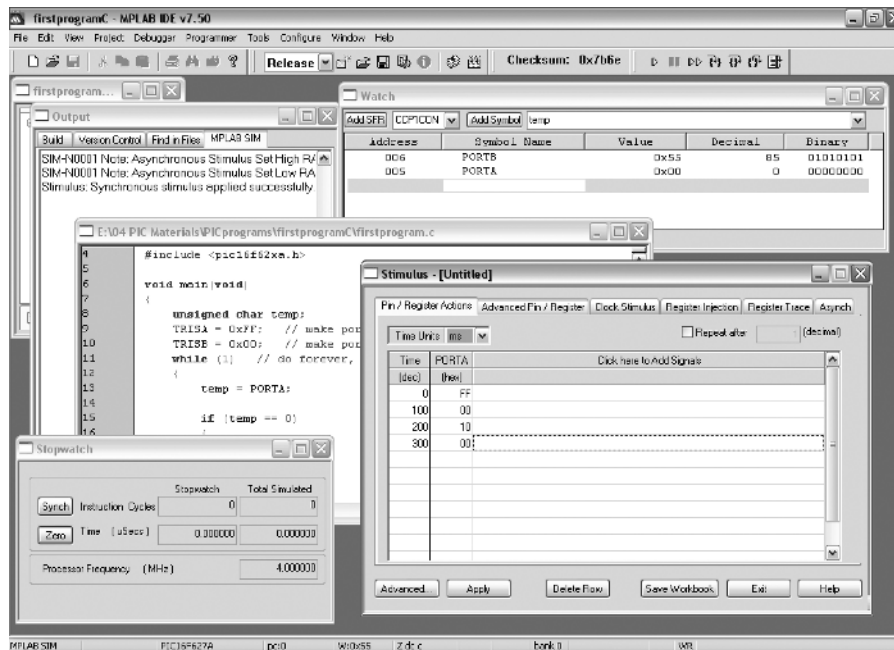


Figure 1.33 Configuration of cyclic stimulus in MPLAB

© Microchip Technology Inc. Reproduced with permission.

Stopwatch feature is also activated to help in following the execution time of the program. Before starting the animation of the program apply this sequence by clicking on the Apply button in the Stimulus window.

Start the animation of the program and observe the changes in the Watch window at specified times. PORTA is changing the value according to specification in the Stimulus window and PORTB is following these changes by switching between two patterns after $100\ \mu\text{s}$, $200\ \mu\text{s}$ and $300\ \mu\text{s}$. The only slightly confusing behaviour might be the value of PORTA during the first $0\text{--}100\ \mu\text{s}$ time interval. Although we set it to FFh, the Watch window shows $\text{PORTA} = 0 \times 3\text{F}$! If you have not noticed it, try resetting the program and animating it again. Can you explain this anomaly? If you can, it is time to start with some simple PIC projects in Chapter 2.

