Please replace the following pages in the book.

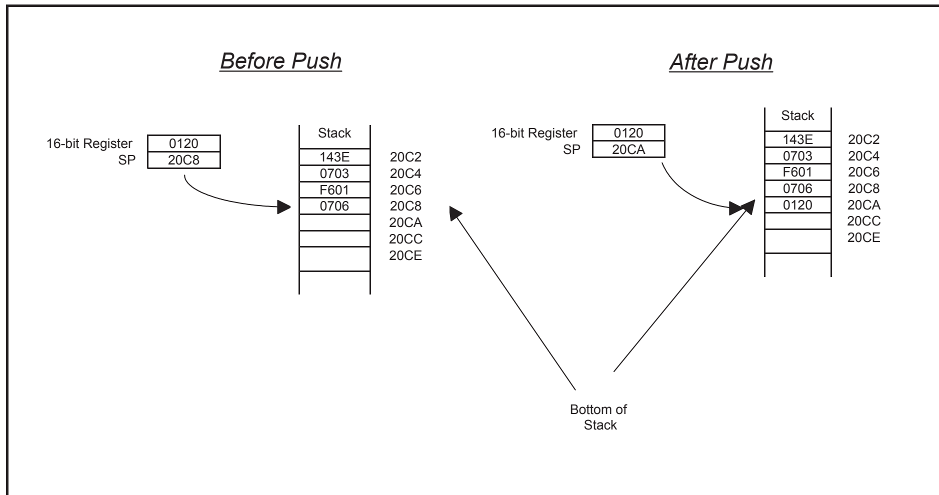


FIGURE 2.12 PUSH operation when accessing a stack from the bottom

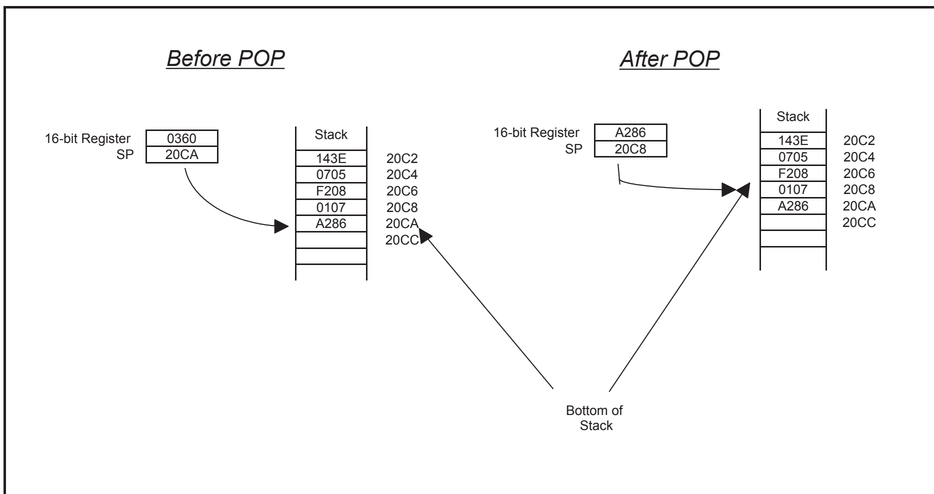


FIGURE 2.13 POP operation when accessing a stack from the bottom

Note that the stack is a LIFO (last in, first out) memory. As mentioned earlier, a stack is typically used during subroutine CALLs. The CPU automatically PUSHes the return address onto a stack after executing a subroutine CALL instruction in the main program. After executing a RETURN from a subroutine instruction (placed by the programmer as the last instruction of the subroutine), the CPU automatically POPs the return address from the stack (previously PUSHed) and then returns control to the main program. Note that the PIC18F accesses the stack from the top. This means that the stack pointer in the PIC18F holds the address of the bottom of the stack. Hence, in the PIC18F, the stack pointer is incremented after a PUSH, and decremented after a POP.

2.3.2 Control Unit

The main purpose of the control unit is to read and decode instructions from the program memory. To execute an instruction, the control unit steps through the appropriate blocks of the ALU based on the op-codes contained in the instruction register. The

Questions and Problems

2.1 What is the difference between a single-chip microcomputer and a microcontroller?

2.2 What is meant by an 8-bit microcontroller? Name one commercially available 8-bit microcontroller.

2.3 What is the difference between
(a) a program counter and a memory address register?
(b) an accumulator and an instruction register?
(c) a general-purpose register-based CPU and an accumulator-based CPU?

2.4 Assuming signed numbers, find the sign, carry, zero, and overflow flags of
(a) $09_{16} + 17_{16}$
(b) $A5_{16} - A5_{16}$
(c) $71_{16} - A9_{16}$
(d) $6E_{16} + 3A_{16}$
(e) $7E_{16} + 7E_{16}$

2.5 What is the difference between PUSH and POP operations in the stack?

2.6 Suppose that an 8-bit microcontroller has a 16-bit stack pointer and uses a 16-bit register to access the stack from the top. Assume that initially the stack pointer and the 16-bit register contain $20C0_{16}$ and 0205_{16}, respectively. After the PUSH operation
(a) What are the contents of the stack pointer?
(b) What are the contents of memory locations $20BE_{16}$ and $20BF_{16}$?

2.7 What is the main purpose of the hardware reset pin on the microcontroller chip?

2.8 How many bits are needed to access a 4 MB data memory? What is the hexadecimal value of the last address in this memory?

2.9 If the last address of an on-chip memory is 0x7FF, determine its size.

2.10 What is the difference between von Neumann and Harvard CPU architectures? Provide an example of a commercially available microcontroller using each type of CPU.

2.11 What is the basic difference between program execution by a conventional CPU and the PIC18F CPU?

2.12 Discuss the basic features of RISC and CISC.

2.13 Discuss briefly the purpose of the functional units (CCP, A/D, serial communication) implemented in the PIC18F.

at hand. A second advantage is that a program written in a particular high-level language can be executed by two different microcontrollers, provided that they both understand that language. For example, a program written in C for a PIC18F microcontroller will run on a Texas Instrument's MSP 430 microcontroller because both microcontrollers have a compiler to translate the C language into their particular machine language; minor modifications are required for I/O programs.

Typical microcontrollers are also provided with a program called an "interpreter." This is provided as part of the software development package. The interpreter reads each high-level statement such as F = A + B and directs the microcontroller to perform the operations required to execute the statement. The interpreter converts each statement into machine language codes but does not convert the entire program into machine language codes prior to execution. Hence, it does not generate an object program. Therefore, an interpreter is a program that executes a set of machine language instructions in response to each high-level statement in order to carry out the function. A compiler, however, converts each statement into a set of machine language instructions and also produces an object program that is stored in memory. This object program must then be executed by the CPU to perform the required task in the high-level program.

In summary, an interpreter executes each statement as it proceeds, without generating an object code, whereas a compiler converts a high-level program into an object program that is stored in memory. This program is then executed. Compilers for microprocessors normally provide inefficient machine codes because of the general guidelines that must be followed for designing them. However, modern C compilers for microcontrollers generate very tight and efficent codes. Also, C is a high-level language that includes input/output instructions. Hence, C is a very populer programming language with microcontrollers.

3.5 Choosing a Programming Language

Compilers used to provide inefficient machine codes because of the general guidelines that must be followed for designing them. However, modern C compilers generate very tight and efficient codes. Hence, C is widely used these days. Assembly language programming, on the other hand, is important in the understanding of the internal architecture of a microcontroller, and may sometimes be useful for writing programs for real-time applications.

3.6 Flowcharts

Before an assembly language program is written for a specific operation, it is convenient to represent the program in a schematic form called a *flowchart.* A brief listing of the basic shapes used in a flowchart and their functions is given in Table 3.3.

Note that the flowchart symbols of Table 3.3 are used for writing some of the PIC18F assembly language programming examples in Chapters 6 and 7.

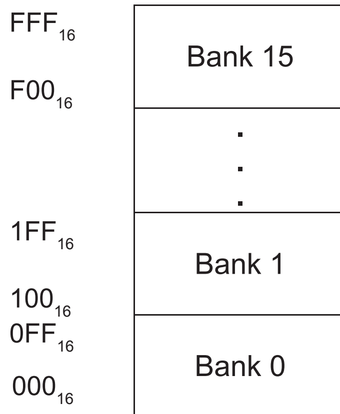


FIGURE 4.1 PIC18F data memory

of 8-bit units called *memory words*. An 8-bit unit of data is termed a *byte*. Therefore, for an 8-bit microcontroller, *memory word* and *memory byte* mean the same thing. For 16-bit microcontrollers, a word contains two bytes (16 bits). A memory word is identified in the memory by an address. For example, the PIC18F 4321 is an 8-bit microcontroller, and can directly address a maximum of two megabytes (2^{21}) of program memory space. The data memory address, on the other hand, is 12 bits wide. Hence, the PIC18F family members can directly address data memory of up to 4 Kbytes (2^{12}). This provides a maximum of 2^{12} = 4096 bytes of data memory addresses, ranging from 000 to FFF in hexadecimal.

An important characteristic of a memory is whether it is volatile or nonvolatile. The contents of a volatile memory are lost if the power is turned off. On the other hand, a nonvolatile memory retains its contents after power is switched off. ROM is a typical example of nonvolatile memory. RAM is a volatile memory unless backed up by batteries.

Large areas of data memory require an efficient addressing scheme to make rapid access to any address possible. Ideally, this means that an entire address does not need to be provided for each read or write operation. For PIC18F, this is accomplished with a RAM banking scheme. This divides the memory space into 16 contiguous banks (bank 0 through 15) of 256 bytes. Depending on the instruction, each location can be addressed directly by its full 12-bit address, or an 8-bit low-order address and a 4-bit bank pointer.

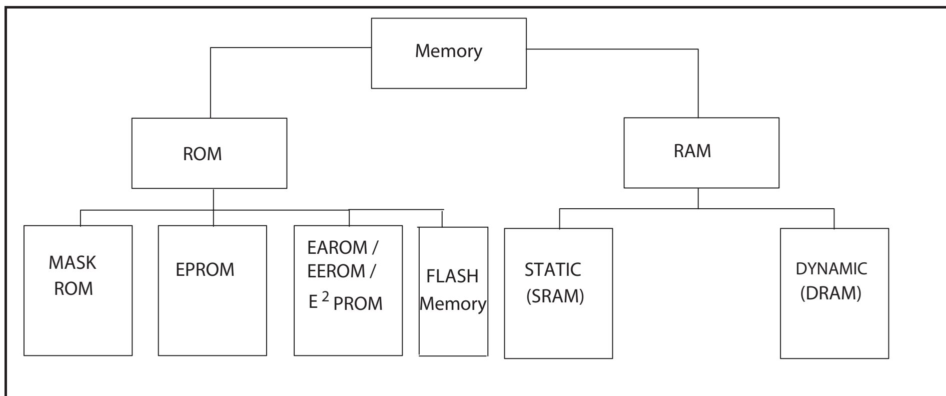


FIGURE 4.2 Summary of available semiconductor memories for microcontroller systems

program as follows:

```
ORG      0x000008  ; Starting address of the service routine
MOVLW    0x02      ; Move 1 to bit 1 of WREG (accumulator) register
MOVWF    PORTD     ; Turn the LED ON
RETFIE             ; Restore PC and SR, and return from interrupt
```

In this service routine, using the MOVLW and MOVWF instructions, the microcontroller turns the LED ON. The return instruction RETFIE, at the end of the service routine loads the address BEGIN and the previous status register contents from the stack, and loads the program counter and status register with them. The microcontroller executes the "MOVWF 0x30" instruction at the address BEGIN and continues with the main program. The basic characteristics of interrupt I/O have been discussed so far. The main features of interrupt I/O provided with a typical microcontroller are discussed next.

Interrupt Types There are typically two types of interrupts: external interrupts and internal interrupts. *External interrupts* are initiated through a microcontroller's interrupt pins by external devices such as the comparator in the previous example. External interrupts can be divided further into two types: maskable and nonmaskable. The nonmaskable interrupt cannot be enabled or disabled by instructions, whereas a microcontroller's instruction set typically contains instructions to enable or disable maskable interrupt. A nonmaskable interrupt has a higher priority than a maskable interrupt. If maskable and nonmaskable interrupts are activated at the same time, the processor will service the nonmaskable interrupt first.

A nonmaskable interrupt is typically used as a power failure interrupt. Microcontrollers normally use +5 V dc, which is transformed from 110 V ac. If the power falls below 90 V ac, the DC voltage of +5 V cannot be maintained. However, it will take a few milliseconds before the ac power drops below 90 V ac. In these few milliseconds, the power-failure-sensing circuitry can interrupt the processor. The interrupt service routine can be written to store critical data in nonvolatile memory such as battery-backed CMOS RAM, and the interrupted program can continue without any loss of data when the power returns.

Internal interrupts are activated internally by conditions such as completion of analog-to-digital conversion, timer interrupt, or interrupt due to serial I/O. Internal interrupts are handled in the same way as external interrupts. The user writes a service routine to take appropriate action to handle the interrupt. Some microcontrollers include software interrupt instructions. When one of these instructions is executed, the microcontroller is interrupted and serviced similarly to external or internal interrupts.

Some microcontrollers such as the Motorola/Freescale HC11/HC12 provide both external (maskable and nonmaskable) and internal (exceptional conditions and software instructions). The PIC18F provides external maskable interrupts only. The PIC18F does not have any external nonmaskable interrupts. However, the PIC18F provides internal interrupts. The internal interrupts are activated internally by conditions such as timer interrupts, completion of analog-to-digital conversion, and serial I/O.

Interrupt Address Vector The technique used to find the starting address of

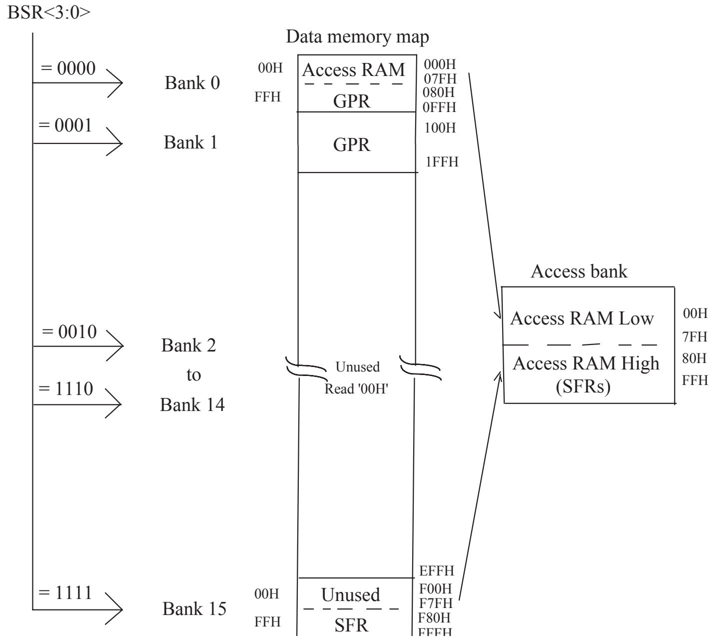


FIGURE 5.7 PIC18F4321 data memory map

$$
\begin{array}{lrl}
 & 68_{16} = & 0\,1\,1\,0\,1\,0\,0\,0 \qquad 68_{16} \\
\text{Add 2's complement of } 06_{16} & = & \underline{1\,1\,1\,1\,1\,0\,1\,0} \qquad \underline{-\ 06_{16}} \\
 & 1 & 0\,1\,1\,0\,0\,0\,1\,0 \qquad 62_{16}
\end{array}
$$

$C_f = 1$ (final carry), $C_p = 1$, $DC = 0$

In the above, C (borrow) = one’s complement of $C_f = 0$, DC = 0 (no carry from bit 3 to bit 4), Z = 0 (nonzero result), OV = $C_f \oplus C_p = 1 \oplus 1 = 0$ (meaning correct result), and N = 0 (most significant bit of the result is 0 indicating positive number). Note that while obtaining two’s complement subtraction using paper and pencil, the correct borrow is always the one’s complement of the borrow obtained analytically. Hence, microcontrollers perform one’s complement operation on the borrow in order to reflect the correct borrow which will be useful in multiprecision subtraction.

5.3 PIC18F Memory Organization

Two types of memories are normally utilized in the PIC18F. They are flash memory and SRAM. The flash memory is used to store programs. The SRAM, on the other hand, contains data. Some versions of the PIC18F family contain EEPROM along with SRAM to hold data. Note that, SRAM is a volatile read/write memory. The EEPROM, on the other hand, is a nonvolatile memory.

The EEPROM is separate from the data SRAM and program flash memory. The EEPROM is used for long-term storage of critical data. The EEPROM is normally used as a read-mostly memory since its read time is faster than write times. It is not directly mapped

in either the register file or program memory space but is indirectly addressed through the special function registers (SFRs). One of the main advantage of including EEPROM in the PIC18F is that all critical data stored in the EEPROM can be protected from reading or writing by other users. This can be accomplished by programming appropriate bits in the corresponding SFR. Note that the PIC18F4321 contains 256 bytes of EEPROM.

The data memory in PIC18F devices is implemented as static RAM. Each register in the data memory has a 12-bit address, allowing up to 4096 bytes (2^{12}) of data memory. The memory space is divided into as many as 16 banks that contain 256 bytes each.

5.3.1 PIC18F Program Memory Map

Figure 5.6 shows the program memory map for the PIC18F4321. Program memory is implemented in flash memory in the PIC18F.

The PIC18F4321 contains 8 Kbytes (13-bit address) of on-chip flash memory, and can store up to 4096 single 16-bit word instructions. Note that most PIC18F instructions are 16 bits wide.

As mentioned before, the program counter (PC) contains the address of the instruction to be fetched for execution. The PC is 21 bits wide. The PC addresses bytes in the program memory. To prevent the PC from becoming misaligned with 16- or 32-bit-wide instructions, the least significant bit of PC is fixed to a value of ‘0’. This is because the address is an even number for 16-bit or 32-bit instructions. The PC increments by 2 or 4 to address sequential 16- or 32-bit-wide instructions in the program memory.

The stack operates as a 31 RAM locations, each location being 21-bit wide and is addressed by a 5-bit stack pointer, STKPTR. The stack space is not part of either program or data space. The stack pointer is readable and writeable. The stack is accessed from the bottom. Data can also be pushed to, or popped from, the stack.

The reset vector address is located at 000000H, where H stands for hex. There are two interrupts. These are high-priority interrupt and low-priority interrupt. The starting address for the high priority service routine is 000008H. There are 16 bytes available to the user for writing the high-priority service routine. The starting address for the low priority service routine is 000018H. There is no specific size for the low-priority service routine. Reset and interrupts will be discussed in more detail later in this book. In the PIC18F4321, the user program should be written after the low-priority service routine to a maximum allowable address of 001FFFH. Addresses 002000H through 1FFFFFH are not implemented, and are read as zeros.

5.3.2 PIC18F Data Memory Map

Figure 5.7 shows the data memory organization for the PIC18F4321. As mentioned before, the PIC18F data memory is implemented in SRAM. The PIC18F can have a data memory of up to 4096 (2^{12}) bytes; 12-bit address is needed to address each location. However, the PIC18F4321 implements two banks with a total of 512 bytes of data SRAM.

The data memory contains SFRs and general purpose registers (GPRs). The GPRs are typically used for storing data and as scratch pad registers during programming. The SFRs, on the other hand, are dedicated registers. These registers are used for control and status of the controller and peripheral functions such as registers associated with I/O ports and interrupts, timers, ADC (analog-to-digital converter), and serial I/O. An unimplemented location will be read as 0’s. The instruction set and architecture allow operations across all banks. The entire data memory may be accessed by direct or indirect addressing modes.

```
              MOVLW   D'100'      ; Move 100 decimal into WREG
              MOVWF   0x20        ; Initialize counter reg (0x20) with 100 decimal
              LFSR    0,0x0044    ; Initialize pointer FSR0 with starting address 044H
REPEAT        CLRF    POSTDEC0    ; Clear a location to 0 and  decrement FSR0 by 1
              DECF    0x20,F      ; Decrement counter by 1
              BNZ     REPEAT      ; Branch to  REPEAT if  Zero flag = 0; otherwise,
                                  ; go to the next instruction
```

FIGURE 5.12 Instruction sequence for illustrating posdecrement mode

These SFRs use the contents of FSR0 through FSR2 to achieve the four submodes. The submodes can be used with various PIC18F instructions. The SFRs are utilized by the submodes as follows:

- Indirect with postincrement mode uses POSTINC0 through POSTINC2 registers. POSTINC0 is associated with FSR0, POSTINC1 with FSR1 and, POSTINC2 with FSR2.
- Indirect with postdecrement mode uses POSTDEC0 through POSTDEC2. POSDEC0 is associated with FSR0, POSDEC1 with FSR1, and POSDEC2 with FSR2.
- Indirect with preincrement mode uses PREINC0 through PREINC2 registers. PREINC0 is associated with FSR0, PREINC1 with FSR1, and PREINC2 with FSR2.
- Indirect with 8-bit indexed mode uses PLUSW0 through PLUSW2. PLUSW0 is associated with FSR0, PLUSW1 with FSR1, and PLUSW2 with FSR2.

Indirect with postincrement mode reads the contents of the FSR specified in the instruction, using the low 12-bit value as the address for the operation to be performed. The specified FSR is then incremented by 1 to point to the next address. The special function register POSTINC is used for this purpose.

As an example, consider CLRF POSTINC0. Prior to execution of this instruction, suppose that the 16-bit contents of FSR0 are 0030H, and the 8-bit contents of the data memory location addressed by 12-bit address 030H are 84H. After execution of the instruction CLRF POSTINC0, the contents of address 030H will be cleared to 00H, and the contents of FSR0 will be incremented by 1 to hold 0031H. This may be used as a pointer to the next data. This is depicted in Figure 5.9. Note that all addresses and data are chosen arbitrarily.

The postincrement mode is typically used with memory arrays stored from LOW to HIGH memory locations. For example, to clear 20 bytes starting at data memory address 030H and above, the instruction sequence in Figure 5.10 can be used. In Figure 5.10, MOVLW D'20' moves 20 decimal into WREG while MOVWF 0x10 moves the contents of WREG (20 decimal) into address 010H. This will initialize the counter register with 20 decimal. The LFSR 0,0x0030 loads FSR0 with 0030H; 030H is the address of the first byte in the array to be cleared to 0. The CLRF POSTINC0 clears the contents of the data memory addressed by FSR0 to 0 and increments FS0 by 1 to hold 031H. This is because POSTINC0 is associated with FSR0. Since the 16-bit contents of FSR0 are 0030H, contents addressed by the the low 12 bits (030H) of FSR0 are cleared to 0. The DECF 0x10,F decrements the contents of data register 010H by one and then places the result in the data register 010H. After the first pass, data register 010H will contain 19 decimal.

The BNZ REPEAT instruction checks if Z flag in the flag register is 0. Note that Z = 0 since the contents of counter are nonzero (19) after execution of DECF. The program branches to label REPEAT, and the loop will be performed 20 times clearing 20 bytes of the array to 0's.

TABLE 6.2 General format for instructions

Byte-oriented file register operations

15 10	9	8	7 0	**Example instruction**
OPCODE	d	a	f = 8-bit File register address	ADDWF f, d, a

d = 0 for result destination to be WREG register

d = 1 for result destination to be file register (f)

a = 0 to force access bank, a = 1 for BSR to select bank

f = 8-bit file register address

Bit-oriented file register operatios

15 11	9	8	7 0	
OPCODE	b (BIT #)	a	f	BSF f, b, a

b = 3-bit position of bit in file register (f)
a = 0 to force access bank

a = 1 for BSR to select bank

f = 8-bit file register address

Literal operations

15 8	7 0	
OPCODE	k (literal)	MOVLW 0x2A

k = 8-bit immediate value

Control operations

Branch operation

15 8	7 0	
OPCODE	n<7:0> (offset) (Relative mode)	BZ n

the WREG register. When compared with the literal-oriented format of Table 6.2, k = 00101010 (0x2A). Since the 8-bit opcode for MOVLW is 00001110, the binary 16-bit code for MOVLW 0x2A is 0000111000101010 (0E2AH).

An example of the control instructions includes conditional branch instructions with the following operand:

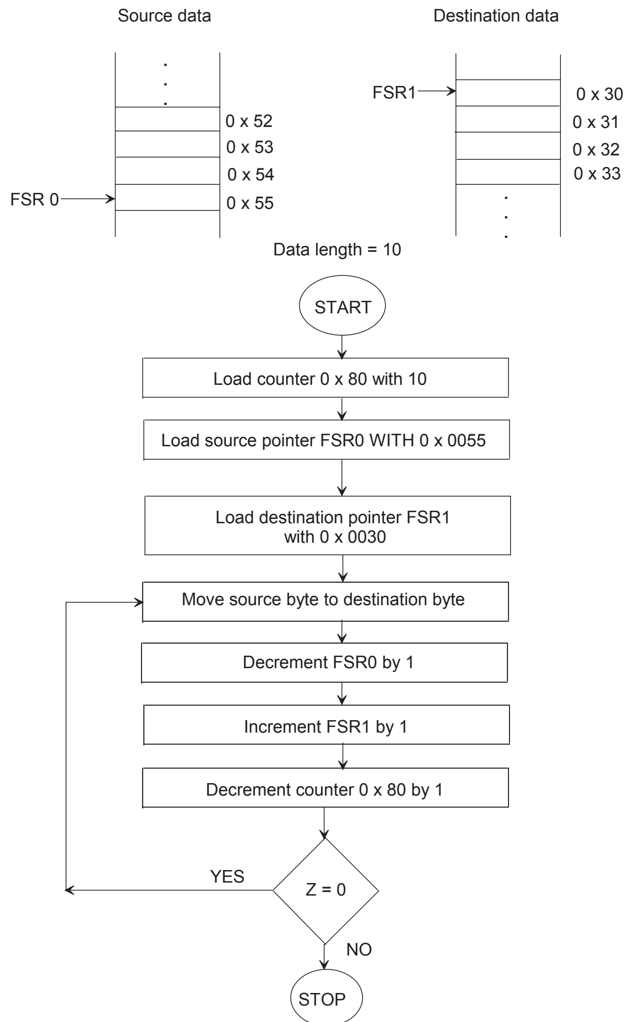
- a signed 8-bit offset (specified by '*n*')

As an example, consider BZ 0x04 where 04 (hex) is the offset (*n*). This instruction branches to an address (PC+2+ 2 x 4) if Z = 1; otherwise, the next instruction is executed. Since the 8-bit opcode for BZ is 11100000_2 and *n* = 00000100_2 (0x04), the binary 16-bit code for BZ 0x04 is 1110000000000100_2 (E004H).

Most PIC18F instructions are a single word; only four instructions are double-word

Solution

(a) The flowchart along with data memory layout is provided below:



(b) The flowchart can be converted to PIC18F assembly language program as follows:

```
        INCLUDE   <P18F4321.INC>
COUNTER EQU       0x80
        ORG       0x100
        MOVLW     D'10'          ; Move 10 decimal into WREG
        MOVWF     COUNTER        ; Initialize counter reg (0x80) with 10
                                 ; decimal
        LFSR      0, 0x0055      ; Initialize FSR0 with source starting
                                 ; address
        LFSR      1, 0x0030      ; Initialize FSR1 with destination
                                 ; starting address
```

Solution

```
        INCLUDE<P18F4321.INC>
        ORG     0x100
        MOVLW   D'4'            ; Move  WREG with 4
        MOVWF   0x50            ; Initialize 0x50 with loop count (4)
        LFSR    0, 0x0072       ; Initialize pointer FSR0 with 0x0072
        LFSR    1, 0x0040       ; Initialize pointer FSR1 with 0x0040
        ADDLW   0x00            ; Clear carry flag
START   MOVF    POSTINC0, W     ; Move byte into WREG and update pointer
        SUBWFB  POSTINC1, F     ; Subtract [WREG] and carry   from byte; store result
                                ; in data register
        DECF    0x50, F         ; Decrement counter 0x50 by 1
        BNZ     START           ; Branch to START  if  Z ≠ 0
        SLEEP                   ; Halt
        END
```

Example 6.13 Write a PIC18F assembly language program at address 0x100 to compute ($X^2 + Y^2$) where X and Y are two 8-bit unsigned numbers stored in data registers 0x40 and 0x41, respectively. Store the 16-bit result in data registers 0x61 (high byte) and 0x60 (low byte). Assume X and Y are already loaded into data registers 0x40 and 0x41.

Solution

```
        INCLUDE   <P18F4321.INC>
        ORG       0x100
        MOVF      0x40, W         ;  Move X into WREG
        MULWF     0x40            ;  Multiply X by X; result in PRODH:PRODL
        MOVFF     PRODL, 0x50     ;  Save low byte of result in data reg 0x50
        MOVFF     PRODH, 0x51     ;  Save high byte of result in data reg 0x51
        MOVF      0x41, W         ;  Move Y into WREG
        MULWF     0x41            ;  Multiply Y by Y; result in PRODH:PRODL
        MOVFF     PRODL, 0x60     ;  Save low byte of result in data reg 0x60
        MOVFF     PRODH, 0x61     ;  Save high byte of result in data reg 0x61
        MOVF      0x50, W         ;  Move low byte of  X times X  into WREG
        ADDWF     0x60, F         ;  Add and store low byte of result in 0x60
        MOVF      0x51, W         ;  Move high byte of  X times X  into WREG
        ADDWFC    0x61, F         ;  Add high bytes with carry. Store result in 0x61
        SLEEP                     ;  Halt
        END
```

Example 6.14 Write a PIC18F assembly language program at address 0x100 to add two packed BCD bytes stored in data registers 0x20 and 0x21. Store the correct packed BCD result in WREG. Load packed BCD bytes 0x72 and 0x45 into data registers 0x20 and 0x21, respectively, using PIC18F instructions. Note that data are arbitrarily chosen.

and then perform the following operations on the contents of data register 0x50:

- Set bits 0 and 3 to one without changing other bits in data register 0x50.
- Clear bit 5 to zero without changing other bits in data register 0x50.
- One's complement bit 7 without changing other bits in data register 0x50.

Use only "logic", and "rotate" instructions. Do not use any multiplication instruction. Assume data are already in data register 0x50. Store result in WREG. Assume that a '1' is not shifted out of the most significant bit each time after rotating to the left.

Solution

```
        INCLUDE <P18F4321.INC>
        ORG     0x100
        BCF     STATUS, C
        RLCF    0x50, W   ; Unsigned multiply [0x50] by 2
        BCF     STATUS, C
        RLCF    0x50, W   ; Unsigned multiply [0x50] by 4; result in W
        IORLW   0x09      ; Set bits 0 and 3 in WREG to one's
        ANDLW   0xDF      ; Clear bit 5 in WREG to zero
        XORLW   0x80      ; One's complement bit 7 in WREG
FINISH  GOTO    FINISH    ; Stop
        END
```

As mentioned before, FINISH GOTO FINISH (unconditionally jumping to the same location) and the instruction SLEEP are equivalent to HALT instruction in other processors. Either can be used in the PIC18F as HALT in the assembly language program.

Example 6.19 Write a PIC18F assembly language program at address 0x100 to check whether an 8-bit signed number (x) is positive or negative. If the number is positive, then compute 16-bit value $y1 = x^2$ and store result in PRODH:PRODL. If the number is negative, then compute the 8-bit value $y2 = 2x$. Store result in WREG. Do not use any logic instructions. Assume that the 8-bit number x is already loaded in data register 0x60.

Solution

```
         INCLUDE  <P18F4321.INC>
         ORG      0x100
         MOVF     0x60, W      ; Move x into WREG
         MOVFF    0x60, 0x70   ; Save x in 0x70
         RLCF     0x60, F      ; Rotate sign bit to carry to check whether 0 or 1
         BN       NEGATIVE     ; Branch if N = 1
         MULWF    0x60         ; Compute y1 and store in PRODH:PRODL
         GOTO     FINISH       ; Jump to FINISH
NEGATIVE ADDWF    0x70, W      ; Compute y2 by adding x to itself
FINISH   GOTO     FINISH
         END
```

Example 6.20 Write a PIC18F assembly language program at address 0x100 that will multiply an 8-bit unsigned number in data register 0x50 by 4 to provide an 8-bit product, and then perform the following operations on the contents of data register 0x50 :

- Set bits 0 and 3 to one without changing other bits in data register 0x50.
- Clear bit 5 to zero without changing other bits in data register 0x50.
- One's complement bit 7 without changing other bits in data register 0x50.

Use only "rotate (RLCF instruction only)" and "bit manipulation" instructions. Do not use any multiplication instruction. Assume data are already in data register 0x50. Store result in WREG. Assume that a '1' is not shifted out of the most significant bit each time after rotating to the left.

This example is a repeat of Example 6.18, but uses "bit manipulation instructions" instead of "logic instructions."

Solution

```
              INCLUDE      <P18F4321.INC>
              ORG          0x100
              BCF          STATUS, C
              RLCF         0x50, F  ; Unsigned multiply [0x50] by 2
              BCF          STATUS, C
              RLCF         0x50, F  ; Unsigned multiply [0x50] by 4; result in F
              BSF          0x50, 0  ; Set bit 0 in [0x50] to one
              BSF          0x50, 3  ; Set bit 3 in [0x50] to one
              BCF          0x50, 5  ; Clear bit 5  in [0x50] to zero
              BTG          0x50,7   ; One's complement  bit 7 in 0x50
              MOVF         0x50, W ; Store result in WREG
FINISH        GOTO         FINISH  ; Stop
              END
```

Example 6.21 Write a PIC18F assembly language program at address 0x100 that will perform $5 \times X + 6 \times Y + [Y/2] \rightarrow [\,0x71][0x70]$, where X is an unsigned 8-bit number stored in data register 0x40 and Y is a 4-bit unsigned number stored in the upper 4 bits of data register 0x50. Discard the remainder of $Y/2$. Save the 16-bit result in 0x71 (upper byte) and in 0x70 (lower byte).

(a) Flowchart the problem.

(b) Convert the flowchart to PIC18F assembly language program starting at address 0x100.

6.8 Write PIC18F instruction sequence that is equivalent to the following C code:

```
if (p <= q)
   p = p + 5;
else
   p = 10;
```

6.9 What is the content of WREG after execution of the following PIC18F instruction sequence?

```
MOVLW   0x33
ADDLW   0x77
DAW
```

6.10 Find two ways to clear [WREG] to 0 using
(a) a single PIC18F instruction
(b) two PIC18F instructions

6.11 Using a single PIC18F instruction, clear the carry flag without changing the contents of any data registers, WREG, or other status flags.

6.12 Write the machine code for the following PIC18F instruction sequence:

```
        ORG     0x200
HERE    BRA     HERE
```

6.13 Write a PIC18F assembly language program at address 0x100 to add two 8-bit numbers (N1 and N2). Data register 0x20 contains N1. The low four bits of N2 are stored in the upper nibble of data register 0x21 while the high four bits of N2 are stored in the lower nibble of data register 0x21. Store result in data register 0x30.

6.14 Write a PIC18F assembly language program at address 0x100 to add two 24-bit data items in memory, as shown in Figure P6.14. Store the result pointed to by 0x50. The operation with sample data is given by

```
        F1  91  B5
PLUS    07  A2  04
---------------------------------
        F9  33  B9
```

Assume that the data pointers and the data are already initialized.

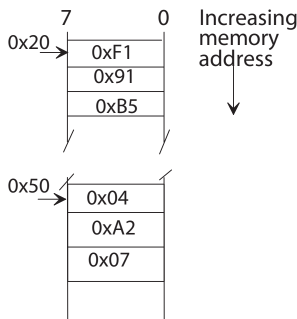


FIGURE P6.14

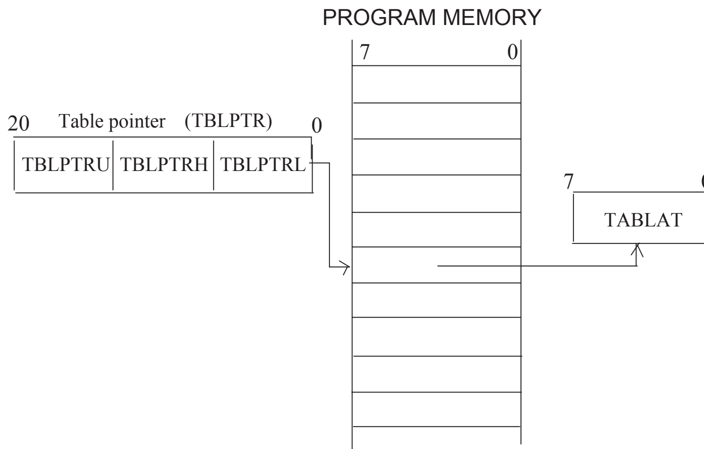


FIGURE 7.1 Table read operation (instruction TBLRD*)

The table read/write instructions will now be explained in the following using numerical examples with similar data for each instruction:

- Consider TBLRD* instruction.

  Prior to execution of TBLRD*, [TBLPTR] = 0x02318, [TABLAT] = 0x24, and [0x002318] = 0xF2.

  After execution of TBLRD*, [TABLAT] = 0xF2, [TBLPTR] = 0x002318 (unchanged), and [0x002318] = 0xF2 (unchanged).

- Consider TBLRD*+ instruction.

  Prior to execution of TBLRD* +, [TBLPTR] = 0x002318, [TABLAT] = 0x24, and [0x002318] = 0xF2.

  After execution of TBLRD*+, [TABLAT] = 0xF2, [TBLPTR] = 0x002319, and [0x002318] = 0xF2 (unchanged).

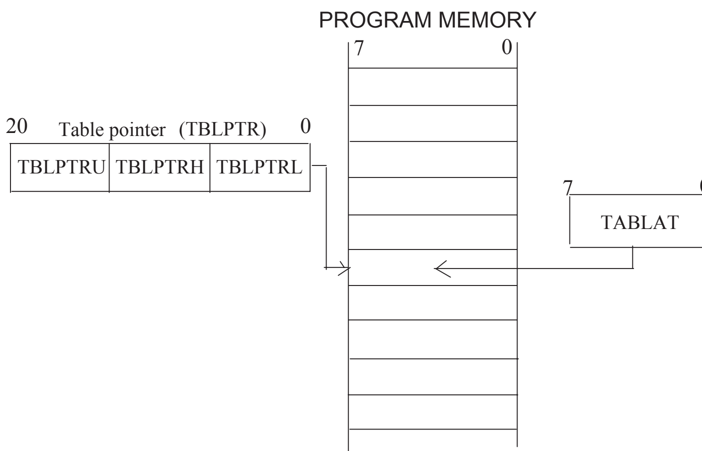


FIGURE 7.2 Table write operation (instruction TBLWT*)

TABLE 7.4 PIC18F subroutine instructions

Instruction	Operation
CALL k, s	Call the subroutine at address k within the two megabytes of program memory. First, return address (PC + 4) is pushed onto the return stack. If ‘s’ = 1, the WREG, STATUS, and BSR registers are also pushed into their respective shadow registers (internal to the CPU), WS, STATUSS and BSRS. If ‘s’ = 0, these registers are unaffected (default). Then, the value ‘k’ is loaded into PC.
POP	Discards top of stack pointed to by SP and decrements PC by 1.
PUSH	PUSHes or writes the PC onto the stack, and increments PC by 1.
RCALL n	Subroutine CALL with relative mode.
RETFIE	Returns from interrupt. Stack is popped and top-of-stack (TOS) is loaded into the PC. Interrupts are enabled by setting either the high or low priority global interrupt enable bit. This instruction is normally used at the end of an interrupt service routine.
RETLW k	WREG is loaded with the eight-bit literal ‘k’. The program counter is loaded from the top of the hardware stack (the return address).
RETURN s	Returns from subroutine. The stack is popped and the top of the stack (TOS) is loaded into the program counter. If ‘s’= 1, the contents of the shadow registers, WS, STATUSS, and BSRS, are loaded into their corresponding registers, WREG, STATUS, and BSR. If ‘s’ = 0, these registers are not affected (default).

- CALL and RETURN instructions are executed in two cycles.
- POP and PUSH are executed in one cycle.
- The size of each instruction except CALL is one word; the size of the CALL instruction is two words.

if needed for storing local variables. The “hardware stack” and “software stack” will be discussed in the next section in more detail.

- POP instruction reads (pops) the TOS (top of stack) value from the return stack and discards it; [STKPTR] is decremented by 1. The TOS value becomes the previous value that was pushed onto the return stack.

  As an example, consider the POP instruction with numerical data in the following:

  Prior to execution of the POP, [STKPTR] = 0x15, [0x15] = TOS (top of stack) = 0x000502, stack (1 level down), and [0x14] = 0x002418.

  After execution of the POP, [STKPTR] = 0x14, TOS, and [0x14] = 0x002418.

  Note that previous TOS (0x000502) is discarded, and previous stack (1 level down) is the current TOS. Figures 7.3 (a) and (b) depict this.

- PUSH writes (pushes) PC+2 onto the top of the return stack; [STKPTR] is incremented

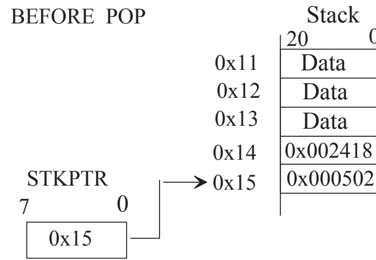


FIGURE 7.3 (a) PIC18F hardware stack with arbitrary data before execution of POP instruction

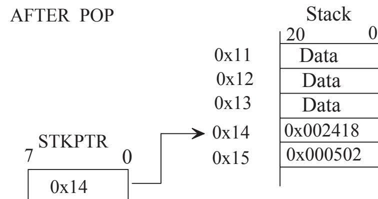


FIGURE 7.3 (b) PIC18F hardware stack with arbitrary data after execution of POP instruction; address 0x65 is assumed to be free

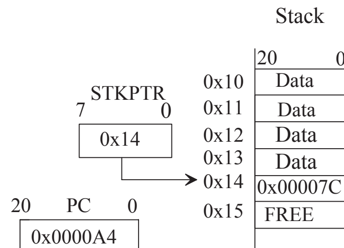


FIGURE 7.4 (a) PIC18F hardware stack with arbitrary data before execution of PUSH instruction

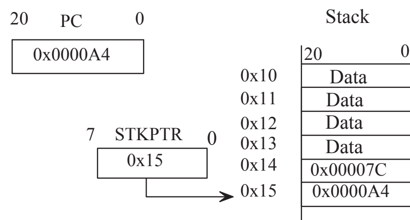


FIGURE 7.4 (b) PIC18F hardware stack with arbitrary data after execution of PUSH instruction

by 1. The previous TOS value is pushed down on the stack.

As an example, consider the PUSH instruction with numerical data in the following: Prior to execution of PUSH, [STKPTR] = 0x14, [0x14] = TOS (top of stack) = 0x00007C, and [PC+ 2] = 0x0000A4; that is, the PUSH instruction is stored at address 0x0000A2. After execution of the PUSH, [STKPTR] = 0x15, [0x15] = TOS = 0x0000A4, previous TOS (one level down), and [0x14] = 0x00007C.

Figures 7.4 (a) and (b) depict this.

- The "CALL k, s" with s = 0 (or CALL k) instruction is the simplest way of CALLing a subroutine; s = 0 is the default case. As an example, the CALL START instruction automatically pushes the current contents of the PC onto the stack, and loads PC with the label called START. Note that address START contains the starting address of the subroutine. The "RETURN s" instruction with s = 0 (or RETURN since s = 0 is default) pops the return address (PC pushed onto the stack by the CALL START instruction) from TOS, and loads PC with this address. Thus, control is returned to the main program, and program execution continues with the instruction next to the CALL START.

Consider the following PIC18F program segment:

Main Program		*Subroutine*	
—	SUB	—	; First instruction of subroutine
—		—	
—		—	
CALL SUB		—	
START —		—	
—		—	
—		RETURN	; Last instruction of the subroutine

Here, the CALL SUB instruction in the main program calls the subroutine SUB. In response to the CALL instruction, the PIC18F pushes the current PC contents (START in this case) onto the stack and loads the starting address SUB of the subroutine into PC. After the subroutine is executed, the RETURN instruction at the end of the subroutine pops the address START from the stack into PC, and program control is then returned to the main program.

7.5 PIC18F System Control Instructions

The system control instructions are associated with the operation of the PIC18F. Table 7.5 lists these instructions.

7.6 PIC18F Hardware vs. Software Stack

As mentioned in Chapter 5, the PIC18F stack is a group of thirty-one 21-bit registers to hold memory addresses. This stack (also called the "hardware stack") is part of neither data memory nor program memory. Note that the size of the stack (21-bit) is the same as the size of the PC (21-bit). The SP (stack pointer) is 5 bits wide in order to address 31 registers.

software stack. Subroutine CALLs and interrupts automatically use the hardware stack pointer (STKPTR). As mentioned before, subroutine CALLs push the current PC onto the hardware stack; RETURN pops the PC from the hardware stack.

The PIC18F accesses the system stack from the top for operations such as subroutine calls or interrupts. This means that stack operations such as subroutine calls or interrupts access the hardware stack automatically from HIGH to LOW memory. As mentioned before, the low five bits of the STKPTR are used as the stack pointer for the hardware stack. Note that the STKPTR can be initialized using PIC18F MOVE instructions. For example, in order to load 5 into the STKPTR, the following PIC18F instruction sequence can be used:

```
MOVLW    5         ; Load 5 into WREG
MOVWF    STKPTR    ; Load [WREG] into STKPTR
```

Also, the STKPTR is incremented by 1 after a push and decremented by one after a pop. As an example, suppose that a PIC18F CALL instruction such as CALL 0x000200 is executed when [PC] = 0x000500; then, after execution of the subroutine call, the PIC18F will push the current contents of PC (0x000500) onto the hardware stack, and then load PC with 0x000200 (starting address of the subroutine specified in the CALL 0x000200 instruction). This is shown in Figures 7.5 (a) and (b). The RETURN instruction at the end of the subroutine will pop 0x000500 from the hardware stack into the PC and return control to the main program. All data are arbitrarily chosen.

In the PIC18F, the software stack can be created using appropriate addressing modes. Typical PIC18F memory instructions such as the MOVFF instruction can be used to access the stack. Also, by using one of the three FSRns (FSR0–FSR2) as software stack pointers, stacks can be filled from either HIGH to LOW memory or vice versa:

Filling a stack from HIGH to LOW memory (top of the stack) is implemented with postdecrement mode for push and preincrement mode for pop.

Filling a stack from LOW to HIGH (bottom of the stack) memory is implemented with preincrement for push and postdecrement for pop.

The programmer can create software stack growing from HIGH to LOW memory addresses using FSRn as the stack pointer. To push the contents of a data register onto the software stack, the MOVFF instruction with appropriate addressing modes can be used. For example, to push contents of a data register 0x30 using FSR0 as the stack pointer, the following PIC18F instruction sequence can be used:

```
LFSR   0, 0x0070          ;  Initialize FSR0 with 0x70 to be used as the SP
MOVFF  0x30, POSTDEC0     ;  Push [0x30] to stack, decrement SP (FSR0) by 1
```

This is shown in Figures 7.6 (a) and (b). Figure 7.6 (a) shows the software stack with arbitrary data prior to execution of the above instructions. Figure 7.6 (b) shows the software stack with arbitrary data after execution of the above instructions. Note that the stack pointer FSR0 in this case is decremented by 1 after PUSH.

The 8-bit data 0xF2 can be popped from the stack into another data register 0x20, for example, using the MOVFF PREINC0, 0x20 instruction. Note that the stack pointer FSR1 in this case is incremented by 1 after POP.

Next, consider the stack growing from LOW to HIGH memory addresses in which the programmer also utilizes FSRn as the stack pointer.

To push the 8-bit contents of a data register onto the software stack, the MOVFF instruction with appropriate addressing modes can be used. For example, to push contents of a data register 0x20 using FSR1 as the stack pointer, the following PIC18F instruction sequence can be used:

```
          LFSR     1, 0x0053            ; Initialize FSR1 to 0x53 to be used as the SP
          MOVFF 0x20, PREINC0  ; Increment SP (FSR1) by 1, and Push [0x20] to stack.
```

This is shown in Figures 7.7 (a) and (b). Figure 7.7 (a) shows the software stack with arbitrary data prior to execution of the above instructions. Figure 7.7 (b) shows the software stack with arbitrary data after execution of the above instructions. Note that the stack pointer FSR1 in this case is incremented by 1 after PUSH.

The 8-bit data 0x17 can be popped from the stack into another data register 0x26, for example, using the MOVFF POSTDEC1, 0x26 instruction. Note that the stack pointer FSR1 in this case is decremented by 1 after POP.

Example 7.2 Write a PIC18F subroutine at address 0x100 to compute $Y = \sum_{i=1}^{N} X_i^2$.

Assume the X_i's are 8-bit unsigned integers, and N = 4. The numbers are stored in consecutive locations. Assume data register 0x40 points to the first element of the array for X_i's. The array elements are stored from LOW to HIGH memory addresses. The subroutine will store the 16-bit result (*Y*) in data memory registers 0x21 (high byte) and 0x20 (low byte), Also, write the main program at address 0x50 that will load data, initialize STKPTR to 0x05, FSR0 to 0x0040, call the subroutine, compute (*Y*/4) by discarding the remainder, and then stop.

Verify the correct operation of the programs using the MPLAB. Show screen shots as necessary.

Solution

```
          INCLUDE<P18F4321.INC>
          ORG       0x50              ; Starting address of the main program
; LOAD FOUR ARBITRARILY CHOSEN DATA  INTO DATA MEM ADDR .
                                      ; 0x40 TO 0x43
          MOVLW   0x7E                ; Move 0x7E into WREG
          MOVWF   0x40                ; Move 0x7E into file register 0x40
          MOVLW   0x08                ; Move 0x08 into WREG
          MOVWF   0x41                ; Move 0x08 into file register 0x41
          MOVLW   0x23                ; Move 0x23 into WREG
          MOVWF   0x42                ; Move 0x23 into file register 0x42
          MOVLW   0x30                ; Move 0x30 into WREG
          MOVWF   0x43                ; Move 0x43 into file register 0x40
; INITIALIZE STKPTR, CALL SUBROUTINE, AND DIVIDE BY 4 BY
                                      ; RIGHT SHIFT TWICE
          MOVLW   0x05                ; Move 0x05 into WREG
          MOVWF   STKPTR              ; Load 0x05 into STKPTR
          LFSR    0,0x0040            ; Load file select register 0 with register 0x0040
          CALL    SQR                 ; Call the function SQR
          BCF     STATUS, C           ; Clear the carry flag
          RRCF    0x21,F              ; Rotate right, or divide by 2
          RRCF    0x20,F
          BCF     STATUS, C           ; Clear the carry flag
          RRCF    0x21.F              ; Divide by 4
          RRCF    0x20,F
```

```
FINISH  GOTO     FINISH        ; Halt
        ORG      0x100         ; Starting address of the subroutine
SQR     MOVLW    0x00
        MOVWF    0x21          ; Clear register 0x21
        MOVWF 0x20             ; Clear register 0x20
        MOVLW    0x04
        MOVWF 0x60             ; Move 0x04 into register 0x60
BACK    MOVFF    INDF0, 0x50   ; Move the value in memory pointed to by FSR0
                               ; into register 0x50.
                               ; 0x50 is used as a holding register in data memory
                               ; It should not be confused with the starting address
                               ; 0x50 of the main program which is in program
                               ; memory of the PIC18F
        MOVF     POSTINC0, W; Move value pointed to by FSR0  into WREG, and
                               ; then increment FSR0 by 1
        MULWF    0x50          ; Multiply WREG by 0x50, or X squared
        MOVF     PRODL, W      ; Move low byte of answer to WREG
        ADDWF    0x20, F       ; Sum with value in 0x20
        MOVF     PRODH, W      ; Move high byte of product to WREG
        ADDWFC 0x21, F         ; Sum with carry with value in 0x21
        DECFSZ    0x60, F      ; Decrement register 0x60 by one, and skip next
                               ; step if 0
        GOTO     BACK          ; Start over
        RETURN                 ; Return to main code
        END
```

Verification of the programs using MPLAB:

The following sample data are used:

[0x40] = 0x7E = 126 (decimal)
[0x41] = 0x08 = 8 (decimal)
[0x42] = 0x23 = 35 (decimal)
[0x43] = 0x30 = 48 (decimal)

$$\sum_{i=1}^{N} X_i^2 = (126)^2 + (8)^2 + (35)^2 + (48)^2 = 15876 + 64 + 1225 + 2304 = 19469 \text{ (decimal)}$$

Hence, result = (19469)/4 = 4867.25, which is approximately 4867 (decimal) or 1303 (hex).

The following example will also demonstrate how the hardware stack on the PIC18F changes with the execution of the CALL and RETURN instructions.

The "PIC18F disassembly" function can be displayed from the "Disassembly Listing" option in the "View" menu as follows:

File Registers

Address	00	01	02	03	04	05	06
000	00	00	00	00	00	00	00
010	00	00	00	00	00	00	00
020	03	13	00	00	00	00	00
030	00	00	00	00	00	00	00
040	7E	08	23	30	00	00	00
050	30	00	00	00	00	00	00
060	00	00	00	00	00	00	00

7.7 Multiplication and Division Algorithms

As mentioned in Chapter 1, an *unsigned binary number* has no arithmetic sign, and therefore, is always positive. Typical examples are your age or a memory address, which are always positive numbers. An 8-bit unsigned binary integer represents all numbers from 00_{16} through FF_{10} (0_{10} through 255_{10}).

A *signed binary number*, on the other hand, includes both positive and negative numbers. It is represented in the microcontroller in two's complement form. For example, the decimal number +15 is represented in 8-bit two's complement form as 00001111 (binary) or 0F (hexadecimal). The decimal number -15 can be represented in 8-bit two's complement form as 11110001 (binary) or F1 (hexadecimal). Also, the most significant bit (MSB) of a signed number represents the sign of the number. For example, bit 7 of an 8-bit number represents the signs of the respective numbers. A "0" at the MSB represents a positive number; a "1" at the MSB represents a negative number. Note that the 8-bit binary number 11111111 is 255_{10} when represented as an unsigned number. On the other hand, 11111111_2 is -1_{10} when represented as a signed number.

As mentioned before, the PIC18F includes only unsigned multiplication instruction. The PIC18F instruction set does not provide any instructions for signed multiplication, or unsigned and signed division instructions. These algorithms are covered in detail in Section 3.3.6 of Chapter 3. A summary of the algorithms is provided in this section for convenience. The PIC18F assembly language programs using these algorithms are written in this section.

7.7.1 Signed Multiplication Algorithm

Signed multiplication can be performed using various algorithms. A simple algorithm follows. Assume that *M* (multiplicand) and *Q* (multiplier) are in two's complement form. Assume that M_n and Q_n are the most significant bits (sign bits) of the multiplicand (*M*) and the multiplier (*Q*), respectively. To perform signed multiplication, proceed as follows:

1. If $M_n = 1$, compute the two's complement of *M*; else, keep *M* unchanged.
2. If $Q_n = 1$, compute the two' s complement of *Q*; else, keep *Q* unchanged.
3. Multiply the $n - 1$ bits of the multiplier and the multiplicand using unsigned multiplication.
4. The sign of the result $S_n = M_n \oplus Q_n$.

```
                MOVWF  COUNTER      ; Move [WREG] into COUNTER
        LOOP    TBLRD*+             ; Read data from program memory into
                                    ; TABLAT, increment TBLPTR by 1
                MOVF   TABLAT, W    ; Move [TABLAT] into WREG
                MOVWF  POSTINC0     ; Move W into data memory pointed to
                                    ; by FSR0, and then increment FSR0 by 1
                                    ; memory address   0x000200
                DECF   COUNTER, F   ; Decrement COUNTER BY 1
                BNZ    LOOP         ; Branch  if  Z = 0
; INITIALIZE STKPTR, LOAD n, INITIALIZE DATA POINTER, CALL SUBROUTINE
                MOVLW  0x15         ; Initialize STKPTR to 0x15
                MOVWF  STKPTR
                MOVLW  4            ; Move n into WREG
                LFSR   1, 0x40      ; Load 0x40 into FSR0 to be used as pointer
                CALL   FIBNUM
        FINISH  BRA    FINISH
; READ  THE FIBONACCI  NUMBER FOR n FROM  DATA MEMORY INTO 'W' USING
; MOVF WITH  INDEXED  ADDRESSING MODE

; SUBROUTINE
                ORG    0x60
        FIBNUM  MOVF   PLUSW1, W    ; Result in WREG
                RETURN              ; Return to FINISH in main
                ORG    0x200
        ADDR    DB     1, 1, 2, 3, 5, 8, 13 ;     Fibonacci numbers
                END
```

Example 7.8 Without using a lookup table and the MOVFF with indexed addressing mode as in Example 7.7, write a subroutine in PIC18F assembly language at address 0x50 to find the nth number (0 to 6) of the Fibonacci sequence. The subroutine will return the desired Fibonacci number in WREG based on 'n' stored by the main program. Also, write the main program at address 0x100 that will store the nth number (0 to 6) in WREG, call the subroutine, and stop. The Fibonacci sequence for n = 0 to 6 is provided below:

n	Fib(n)
0	1
1	1
2	2
3	3
4	5
5	8
6	13

Solution

This program can be written with the RETLW instruction that is ideal for returning the desired value using an operation alternate to using a table lookup with indexed addressing mode shown in Example 7.7. Note that, the RETLW k loads the 8-bit immediate data k into WREG, and returns to the main program by loading the program counter with the address

```
; STEP4: ADJUST (BCD CORRECTION) [WREG] TO CONTAIN CORRECT PACKED BCD
          DAW                       ; #17 ADJUST THE RESULT TO CONTAIN
                                    ; CORRECT PACKED BCD
FINISH    BRA        FINISH         ; HALT
          END
```

Note: The above program will be explained in the following. Note that the # sign along with the line number is placed before each comment in order to explain the program. ASCII data to be added are assumed to be 3439H and 3231H. The purpose of the program is to convert the first number, ASCII 3439H to unpacked BCD 0409H, and then to packed BCD 49H, and similarly, the second number, ASCII 3231H, to unpacked BCD 0201H, and then to packed BCD 21H. Finally, the two packed BCD numbers will be added in binary using PIC18F's ADDWF instruction, and then the result in WREG will be converted to correct packed BCD using DAW.

Line #'s 1 through 4 initialize N1 and N2 so that [0x40] = 39H, [0x41] = 34H, [0x50] = 31H, and [0x51] = 32H. Line #5 initializes COUNTER with loop count of 2 for converting the numbers from ASCII to unpacked BCD. Line #'s 6 and 7 initialize FSR0 and FSR1 with 0x40 and 0x50, respectively. Line #'s 8 through 11 convert the two bytes of ASCII codes in 0x41 (high byte) and 0x40 (low byte) into unpacked BCD in 0x41 (high byte) and 0x40 (low byte). Also, Line #'s 8 through 11 convert the ASCII numbers, N1 and N2 into their corresponding unpacked BCD bytes.

Line #'s 12 through 15 convert the unpacked BCD numbers (N1 and N2) into packed BCD bytes. This is done by swapping high unpacked bytes of N1 and N2 , and then ORing with the corresponding low unpacked bytes. Line #16 performs binary addition of the two packed BCD bytes (N1 and N2), and stores the binary result in WREG. The DAW instruction at Line #17 adjusts the contents of WREG to provide the correct packed BCD result.

7.9 PIC18F Delay Routine

Typical PIC18F software delay routines can be written by loading a "counter" with a value equivalent to the desired delay time, and then decrementing the "counter" in a loop, using typically MOVE, DECREMENT, and conditional BRANCH instructions. For example, the following PIC18F instruction sequence can be used for a delay loop:

```
         MOVLW    COUNT
         MOVWF    0x20
DELAY    DECF     0X20, F
         BNZ      DELAY
```

Note that DECF in the above decrements the register 0x20 by one, and if [0x20] ≠0, branches to DELAY; if [0x20] = 0, the PIC18F executes the next instruction. The initial loop counter value of "COUNT" can be calculated using the machine cycles (Appendix D) required to execute the following PIC18F instructions:

```
MOVLW     (1 cycle)
MOVWF     (1 cycle)
DECF      (1 cycle)
BNZ       (2/1 cycles)
```

Note that the BNZ instruction requires two different execution times. BNZ requires two cycles when the PIC18F branches if Z = 0. However, the PIC18F goes to the next instruction and does not branch when Z = 1. This means that the DELAY loop will require two cycles for "COUNT" times, and the last iteration will take one cycle. The desired delay time can be obtained by loading register 0x20 with the appropriate COUNT value.

Assuming 1 MHz default crystal frequency, the PIC18F's clock period will be 1 μsec. Note that the PIC18F divides the crystal frequency by 4. This is equivalent to multiplying the clock period by 4. Hence, each instruction cycle will be 4 microseconds. For a 100-microsecond delay, total cycles = $\frac{100 \text{ micro sec}}{4 \text{ micro sec}} = 25$. The BNZ in the loop will require two cycles for (COUNT - 1) times when Z = 0 and the last iteration will take 1 cycle when no branch is taken (Z = 1). Thus, total cycles including the MOVLW = $1 + 1 + 3 \times (\text{COUNT} - 1) + 1 = 25$. Hence, COUNT = 8.3. Therefore, register 0x20 should be loaded with an integer value of 9 for an approximate delay of 100 microseconds.

Now, in order to obtain delay of one millisecond, the above DELAY loop of 100 miicroseconds can be used with an external counter. Counter value = $\frac{1 \text{ milli sec}}{100 \text{ micro sec}} = 10$. The following instruction sequence will provide an approximate delay of one millisecond:

```
        MOVLW  D'10'
        MOVWF  0x30     ; Initialize counter 0x30 for one-millisecond delay
BACK    MOVLW  D'9'
        MOVWF  0x20     ; Initialize counter 0x20 for 100-microsecond delay
DELAY   DECF   0X20, F  ; 100-microsec delay
        BNZ    DELAY
        DECF   0X30, F
        BNZ    BACK
```

Next, the delay time provided by the above instruction sequence can be calculated.

As before, assuming 1 MHz crystal, each instruction cycle is 4 microseconds.

Total delay in seconds from the above instruction sequence

= Execution time for MOVLW + Execution time for MOVWF +
10 x (100-microsecond delay) + (10 - 1) x (Execution time for DECF +
Execution time for BNZ (Z = 0)) + Execution time for BNZ (Z = 1)

= 1 x (4 microsec) + 1 x (4 microsec) + (1000 microseconds)
+ 9 x (3 x 4 microsec) + 1 x (4 microsec)

= 1.112 milliseconds.

This is approximately equivalent to the desired 1-millisecond delay. In other words, the delay is 1.112 milliseconds rather than 1 millisecond. This is because the execution times of MOVLW D'10', and MOVWF 0x30, are discarded.

Example 7.10 Assume 1 MHz PIC18F. Consider the following subroutine:

```
DELAY   MOVLW   D'100'
        MOVWF   0x20
DLOOP   DECFSZ  0x20, F
        BRA     DLOOP
        RETURN
```

(a) Calculate the time delay provided by the above subroutine.

stack pointer FSR0 to 0x60.

7.9. Write a PIC18F assembly program at address 0x100 to divide a 9-bit unsigned number in the high 9 bits (bits 8-1 in bits 7-0 of register 0x30 and bit 0 in bit 7 of register 0x31) by 8_{10}. Do not use any division instruction. Store the result in register 0x50. Discard the remainder.

7.10 Write a PIC18F assembly language program at address 0x200 that will check whether the 16-bit signed number in registers [0x31][0x30] is positive or negative. If the number is positive, the program will multiply the 16-bit unsigned number (bits 12 through 15 as 0's) in [0x21][0x20] by 16, and provide a 16-bit result; otherwise, the program will set the byte in register 0x40 to all ones. Use only data movement, shift, bit manipulation, and program control instructions. Assume the 16-bit signed and unsigned numbers are already loaded into the data registers.

7.11 Assume that several 8-bit packed BCD numbers are stored in data memory locations from 0x10 through 0x2D. Write a PIC18F assembly language program at address 0x100 to find how many of these numbers are divisible by 5, and save the result in data memory location 0x40.

7.12 Write a program at address 0x100 in PIC18F assembly language to add two 32-bit packed BCD numbers. BCD number 1 is stored in data registers starting from 0x20 through 0x23, with the least significant digit at register 0x23 and the most significant digit at 0x20. BCD number 2 is stored in data registers starting from 0x30 through 0x33, with the least significant digit at 0x33 and the most significant digit at 0x30. Store the result as packed BCD digits in 0x20 through 0x23.

7.13 Write a subroutine at address 0x100 in PIC18F assembly language program to find the square of a BCD digit (0 to 9) using a lookup table. The subroutine will store the desired result in WREG based on the BCD digit stored by the main program. The lookup table will store the square of the BCD numbers starting at program memory address 0x300. Also, write the main program at address 0x200 that will initialize STKPTR to 0x10, store the BCD digit (0 to 9) in WREG, call the subroutine, and stop. Use indexed addressing mode.

7.14 Write a subroutine in PIC18F assembly language program at address 0x100 to find the square of a BCD digit (0 to 9) and store it in WREG. The subroutine will return the desired result based on the BCD digit stored by the main program. Also, write the main program at address 0x200 that will initialize STKPTR to 0x12, store the BCD digit (0 to 9) in WREG, call the subroutine, and stop. Do not use indexed addressing mode.

7.15 Write a subroutine at address 0x150 in PIC18F assembly language to convert a 3-digit unpacked BCD number to binary using unsigned multiplication by 10, and additions. The most significant digit is stored in a memory location starting at register 0x30, the next digit is stored at 0x31, and so on. Store the 8-bit binary result (*N)* in register 0x50. Note that arithmetic operations for obtaining *N* will provide binary result. Use the value of the 3-digit BCD number,

$N = N2 \times 10^2 + N1 \times 10^1 + N0$
$= ((10 \times N2)+N1) \times 10+N0$

7.16 Write a subroutine in PIC18F assembly language at 0x100 to compute

$$Z = \sum_{i=1}^{8} X_i$$

Assume the X_i's are unsigned 8-bit and stored in consecutive locations starting at 0x30 and Z is 8-bit. Also, assume that FSR1 points to the X_i's. Write the main program in PIC18F assembly language at 0x150 to perform all initializations (FSR1 to 0x30, STKPTR to 20 decimal), call the subroutine, and then compute $Z/8$. Discard remainder of Z/8. Assume data are already loaded in data registers.

7.17 Write a PIC18F assembly language program to estimate the square root of an 8-bit integer number P using the algorithm provided in the following:
The sum of odd integers is always a perfect square. For example, $1 = 1^2$, $1+3 = 2^2$, $1+3+5 = 3^2$, and so on. Specifically, $\sum_{i=1}^{k}(2i - 1) = k^2$. This property is useful in approximating the square root of an 8-bit unsigned number P. For example, if P = 17, the square root of P can be estimate as follows:
Subtract 1 from P so that P becomes 16; since the subtraction went through, add 1 to a counter. Hence, counter value is 1.
Subtract 3 from P so that P becomes 13; since the subtraction went through, add 1 to a counter. Hence, counter value is 2.
Subtract 5 from P so that P becomes 8; since the subtraction went through, add 1 to a counter. Hence, counter value is 3.
Subtract 7 from P so that P becomes 1; since the subtraction went through, add 1 to a counter. Hence, counter value is 4. Now, when 9 is subtracted from the existing value of P (P = 1), the result becomes negative (-8) meaning that the subtraction did not go through. The process terminates, and the integer square root approximation for 17 is 4.

7.18 Consider the following PIC18F DELAY subroutine:

```
DELAY   MOVLW   Q
        MOVWF   Q
LOOP1   MOVLW   100
        MOVWF   P
LOOP2   DECF    P, F
        BNZ     LOOP2
        DECF    Q, F
        BNZ     LOOP1
        RETURN
```

Assuming 1 MHz PIC18F, determine the value of Q such that when this subroutine is called, a delay of 145.940 msec will be generated.

TABLE 8.1 PIC18F4321 Pinout description (continued)

Pin number	Pin name	Pin type	Description
10	RE2	Input/Output	Digital I/O; Port E bit 2
	$\overline{CS}$	Input	Chip Select control for parallel slave port (see related $\overline{RD}$ and $\overline{WR}$).
	AN7	Input	Analog input 7
11	VDD	Power	Positive supply for logic and I/O pins.
12	VSS	Ground	Ground reference for logic and I/O pins.
13	OSC1	Input	Oscillator crystal input or external clock source input.
	CLKI	Input	External clock source input. Always associated with pin function OSC1. (See related OSC1/CLKI, OSC2/ CLKO pins.)
	RA7	Input/Output	Digital I/O; Port A bit 7
14	OSC2	Output	Oscillator crystal output. Connects to crystal or resonator in crystal oscillator mode. In RC, EC, and INTIO modes, OSC2 pin outputs.
	CLKO	Output	CLKO which has one-fourth the frequency of OSC1 and denotes the instruction cycle rate.
	RA6	Input/Output	Digital I/O; Port A bit 6
15	RC0	Input/Output	Digital I/O; Port C bit 0
	T1OSO	Output	Timer1 oscillator analog output
	T13CKI	Input	Timer1/Timer3 external clock input
16	RC1	Input/Output	Digital I/O; Port C bit 1
	T1OSI	Input	Timer1 oscillator analog input
	CCP2	Input/Output	Capture 2 input/Compare 2 output/ PWM 2 output; default assignment for CCP2 when Configuration bit, CCP2MX, is set.
17	RC2	Input/Output	Digital I/O; Port C bit 2
	CCP1	Input/Output	Capture 1 input/Compare 1 output/ PWM 1 output
	P1A	Output	Enhanced CCP1 output
18	RC3	Input/Output	Digital I/O; Port C bit 3
	SCK	Input/Output	Synchronous serial clock input/output for SPI mode.
	SCL	Input/Output	Synchronous serial clock input/output for I^{2}C™ mode.
19	RD0	Input/Output	Digital I/O; Port D bit 0
	PSP0	Input/Output	Parallel Slave Port data
20	RD1	Input/Output	Digital I/O; Port D bit 1
	PSP1	Input/Output	Parallel slave port data
21	RD2	Input/Output	Digital I/O; Port D bit2
	PSP2	Input/Output	Parallel slave port data
22	RD3	Input/Output	Digital I/O; Port D bit 3
	PSP3	Input/Output	Parallel slave port data

TABLE 8.1 PIC18F4321 Pinout description (continued)

Pin number	Pin name	Pin type	Description
23	RC4	Input/Output	Digital I/O
	SDI	Input	SPI data in
	SDA	Input/Output	I^2C data I/O
24	RC5	Input/Output	Digital I/O; Port C bit 5
	SDO	Output	SPI data out
25	RC6	Input/Output	Digital I/O; Port C bit 6
	TX	Output	EUSART asynchronous transmit
	CK	Input/Output	EUSART synchronous clock (see related RX/DT).
26	RC7	Input/Output	Digital I/O; Port C bit 7
	RX	Input	EUSART asynchronous receive
	DT	Input/Output	EUSART synchronous data (see related TX/CK)
27	RD4	Input/Output	Digital I/O; Port D bit 4
	PSP4	Input/Output	Parallel slave port data
28	RD5	Input/Output	Digital I/O; port D bit 5
	PSP5	Input/Output	Parallel slave port data
	P1B	Output	Enhanced CCP1 output
29	RD6	Input/Output	Digital I/O; port D bit 6
	PSP6	Input/Output	Parallel slave port data
	P1C	Output	Enhanced CCP1 output
30	RD7	Input/Output	Digital I/O; Port D bit 7
	PSP7	Input/Output	Parallel slave port data
	P1D	Output	Enhanced CCP1 output
31	VSS	Ground	Ground reference for logic and I/O pins
32	VDD	Power	Positive supply for logic and I/O pins
33	RB0	Input/Output	Digital I/O; Port B bit 0
	INT0	Input	External interrupt 0
	FLT0	Input	PWM fault input for enhanced CCP1
	AN12	Input	Analog input 12
34	RB1	Input/Output	Digital I/O; Port B bit 1
	INT1	Input	External interrupt 1
	AN10	Input	Analog input 10
35	RB2	Input/Output	Digital I/O; Port B bit 2
	INT2	Input	External interrupt 2
	AN8	Input	Analog input 8
36	RB3	Input/Output	Digital I/O; Port B bit 3
	AN9	Input	Analog input 9
	CCP2	Input/Output	Capture 2 input/compare 2 output/ PWM 2 output; alternate assignment for CCP2 when configuration bit CCP2MX, is cleared.
37	RB4	Input/Output	Digital I/O; Port B bit 7
	KBI0	Input	Interrupt-on-change pin
	AN11	Input	Analog input 11
38	RB5	Input/Output	Digital I/O; Port B bit 5
	KBI1	Input	Interrupt-on-change pin

TABLE 8.1 PIC18F4321 Pinout description (continued)

Pin number	Pin name	Pin type	Description
	PGM	Input/Output	Low-voltage programming enable pin
39	RB6	Input/Output	Digital I/O; Port B bit 6
	KBI2	Input	Interrupt-on-change pin
	PGC	Input/Output	In-circuit debugger and programming clock pin
40	RB7	Input/Output	Digital I/O; Port B bit 7
	KBI3	Input	Interrupt-on-change pin
	PGD	Input/Output	In-circuit debugger and ICSP programming data pin.

The PIC18F pins associated with clock, reset, and I/O will be discussed in the following.

8.1.1 Clock

Upon reset, the PIC18F4321 operates at an internal clock frequency of 1 MHz (default). After reset, the internal frequency can be changed from 31 KHz to 8 MHz by writing appropriate data into the OSCCON register (page 31 of the PIC18F4321 data sheet).

The PIC18F4321 can also be operated in ten different oscillator modes. The user can program the Configuration bits, FOSC3:FOSC0, in Configuration Register to select one of these ten modes. Note that this configuration register is mapped as address 300001H in program memory, and can be accessed using TBLRD and TBLWT instructions. These modes can be classified into following groups:

1. By connecting a crystal or ceramic resonator at OSC1 and OSC2 pins.
2. By connecting an external clock source at the OSC1 pin. The oscillator frequency divided by 4 is output at the OSC2 pin.
3. By connecting an RC oscillator circuit at the OSC1 pin. The oscillator frequency divided by 4 is output at the OSC2 pin.
4. Using a frequency multiplier for a crystal oscillator to produce an internal clock frequency of up to 40MHz. The "frequency multiplier mode" is only available to the crystal oscillator when the FOSC3:FOSC0 configuration bits are programmed for this mode. This will be useful for applications requiring higher clock speed.
5. Using an internal oscillator block which generates two different clock signals; either can be used as the microcontroller's clock source. This may eliminate the need for external oscillator circuits on the OSC1 and/or OSC2 pins.
6. By switching the PIC18F4321 clock source from the main oscillator to an alternate clock source. When an alternate clock source is enabled, the various power-managed operating modes are available.

In the following, typical oscillator circuits using a crystal and an RC circuit will be provided. Figure 8.2 shows a typical quartz crystal oscillator circuit for the PIC18F. The crystal frequency can vary from 4MHz to 25 MHz.

Figure 8.3 shows how the PIC18F clock is generated at the OSC2 pin by connecting an RC oscillator circuit at the OSC1 pin. The oscillator frequency at the OSC1 pin is divided by 4 by the PIC18F and then generated on the OSC2 output pin. The OSC2 clock may be used for test purposes to synchronize other logic.

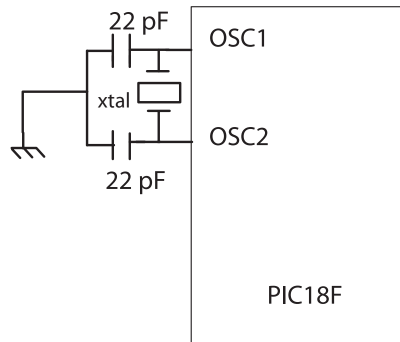


FIGURE 8.2 Typical crystal oscillator circuit

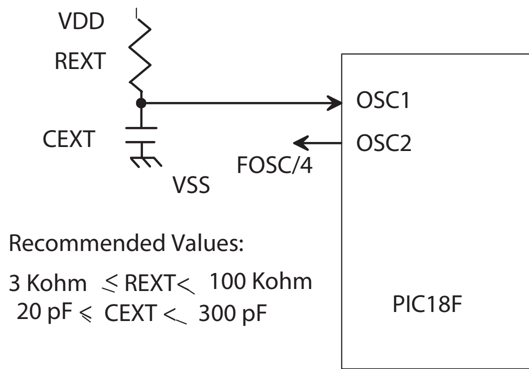


FIGURE 8.3 RC oscillator

8.1.2 PIC18F Reset

Upon reset, the PIC18F loads '0' into program counter. Thus the PIC18F reads the first instruction from the contents of address 0 in the program memory. Most registers are unaffected by a Reset. Howerver, WREG and STKPTR are cleared to zero. All TRISX (Data Direction Registers) are loaded with FFH. The PIC18F4321 can be reset in several different ways. For simplicity, the two most commonly used RESET techniques are Power-on and Manual resets. These two resets will be discussed in this following. A summary of some of the other resets will then be provided.

Power-On Reset (POR) A power-on reset pulse is generated on-chip upon power-up whenever VDD rises above a certain threshold. This allows the device to start in the initialized state when VDD is adequate for operation. The reset circuit in Figure 8.4 provides a simple power-on reset circuit with a pushbutton (manual) switch. When the power is turned ON, the resistors in Figure 8.4 with the switch open will provide power-on reset. When the PIC18F exits the reset condition, and starts normal operation, a program can be executed by pressing the pushbutton, and program execution can be restarted upon activation of the pushbutton.

Power-on reset events are captured by the $\overline{\text{POR}}$ bit (bit 1 of RCON, Figure 8.5). The state of the bit is set to '0' whenever a POR occurs; $\overline{\text{POR}}$ bit = 1 indicates that a POR has not occurred.

TABLE 8.2 PIC18F4321 I/O PORTS, TRISx REGISTERS, ALONG WITH ADDRESSES

(Upon RESET, all ports are configured as inputs)

Port Name	Size	Mapped SFR address	Comment
Port A	8-bit	0xF80	Port A
TRISA	8-bit	0xF92	Data Direction Register for Port A
Port B	8-bit	0xF81	Port B
TRISB	8-bit	0xF93	Data Direction Register for Port B
Port C	8-bit	0xF82	Port C
TRISC	8-bit	0xF94	Data Direction Register for Port C
Port D	8-bit	0xF83	Port D
TRISD	8-bit	0xF95	Data Direction Register for Port D
Port E	4-bit	0xF84	Port E
TRISE	4-bit	0xF96	Data Direction Register for Port E

inputting from or outputting to ports. As an example, consider the PIC18F instruction, MOVF PORTD, W will input the contents of PORTD into WREG.

The MOVWF PORTC instruction, on the other hand, will output the contents of WREG into PORTC. Data can also be output from one port to another. For example, the MOVFF PORTC, PORTD instruction will output the contents of PORTC to PORTD.

The PIC18F bit-oriented instructions such as BSF and BCF can be used to output a ‘1’ or ‘0’ to a specific bit of an I/O port. For example, the instruction BSF PORTD, 6 will set bit 6 of Port D; in other words, the PIC18F will output a ‘1’ to bit 6 of Port D. The instruction BCF PORTC, 3 , on the other hand, will clear bit 3 of Port A to zero; in other words, the PIC18F4321 will output a ‘0’ to bit 3 of PORTC.

8.2.2 Configuring PIC18F4321 I/O Ports

As mentioned before, writing a ‘1’ at a particular bit position in the TRISx register will make the corresponding bit in the associated port as an input. On the other hand, writing a ‘0’ at a particular bit position in the TRISx register will make the corresponding bit in the associated port as an output. Upon reset all TRIS registers are automatically loaded with 1’s, and hence, all ports will be configured as inputs.

Next, in order to illustrate how PIC18F4321 ports are configured using the associated TRISx registers, consider the following PIC18F instruction sequence:

```
MOVLW    0x34    ; Move 0x34 into WREG
MOVWF    TRISD   ; Configure PORT D
```

In the above instruction sequence, MOVLW loads WREG with 34 (hex), and then moves these data into TRISD (8-bit data direction register for PORTD) which then contains 34(Hex); the corresponding port is defined as shown in Figure 8.8. In this example, because 34H (0011 0100) is written into TRISD, bits 0, 1, 3, 6, and 7 of the port are set up as outputs, and bits 2, 4, and 5 of the port are defined as inputs. The microcontroller can then send output to external devices, such as LEDs, connected to bits 0, 1, 3, 6, and 7 through a proper interface. Similarly, the PIC18F4321 can input the status of external devices, such as switches, through bits 2, 4, and 5. To input data from the input switches, the PIC18F4321 inputs the complete byte, including the bits to which output devices such as LEDs are connected. While receiving input data from an I/O port, however, the PIC18F4321 places a value, probably 0, at the bits configured as outputs and the program must interpret them as “don’t cares.” At the same time, the PIC18F4321’s outputs to bits configured as inputs are disabled.

The PIC18F instructions such as SETF and CLRF can be used to configure I/O

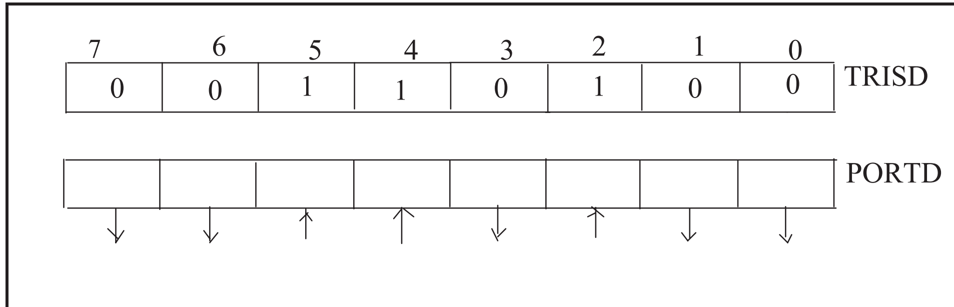


FIGURE 8.8 PORT D along with TRISD

ports. For example, to configure all bits in Port C as inputs, and Port D as outputs, SETF or CLRF instructions can be used as follows:

```
SETF    TRISC   ; Set all bits in TRISC to 1's and configure
                ; configure Port C as an input port.
CLRF    TRISD   ; Clear all bits in TRISD to 0's and configure
                ; configure Port D as an output port
```

Also, a specific bit in a port can be configured as an input or as an output using PIC18F bit-oriented instructions such as BSF and BCF. For example, the instruction BSF TRISD, 7 will make bit 7 of Port D as an input bit. On the other hand, BCF TRISC, 1 will make bit 1 of Port C as an output.

Note that configuring Port A, Port B and Port E is different than configuring Port C and Port D. This is because, certain bits of Port A, Port B, and Port E are multiplexed with analog inputs (Figure 8.1). Upon power-on reset (default), all bits of ports A, B, and E multiplexed with AN0-AN12 are configured as analog inputs. However, writing 0x0F to bits 0-3 of the ADCON1 register (Figure 8.9) will configure these multiplexed port bits as digital I/O. As mentioned before, upon power-on reset, all TRISx registers are loaded with 0xFF and the associated port bits are configured as inputs. Hence, upon loading 0x0F into ADCON1 register will make ports A, B and E as inputs in default mode.

However, for configuring these ports as outputs, the corresponding TRISx bits must be loaded with 0's; the ADCON1 register is not required for configuring these port bits as outputs. The following examples will illustrate this.

For example, the following instruction sequence will configure all 13 port bits multiplexed with AN0 - AN12 as inputs:

```
MOVLW   0x0F      ; Move 0xF into WREG
MOVWF   ADCON1    ; Move WREG into ADCON1
```

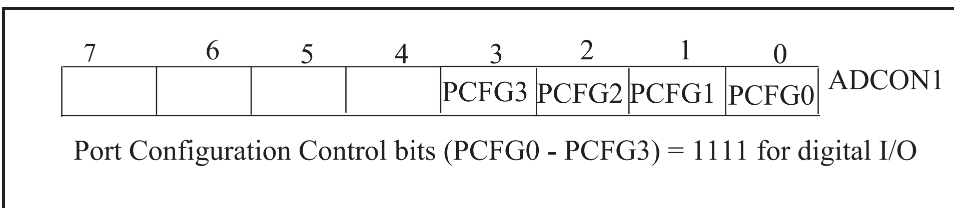


FIGURE 8.9 ADCON1 register for digital I/O

Next, in order to configure bit 1 of Port A, and bits 2 and 4 of Port B as outputs, the following instruction sequence can be used:

```
BCF    TRISA, 1    ; Configure bit 1 of Port A as output
BCF    TRISB, 2    ; Configure bit 2 of Port B as output
BCF    TRISB, 4    ; Configure bit 4 of Port B as output
```

It should be mentioned that if a bit of an I/O port in the PIC18F family of microcontrollers such as the PIC18F4321 and PIC18F4520 is multiplexed with an analog input, the bit must be configured as an input using the ADCON1 register; the same bit can be configured as an output using the corresponding bit in the associated TRISx register. However, if a port bit is not multiplexed with an analog input, it can be configured as an input or an output using the associated TRISx register.

For simplicity, Port C, Port D, and multiplexed bits of Port A, Port B, and Port E with analog inputs will be used to illustrate the concept of programmed I/O associated with the PIC18F4321 in this book.

8.2.3 Interfacing LEDs (Light Emitting Diodes) and Seven-segment Displays

The PIC18F sources and sinks adequate currents so that LEDs and seven-segment displays can be interfaced to the PIC18F without buffers (current amplifiers) such as 74HC244. An LED can be connected in two ways. Figure 8.10 (a) and (b) shows these configurations.

In Figure 8.10 (a), the PIC18F will output a HIGH to turn the LED ON; the PIC18F will output a 'LOW' will turn it OFF. In Figure 8.10 (b), the PIC18F will output a LOW to turn the LED ON; the PIC18F will output a 'HIGH' will turn it OFF. Also, when an LED is turned on, a typical current of 10 mA flows through the LED with a voltage drop of 1.7 V. Hence,

$$R = \frac{5 - 1.7}{10 \text{ mA}} = 330\ \Omega$$

As discussed in Chapter 3, a seven-segment display can be used to display, for example, decimal numbers from 0 to 9. The name "seven segment" is based on the fact that there are seven LEDs — one in each segment of the display. Figure 8.11 shows a typical seven-segment display.

Figure 8.12 shows two different seven-segment display configurations, namely, common cathode and common anode. Note that Figures 8.11 and 8.12 are redrawn from Chapter 3 for convenience. In Figure 8.12, each segment contains an LED. All decimal

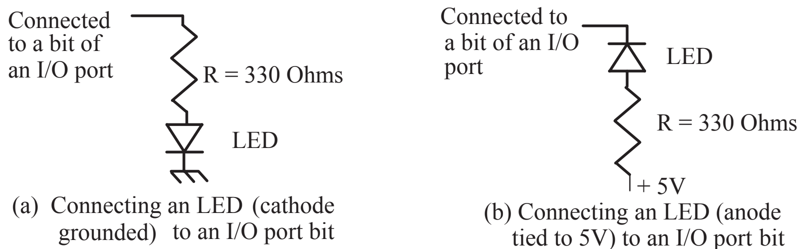


FIGURE 8.10 Interfacing LED to PIC18F

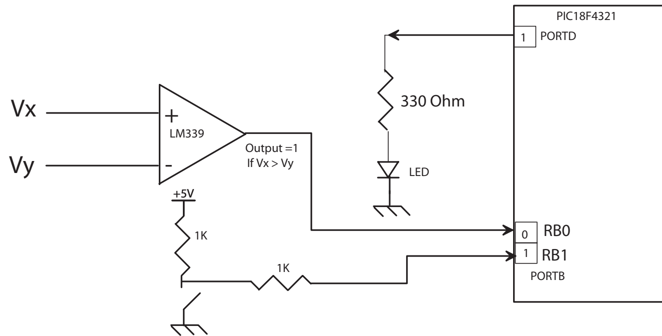


FIGURE 8.19 Figure for Example 8.4 (a) using programmed I/O

(b) Repeat part (a) using INT0 external interrupt. Use Port D for the LED and bit 1 of Port B for the switch as above. Write the main program at address 0x100 in PIC18F assembly language. Connect INT0 to the output of the comparator. The main program will initialize hardware stack pointer (STKPTR) to 0x12, configure Port B and Port D and enable GIE and INT0IE. Write the service routine in PIC18F assembly language which will clear the INT0 flag, input the switch, output to LED, and then return to the main program.

Solution

Example 8.4(a) uses programmed (polled or conditional) I/O while Example 8.4(b) uses interrupt I/O.

(a) In this example, an LM339 comparator is connected to the PIC18F4321 in order to control when the LED will be turned ON or OFF, based on the switch. In Figure 8.19, when Vx > Vy, then the comparator will output a one, and the PIC18F4321 will turn the LED ON or OFF depending on the switch status. If Vx < Vy, then the comparator will output a zero and the LED will be turned OFF. In the program, the ADCON1 register is used to configure RB0 and RB1 as inputs. The TRISD is used to make RD1 of Port D as output.

The PIC18F assembly language program for programmed I/O is provided below:

```
            INCLUDE <P18F4321.INC>
            ORG     0            ; RESET VECTOR
            GOTO    MAIN         ; JUMP TO MAIN
            ORG     0x100
MAIN        BCF     TRISD, 1     ; CONFIGURE BIT 1 OF PORTD AS OUTPUT
            MOVLW   0x0F         ; CONFIGURE BITS 0 AND 1 OF PORTB
            MOVWF   ADCON1       ; AS DIGITAL INPUTS
BEGIN       BCF     PORTD, 1     ; TURN LED OFF
CHECK       BTFSS   PORTB, 0     ; CHECK IF COMPARATOR IS ONE
            BRA     BEGIN        ; WAIT IN LOOP UNTIL ONE
            BTFSS   PORTB, 1     ; CHECK IF SWITCH IS OPEN
            BRA     BEGIN
            BSF     PORTD, 1     ; IF SWITCH OPEN, TURN LED ON
```

```
        BRA     CHECK
        END
```

In the above program, upon reset, the PIC18F starts executing the program at address 0x000000 in program memory. After execution of the instruction GOTO MAIN, the program will jump to address 0x100.

Next, Port B and Port D are configured. The instruction BCF PORTD, 0 turns the LED OFF. In order to check whether the comparator is outputting a one, the instruction BTFSS PORTB, 0 is used in the program. After execution of this instruction, if bit 0 of PORTB (comparator output) is 0, the next instruction, BRA BEGIN, continues looping until the comparator outputs a one. However, if the comparator output is 1, the BTFSS PORTB, 0 will skip the next instruction (BRA BEGIN), and will execute BTFSS PORTB, 1 to see whether the switch is ON or OFF. If it is OFF, the program will branch to BEGIN where it will turn the LED OFF. If the switch is OPEN (bit 1 of PORTB is 1), the program will skip the next instruction (BRA BEGIN), output a 1 to bit 1 of PORTD, and then turn the LED ON. The program will then branch to CHECK to make the loop is continuous.

(b) Figure 8.20 shows the relevant connections of the comparator to the PIC18F4321 using interrupt I/O. Note that the comparator output is connected to bit 0 of Port B to be used as an INT0 pin. In this example, using ADCON1 register, bit 0 of Port B can be configured as digital input to accept interrupt via INT0. The INT0IE bit of the INTCON register must be set to one in order to enable the external interrupt along with GIE to enable global interrupts.

The PIC18F assembly language program using external interrupt INT0 is provided in the following:

```
            INCLUDE <P18F4321.INC>
            ORG     0               ; RESET
            GOTO    MAIN_PROG
; MAIN PROGRAM
            ORG     0x00100         ; MAIN PROGRAM
MAIN_PROG   MOVLW   0x12            ; Initialize STKPTR to 0x12
            MOVWF   STKPTR
            BCF     TRISD,1         ; Configure bit 1 OF PORTD as output
            MOVLW   0x0F            ; Configure bit 0 of PORTB as INT0
            MOVWF   ADCON1          ; and bit 1 as input
```

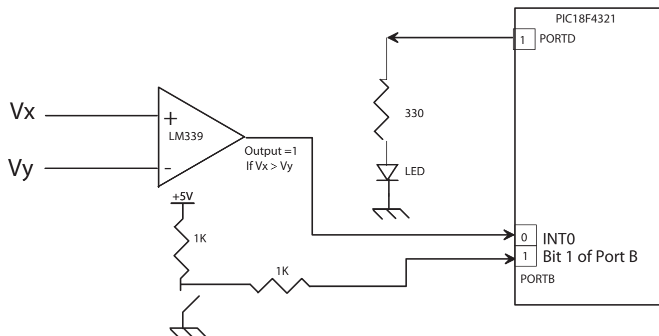


FIGURE 8.20 Figure for Example 8.3(b) using interrupt I/O

```
                BSF     INTCON, INT0IE  ; Enable the external interrupt
                BCF     INTCON, INT0IF  ; Clear interrupt flag
                BSF     INTCON, GIE     ; Enable global interrupts
OVER            BRA     OVER            ; Wait for interrupt
                GOTO    MAIN_PROG       ; Repeat
; INTERRUPT  SERVICE ROUTINE
                ORG     0x000008        ; Interrupt Address Vector
INT_SERV        MOVFF   PORTB, PORTD    ; Output switch status to turn LED ON/OF
                BCF     INTCON, INT0IF  ; Clear flag to avoid double interrupt
                RETFIE                  ; Enable interrupt and return
                END
```

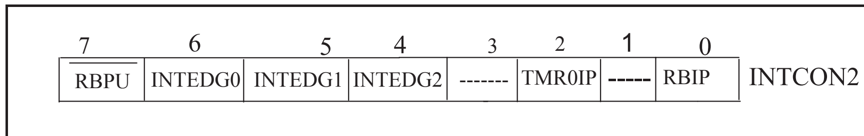
In the above, upon recognition of the interrupt, the PIC18F4321 pushes the program counter onto the stack, and automatically jumps to address 0x000008 (interrupt address vector, 0x000008 for INT0). The interrupt service routine is written at address 0x000008. The interrupt flag bit does not need to be checked to determine the source of interrupt for the single interrupt in this example. It will be shown in the next section that for multiple interrupts, the interrupt flag bit for each individual interrupt must be checked in the routine at the interrupt address vector to find the source of interrupt.

8.3.4 Interrupt Registers and Priorities

The PIC18F4321 contains ten registers which are used to control interrupt operation. These registers are

- RCON (Figure 8.5)
- INTCON (Figure 8.17)
- INTCON2 (Figure 8.21)
- INTCON3 (Figure 8.17)
- PIR1, PIR2 (to be discussed in Chapter 9)
- PIE1, PIE2 (to be discussed in Chapter 9)
- IPR1, IPR2 (discussed in Microchip's PIC18F4321 manual)

Registers RCON, INTCON, INTCON2, and INTCON3 are associated with external and port change interrupts. Hence, they will be covered in this section. Registers



bit 7 **RBPU:** PORTB Pull-up Enable bit
1 = All PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values
bit 6 **INTEDG0:** External Interrupt 0 Edge Select bit, 1 = Interrupt on rising edge, 0 = Interrupt on falling edge
bit 5 **INTEDG1:** External Interrupt 1 Edge Select bit, 1 = Interrupt on rising edge, 0 = Interrupt on falling edge
bit 4 **INTEDG2:** External Interrupt 2 Edge Select bit, 1 = Interrupt on rising edge, 0 = Interrupt on falling edge
bit 3 **Unimplemented:** Read as '0'
bit 2 **TMR0IP:** TMR0 Overflow Interrupt Priority bit, 1 = High priority, 0 = Low priority
bit 1 **Unimplemented:** Read as '0'
bit 0 **RBIP:** RB Port Change Interrupt Priority bit, 1 = High priority, 0 = Low priority

FIGURE 8.21 INTCON2 register

in the main program. The "RETFIE 1" instruction, on the other hand, pops the contents of WREG, BSR, and STATUS registers (previously PUSHed) from shadow registers WS, STATUSS, and BSRS before going to the main program, enables the global interrupt enable bit, and returns control to the appropriate place in the main program.

8.3.7 PORTB Interrupt-on-Change

The PIC18F4321 provides four interrupt-on-change pins (KB10 through KB13). These pins are multiplexed among others with bits 4 through 7 of Port B.

An input change (HIGH to LOW or LOW to HIGH) on one or more of these interrupts sets the flag bit RBIF (bit 0 of INTCON register). Note that a single flag bit is assigned to all four interrupts.

The interrupt can be enabled/disabled by setting/clearing a single enable bit, RBIE (bit 3 of INTCON register). Interrupt priority for PORTB interrupt-on-change is determined by the value contained in the interrupt priority bit RBIP (bit 0 of INTCON2 register). As before, BSF and BCF instructions can be used to set or clear a bit in a register. The PORTB interrupt-on-charge is typically used for interfacing devices such as keyboard.

8.3.8 Context Saving During Interrupts

During interrupts, the return PC address is saved onto the hardware stack. For high-priority interrupts, the PIC18F also saves WREG, STATUS, and BSR registers automatically in the associated shadow registers (internal to the PIC18F) called WS, STATUSS, and BSRS. Note that these three registers are saved internally upon recognition of a high-priority interrupt, and before going to the ISR. The contents of WREG, STATUS, and BSR are normally changed by the instructions in the ISR. Hence, it is desirable for the user to restore the contents of these registers before returning to the main program. This can be accomplished by placing the "RETFIE 1" at the end of the ISR which will restore these registers, enable global interrupt, and return control to the appropriate place in the main program.

For low priority interrupts, only the return PC address is saved onto the hardware stack. Additionally, the user may need to save WREG, STATUS and BSR registers in data memory on entry to the ISR. Depending on the user's application, other registers may also need to be saved. For low-priority interrupts, the user may save the desired registers such as WREG, STATUS, and BSR in data registers, and retrieve them before returning to the main program. The following PIC18F instruction sequence illustrates this:

```
; MAIN PROGRAM
W_TEMP              EQU     0x20
STATUS_TEMP         EQU     0x30
BSR_TEMP            EQU     0x40
                    ---
                    ---
; INTERRUPT SERVICE ROUTINE
; SAVING STATUS, WREG AND BSR REGISTERS IN  DATA MEMORY
          MOVWF     W_TEMP                  ; Save WREG in 0x20
          MOVFF     STATUS, STATUS_TEMP     ; Save STATUS  in 0x30
          MOVFF     BSR, BSR_TEMP           ; Save BSR in 0x40
          ---
          ---
```

```
        MOVFF   BSR_TEMP, BSR             ; Restore BSR
        MOVF    W_TEMP, W                 ; Restore WREG
        MOVFF   STATUS_TEMP, STATUS       ; Restore STATUS
        RETFIE                            ; POP  PC  from hardware stack,
                                          ; Enable global interrupt, and
                                          ; return to the main program
```

Example 8.5 In Figure 8.22, if Vx > Vy, the PIC18F4321 is interrupted via INT0. On the other hand, opening the switch will interrupt the microcontroller via INT1. Note that in the PIC18F4321, INT0 has higher priority than INT1. Write the main program in PIC18F assembly language at address 0x100 that will perform the following:

- Initialize STKPTR to 0x10.
- Configure PORTB as interrupt inputs.
- Clear interrupt flag bits of INT0 and INT1.
- Set INT1 as low-priority interrupt.
- Enable global HIGH and LOW interrupts.
- Turn both LEDs at PORTD OFF (comparator LED at bit 0 of PORTD and switch LED at bit 1 of PORTD)
- Wait in an infinite loop for one or both interrupts to occur.

Also, write a service routine for the high-priority interrupt (INT0) in PIC18F assembly language at address 0x200 that will perform the following:

- Clear interrupt flag for INT0.
- Turn LED on at bit 0 of PORTD.

Finally, write a service routine for the low-priority interrupt (INT1) in PIC18F assembly language at address 0x300 that will perform the following:

- Clear interrupt flag for INT1.
- Turn LED on at bit 1 of PORTD.

Solution

This example will demonstrate the interrupt priority scheme of the PIC18F4321 microcontroller. With interrupt priority, the user has the option to have the interrupts declared as either low or high interrupts. If, at anytime, the low- and high-priority interrupts occur at the same time, the microcontroller will always service the high-priority interrupt.

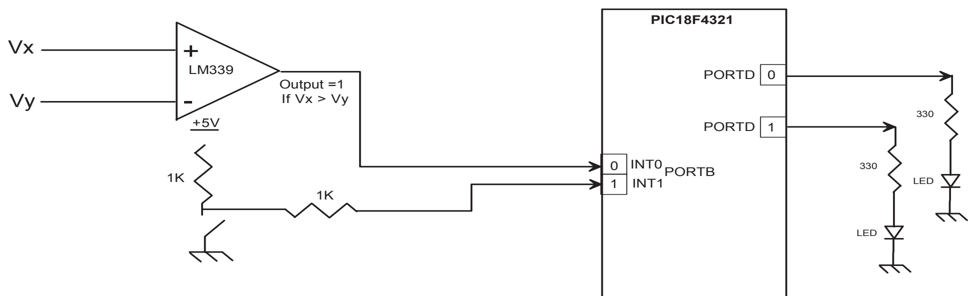


FIGURE 8.22 Figure for Example 8.5

In the above example, the comparator is set as the high-priority and the switch is set as the low-priority, so if both interrupts are triggered simultaneously, then only the LED associated with the comparator will be turned ON.

Note that the external interrupt INT0 can only be a high-priority interrupt. When implementing a single interrupt, the interrupt service routine is written at address 0x08. On the other hand, when priority interrupts are enabled, the service routine for the high-priority interrupt is written at address 0x08 while the service routine for the low-priority interrupt is written at address 0x018.

In order to enable the second external interrupt INT1, the register INTCON3 is configured. Also, INT1IE must be enabled, and INT1IF must be cleared to 0. Furthermore, the INT1IP bit in INTCON3 register that sets the priority of INT1 in the INTCON3 register must be cleared to 0 for low priority. Next, the IPEN bit in the RCON register that enables the interrupt priority functionality of the PIC18F4321 must be set to one. Finally, the GIEH and GIEL bits in the INTCON register must be set to one in order to enable global high and low interrupts. The following code implements priority interrupts on the PIC18F4321 using assembly language:

```
INCLUDE <P18F4321.INC>
; RESET
              ORG 0                         ; Reset vector
              GOTO      MAIN                ; Jump to main program
; HIGH PRIORITY INTERRUPT ADDRESS VECTOR
              ORG       0x0008              ; High-priority interrupt
              BRA       HIGH_INT_ISR        ; Jump to service routine for the comparator
; LOW PRIORITY INTERRUPT ADDRESS VECTOR
              ORG       0x0018              ; Low-priority interrupt
              BRA       LOW_INT_ISR         ; Jump to service routine for the switch
                                            ; Main Program
              ORG       0x0100
MAIN          MOVLW     0x10                ; Initialize STKPTR to 0x10
              MOVWF     STKPTR
              CLRF      TRISD               ; PORTD is output
              MOVLW     0x0F                ; Configure ADCON1 to set up
              MOVWF     ADCON1              ; INT0 and INT1 as digital inputs
              BSF       INTCON, INT0IE      ; Enable the external interrupt INT0
              BSF       INTCON3,INT1IE      ; Enable the external interrupt INT1
              BCF       INTCON,INT0IF       ; Clear the INT0 flag
              BCF       INTCON3,INT1IF      ; Clear the INT1 flag
              BCF       INTCON3, INT1IP     ; Set INT1 as low priority
              BSF       RCON, IPEN          ; Enable interrupt priority
              BSF       INTCON, GIEH        ; Enable global high interrupts
              BSF       INTCON, GIEL        ; Enable global low interrupts
              CLRF      PORTD               ; Turn both LEDs off
OVER          BRA       OVER                ; Wait for interrupt
              BRA       MAIN                ; Halt

; SERVICE ROUTINE FOR HIGH PRIORITY
              ORG       0x200
HIGH_INT_ISR  BCF       INTCON,INT0IF       ; Clear the interrupt flag
```

```
                MOVLW   0x01            ; Turn on LED at bit 0 of PORTD
                MOVWF   PORTD
                RETFIE
; SERVICE ROUTINE FOR LOW PRIORITY
                ORG     0x300
LOW_INT_ISR     BCF     INTCON3, INT1IF ; Clear the interrupt flag
                MOVLW   0x02            ; Turn on LED at bit 1 of PORTD
                MOVWF   PORTD
                RETFIE
                END
```

8.4 PIC18F Interface to an LCD (Liquid Crystal Display)

Seven-segment LEDs are easy to use, and can display only numbers and limited characters. LCDs are very useful for displaying numbers, and several ASCII characters along with graphics. Furthermore, LCDs consume low power. Because of inexpensive price of LCDs these days, they have been becoming popular. LCDs are widely used in notebook computers.

Figure 8.23 shows the PIC18F4321's interface to a typical LCD display such as the Optrex DMC16249 LCD with a 2-line x 16-character display screen. In order to illustrate the basic concepts associated with LCDs, the phrase "Switch Value:" along with the numeric BCD value (0 through 9) of the four switch inputs will be displayed.

The Optrex DMC16249 LCD shown in Figure 8.23 contains 14 pins. The VCC pin is connected to +5 V and the VSS pin is connected to ground. The VEE pin is the contrast control for brightness of the display. VEE is connected to a potentiometer with a value between 10k and 20k. The eight data pins (D0-D7) are used to input data and commands to display the desired message on the screen.

The three control pins EN, R/$\overline{W}$, and RS allow the user to let the display know what kind of data is sent. The EN pin latches the data from the D0-D7 pins into the LCD

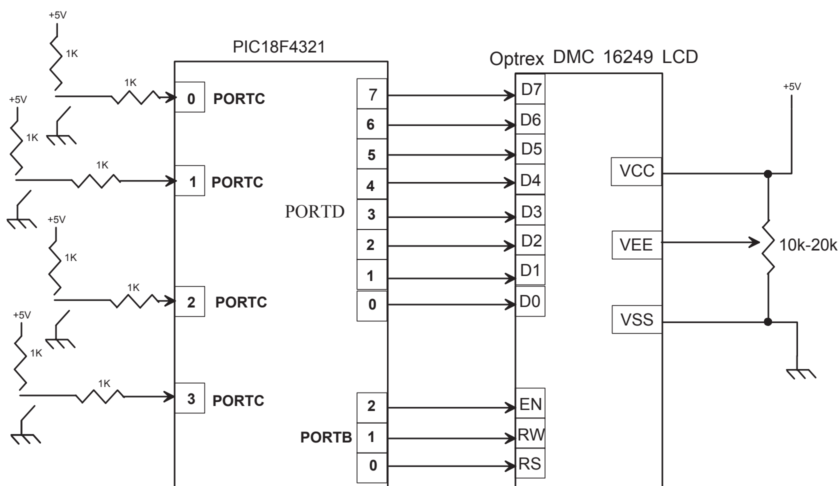


FIGURE 8.23 PIC18F4321 interface to Optrex DMC 16249 LCD

The PIC18F assembly language program is shown in Figure 8.24. Note that time delay rather than the busy bit is used before outputting the next character to the LCD. Three subroutines are used: one for outputting command code, one for delay, and one for LCD data. Since subroutines are used, the hardware STKPTR is initialized in the main program with an arbitrarily chosen value of 5. PORTB and PORTD are configured as output ports, and PORTC is set up as an input port. Also, assume 1-MHz default crystal frequency for the PIC18F4321.

As an example, let us consider the code for outputting a command code such as the command "move cursor to the beginning of the first line" to the LCD. From Table 8.4, the command code for this is 0x80. From Figure 8.24, the code MOVLW 0x80 moves 0x80 into WREG. The CALL CMD calls the subroutine CMD. The CMD subroutine first outputs the command code 0x80 to PORTD using MOVWF PORTD. Since PORTD is connected to LCD's D0-D7 pins, these data will be available to be latched by the LCD. The following few lines of the code of the CMD subroutine are for outputting 0's to RS and $R/\overline{W}$ pins, and a trailing edge (1 to 0) pulse to EN pin along with a delay of 20 msec. Hence, the LCD will latch 0x80, and the cursor will move to the start of the first line. Note that an external counter of 10 loaded into a register 0x21 with a 2 msec inner loop for LOOP2 is used for the 20 msec delay. Typical delays should be 10 to 30 milliseconds. Also, 1 MHz default crystal frequency for the PIC18F4321 is assumed. The program then returns to the main program.

The first few lines of the main program at address MAIN perform initializations. Next, in order to display 'S', the MOVLW D'10' moves 10 (decimal) into WREG, and CALL DELAY provides 20 msec delay using this value in the routine. After executing the DELAY routine, MOVLW A'S' moves the 8-bit ASCII code for S into WREG. The instruction CALL LCDDATA calls the subroutine LCDDATA. The MOVWF PORTD instruction in this subroutine outputs the ASCII code for S into the D0-D7 pins of the LCD via PORTD. The next few instructions in the LCDDATA subroutine outputs 1 to the RS pin (for selecting LCD data register to display data), 0 to the $R/\overline{W}$ pin, and a trailing edge (1 to 0) pulse to EN pin along with delay so that the LCD will latch ASCII code for S, and will display S on the screen.

Similarly, the program logic in Figure 8.24 for outputting other ASCII characters and switch input data can be explained.

The PIC18F assembly language program is provided in Figure 8.24 as follows.

For 1 MHz default crystal frequency, the PIC18F clock period will be 1 μsec. Hence, each instruction cycle will be 4 microseconds. For 2 msec delay, total cycles = (2 msec)/(4 μsec)= 500. The DECFSZ in the loop will require 2 cycles for (COUNT - 1) times when Z = 0 and the last iteration will take 1 cycle when skip is taken (Z = 1). Thus, total cycles including the MOVLW = 1 + 1 + 1 + 2 × (COUNT - 1) + 2 = 500. Hence, COUNT will be approximately 255 (decimal), discarding execution times of certain instructions.. Therefore, register 0x21 should be loaded with an integer value of 255 for an approximate delay of 2 msec.

8.5 Interfacing PIC18F4321 to a Hexadecimal Keyboard and a Seven-segment Display

In this section we describe the basics of interfacing the PIC18F4321 microcontroller to a hexadecimal keyboard and a seven-segment display.

```
            INCLUDE   <P18F4321.INC>
            ORG       0x100       ; Start of the MAIN program
MAIN        MOVLW     5           ; Initialize STKPTR with arbitrary value of 5
            MOVWF     STKPTR
            CLRF      TRISD       ; PORTD is output
            CLRF      TRISB       ; PORTB is output
            SETF      TRISC       ; PORTC is input
            CLRF      PORTB       ; rs=0 rw=0 en=0
            MOVLW     D'10'       ; 20 msec delay
            CALL      DELAY
            MOVLW     0x0C        ; Display on, Cursor off
            CALL      CMD
            MOVLW     D'10'       ; 20 msec delay
            CALL      DELAY
            MOVLW     0x01
            CALL      CMD         ; Clear Display
            MOVLW     D'10'       ; 20 msec delay
            CALL      DELAY
            MOVLW     0x06        ; Shift Cursor to the right
            CALL      CMD
            MOVLW     D'10'       ; 20 msec delay
            CALL      DELAY
            MOVLW     0x80        ; Move cursor to the start of the first line
            CALL      CMD
            MOVLW     D'10'       ; 20 msec delay
            CALL      DELAY
            MOVLW     A'S'        ; Send ASCII S
            CALL      LCDDATA
            MOVLW     A'w'        ; Send ASCII w
            CALL      LCDDATA
            MOVLW     A'i'        ; Send ASCII i
            CALL      LCDDATA
            MOVLW      A't'       ; Send ASCII t
            CALL      LCDDATA
            MOVLW     A'c'        ; Send ASCII c
            CALL      LCDDATA
            MOVLW     A'h'        ; Send ASCII h
            CALL      LCDDATA
            MOVLW     A' '        ; Send ASCII space
            CALL      LCDDATA
            MOVLW     A'V'        ; Send ASCII V
            CALL      LCDDATA
            MOVLW     A'a'        ; Send ASCII a
            CALL      LCDDATA
            MOVLW     A'l'        ; Send ASCII l
            CALL      LCDDATA
            MOVLW     A'u'        ; Send ASCII u
```

FIGURE 8.24 Assembly language program for the PIC18F4321-LCD interface

```
            CALL     LCDDATA
            MOVLW    A'e'      ; Send ASCII e
            CALL     LCDDATA
            MOVLW    A':'      ; Send ASCII :
            CALL     LCDDATA
AGAIN       MOVF     PORTC, W  ; Move switch value to WREG
            ANDLW    0x0F      ; Mask lower 4 bits
            IORLW    0x30      ; Convert to ASCII data by Oring with 0x30
            CALL     LCDDATA   ; Display switch value on screen
            MOVLW    0x10
            CALL     CMD
            BRA      AGAIN
CMD         MOVWF    PORTD     ; Command is sent to PORTD
            MOVLW    0x04
            MOVWF    PORTB     ; rs=0 rw=0 en=1
            MOVLW    D'10'     ; 20 msec delay
            CALL     DELAY
            CLRF     PORTB     ; rs=0 rw=0 en=0
            RETURN
LCDDATA     MOVWF    PORTD     ; Data sent to PORTD
            MOVLW    0x05      ; rs=1 rw=0 en=1
            MOVWF    PORTB
            MOVLW    D'10'     ; 20 msec delay
            CALL     DELAY
            MOVLW    0x01
            MOVWF    PORTB     ; rs=1 rw=0 en=0
            RETURN
DELAY       MOVWF    0x20
LOOP1       MOVLW    D'255'    ; LOOP2 provides 2 msec delay with a count of 255
            MOVWF    0x21
LOOP2       DECFSZ   0X21
            GOTO     LOOP2
            DECFSZ   0x20
            GOTO     LOOP1
            RETURN
            END
```

FIGURE 8.24 Assembly language program for the PIC18F4321-LCD interface (continued)

8.5.1 Basics of Keyboard and Display Interface to a Microcontroller

A common method of entering programs into a microcontroller is via a keyboard. An inexpensive way of displaying microcontroller results is by using seven-segment displays. The main functions to be performed for interfacing a keyboard are

- Sense a key actuation.
- Debounce the key.
- Decode the key.

Let us now elaborate on keyboard interfacing concepts. A keyboard is arranged

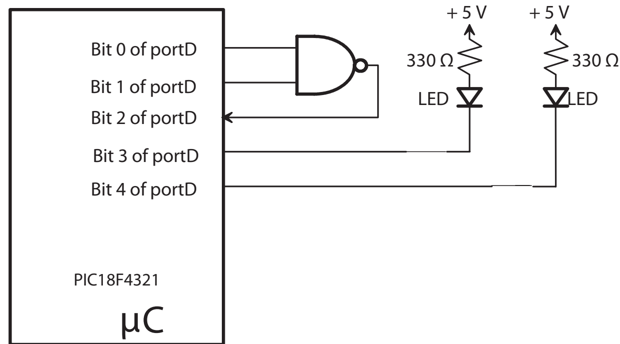


FIGURE P 8.9 (Assume that both LEDs are OFF initially.)

8.10 The PIC18F4321 microcontroller is required to add two 3-bit numbers stored in the lowest 3 bits of data registers 0x20 and 0x21 and output the sum (not to exceed 9) to a common-cathode seven-segment display connected to Port C as shown in Figure P8.10. Write a PIC18F assembly language program at address 0x200 to accomplish this by using a look-up table.

8.11 The PIC18F4321 microcontroller is required to input a number from 0 to 9 from an ASCII keyboard interfaced to it and output to an EBCDIC printer. Assume that the keyboard is connected to Port C and the printer is connected to Port D. Store the EBCDIC codes for 0 to 9 starting at an address 0x30, and use this lookup table to write a PIC18F assembly language program at address 0x100 to accomplish this.

8.12 In Figure P8.12, the PIC18F4321 is required to turn on an LED connected to bit 1 of Port C if the comparator voltage Vx > Vy; otherwise, the LED will be turned off. Write a PIC18F assembly language program at address 0x200 to accomplish this using conditional or polled I/O.

8.13 Repeat Problem 8.12 using Interrupt I/O by connecting the comparator output to INT1. Note that RB1 is also multiplexed with INT1. Write main program at 0x80 and interrupt service routine at 0x150 in PIC18F assembly language. The main program will configure the I/O ports, enable interrupt INT1, initialize STKPTR to 0x05, turn the LED OFF, and then wait for interrupt. The interrupt service routine will turn the LED ON and return to the main program at the appropriate location so that the LED is turned ON continuously until the next interrupt.

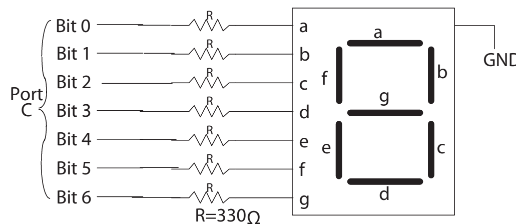


FIGURE P8.10

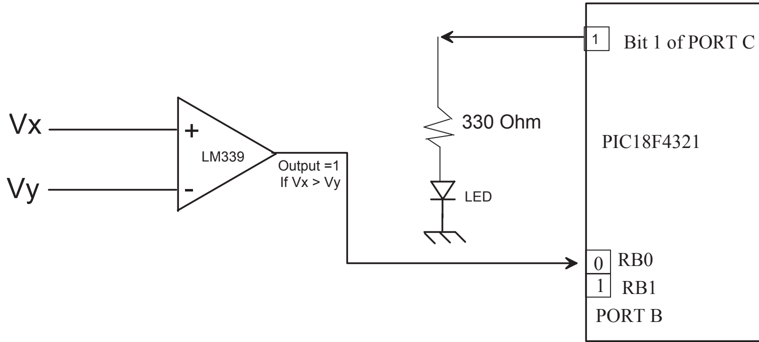


FIGURE P8.12

8.14 In Figure P8.14, if $V_x > V_w$, turn an LED ON connected at bit 3 of Port C. If $V_y > V_z$, turn the LED OFF. Assume that $V_x > V_w$ and $V_y > V_z$ will not occur at the same time. Using ports, registers, and memory locations as needed and INT0 interrupt:

(a) Draw a neat block diagram showing the PIC18F4321 microcontroller and the connections to ports in the diagram in Figure P8.14.

(b) Write the main program at 0x150 and the service routine at 0x200 in PIC18F assembly language. The main program will initialize STKPTR to 0x10, initialize the ports and wait for an interrupt. The service routine will accomplish the task and stop.

8.15 What is the interrupt address vector upon power-on reset?

8.16 Identify the PIC18F4321 external interrupts as maskable or nonmaskable.

8.17 What are the interrupt address vectors for high-priority and low-priority interrupts?

8.18 What are the priority levels for INT0 through INT2 external interrupts of the PIC18F4321 upon power-on reset?

8.19 What is the difference between PIC18F “RETFIE” and “RETFIE 1” instructions?

8.20 Write PIC18F instruction sequence tin PIC18F assembly language to set interrupt priority of INT1 as the high level, and interrupt priority for INT2 level as low level.

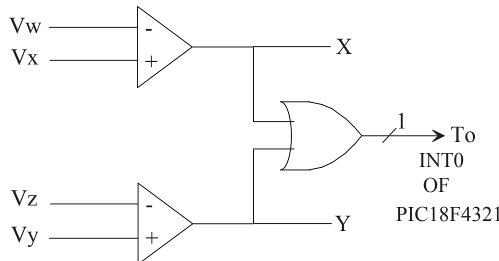


FIGURE P8.14

9

PIC18F HARDWARE AND INTERFACING: PART 2

In this chapter we describe the second part of hardware aspects of the PIC18F4321. Topics include PIC18F4321's on-chip timers, analog interfaces (ADC and DAC), serial I/O, and CCP (Capture/Compare/Pulse Width Modulation).

9.1 PIC18F Timers

The PIC18F microcontroller family contains four to five on-chip hardware timers. The PIC18F4321 microcontroller includes four timers, namely, Timer0, Timer1, Timer2, and Timer3. These timers can be used to generate time delays using on-chip hardware. Note that the basic hardware inside each of these timers is a register that can be incremented or decremented at the rising or falling edge of a clock. The register can be loaded with a count for a specific time delay. Time delay is computed by subtracting the initial starting count from the final count in the register, and then multiplying the subtraction result by the clock frequency.

These timers can also be used as event counters. Note that an event counter is basically a register with the clock replaced by an event such as a switch. The counter is incremented or decremented whenever the switch is activated. Thus the number of times the switch is activated (occurrence of the event) can be determined.

Finally, the PIC18F CCP module utilizes these timers to perform capture, compare, or PWM (pulse width modulation) functions. These topics will be discussed later in this chapter.

All PIC18F4321 timers have prescaler option. This means that the input clock frequency is divided by factors such as 2,4,8,16. The PIC18F4321 Timer2 also includes postscaler option. That is, Timer2 output clock frquency is divided by factors such a 2,4,8,16. Prescaler and Postscaler options can be used to obtain higher time delays Also, all timers can use either their respective timer flags or interrupts. The interrupt address vector is 0008H upon hardware reset.

Figure 9.4 (redrawn from Figure 8.17) shows the INTCON register with the TMR0IE and TMR0IF bits. The TMR0 interrupt is generated (if enabled by setting TMR0IE to one using BSF INTCON, TMR0IE) when the TMR0 register overflows from 0xFF to 0x00 in 8-bit mode, or from 0xFFFF to 0x0000 in 16-bit mode. This overflow sets the TMR0IF flag bit shown in Figure 9.4 (bit 2 of INTCON). The interrupt can be masked by clearing the TMR0IE bit. Before reenabling the interrupt, the TMR0IF bit must be cleared in software by the Interrupt Service Routine. Note that interrupt address vector is 0x0008 upon hardware reset.

In the 8-bit mode, the TMR0IF bit is set to one when the timer value in 8-bit register TMR0L is incremented from 0xFF to 0x00 (overflow). In the 16-bit mode, the TMR0IF bit is set to one when the timer value in 16-bit register TMR0H:TMR0L is incremented from 0xFFFF to 0x0000 (overflow). An extra clock is required when Timer0 rolls over from 0xFF to 0x00 in 8-bit mode or from 0xFFFF to 0x0000.

Example 9.1 Assuming a 4 MHz crystal oscillator, calculate the time delay for the following PIC18F instruction sequence:

```
MOVLW    0xD4
MOVWF    T0CON    ; Initialize T0CON with 0xD4
MOVLW    0x80     ; Load 8-bit timer with count 0x80
MOVWF    TMR0L
```

Solution

The above PIC18F instruction sequence loads 0xD4 into the T0CON register. Note that 0xD4 = 11010100_2. Hence, from Figure 9.1, the T0CON register can be drawn with the binary data, as shown in Figure 9. 5. Comparing data of Figure 9. 5 with data of Figure 9.1, the following results are obtained:
TMR0ON = 1 meaning TIMER0 is ON, T08BIT = 1 meaning 8-bit timer, T0CS = 0 meaning internal instruction clock, PSA = 0 meaning prescaler enabled, and TOPS2 TOPS1 TOPS0 = 100 meaning 1:32 prescale value.

Clock period = 1/(4 MHz) = 0.25 μ sec, Instruction cycle clock period = 4 x 0.25 μ sec = 1 μ sec. Since the prescaler multiplies the Instruction cycle clock period by the prescaler value,
Time Delay = (Instruction cycle clock perod) x (Prescaler value) x (Counter value)
= (1 μ sec) x (32) x (128) = 4096 μ sec = 4.096 msec

Note that, in the above, Counter value = 0x80 = 128 in decimal. This value determines the desired time delay. Also, the last two instructions, MOVLW and MOVWF, account for the two instruction cycles, during which the increment operation is inhibited before writing to the TMR0L register.

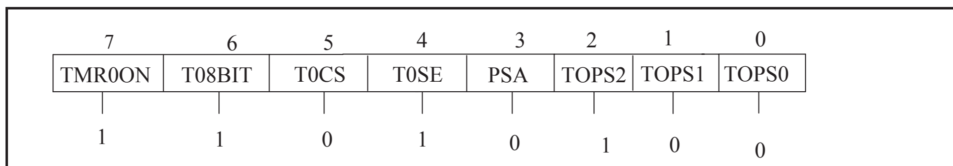


FIGURE 9.5 T0CON register with binary data 11010100_2

Example 9.2 Using Timer0 in 16-bit mode, write a PIC18F assembly language program to obtain a time delay of 1 ms. Assume 8 MHz crystal, and a prescale value of 1:128.

Solution

Since the timer works with divide by 4, crystal frequency = (8MHz)/4 = 2 MHz. Instruction cycle clock period = (1/2 MHz) = 0.5 μ sec.

The bits in register T0CON of Figure 9.1 are as follows:
TMR0ON(bit 7) = 0, T08BIT (bit 6) = 0, T0CS (bit 5) = 0, PSA (bit 3) = 0, and TOPS2 TOPS1 TOPS0 = 110 for a prescale value of 1:128. Hence, the T0CON register will be initialized with 0x06.

Time delay = Instruction cycle x Prescale value x Count
Hence, Count = (1 ms) / (0. 5 μ sec x 128) = 15.625 which can be approximated to an integer value of 16 (0x0010). The timer counts up from an initialized value to 0xFFFF, and then rolls over (increments) to 0000H. The number of counts for rollover is (0xFFFF - 0x0010) = 0xFFEF.

Note that an extra cycle is needed for the rollover from 0xFFFF to 0x0000, and the TMR0IF flag is then set to 1. Because of this extra cycle, the total number of counts for rollover = 0xFFEF + 1 = 0xFFF0.

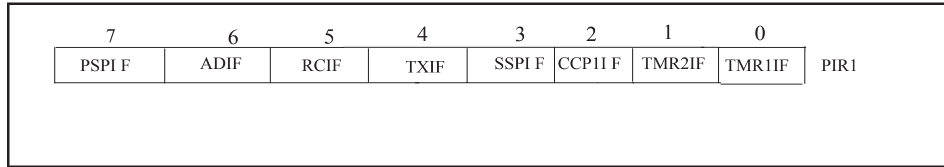
The following PIC18F assembly language program will provide a time delay of 1 ms:

```
            INCLUDE  <P18F4321.INC>
            MOVLW    0x06               ; Initialize T0CON
            MOVWF    T0CON
            MOVLW    0xFF               ; Initialize TMR0H first with 0xFF
            MOVWF    TMR0H
            MOVLW    0xF0               ; Initialize TMR0L next
            MOVWF    TMR0L
            BCF      INTCON, TMR0IF     ; Clear Timer0 flag bit
            BSF      T0CON, TMR0ON      ; Start Timer0
BACK        BTFSS    INTCON, TMR0IF     ; Check Timer0 flag bit for 1
            GOTO     BACK               ; Wait in loop
            BCF      T0CON, TMR0ON      ; Stop Timer0
FINISH      BRA      FINISH             ; Halt
            END
```

9.1.2 Timer1

The Timer1 can be used as a 16-bit timer or a counter. It consists of two 8-bit registers, namely, TMR1H and TMR1L. The TMR1IF flag in PIR1 register goes to HIGH when the Timer1 overflows from 0xFFFF to 0x0000. An extra cycle is required when the Timer1 rolls over from 0xFFFF to 0x0000.

Timer1 is controlled through the T1CON Control register (Figure 9.6). It also contains the Timer1 Oscillator Enable bit (T1OSCEN). Timer1 can be enabled or disabled by setting or clearing the TMR1ON (bit 0 of T1CON) control bit.



bit 7 **PSPIF:** Parallel Slave Port Read/Write Interrupt Flag bit
1 = A read or a write operation has taken place (must be cleared in software)
0 = No read or write has occurred

bit 6 **ADIF:** A/D Converter Interrupt Flag bit
1 = An A/D conversion completed (must be cleared in software),
0 = The A/D conversion is not complete

bit 5 **RCIF:** EUSART Receive Interrupt Flag bit
1 = The EUSART receive buffer, RCREG, is full (cleared when RCREG is read)
0 = The EUSART receive buffer is empty

bit 4 **TXIF:** EUSART Transmit Interrupt Flag bit
1 = The EUSART transmit buffer, TXREG, is empty (cleared when TXREG is written)
0 = The EUSART transmit buffer is full

bit 3 **SSPIF:** Master Synchronous Serial Port Interrupt Flag bit
1 = The transmission/reception is complete (must be cleared in software)
0 = Waiting to transmit/receive

bit 2 **CCP1IF:** CCP1 Interrupt Flag bit
Capture mode:
1 = A TMR1 (or TMR3) register capture occurred (must be cleared in software),
0 = No TMR1 (or TMR3) register capture occurred
Compare mode:
1 = A TMR1 (or TMR3) register compare match occurred (must be cleared in software)
0 = No TMR1 (or TMR3) register compare match occurred
PWM mode:
Unused in this mode.

bit 1 **TMR2IF:** TMR2-to-PR2 Match Interrupt Flag bit
1 = TMR2-to-PR2 match occurred (must be cleared in software), 0 = No TMR2-to-PR2 match occurred

bit 0 **TMR1IF:** TMR1 Overflow Interrupt Flag bit,
1 = TMR1 register overflowed (must be cleared in software)
0 = TMR1 register did not overflow

FIGURE 9.7 PIR1 (Peripheral Interrupt Request) Register1

Timer1 Interrupt Enable bit, TMR1IE (bit 0 of PIE1), shown in Figure 9.8. The other bits in the PIR1 and PIE1 registers contain the individual flag and enable bits for the peripheral interrupts. The interrupt address vector is 0x0008 upon hardware reset.

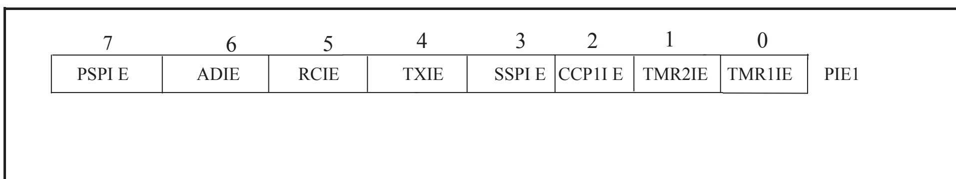
Example 9.3 Write a PIC18F assembly language program to provide a delay of 1 msec using Timer1 with an internal clock of 4 MHz. Use 16-bit mode of Timer1 and the prescaler value of 1:4.

Solution

For 4 MHz clock, each instruction cycle = 4 x (1/4 MHz) = 1 μ sec
Total instruction cycles for 1 msec delay = (1 x $10^{-3}/10^{-6}$) = 1000

With the prescaler value of 1:4, instruction cycles = 1000 / 4 = 250
Number of counts for rollover = 65535_{10} - 250_{10} = 65285_{10} = 0xFF05
An extra cycle is required for rollover from 0xFFFF to 0x0000 which sets the TMR1IF to 1.
Hence, total number of counts = 0xFF05 + 1 = 0xFF06
Therefore, TMR1H must be loaded with 0xFF, and TMR1L with 0x06.
The PIC18F assembly language program for one msec delay is provided below:

```
        INCLUDE   <P18F4321.INC>
        MOVLW     0xC0          ; 16-bit mode, 1:4 prescaler, Timer1 OFF
        MOVWF     T1CON         ; Load into T1CON register
        MOVLW     0xFF          ; Initialize TMR1H with 0xFF
        MOVWF     TMR1H
```



bit 7 **PSPIE:** Parallel Slave Port Read/Write Interrupt Enable bit
1 = Enables the PSP read/write interrupt
0 = Disables the PSP read/write interrupt

bit 6 **ADIE:** A/D Converter Interrupt Enable bit
1 = Enables the A/D interrupt
0 = Disables the A/D interrupt

bit 5 **RCIE:** EUSART Receive Interrupt Enable bit
1 = Enables the EUSART receive interrupt
0 = Disables the EUSART receive interrupt

bit 4 **TXIE:** EUSART Transmit Interrupt Enable bit
1 = Enables the EUSART transmit interrupt
0 = Disables the EUSART transmit interrupt

bit 3 **SSPIE:** Master Synchronous Serial Port Interrupt Enable bit
1 = Enables the MSSP interrupt
0 = Disables the MSSP interrupt

bit 2 **CCP1IE:** CCP1 Interrupt Enable bit
1 = Enables the CCP1 interrupt
0 = Disables the CCP1 interrupt

bit 1 **TMR2IE:** TMR2-to-PR2 Match Interrupt Enable bit
1 = Enables the TMR2-to-PR2 match interrupt
0 = Disables the TMR2-to-PR2 match interrupt

bit 0 **TMR1IE:** TMR1 Overflow Interrupt Enable bit
1 = Enables the TMR1 overflow interrupt
0 = Disables the TMR1 overflow interrupt

FIGURE 9.8 PIE1 (Peripheral Interrupt Enable) Register 1

```
        MOVLW   0x06            ; Initialize TMR1L with 0x06
        MOVWF   TMR1L
        BCF     PIR1, TMR1IF    ; Clear Timer1 overflow flag in PIR1
        BSF     T1CON, TMR10N   ; Start timer
        BSF     T1CON, TMR1ON
BACK    BTFSS   PIR1, TMR1IF    ; If TMR1IF=1, skip next instruction to halt
        BCF     T1CON, TMR10N   ; Stop timer
        GOTO    BACK
HERE    BRA     HERE            ; Halt
        END
```

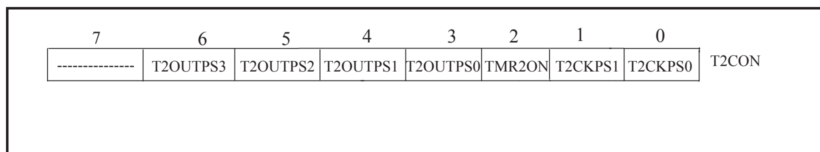
Note that external loop can be used with the above 1 msec delay routine as the inner loop to obtain higher time delays.

9.1.3 Timer2

Timer2 contains an 8-bit timer register and 8-bit period register (TMR2 and PR2). The Timer2 can only be programmed as a timer (not as a counter) with prescale values of 1:1, 1:4, and 1:16, and postscale values of 1:1 through 1:16.

The module is controlled through the T2CON register shown in Figure 9.9. The T2CON register enables or disables the timer and configures the prescaler and postscaler. Timer2 can be shut off by clearing control bit, TMR2ON (bit 2 of T2CON), to minimize power consumption.

Timer2 Operation In normal operation, the PR2 register is initialized to a specific value, and the 8-bit timer register (TMR2) is incremented from 0x00 on each internal clock (FOSC/4). A 4-bit counter/prescaler on the clock input gives direct input, divide-by-4 and divide-by-16 prescale options. These are selected by the prescaler control bits, T2CKPS1:T2CKPS0 (bits 1, 0 of T2CON). The value of TMR2 is compared to that of the



bit 7 **Unimplemented:** Read as ‘0’

bit 6-3 **T2OUTPS3:T2OUTPS0:** Timer2 Output Postscale Select bits
0000 = 1:1 postscale
0001 = 1:2 postscale
•
•
•
1111 = 1:16 postscale

bit 2 **TMR2ON:** Timer2 On bit
1 = Timer2 is on
0 = Timer2 is off

bit 1-0 **T2CKPS1:T2CKPS0:** Timer2 Clock Prescale Select bits
00 = Prescaler is 1
01 = Prescaler is 4
1x = Prescaler is 16

FIGURE 9.9 T2CON (Timer2 Control) Register

8-bit period register, PR2, on each clock cycle. When the two values match, the Timer2 outputs a HIGH on the TMR2IF flag in the PIR1 register, and also resets the value of TMR2 to 0x00 on the next cycle.The output frequency is divided by a counter/postscale value (1:1 to 1:16) as specified in the T2CON register. Note that the interrupt is generated and the TMR2IF flag bit in the PIR1 register (Figure 9.7) is set to 1, indicating the match between TMR2 and PR2 registers. The TMR2IF bit must be cleared to 0 using software.

Timer2 Interrupt Timer2 can also generate an optional device interrupt. The Timer2 output signal (TMR2-to-PR2 match) provides the input for the 4-bit output counter/postscaler. This counter generates the TMR2 match interrupt flag which is latched in TMR2IF (bit 1 of PIR1, Figure 9.7). The interrupt is enabled by setting the TMR2 Match Interrupt Enable bit, TMR2IE (bit 1 of PIE1, Figure 9.8). A range of 16 postscale options (from 1:1 through 1:16 inclusive) can be selected with the postscaler control bits, T2OUTPS3:T2OUTPS0 (bits 6-3 of T2CON, Figure 9.9).

Example 9.4 Write a PIC18F assembly language program using Timer2 to turn on an LED connected at bit 0 of Port D after 10 sec. Assume an internal clock of 4 MHz, a prescaler value of 1:16, and a postscaler value of 1:16.

Solution

For 4 MHz clock, each instruction cycle = 4 x 1/(4MHz) = 1 μ sec. TMR2 is incremented every 1 μ sec. When the TMR2 value matches with the value in PR2, the value in TMR2 is cleared to 0 in one instruction cycle. Since the PR2 is 8-bit wide, we can have a maximum PR2 value of 255. Let us calculate the delay with this PR2 value.

Delay = (Instruction cycle) x (Prescale value) x (Postscale value) x (PR2 value + 1)
= (1 μ sec) x (16) x (16) (255 + 1)
= 65.536 msec

Note that, in the above, one is added to the PR2 value since an additional clock is needed when it rolls over from 0xFF to 0x00, and sets the TMR2IF to 1.

External counter value for 10 sec delay using 65.536 msec as the inner loop = (10 sec)/(65.536 msec), which is approximately 153 in decimal.

The PIC18F assembly language is provided below:

```
         INCLUDE  <P18F4321.INC>
EXT_CNT  EQU      0x50
         BCF      TRISD, 0         ; Configure bit 0 of PORT D as an output
         BCF      PORTD, 0         ; Turn LED OFF
         MOVLW    0x7A             ; 1:16 prescaler, 1:16 postscaler Timer1 off
         MOVWF    T2CON            ; Load into T2CON register
         MOVLW    0x00             ; Initialize TMR2 with 0x00
         MOVWF    TMR2
         MOVLW    D'153'           ; Initialize EXT_CNT with 153
         MOVWF    EXT_CNT
LOOP     MOVLW    D'255'           ; Load PR2 with 255
         MOVWF    PR2
         BCF      PIR1, TMR2IF     ; Clear Timer2 interrupt flag in PIR1
```

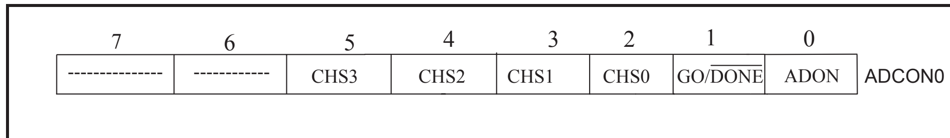
energy (electrical current, in this case) to another form (air pressure, in this example).

9.2.1 On-chip A/D Converter

The PIC 18F4321 contains an on-chip A/D converter (or sometimes called ADC) module with 13 channels (AN0-AN12). An analog input can be selected as an input on one of these 13 channels, and can be converted to a corresponding 10-bit digital number. Three control registers, namely, ADCON0 through ADCON2, are used to perform the conversion.

The ADCON0 register, shown in Figure 9.14, controls the operation of the A/D module. The ADCON0 register can be programmed to select one of 13 channels using bits CHS3 through CHS0 (bits 5 through 2). The conversion can be started by setting the GO/$\overline{\text{DONE}}$ (bit 1) to 1. Once the conversion is completed, this bit is automatically cleared to 0 by the PIC18F4321.

The ADCON1 register, shown in Figure 9.15, configures the functions of the port pins as Analog (A) input or Digital (D) I/O. The table shown in Figure 9.15 shows how the port bits are defined as analog or digital signals by programming the PCFG3 through PCFG0 bits (bits 3 through 0) of the ADCON1 register. This register can also be



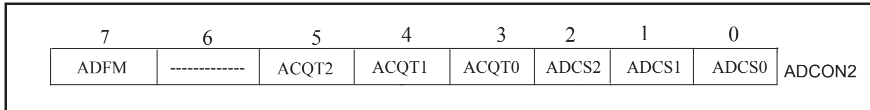
bit 7-6 **Unimplemented:** Read as '0'
bit 5-2 **CHS3:CHS0:** Analog Channel Select bits
0000 = Channel 0 (AN0)
0001 = Channel 1 (AN1)
0010 = Channel 2 (AN2)
0011 = Channel 3 (AN3)
0100 = Channel 4 (AN4)
0101 = Channel 5 (AN5)
0110 = Channel 6 (AN6)
0111 = Channel 7 (AN7)
1000 = Channel 8 (AN8)
1001 = Channel 9 (AN9)
1010 = Channel 10 (AN10)
1011 = Channel 11 (AN11)
1100 = Channel 12 (AN12)
1101 = Unimplemented
1110 = Unimplemented
1111 = Unimplemented

bit 1 **GO/ $\overline{\text{DONE}}$:** A/D Conversion Status bit

When GO/$\overline{\text{DONE}}$ = 1:
1 = A/D conversion in progress
0 = A/D idle

bit 0 **ADON:** A/D On bit
1 = A/D converter module is enabled
0 = A/D converter module is disabled

FIGURE 9.14 ADCON0 (A/D Control Register0)



bit 7 **ADFM:** A/D Result Format Select bit
1 = Right justified; 10-bits in lower 2 bits of ADRESH with upper 6 bits as 0's and in 8 bits of ADRESL
0 = Left justified; 8-bit result in ADRESH and the contents of ADRESL are ignored. Used for 8-bit conversion.
bit 6 **Unimplemented:** Read as '0'
bit 5-3 **ACQT2:ACQT0:** A/D Acquisition Time Select bits
111 = 20 TAD
110 = 16 TAD
101 = 12 TAD
100 = 8 TAD
011 = 6 TAD
010 = 4 TAD
001 = 2 TAD
000 = 0 TAD**(1)**
bit 2-0 **ADCS2:ADCS0:** A/D Conversion Clock Select bits
111 = FRC (clock derived from A/D RC oscillator)**(1)**
110 = FOSC/64
101 = FOSC/16
100 = FOSC/4
011 = FRC (clock derived from A/D RC oscillator)**(1)**
010 = FOSC/32
001 = FOSC/8
000 = FOSC/2
Note 1: If the A/D FRC clock source is selected, a delay of one TCY (instruction cycle) is
added before the A/D clock starts. This allows the SLEEP instruction to be executed before starting a conversion.

FIGURE 9.16 ADCON2 (A/D Control Register 2)

The following steps should be followed to perform an A/D conversion:

1. Configure the A/D module:
- Configure analog pins, voltage reference, and digital I/O (ADCON1)
- Select A/D input channel (ADCON0)
- Select A/D acquisition time (ADCON2)
- Select A/D conversion clock (ADCON2)
- Turn on A/D module (ADCON0)

2. Configure A/D interrupt (if desired):
- Clear ADIF bit (bit 6 of PIR1, Figure 9.7)
- Set ADIE bit (bit 6 of PIE1, Figure 9.8)
- Set GIE bit (bit 7) and PEIE (bit 6) of INTCON register, Figure 8.17(a)
- All interrupts including A/D Converter interrupt, branch to address 0x000008 (default) upon power-on reset. However, the A/D Converter interrupt can be configured as low priority by setting the ADIP bit (bit 6) of the IPRI register (See Microchip manual) to branch to address 0x000018. The instruction, BSF IPR1, ADIP can be used for this purpose.

3. Wait for the required acquisition time (if required).
4. Start conversion:

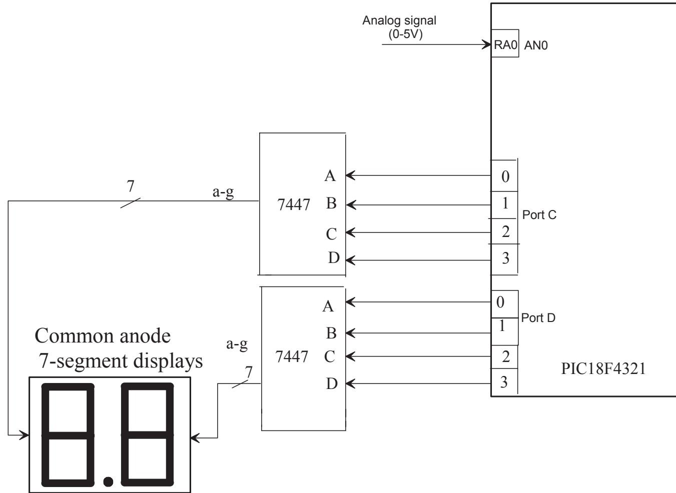


FIGURE 9.18 Figure for Example 9.5

Example 9.5 A PIC18F4321 microcontroller shown in Figure 9.18 is used to implement a voltmeter to measure voltage in the range 0 to 5 V and display the result in two decimal digits: one integer part and one fractional part. Using polled I/O, write a PIC18F assemble language program to accomplish this.

Solution

In order to design the voltmeter, the PIC18F4321 on-chip A/D converter will be used. Three registers, ADCON0-ADCON2, need to be configured. In ADCON0, bit 0 of PORT A (RA0/AN0) is designated as the analog signal to be converted. Hence, CHS3-CHS0 bits (bits 5-2) are programmed as 0000 to select channel 0 (AN0). The ADCON0 register is also used to enable the A/D, start the A/D, and then check the "end of conversion" bit. In the PIC18F assembly language program provided below, the ADCON0 is loaded with 0x01 which will select AN0, and enable A/D.

The reference voltages are chosen by programming the ADCON1 register. In this example, V_{DD} (by clearing bit 4 of of ADCON1 to 0), and V_{SS} (by clearing bit 5 of ADCON1 to 0) will be used. Note that V_{DD} and V_{SS} are already connected to the PIC18F4321. The ADCON1 register is also used to configure AN0 (bit 0 of Port A) as an analog input by writing 1101 (13 decimal in Figure 9.15) at PCFG3-PCFG0 (bits 3-0 of ADCON1). Note that there are several choices to configure AN0 as an analog input. In the program, the ADCON1 is loaded with 0x0D which will select VSS and VDD as reference voltage sources, and AN0 as analog input.

In the program, the ADCON2 is loaded with 0x29 which will provide the 8-bit result left justified (when 8 bits are obtained from 10 bits, the Lower two bits in ADRESL are discarded), select 12 TAD (requires at least 11 TAD for 10-bit conversion), and select Fosc/8.

The ADCON2 is used to set up the acquisition time, conversion clock, and, also, if the result is to be left or right justified. In this example, 8-bit result is assumed. The A/D result is configured as left justified, and, therefore, the 8-bit register ADRESH will contain the result. The contents of ADRESL are ignored.

Note that the maximum decimal value that can be accommodated in 8 bits of

ADRESH is 255_{10} (FF_{16}). Hence, the maximum voltage of 5 V will be equivalent to 255_{10}. This means that 1 volt = 51 (decimal). The display (D) in decimal is given by

$$D = 5 \times (\text{input}/255)$$
$$= \text{input}/51$$
$$= \underbrace{\text{quotient}}_{\text{Integer part}} + \text{remainder}$$

This gives the integer part. The fractional part in decimal is

$$F = (\text{remainder}/51) \times 10$$
$$\simeq (\text{remainder})/5$$

For example, suppose that the decimal equivalent of the 8-bit output of A/D is 200.

$$D = 200/51 \Rightarrow \text{quotient} = 3, \text{ remainder} = 47$$

$$\text{integer part} = 3$$

$$\text{fractional part}, F = 47/5 = 9$$

Therefore, the display will show 3.9 V.

From these equations, the final result will be in BCD. Both integer and fractional parts of the result will be output to two 7447s (BCD to seven-segment decoder) in order to display them on two seven-segment displays arranged in a row, as shown in Figure 9.18. The PIC18F assembly language program for the voltmeter is provided below:

```
              INCLUDE <P18F4321.INC>
D0            EQU     0x30          ;Contains data for right (fractional) 7-seg
D1            EQU     0x31          ;Contains data for left (integer) 7-seg
ADCONRESULT   EQU     0x34          ;Contains 8-bit A/D result
              ORG     0x100         ;Starting address of the program
              MOVLW   0x12          ;Initialize STKPTR to 0x12 (arbitrary value)
              MOVWF   STKPTR        ;Since subroutines are used
              CLRF    TRISC         ;Configure PortC  as output
              CLRF    TRISD         ;Configure PortD  as output
              MOVLW   0x01
              MOVWF   ADCON0        ;Select AN0 for input and enable ADC
              MOVLW    0x0D
              MOVWF   ADCON1        ;Select VDD and VSS as reference
                                    ;voltages and AN0 as analog input.
              MOVLW   0x29
              MOVWF   ADCON2        ;Select left justified, 12TAD and Fosc/8
START         BSF     ADCON0, GO    ;Start A/D conversion
INCONV        BTFSC   ADCON0, DONE  ;Wait until A/D conversion is done
              BRA     INCONV
```

```
                MOVFF   ADRESH,ADCONRESULT  ;Move ADRESH of result into
                                            ;ADCONRESULT register
                CALL    DIVIDE              ;Call the divide subroutine
                CALL    DISPLAY             ;Call display subroutine
                BRA     START
DIVIDE          CLRF    D0                  ;Clears  D0
                CLRF    D1                  ;Clears D1
                MOVLW    D'51'              ;#1 Load 51 into WREG
EVEN            CPFSEQ  ADCONRESULT         ;#2
                BRA     QUOTIENT            ;#3
                INCF    D1, F               ;#4
                SUBWF    ADCONRESULT, F     ;#5
QUOTIENT        CPFSGT  ADCONRESULT         ;#6 Checks if ADCONRESULT
                                            ;still greater than 51
                BRA     DECIMAL             ;#7
                INCF    D1, F               ;#8 Increment D1 for each time
                                            ;ADCONRESULT is greater
                                            ;than 51
                SUBWF   ADCONRESULT, F      ;#9 Subtract 51 from
                                            ;ADCONRESULT
                BRA     EVEN                ;#10
DECIMAL         MOVLW   0x05                ;#11
REMAINDER       CPFSGT  ADCONRESULT         ;#12 Checks if ADCONRESULT
                                            ;greater than 5
                BRA     DIVDONE             ;#13
                INCF    D0, F               ;#14 Increment D0
                SUBWF   ADCONRESULT, F      ;#15 Subtract 5
                                            ;from ADCONRESULT
                BRA     REMAINDER
DIVDONE         RETURN                      ;#16
DISPLAY         MOVFF   D1, PORTC           ;#17 Output D1  on integer 7-seg
                MOVFF   D0, PORTD           ;#18 Output D0  on fractional 7-seg
                RETURN
                END
```

In the above, since the PIC18F does not have any unsigned division instruction, a subroutine called DIVIDE is written to perform unsigned division using repeated subtraction. In the DIVIDE subroutine, the output of the A/D contained in the ADCONRESULT register is subtracted by 51. Each time the subtraction result is greater than 51, the contents of register D1 (address 0x31) is incremented by one, this will yield the integer part of the answer. Once the contents of the ADCONRESULT reaches a value below 51, the remainder part of the answer is determined. This is done by subtracting the number in ADCONRESULT subtracted by 5. Each time the subtraction result is greater than 5, register D0 (address 0x30) is incremented by one. Finally, the integer value is placed in D1 and the remainder part is placed in D0. Now the only task left is to display the result on the seven-segment display.

The # symbol along with a number in the comment field is used in some of the lines in the above program in order to explain the program logic. Line#1 moves 51 (decimal)

Write a PIC18F assembly language at 0x70 for the master PIC18F4321 that will configure PORTB and PORTC, initialize STKPTR to 0x10, initialize SSPSTAT and SSPCON1, input switches, and call a subroutine called SERIAL_WRITE to place this data into its SSPBUF register.

Also, write a PIC18F assembly language program at 0x100 for the slave PIC18F4321 that will configure PORTC and PORTD, initialize SSPSTAT and SSPCON1 registers, input data from its SDI pin, places the data in the slave's SSPBUF, and then output to the LEDs.

Solution

The PIC18F assembly language programs for the master PIC18F4321 and the slave PIC18F4321 in Figure 9.24 are written using the following steps as the guidelines:

Master PIC18F4321

1. Configure PORTB as input and SD0 and SCK as outputs.
2. Select CKE (SPI clock select bit) using the master's SSPSTAT register.
3. Enable serial functions, select master mode with clock such as fosc/4 using the SSPCON1 register.
4. Input switches into WREG, and then CALL a subroutine called SERIAL_WRITE to move switch data into the master's Serial Buffer register (SSPBUF).
5. Wait in a loop, and check whether BF bit in the master's SSPSTAT register is 1, indicating completion of transmission.
6. As soon as BF = 1, the program returns from the subroutine, and branches to Step 4.

Slave PIC18F4321

1. Initialize SDI and SCK pins as inputs, and PORTD as output. Note that the SCK is controlled by the master, and, therefore, it is configured as an input by the slave.
2. Select CKE same as the master CKE (high to low clock in this example) using the slave's SSPSTAT register.
3. Enable serial functions, disable the $\overline{SS}$ pin, and select slave mode using the slave's SSPCON1 register. Note that the $\overline{SS}$ pin is used by multiple slaves.
4. Wait in a loop, and check whether BF = 1 in the slave's SSPSTAT register.
5. If BF = 0; wait. However, if BF = 1, output the contents of the slave's Serial Buffer register (SSPBUF) to slave's PORTD.
6. Go to Step 5.

Figure 9.24 provides the PIC18F assembly language programs for the master and the slave microcontrollers.

Let us now explain the program of Figure 9.24. First, consider the master PIC18F4321. The CKE bit (bit 6) in the SSPSTAT is set to one so that data transmission will occur from an active to an idle (HIGH to LOW) clock. Next, the register SSPCON1 is configured in order to set up the parameters for serial transmission. The bit SSPEN (bit 5) in the SSPCON1 is set to HIGH in order to enable the three pins, namely, SCK, SDO, and SDI. Writing 0000 to bits 3-0 of the SSPCON1 register define the master mode operation with a clock of Fosc/4.

```
;Program for  the master PIC18F4321
              INCLUDE <P18F4321.INC>
              ORG     0x00          ;Reset
              GOTO    MAIN
              ORG     0x70
MAIN          BCF     TRISC, RC5    ;Configure RC5/SD0  as output
              BCF     TRISC, RC3    ;Configure RC3/SCK as output
              MOVLW   0x0F
              MOVWF   ADCON1        ;Make PORTB digital input
              MOVLW   0x10          ;Initialize STKPTR to 0x10 since subroutine
              MOVWF   STKPTR        ;called SERIAL_WRITE is used in the
                                    ;program
              MOVLW   0x40
              MOVWF   SSPSTAT       ;Set data transmission on high to low clock
              MOVLW    0x20
              MOVWF    SSPCON1      ;Enable serial functions and set to
                                    ; master device, and  Fosc/4
GET_DATA      MOVF    PORTB,W       ;Move switch value to WREG
              CALL    SERIAL_WRITE  ;Call SERIAL_WRITE function
              BRA     GET_DATA
SERIAL_WRITE  MOVWF   SSPBUF        ;Move switch value to serial buffer
WAIT          BTFSS   SSPSTAT, BF   ;Wait until transmission is complete
              BRA     WAIT
              RETURN
              END
; Program for the slave PIC18F4321
              INCLUDE <P18F4321.INC>
              ORG     0x00          ;Reset
              GOTO    MAIN
              ORG     0x100
MAIN          BSF     TRISC, RC4    ;Configure RC4/SDI  as input
              BSF     TRISC, RC3    ;Configure RC3/SCK as input
              CLRF    TRISD         ;Configure PORTD as output
              MOVLW    0x40
              MOVWF    SSPSTAT      ;Set data transmission on high to low clock
              MOVLW    0x25
              MOVWF    SSPCON1      ;Enable serial functions and set to the slave
WAIT          BTFSS    SSPSTAT, BF  ;Wait until transmission is complete (BF=1)
              BRA     WAIT          ;If BF=0, wait
              MOVFF   SSPBUF, PORTD ;Output serial buffer data to PORTD LEDs
              BRA     WAIT
              END
```

FIGURE 9.24 PIC18F assembly language program for Example 9.7

The following PIC18F instructions accomplish this:

```
MOVLW    0x20
MOVWF    SSPCON1   ;Enable serial functions and set to master and Fosc/4
```

TABLE 9.1 Assignment of timers for the PIC18F4321 CCP mode

CCP mode selected	Timer
Capture mode	Timer1 or Timer3
Compare mode	Timer1 or Timer3
PWM mode	Timer2

is determined by the Timer to CCP enable bits in the T3CON register (Figure 9.10). Both modules may be active at any given time and may share the same timer resource if they are configured to operate in the same mode (Capture, Compare, or PWM) at the same time. The assignment of the timers is summarized in Table 9.1.

9.4.3 PIC18F4321 Capture Mode

In Capture mode, the CCPRxH:CCPRxL register pair captures the 16-bit value of the TMR1 or TMR3 registers when an event (such as every rising or falling edge) occurs on the corresponding CCPx pin. The event is selected by the mode select bits, CCPxM3:CCPxM0 (bits 3-0 of CCPxCON, Figure 9.25). When a capture is made, the interrupt request flag bit, CCPxIF (PIR1 register in Figure 9.7), is set; it must be cleared in software. If another capture occurs before the value in register CCPRx is read, the old captured value is overwritten by the new captured value.

In Capture mode, the appropriate CCPx pin (RC2/CCP1/P1A, pin 17 or RC1/T1OSI/CCP2, pin 16) of the PIC18F4321 should be configured as an input by setting the corresponding TRIS direction bit. Also, the timers that are to be used with the capture feature (Timer1 and/or Timer3) must be running in Timer mode or Synchronized Counter mode. In Asynchronous Counter mode, the capture operation will not work. The timer to be used with each CCP module is selected in the T3CON register (Figure 9.10).

When the Capture mode is changed, a false capture interrupt may be generated. The user should keep the CCPxIE interrupt enable bit clear to avoid false interrupts. The interrupt flag bit, CCPxIF, should also be cleared following any such change in operating mode.

In summary, the following steps can be used to program the PIC18F4321 in capture mode to determine the period of a waveform (assume CCP1; similar procedure for CCP2):

1. Load the CCP1CON register (Figure 9.25) with appropriate data for capture mode.
2. Configure RC2/CCP1/P1A as an input pin using the TRISC register.
3. Select Timer1 and/or Timer3 by loading appropriate data respectively into T1CON register (Figure 9.6) and/or T3CON register (Figure 9.10).
4. Clear the interrupt request flag, CCP1IF for CCP1 (Register PIR1 of Figure 9.7) or CCP2IF for CCP2 (Register PIR2 of Figure 9.11), after a capture so that the next capture can be made.
5. Clear the interrupt enable bit, CCP1IE for CCP1 (Register PIE1 of Figure 9.8) or CCP2IE for CCP2 (Register PIE2 of Figure 9.12), to avoid false interrupts.
6. Clear CCPR1H and CCPR1L to 0.
7. Check CCP1IF flag in PIR1 and wait in a loop until CCP1IF is 1 for the first rising edge. As soon as the first rising edge is detected, start Timer1 (or Timer3).
8. Save CCPR1H and CCPR1L in data memory such as REGX and REGY.
9. Clear CCP1IF to 0.
10. Check CCP1IF flag in PIR1 and wait in a loop until CCP1IF is 1 for the second rising edge. As soon as the second rising edge is detected, stop Timer1 (or Timer3).

11. Disable capture by clearing CCP1CON register.
12. Perform 16-bit subtraction: [CCPR1H:CCPR1L] - [REGX:REGY].
13. 16-bit result in register pair [REGX:REGY] will contain the period of the incoming waveform in terms of the number of clock cycles.

Typical applications of the capture mode include:

- measurement of the pulse width of an unknown periodic signal by capturing the subsequent leading (rising) and trailing (falling) edges of a pulse.
- measurement of the period of a signal by capturing two subsequent leading or trailing edges.
- measurement of duty cycle . Note that the duty cycle is defined as (t1/T) x 100 where t1 is the fraction of the time the signal is HIGH in a period T.

Example 9.8 Assume PIC18F4321. Write a PIC18F assembly language program at address 0x200 to measure the period (in terms of the number of clock cycles) of an incoming periodic waveform connected at RC2/CCP1/P1A pin. Store result in registers 0x21 (high byte) and 0x20 (low byte). Use Timer3, and capture mode of CCP1.

Solution

The PIC18F assembly language program is provided below:

```
        INCLUDE<P18F4321.INC>
        ORG     0x200
        MOVLW   B'00000101'       ;Select capture mode rising  edge
        MOVWF   CCP1CON
        BSF     TRISC, CCP1       ;Configure RC2/CCP1/P1A pin as input
        MOVLW   B'01000000'       ;Select TIMER3 as clock source for capture
        MOVWF   T3CON             ;Select TIMER3 internal clock, 1:1 prescale
                                  ;TIMER3 OFF
        BCF     PIE1, CCP1IE      ;Disable CCP1IE  to avoid false interrupt
        MOVLW   0X00
        MOVWF   CCPR1H            ;Clear CCPR1H to 0
        MOVWF   CCPR1L            ;Clear CCPR1L to 0
        BCF     PIR1, CCP1IF      ;Clear CCP1IF
WAIT    BTFSS   PIR1, CCP1IF      ;Wait for the first rising edge
        GOTO    WAIT
        BSF     T3CON, TMR3ON     ;Turn Timer3 ON
        MOVFF   CCPR1L, 0x20      ;Save CCPR1L in 0x20 at 1st rising edge
        MOVFF   CCPR1H, 0x21      ;Save  CCPR1H in 0x21 at 1st rising edge
        BCF     PIR1, CCP1IF      ;Clear CCP1IF
WAIT1   BTFSS   PIR1, CCP1IF      ;Wait for next rising edge
        GOTO    WAIT1
        BCF     T3CON, TMR3ON     ;Turn OFF  Timer3
        CLRF    CCP1CON           ;Disable capture
        MOVF    0x20, W           ;Move 1st low byte to WREG
        SUBWF   CCPR1L, F         ;Subtract WREG from 2nd low byte
                                  ;Result in 0x20
        MOVF    0x21, W           ;Move 1st High byte to WREG
        SUBWFB  CCPR1H, F         ;Subtract  WREG with borrow
```

9. As soon as match occurs (CCP1IF or CCP2IF HIGH), stop Timer1 (or Timer3).

Example 9. 9 Assume PIC18F4321 with an internal crystal clock of 20 MHz. Write a PIC18F assembly language program at address 0x100 that will toggle the RC2/CCP1/P1A pin after a time delay of 10 msec. Use Timer3, and compare mode of CCP1.

Solution

With 20 MHz internal crystal, Fosc = 20 MHz. Since Timer3 uses Fosc/4, Timer clock frequency = Fosc/4 = 5 MHz. Hence, clock period of Timer3 = 0.2 μ sec. Counter value = (10 msec)/(0.2 μ sec) = $500_{10} = 01F4_{16}$. Hence, CCPR1H :CCPR1L should be loaded with 0x01F4 for the PIC18F4321 compare mode.

The PIC18F assembly language program is provided below:

```
        INCLUDE <P18F4321.INC>
        ORG     0x100
        MOVLW   0x02            ;Select compare mode, toggle CCP1 pin
        MOVWF   CCP1CON         ;on match
        BCF     TRISC, CCP1     ;Configure CCP1 pin as output
        MOVLW   0x40            ;Select TIMER3 as clock source for
                                ;compare
        MOVWF   T3CON           ;Select TIMER3 internal clock, 1:1 prescale
                                ;TIMER3 OFF
        MOVLW   0x01            ;Load CCPR1H with 0x01
        MOVWF   CCPR1H
        MOVLW   0xF4            ;Load CCPR1L  with 0xF4
        MOVWF   CCPR1L
        CLRF    TMR3H           ;Initialize TMR3H to 0
        CLRF    TMR3L           ;Initialize TMR3L to 0
        BCF     PIR1, CCP1IF    ;Clear CCP1IF
        BSF     T3CON, TMR3ON   ;Start Timer3
WAIT    BTFSS   PIR1, CCP1IF    ;Wait in a loop until CCP1IF is 1. CCP1 pin
        BRA     WAIT            ;toggles when match occurs
        BCF     T3CON, TMR3ON   ;Stop Timer3
HERE    BRA     HERE            ; Halt
        END
```

9.4.5 PIC18F4321 PWM (Pulse Width Modulation) Mode

In PWM mode, the CCPx pin can be configured as an output to generate a periodic waveform with a specified frequency, and a 10-bit duty cycle. Timer2 is used for the PWM mode. The PWM period is specified by writing to the 8-bit PR2 register in the CCP module. Note that PWM waveform can be generated using timers. However, it is easier to produce PWM waveform using the CCPx module.

The PWM period is specified by writing to the PR2 register. From the data sheet, the PWM period can be calculated using the following formula:

PWM Period = [(PR2) + 1] x 4 x Tosc x (TMR2 Prescale Value)

where Tosc = (1/Fosc), Fosc is the crystal frequency, and TMR2 Prescale Value can be

initialized as 1, 4, or 16 using the T2CON register.
Hence, PR2 = [(Fosc)/(4 x Fpwm x TMR2 Prescale Value)] - 1
Note that PWM frequency (Fpwm) is defined as 1/[PWM period].

The PWM duty cycle is specified by writing to the CCPRxL register and to the CCPxCON<5:4> bits. Up to 10-bit resolution is available. The CCPRxL contains the eight most significant bits, and the CCPxCON (bits 5 and 4) contains the two least significant bits. This 10-bit value is represented by CCPRxL:CCPxCON (bits 5 and 4). The following equation is used to calculate the PWM duty cycle in time:

PWM Duty Cycle = (CCPRXL:CCPXCON<5:4>) x Tosc x (TMR2 Prescale Value).

As mentioned before, the duty cycle is defined as the percentage of the time the pulse is high in a clock period. Note that the upper eight bits in the CCPRxL are the decimal part of the duty cycle while bits 5 and 4 of the CCPxCON register contain the fractional part of the duty cycle. For example, consider 25% duty cycle. Since duty cycle is a fraction of the PR2 register value, decimal value for the duty cycle with a PR2 value of 30 is 7.5 (0.25 x 30). Hence, the 8-bit binary number 00000111_2 must be loaded into CCPRxL, and $10_2(0.5_{10})$ must be loaded for DCxB1 and DCxB0 bits in the CCPxCON register (Figure 9.25).

The CCP1 PWM output waveform with the specified duty cycle in CCPR1L:CCPICON register is generated as follows: CCPR1L is copied to CCPR1H. Two bits of CCP1CON <5:4> are latched internally to provide 10-bit duty cycle. Also, 8-bit TMR2 value is concatenated with 2-bit internal latch to create 10-bit duty cycle. After TMR2 is started from 0, the CCP1 pin goes to HIGH to indicate the start of a cycle. The CCPR1H and 2-bit latch values are compared with TMR2 and 2-bit latch values for a match. As soon as the match occurs, the CCP1 pin goes to LOW. At this point, the waveform will be HIGH for the duration specified by the duty cycle. TMR2 keeps incrementing. As soon as the contents of TMR2 and PR2 match, CCP1 pin is driven to HIGH, and TMR2 is cleared to 0. This competes a cycle, and another cycle is then started. The same proess continues for subsequent cycles.

The following procedure should be followed when configuring the CCP module for PWM operation:

1. The PR2 register should be initialized with the PWM period.
2. Load the PWM duty cycle by writing to the CCPRxL register for higher eight bits, and bits 5, 4 of CCPxCON (Figure 9.25) for lower two bits.
3. Make the CCPx pin an output by clearing the appropriate TRIS bit.
4. Set the TMR2 prescale value, then enable Timer2 by writing to T2CON.
5. Initialize TMR2 register to 0.
6. Set up the CCPx module for PWM operation, and turn Timer2 ON.

Example 9.10 Write a PIC18F assembly language program at 0x100 to generate a 4 KHz PWM with a 50% duty cycle on the RC2/CCP1/P1A pin of the PIC18F4321. Assume 4 MHz crystal.

Solution

PR2 = [(Fosc)/(4 x Fpwm x TMR2 Prescale Value)] - 1
PR2 = [(4 MHz)/(4 x 4 KHz x 1)] - 1 assuming Prescale value of 1
PR2 = 249. With 50% duty cycle, duration for HIGH PWM waveform in a cycle = 0.5 x 249 = 124.5. Hence, the CCPR1L register will be loaded with 124, and bits DC1B1:DC0B0 (CCP1CON register) with 10 (binary).

The PIC18F assembly language program is provided below:

```
        INCLUDE   <P18F4321.INC>
        ORG       0x100
        MOVLW     D'249'           ;Initialize PR2 register
        MOVWF     PR2
        MOVLW     D'124'           ;Initialize CCPR1L
        MOVWF     CCPR1L
        MOVLW     0x20             ;CCP1 OFF,
        MOVWF     CCP1CON          ;DC1B1:DC0B0=10
        BCF       TRISC, CCP1      ;Configure CCP1 pin as output
        CLRF      T2CON            ;1:1 prescale, Timer2 OFF
        MOVLW     0x2C             ;PWM mode
        MOVWF     CCP1CON
        CLRF      TMR2             ;Clear Timer2 to 0
BACK    BCF       PIR1, TMR2IF     ;Clear TMR2IF to 0
        BSF       T2CON, TMR2ON    ;Turn Timer2 ON
WAIT    BTFSS     PIR1, TMR2IF     ;Wait until TMR2IF is HIGH
        GOTO      WAIT
        BRA       BACK
        END
```

9.5 DC Motor Control

Typical applications of the PWM mode include DC motor control. The speed of a DC motor is directly proportional to the driving voltage. The speed of a motor increases as the voltage is increased. In earlier days, voltage regulator circuits were used to control the speed of a DC motor. But voltage regulators dissipate lots of power. Hence, the PIC18F in the PWM mode is used to control the speed of a DC motor. In this scheme, power dissipation is significantly reduced by turning the driving voltage to the motor ON and OFF. The speed of the motor is a direct function of the ON time divided by the OFF time.

Sometimes, it is desirable to change direction of rotation of the DC motor. This can be accomplished by reversing the direction of the motor via software by interfacing a device called an H-Bridge to an I/O port of the PIC18F. Note that the speed of the motor, on the other hand, can be controlled using the PWM mode, and by connecting the DC motor to a PWM pin such as the PIC18F CCP1. The basic concepts associated with the DC motor control using the PIC18F4321's PWM mode will be illustrated in Example 9.11.

Microcontrollers such as the PIC18F4321 are not capable of outputting the required large current and voltage to control a typical DC motor. Hence, a driver such as the CNY17F Optocoupler is needed to amplify the current and voltage provided by the

PIC18F's output, and provide appropriate levels for the DC motor. One of the many useful applications for employing a PWM signal is its ability to control a mechanical device, such as a motor.

Note that the motor will run faster or slower based on the duty cycle of the PWM signal. The motor runs faster as the duty cycle of the PWM signal at the CCPx pin is increased. To illustrate this concept, two different duty cycles will be used in the following example (Example 9.11).

Example 9.11 Figure 9.26 shows a simplified diagram interfacing the PIC18F4321 to a DC motor via the CNY17F Optocoupler. The purpose of this example is to control the speed of a DC motor by inputting two switches connected at bit 0 and bit 1 of PORTD. The motor will run faster or slower based on the switch values (00 or 11), but will not provide any measure of the exact RPM of the motor.

When both switches are closed (00), a PWM signal at the CCP1 pin of the PIC18F4321 with 50% duty cycle will be generated. When both switches are open (11), a PWM signal at the CCP1 pin of the PIC18F4321 with 75% duty cycle will be generated. Otherwise, the motor will stop, and the program will wait in a loop.

If switches are closed (00), the motor will run using the 4 KHz PWM pulse of Example 9.10 with 50% duty cycle. If both switches are open (11), the motor will run using the same PWM pulse at a faster speed with a duty cycle of 75%. The program will first perform initializations, and wait in a loop until the switches are 00 or 11.
Write a PIC18F assembly language program to accomplish this.

Solution

The schematic of Figure 9.26 uses a CNY17F Optocoupler which serves two purposes. The first purpose is to protect the PIC18F4321 microcontroller by isolating the motor from the microcontroller. The second purpose the optocoupler serves is allowing the user to take a 0-5 V PWM signal and boost it to a 0-12 V source, where any voltage could be used that is safe for the optocoupler.

From Example 9.10, PR2 = 249. With 50% duty cycle, duration during which PWM is HIGH in a cycle = 0.5 x 249 = 124.5. Hence, the CCPR1L register will be loaded with 124, and bits DC1B1:DC0B0 (CCP1CON register) with 10_2.

With 75% duty cycle, duration during which PWM is HIGH in a cycle = 0.75 x 249 = 186.75. Hence, the CCPR1L register will be loaded with 186, and bits DC1B1:DC0B0 (CCP1CON register) with 11_2.

The PIC18F assembly language program is provided below:

```
        INCLUDE <P18F4321.INC>
        ORG     0x100
        MOVLW   D'249'          ;Initialize PR2 register
        MOVWF   PR2
        BCF     TRISC, CCP1     ;Configure CCP1 pin as output
        BSF     TRISD, RD0      ;Configure RD0 as an input bit
        BSF     TRISD, RD1      ;Configure RD1 as an input bit
        CLRF    T2CON           ;1:1 prescale, Timer2 OFF
        MOVLW   0x3C            ;PWM mode,DC1B1:DC0B0=11
        MOVWF   CCP1CON
```

FIGURE 9.26 Figure for Example 9.11

```
SWITCH   BTFSC   PORTD, RD0       ;If switch0 is  LOW, check switch1
                                  ;for LOW
         BRA     SWITCH1          ;If switch0 is HIGH,  branch to
                                  ;check switch1 for HIGH
         BTFSC   PORTD, RD1       ;If both switches are LOW, branch
                                  ;to DUTY50 and generate
                                  ;PWM with 50%  duty cycle;
         BRA     SWITCH           ;else, go back and wait
                                  ;Both switches are HIGH, go to
                                  ;75% duty cycle
         BRA     DUTY50           ;Both switches LOW, go to 50%
                                  ;duty cycle
SWITCH1  BTFSS   PORTD, RD1       ;If both switches are HIGH, branch
                                  ;to DUTY75, and generate PWM
                                  ;with 75% duty cycle;
         BRA     SWITCH           ;else, go back and wait
DUTY75   MOVLW   D'186'           ;For 75% duty cycle
         MOVWF   CCPR1L
         MOVLW   0x3C             ;PWM mode,DC1B1:DC0B0=11
         MOVWF   CCP1CON
         BRA     TIMER
DUTY50   MOVLW   D'124'           ;For 50% duty cycle
         MOVWF   CCPR1L
         MOVLW   0x2C             ;PWM mode,DC1B1:DC0B0=10
         MOVWF   CCP1CON          ;Initialize CCP1CON
TIMER    CLRF    TMR2             ;Clear Timer2 to 0
BACK     BCF     PIR1, TMR2IF     ;Clear TMR2IF to 0
         BSF     T2CON, TMR2ON    ;Turn Timer2 ON
WAIT     BTFSS   PIR1, TMR2IF     ;Wait until TMR2IF is HIGH
         BRA     WAIT             ;(end of period)
         BRA     SWITCH           ;Repeat to initialize and read switch
                                  ;inputs
         END
```

Questions and Problems

9.1 Find the contents of T0CON register to program Timer0 in 8-bit mode with 1:16 prescaler using the external clock, and incrementing on negative edge.

9.2 Write a PIC18F assembly language instruction sequence to initialize Timer0 as an 8-bit timer to provide a time delay with a count of 100. Assume 4 MHz internal clock with a prescaler value of 1:16.

9.3 Write a PIC18F assembly language program to generate a square wave with a period of 4 ms on bit 0 of PORTC using a 4 MHz crystal. Use Timer0.

9.4 Write a PIC18F assembly language program to generate a square wave with a period of 4 ms on bit 7 of PORTD using a 4 MHz crystal. Use Timer1.

9.5 Write a PIC18F assembly language program to turn an LED ON connected at bit 0 of PORTC with a PR2 value of 200. Assume a 4 MHz crystal. Use TMR2 prescaler and postscaler values of 1:1.

9.6 Write a PIC18F assembly language program to generate a square wave on pin 3 of PORTC with a 4 ms period using Timer3 in 16-bit mode with a prescaler value of 1:8. Use a 4 MHz crystal.

9.7 Repeat Example 9.5 using A/D converter's interrupt bit indicating completion of conversion. Use addresses, and other parameters of your choice.

9.8 Design and develop hardware and software for a PIC18F4321-based system (Figure P9.8) that would measure, compute, and display the Root-Mean-Square (RMS) value of a sinusoidal voltage. The system is required to:

1. Sample a 5 V (zero-to-peak voltage), 60 Hz sinusoidal voltage 128 times.

2. Digitize the sampled value using the on-chip ADC of the PIC18F4321 along with its interrupt upon completion of conversion signal.

3, Compute the RMS value of the waveform using the formula,
RMS Value = SQRT $\left[\sum_{n=1}^{N}(X_n^2)/N\right]$, where X_n's are the samples, and N

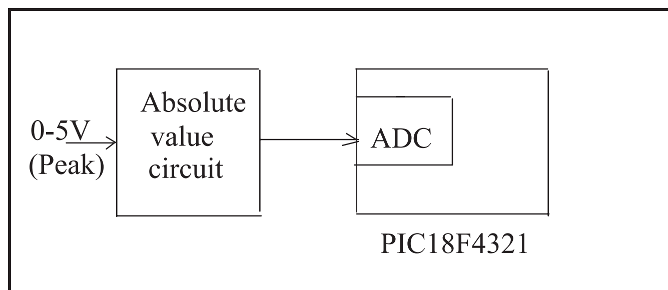


FIGURE P9.8

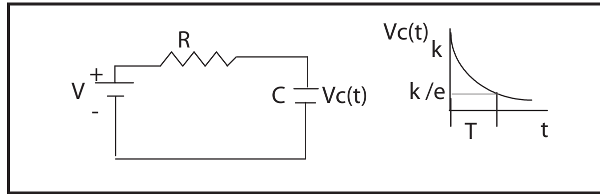


FIGURE P9.9

is the total number of samples**.** Display the RMS value on seven-segment displays.

(a) Flowchart the problem.

(b) Convert the flowchart to a PIC18F assembly language program.

9.9 *Capacitance meter.* Consider the RC circuit of Figure P9.9. The voltage across the capacitor is Vc (t) = k $e^{-t/RC}$. In one-time constant RC, this voltage is discharged to the value k/e. For a specific value of R, value of the capacitor C = T/R, where T is the time constant that can be counted by the PIC18F4321. Design the hardware and software for the PIC18F4321 to charge a capacitor by using a pulse to a voltage of your choice. The PIC18F4321 will then stop charging the capacitor, measure the discharge time for one time constant, and compute the capacitor value.

(a) Draw a hardware schematic.

(b) Write a PIC18F assembly language program to accomplish the above.

9.10 Design a PIC18F4321-based digital clock. The clock will display time in hours, minutes, and seconds. Write a PIC18F assembly language program to accomplish this.

9.11 Design a PIC18F4321-based system to measure the power absorbed by a 2K resistor (Figure P9.11). The system will input the voltage (V) across the 2K resistor, convert it to an 8-bit input using the PIC18F4321's on-chip A/D converter, and then compute the power using V^2/R. Write a PIC18F assembly language program to accomplish this.

9.12 Design a PIC18F4321-based system (Figure P9.12) as follows: The system will drive two seven-segment digits, and monitor two key switches. The system will start displaying 00. If the increment key is pressed, it will increment the display by one. Similarly, if the decrement key is pressed, the display will be decremented by

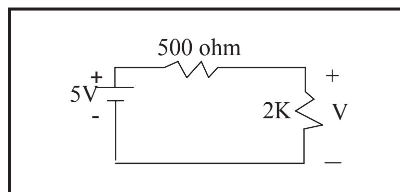


FIGURE P9.11

9.15 Assume PIC18F4321. Write a PIC18F assembly language program at address 0x200 that will measure the period of a periodic pulse train on the CCP1 pin using the capture mode. The 16-bit result will be performed in terms of the number of internal (Fosc/4) clock cycles, and will be available in the TMR1H:TMR1L register pair. Use 1:1 prescale value for Timer1.

9.16 Assume PIC18F4321. Write a PIC18F assembly language program at address 0x200 that will generate a square wave on the CCP1 pin using the Compare mode. The square wave will have a period of 20 ms with a 50% duty cycle. Use Timer1 internal clock (Fosc/4 from XTAL) with 1:2 prescale value. Assume 4-MHz crystal.

9.17 Write a PIC18F assembly language program at 0x100 to generate a 16 KHz PWM with a 75% duty cycle on the RC2/CCP1/P1A pin of the PIC18F4321. Assume 10 MHz crystal.

9.18 It is desired to change the speed of a DC motor by dynamically changing its pulse width using a potentiometer connected at bit 0 of PORTB (Figure P9.18). Note that the PWM duty cycle is controlled by the potentiometer. Write a PIC18F assembly language program that will input the potentiometer voltage via the PIC18F4321's on-chip A/D converter using interrupts, generate an 800-Hz PWM waveform on the CCP1 pin, and then change the speed of the motor as the potentiometer voltage is varied.
Assume 4MHz crystal and a TMR2 prescaler value of 16. Ignore fractional part of the duty cycle.

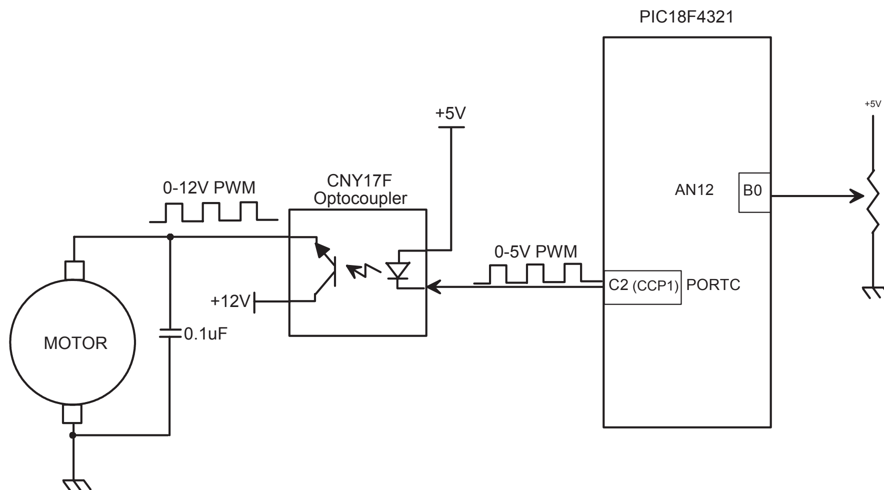


FIGURE P9.18

language. For example, a program written in C for a PIC18F microcontroller will run on the HC12 microcontroller because both microcontrollers have a compiler to translate the C language into their particular machine language; minor modifications are required for I/O programs. C is a high-level language that includes I/O instructions.

Compilers for microprocessors normally provide inefficient machine codes because of the general guidelines that must be followed for designing them. However, modern C compilers for microcontrollers generate very tight and efficient codes. C is widely used these days for writing programs for both real-time and non-real-time applications with microcontrollers. *Real time* indicates that the task required by the application must be completed before any other input to the program can occur that would change its operation.

The C Programming language was developed by Dennis Ritchie of Bell Labs in 1972. C has become a very popular language for many engineers and scientists, primarily because it is portable except for I/O and, however, can be used to write programs requiring I/O operations with minor modifications. This means that a program written in C for the PIC18F4321 will run on the Texas Instruments MSP430 with some modifications related to I/O as long as C compilers for both microcontrollers are available.

C is a general-purpose programming language and is found in numerous applications as follows:

- **Systems Programming.** Many operating systems (such as UNIX and its variant LINUX), compilers, and assemblers are written in C. Note that an operating system typically is included with the personal computer when it is purchased. The operating system provides an interface between the user and the hardware by including a set of commands to select and execute the software on the system.
- **Computer-Aided Design (CAD) Applications.** CAD programs are written in C. Typical tasks to be accomplished by a CAD program are logic synthesis and simulation.
- **Numerical Computation.** Software written in C is used to solve mathematical problems such as solving linear system of equations and matrix inversion. Industry standard MATLAB software is written in C.
- **Other Applications.** These include programs for printers and floppy disk controllers, and digital control algorithms such as PI (Proportional Integral) and PID (Proportional Integral Derivative) algorithms using microcontrollers.

A C program may be viewed as a collection of functions. Execution of a C program will always begin by a call to the function called "main." This means that all C programs should have its main program named as **main**. However, one can give any name to other functions.

A simple C program that prints "I wrote a C-program" is

```
/* First C-program */
# include <stdio.h>
void main ( )
{
        printf ("I  wrote a C-program") ;
}
```

$p \oplus (p \oplus q) = q$
Also, the above statements can be represented in compact form as:

```
y ^ = x;
x ^ = y;
y ^ = x;
```

Example 10.1 Write a C program to convert a 16-bit number, each byte containing an ASCII digit into a packed BCD byte.
Solution

```
# include   <p18f4321.h>
void        main  ()
   {
            unsigned char  a, b, c;
            unsigned char  addr1 = 0x32;
            unsigned char  addr2 = 0x31;
            unsigned char  addr3;
            addr1 = addr1 &  0x0f;          // Mask  off upper four bits of  the low byte
            addr2 =  addr2 & 0x0f;          // Mask  off upper four bits of the high byte
            addr2 =  addr2 << 4;            // Shift high byte  4 times to left
            addr3 = addr2 | addr1;          // Packed BCD byte in addr3
   }
```

10.4 Control Structures

Control structures allow programmers to modify control flow which is a sequential flow by default. Structures allow one to make decisions and create loops which make the hardware to replicate execution of statements. Typical structured control structures in C include `if-else`, `switch`, `while`, `for` and, `do-while`.

10.4.1 The `if-else` Construct

The syntax for the `if-else` construct is as follows:

```
if  (cond)
    statement1;
else
    statement2;
```

Figure 10.1 shows the flowchart for the `if-else` construct.

This is a one-entry-one-exit structure in that if the condition is true, the statement1 is executed; else (if the condition is false), statement1 is skipped, and the statement2 is executed. An example of the `if-else` structure (flowchart in Figure 10.2) is provided in the following:

B are multiplexed with analog inputs AN8 through AN12, and bits 0 through 2 of Port E are multiplexed with analog inputs AN5 through AN7 (Figure 8.1). When a port bit is multiplexed with an analog input, then bits 0-3 of a special function register (SFR) called ADCON1 (A/D Control Register 1 with mapped data memory address 0xFC1) must be used to configure the port bit as input. The other bits in ADCON1 are associated with the A/D converter. Figure 8.9 shows the ADCON1 register along with the associated bits for digital I/O. When bits 0 through 3 of the ADCON1 register are loaded with 1111, the analog inputs (AN0- AN12) multiplexed with the associated bits of Port A, Port B, and Port E are configured as digital I/O. This will also make these port bits as inputs automatically; the corresponding TRISx registers are not required to configure the ports. However, for configuring these ports as outputs, the corresponding TRISx bits must be loaded with 0's; the ADCON1 register is not required for configuring these port bits as outputs. The following examples will illustrate this.

For example, the following C statement will configure all 13 port bits multiplexed with AN0 - AN12 as inputs:

```
ADCON1 = 0x0F ; // Configure 13 bits of  Ports A, B, and E as inputs
```

Note that the TRISx registers associated with Ports A, B, and E can be used to configure these ports as outputs.

Next, in order to configure bit 1 of PORTA as output in PIC18F assembly language, the following instruction can be used:

```
BCF        TRISA, 1      ; Configure bit 1 of PORTA as output
```

The MPLAB C18 compiler provides built-in unions for configuring a port bit. This allows the programmer to address a single bit in a port without changing the other bits in the port. For example, bit 2 of Port C can be configured as an output by writing a '0' at bit 2 of TRISC as follows:

```
# define portbit  PORTCbits.RC2    // Declare a bit (bit 2) of  Port C
TRISCbits.TRISC2 = 0 ;           // Configure bit 2 of Port C as an output
```

Now, a '1' can be output to bit 2 of Port C using the following statement:

```
portbit = 1;
```

Similarly, the statement, `portbit = 0;` will output a '0' to bit 2 of Port C.

Next, bit 3 of Port D can be configured as an input by writing a '1' at bit 3 of TRISD as follows:

```
# define portbit  PORTDbits.RD3    // Declare a bit (bit 3) of  Port D
TRISDbits. TRISD3 = 1 ;          // Configure bit 3 of Port D as an input
```

Example 10. 2 Assume PIC18F4321. Suppose that three switches are connected to bits 0-2 of Port C and an LED to bit 6 of Port D. If the number of HIGH switches is even, turn

the LED on; otherwise, turn the LED off. Write a C language program to accomplish this.

Solution

The C language program is shown below:

```
#include <p18f4321.h>
#define    portc0  PORTCbits.RC0
#define    portc1  PORTCbits.RC1
#define    portc2  PORTCbits.RC2
void main (void)
{
        unsigned char mask = 0x07;              // Data for masking off upper 5 bits
                                                // of Port C
        unsigned char masked_in;
        unsigned char xor_bit;
        TRISC = 0xFF;                           // Configure Port C as an input port
        TRISD = 0;                              // Configure Port D as an output port
        while(1)
        {
            masked_in  =  PORTC & mask;          // Mask input  bits
            if (masked_in == 0)
            PORTD = 0x40;                       // For all low switches (even), turn led on
            else
            xor_bit = portc0 ^ portc1 ^ portc2; // Xor input bits
            if (xor_bit == 0)                   // For even # of high switches,
            PORTD = 0x40;                       // turn led on

        else
            PORTD = 0;                          // For odd # of high switches, turn led off
        }
}
```

Example 10.3 Assume PIC18F4321. Suppose that it is desired to input a switch connected to bit 4 of Port C, and then output it to an LED connected to bit 2 of Port D. Write a C language program to accomplish this.

Solution

The following C code will accomplish this:

```
# include  <P18F4321.h>
# define portc_bitin  PORTCbits.RC4     // Declare a bit (bit 4) of  Port C
# define portd_bitout  PORTDbits.RD2  // Declare a bit (bit 2) of  Port D
        void main  (void )
        {
        TRISCbits.TRISC4 = 1;           // Configure bit 4 of Port C as an input bit
        TRISDbits.TRISD2 = 0;           // Configure bit 2 of Port D as an output bit
        while  (1)                      //  Halt
                                        {
```

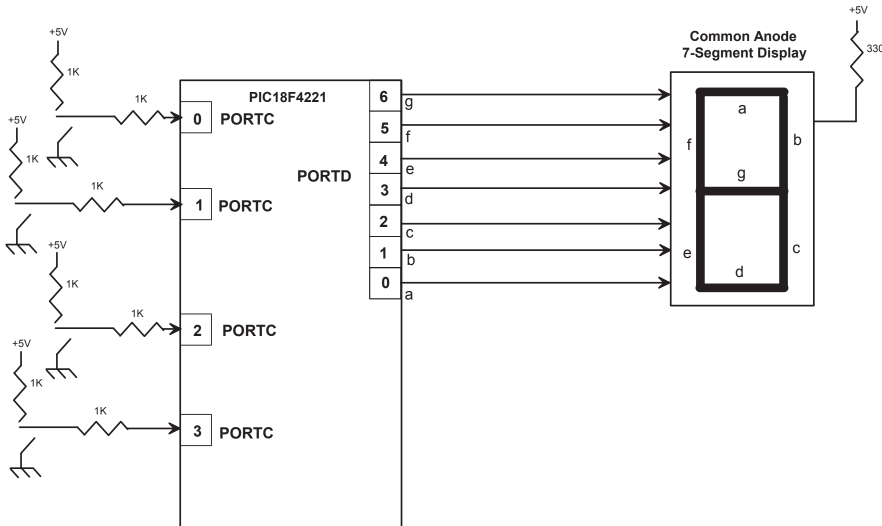


FIGURE 10.9 Figure for Example 10.4

```
portd_bitout = portc_bitin;      // Output switch to LED
                                 }
}
```

Example 10. 4 The PIC18F4321 microcontroller shown in Figure 10.9 is required to output a BCD digit (0 to 9) to a common-anode seven-segment display connected to bits 0 through 6 of Port D. The PIC18F4321 inputs the BCD number via four switches connected to bits 0 through 3 of Port C. Write a C language program that will display a BCD digit (0 to 9) on the seven-segment display based on the switch inputs.

Solution

The C code is provided below:

```
#include <p18f4321.h>
void main ()
unsigned char input;
unsigened char code[10] = {0x40, 0x79, 0x24, 0x30, 0x19, 0x12, 0x03, 0x78, 0x00, 0x18};
unsigned char oput;
TRISD = 0;          //Configure PortD as Output
TRISC = 0xFF;       //Configure PortC as Input
while (1) {
        input = PORTC & 0x0F;
        oput = code [input];
        PORTD = oput;
        }
```

In the above, the last two lines can be combined as PORTD = Code [input];

In the above program, first the PORTB is set as an output port and PORTC is set as an input port. A variable 'input' is then declared. The program moves to an infinite 'while' loop where it will first take the input from the four switches via PORTC, and mask the

In the above code, the register ADCON1 is used to configure Port B. Within the infinite `while` loop, the code checks to see when the comparator output is one indicating Vx > Vy. The LED is then turned ON or OFF based on the state of the switch.

10.9 Programming PIC18F4321 Interrupts Using C

The PIC18F4321 interrupts are covered in detail in Section 8.3 of chapter 8. The PIC18F4321 interrupts can be classified into two groups: high-priority interrupt levels and low-priority interrupt levels. The high-priority interrupt vector is at address 0x000008 and the low-priority interrupt vector is at address 0x000018 in the program memory. High-priority interrupt events will interrupt any low-priority interrupts that may be in progress.

As mentioned before, upon power-on reset, the interrupt address vector is 0x000008 (default), and no interrupt priorities are available. The IPEN bit (bit 7 of the RCON register) of the RCON register in Figure 8.5 can be programmed to assign interrupt priorities. Upon power-on reset, IPEN is automatically cleared to 0, and the PIC18F operates as a high-priority interrupt (single interrupt) system. Hence, the interrupt vector address is 0x000008. During normal operation, the IPEN bit can be set to one by executing the RCONbits.IPEN=1; to assign priorities in the system.

When interrupt priority is enabled (IPEN = 1), there are two bits which enable interrupts globally. Setting the GIEH bit (bit 7 of INTCON register of Figure 8.17) enables all interrupts that have the priority bit set (high priority). Setting the GIEL bit (bit 6 of INTCON register of Figure 8.17) enables all interrupts that have the low priority. When the interrupt flag, enable bit, and appropriate global interrupt enable bit are set, the interrupt will vector immediately to address 0x000008 or 0x000018, depending on the priority bit setting. Individual interrupts can be disabled through their corresponding enable bits.

Note that the C18 compiler does not allow the program to automatically jump to the interrupt service routine from the interrupt address vector. Hence, the PIC18F assembly language instructions `GOTO` or `BRA` must be used to jump to the interrupt service routine.

If interrupt priority levels are used, high-priority interrupt sources can interrupt a low priority interrupt. Low-priority interrupts are not processed while high-priority interrupts are in progress. The return address is pushed onto the stack and the PC is loaded with the interrupt vector address (0x000008 or 0x000018). Once in the Interrupt Service Routine, the source(s) of the interrupt must be determined for the priority interrupt system by polling the interrupt flag bits. The interrupt flag bits must be cleared in software before re-enabling interrupts to avoid recursive interrupts. In order to jump to the interrupt service routine from the interrupt address vector such as 0x000008 or 0x000018, the programmer should first check the interrupt flag bits to find the source of interrupt in a priority interrupt system, and then use the GOTO or BRA instruction of the assembly language to jump to the interrupt service routine.

Based upon discussion on interrupts in Section 8.3 of Chapter 8, INT0 can be initialized to recognize interrupts using the following C code:

```
ADCON1=0x0F;              //Configure PORTB to be digital input
                          //since PORTB contains interrupt pins
INTCONbits.INT0IE=1;      //Enable external interrupt
INTCONbits.INT0IF=0;      //Clear the external interrupt flag
INTCONbits.GIE=1;         //Enable global interrupts
```

Based upon detailed coverage of interrupts in Section 8.3 of Chapter 8, INT0 (High Priority) and INT1 (Low Priority) can be initialized to recognize interrupts using the following C code:

```
ADCON1=0x0F;                //Configure PORTB to be digital input
                            //Since PORTB contains interrupt pins
INTCONbits.INT0IE=1;        //Enable external interrupt INT0
INTCON3bits.INT1IE=1;       //Enable external interrupt INT1
INTCONbits.INT0IF=0;        //Clear INT0 external interrupt flag
INTCON3bits.INT1IF=0;       //Clear INT1 external interrupt flag
INTCON3bits.INT1IP=0;       //Set INT1 to low priority interrupt
RCONbits.IPEN=1;            //Enable priority interrupts
INTCONbits.GIEH=1;          //Enable global high priority interrupts
INTCONbits.GIEL=1;          //Enable global low priority interrupts
```

10.9.1 Specifying Interrupt Address Vector using the C18 Compiler

As mentioned before, using the MPLAB assembler, the programmer uses the ORG directive to specify the starting address of a program or data. Using the C18 C compiler, the programmer can use the directive `#pragma code begin` to specify an address to a program or to data at address `begin`. For example, the C statement `#pragma code int_vect = 0x000008` will assign the address 0x000008 to label `int_vect`. Note that `pragma` and `code` are keywords of the C18 compiler.

10.9.2 Assigning Interrupt Priorities Using the C18 Compiler

The C18 compiler uses the keywords `interrupt` and `interruptlow` to specify high- and low-priority interrupt levels. Note that the PIC18F interrupt address vectors for the high- and low-priority levels are 0x000008 and 0x000018 respectively. The programmers can use these keywords which allow a program to branch automatically from the respective interrupt address vector to a different program to find the source of the interrupt (for multiple interrupts with priorities), and then to the appropriate service routine.

10.9.3 A Typical Structure for Interrupt Programs Using C

The default interrupt INT0 with vector address 0x000008 is used in the following to illustrate the interrupt programs using C. Typical structures for the main program and the service routine are provided below:

```
#include  <P18F4321.h>
void        ISR (void);
#pragma code Int=0x08  //At interrupt code jumps here
void  Int(void)
{
_asm  //Using assembly language
GOTO ISR
_endasm
}
#pragma code //  End of code
void main( )
```

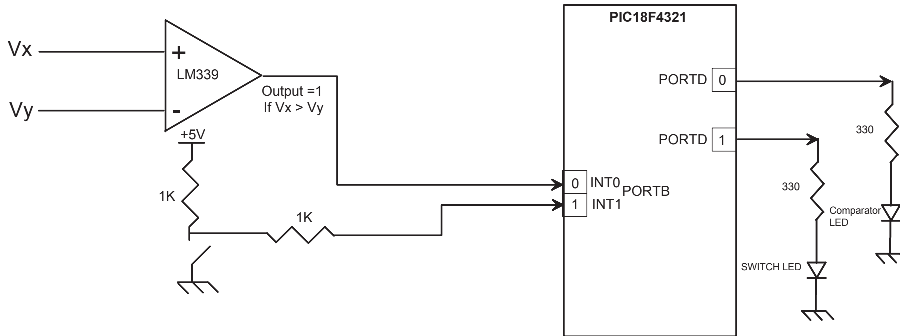


FIGURE 10.12 **Figure for Example 10.7**

```
        INTCONbits.INT0IF=0;  //Clear external interrupt flag
                while(PORTBbits.RB0==1){          //Check if comparator is high
                        PORTD = PORTB;            //Move PORTB into PORTD
        }
}
```

Example 10.7 In Figure 10.12, if Vx > Vy, the PIC18F4321 is interrupted via INT0. On the other hand, opening the switch will interrupt the microcontroller via INT1. Note that in the PIC18F4321, INT0 has the higher priority than INT1. Write the main program in C that will perform the following:

- Configure PORTB as interrupt inputs.
- Clear interrupt flag bits of INT0 and INT1.
- Set INT1 as low priority interrupt.
- Enable IPEN in RCON register.
- Enable global HIGH and LOW interrupts.
- Turn both LEDs at PORTD OFF.
- Wait in an infinite loop for one or both interrupts to occur.

Also, write a service routine for the high priority interrupt (INT0) in C that will perform the following:

- Turn LED on at bit 0 of PORTD.

Finally, write a service routine for the low priority interrupt (INT1) in C that will perform the following:

- Turn LED on at bit 1 of PORTD.

Solution

This example will demonstrate the interrupt priority system of the PIC18F microcontroller. Using interrupt priority, the user has the option to have various interrupts assigned as either low-priority or high-priority interrupts. If a low-priority interrupt and a high-priority interrupt occur at the same time, the PIC18F will always service the high priority interrupt first.

In the above example, the high priority is assigned to the comparator while the switch is assigned with the low priority. Hence, if both interrupts were triggered at the

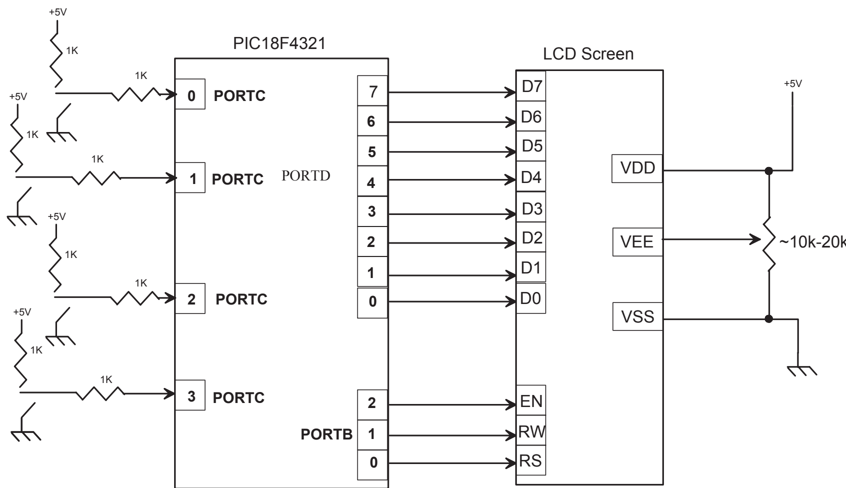


FIGURE 10.13 PIC18F4321 interface to LCD and switches

```
        PORTD=0;                    //LED is off
}

}
#pragma  interrupt  HP_COMP_ISR
void HP_COMP_ISR(void){                     //High-priority interrupt service
        INTCONbits.INT0IF=0;                //Clear external interrupt flag
{
                PORTD=0x01;                 //Turn on LED
        }
}

#pragma   interrupt low  LP_SWITCH_ISR
void LP_SWITCH_ISR(void){                   //Low-priority interrupt service
        INTCON3bits.INT1IF=0;               //Clear external interrupt flag
        PORTD=0x02;                         //Turn on LED

}
```

10.10 Programming the PIC18F4321 Interface to LCD Using C

The PIC18F4321 is interfaced to the Optrex DMC 16249 LCD in Section 8.4 of Chapter 8, and the programs are written using PIC18F assembly language. In this section, the same program will be written using C. For convenience, some of the concepts described in Chapter 8 will be repeated in this section.

Note that the PIC18F4321 is also interfaced to the seven-segment LED display in Chapter 8. The seven-segment LEDs are easy to use, and can display only numbers and limited characters. An LCD is very useful for displaying numbers and several ASCII

PORTB of PIC18F4321.

The complete LCD program in C is shown in the following. Note that time delay rather than the busy bit is used before outputting the next character to the LCD. Two functions are used: one for outputting command code, and the other for the delay. PORTB and PORTD are configured as input ports, and PORTC is set up as an input port. Also, assume 1-MHz default crystal frequency for the PIC18F4321.

As an example, let us consider the code for outputting a command code such as the command "move cursor to the beginning of the first line" (Start at line 1 position 0) to the LCD. From Table 9.1, the command code for this is 0x80. From the LCD program shown below, the statement `cmd(0x80);` will execute the following C code:

```
void cmd(unsigned char value)
{
        PORTD=value;            //Command is sent to PORTD
        PORTB=0x04;             //rs=0 rw=0 en=1
        delay(10);              //20msec delay
        PORTB=0x00;             //rs=0 rw=0 en=0
}
```

The above CMD function first outputs the value=0x80 to PORTD. Since PORTD is connected to LCD's D0-D7 pins, these data will be available to be latched by the LCD. The following few lines of the above code of the CMD function are for outputting 0's to RS and R/$\overline{W}$ pins, and a trailing edge (1 to 0) pulse to EN pin along with a delay of 20 msec. Hence, the LCD will latch 0x80, and the cursor will move to the start of the first line.

The following C loop will provide 2 msec delay:

```
void delay(unsigned int itime)      //2 msec delay
{
        unsigned int i,j;
        for(i=0; i<itime; i++)
                for(j=0; j<255;j++);
}
```

The above C code along with the statement `delay(10);` will provide 20 msec delay.

Similarly, the program logic (shown below) for outputting other ASCII characters and switch input data can be explained.

The complete LCD program using C is provided below:

```
#include <P18F4321.h>
void cmd(unsigned char);
void data(unsigned char);
void delay(unsigned int);
void main(void)
{
unsigned char input,output,i;
unsigned char tstr [13]={'S', 'w', 'i', 't', 'c', 'h', ' ', 'V', 'a', 'l', 'u', 'e', ':'}
```

```
        TRISD=0;                    //PORTD is output
        TRISB=0;                    //PORTB is output
        TRISC=0xFF;                 //PORTC is input
        PORTB=0x00;                 //rs=0 rw=0 en=0
        delay(10);                  //20msec delay
        cmd(0x0C);                  //Display On, Cursor Off
        delay(10);                  //20msec delay
        cmd(0x01);                  //Clear Display
        delay(10);                  //20msec delay
        cmd(0x06);                  //Shift cursor to the right
        delay(10);                  //20msec delay
        cmd(0x80);                  //Start at line 1 position 0
        delay(10);                  //20msec delay
for (i = 0; i<13; i++)
        data (tstr [i]);
while(1)
{
        input= PORTC&0x0F;          //Mask  switch value
        output=0x30 | input;        //Logically OR switch inputs with 0x30 to obtain
                                    //the ASCII code
        data(output);               //Display switch value on screen
        delay(10);                  //20msec delay
        cmd(0x10);                  //Shift cursor left one
}
}
void cmd(unsigned char value)
{
        PORTD=value;                //Command is sent to PORTD
        PORTB=0x04;                 //rs=0 rw=0 en=1
        delay(10);                  //20msec delay
        PORTB=0x00;                 //rs=0 rw=0 en=0
}

void data(unsigned char value)
{
        PORTD=value;                //Data sent to PORTD
        PORTB=0x05;                 //rs=1 rw=0 en=1
        delay(10);                  //20msec delay
        PORTB=0x01;                 //rs=1 rw=0 en=0
        }
void delay(unsigned int itime)      //2 msec delay
{
        unsigned int i,j;
        for(i=0; i<itime; i++)
                for(j=0; j<255;j++);
}
```

```
        T0CON=0x06;                    // Initialize T0CON
        TMR0H=0xFF;                    // Initialize TMR0H first with 0xFF
        TMR0L=0xF0;                    // Initialize TMR0L next
        INTCONbits.TMR0IF=0;           // Clear Timer0 flag bit
        T0CONbits.TMR0ON=1;            // Start Timer0
        while(INTCONbits.TMR0IF==0);   // Wait for Timer0 flag bit to be 1
        T0CONbits.TMR0ON=0;            // Stop Timer0
        while(1);                      // Halt
    }
```

Example 10.9 Write a C language program to provide a delay of 1 msec using Timer1 with an internal clock of 4 MHz. Use 16-bit mode of Timer1 and the prescale value of 1:4.

Solution

For 4 MHz clock, each instruction cycle = 4 x (1/4 MHz) = 1 μsec
Total instruction cycles for 1 msec delay = (1 x 10^{-3} /10^{-6}) = 1000
With the prescale value of 1:4, instruction cycles = 1000 / 4 = 250
Counter value = 65536_{10} - 250_{10} = 65286_{10} = 0xFF06
Hence, TMR1H must be loaded with 0xFF, and TMR1L with 0x06

The C language program for one msec delay is provided below:

```
#include<p18f4321.h>
void main(void)
{
        T1CON=0xC0;                    //16-bit mode, 1:4 prescaler, Timer1 OFF
        TMR1H=0xFF;                    //Initialize TMR1H with 0xFF
        TMR1L=0x06;                    //Initialize TMR1L with 0x06
        PIR1bits.TMR1IF=0;             //Clear Timer1 overflow flag in PIR1
        T1CONbits.TMR1ON=1;            //Start Timer1
        while(PIR1bits.TMR1IF==0);     //Wait for Timer1 interrupt to be 1
        T1CONbits.TMR1ON=0;            //Stop Timer1
        while(1);                      //Halt
}
```

Example 10.10 Write a C language program using Timer2 to turn an LED connected at bit 0 of PORT D after 10 sec. Assume an internal clock of 4 MHz. a prescale value of 1:16, and a postscale value of 1:16.

Solution

For 4 MHz clock, each instruction cycle = 4 x 1/(4MHz) = 1 μ sec. TMR2 is incremented every 1 μ sec. When the TMR2 value matches with the value in PR2, the value in TMR2 is cleared to 0 in one instruction cycle. Since, the PR2 is 8-bit wide, we can have a maximum PR2 value of 255. Let us calculate the delay with this PR2 value.

Delay = (Instruction cycle) x (Prescale value) x (Postscale value) x (PR2 value + 1)
= (1 μ sec) x (16) x (16) (255 + 1)
= 65.536 msec

Note that, in the above, one is added to the PR2 value since an additional clock is needed when it rolls over from 0xFF to 0x00, and sets the TMR2IF to 1.
External counter value for 10 sec delay using 65.536 msec as the inner loop = (10 sec)/(65.536 msec), which is approximately 153 in decimal.

The C language is provided below:

```
#include<p18f4321.h>
void main(void)
{
      unsigned char i;
      TRISDbits.TRISD0=0;            // Configure bit 0 of Port D as an output
      PORTDbits.RD0=0;               // Turn LED OFF
      T2CON=0x7A;                    // 1:16 prescaler, 1:16 postscaler Timer1 off
      TMR2=0x00;                     // Initialize TMR2 with 0x00
      for(i=0;i<153;i++)
      {
      PR2=255;                       // Load PR2 with 255
      PIR1bits.TMR2IF=0;             // Clear Timer2 interrupt flag in PIR1
      T2CONbits.TMR2ON=1;            // Set TMR2ON bit in T2CON to start timer
      while(PIR1bits.TMR2IF==0);     // Wait for Timer2 interrupt to be 1
      }
      PORTDbits.RD0=1;               // Turn LED ON
      T2CONbits.TMR2ON=0;            // Turn off Timer2
      while(1);                      // Halt
}
```

10.12 Programming the PIC18F4321 on-chip A/D Converter Using C

The PIC18F4321 on-chip A/D converter is covered in detail in Section 9.2.1 of Chapter 9. Also, the concepts associated with the PIC18F A/D converter interface were illustrated in Chapter 9 using examples in PIC18F assembly language.

In summary, the PIC18F4321 contains an on-chip A/D converter (or sometimes called ADC) module with 13 channels (AN0-AN12). An analog input can be selected as an input on one of these 13 channels, and can be converted to a corresponding 10-bit digital number. Three control registers, namely, ADCON0 through ADCON2, are used to perform the conversion. Example 10.11 illustrates these concepts by designing a voltmeter and writing the programs in C.

Example 10.11 A PIC18F4321 microcontroller shown in Figure 10.14 is used to implement a voltmeter to measure voltage in the range 0 to 5 V and display the result in two decimal digits: one integer part and one fractional part. Using polled I/O, write a C language program to accomplish this.

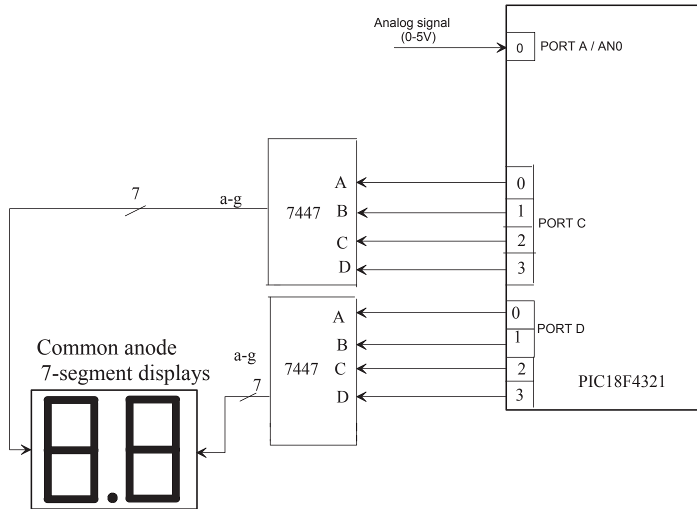


FIGURE 10.14 Figure for Example 10.11

Solution

In order to design the voltmeter, the PIC18F4321 on-chip A/D converter available will be used. Three registers, ADCON0-ADCON2, need to be configured. In ADCON0, bit 0 of Port A (RA0/AN0) is designated as the analog signal to be converted. Hence, CHS3-CHS0 bits (bits 5-2) are programmed as 0000 to select channel 0 (AN0). The ADCON0 register is also used to enable the A/D, start the A/D, and then check the “End of conversion” bit.

The reference voltages are chosen by programming the ADCON1 register. In this example, V_{DD} (by clearing bit 4 of of ADCON1 to 0) and V_{SS} (by clearing bit 5 of ADCON1 to 0) will be used. Note that V_{DD} and V_{SS} are already connected to the PIC18F4321. The ADCON1 register is also used to configure AN0 (bit 0 of Port A) as an analog input by writing 1101 at PCFG3-PCFG0 bits (bits 3-0 of ADCON1). Note that there are several choices to configure AN0 as an analog input.

The ADCON2 is used to set up the acquisition time, conversion clock, and, also, if the result is to be left or right justified. In this example, 8-bit result is assumed. The A/D result is configured as left justified, and, therefore, the 8-bit register ADRESH will contain the result. The contents of ADRESL are ignored. When 8 bits are obtained from 10 bits using left justified, lower two bits in ADRESL are discarded.

As mentioned in Chapter 9, because the maximum decimal value that can be accommodated in 8 bits of ADRESH is 255_{10} (FF_{16}), the maximum voltage of 5 V will be equivalent to 255_{10}. This means the display in decimal is given by

$$
\begin{aligned}
D &= 5 \times (\text{input}/255) \\
&= \text{input}/51 \\
&= \underbrace{\text{quotient}}_{\text{Integer part}} + \text{remainder}
\end{aligned}
$$

This gives the integer part. The fractional part in decimal is

$$F = (\text{ remainder}/51\ )\times 10$$

$$\simeq (\text{remainder })/5$$

For example, suppose that the decimal equivalent of the 8-bit output of A/D is 200.

$$D = 200/51 \Rightarrow \text{quotient } = 3, \text{ remainder } = 47$$

$$\text{integer part } = 3$$

$$\text{fractional part, } F = 47/5 = 9$$

Therefore, the display will show 3.9 V.

From these equations, the final result will be in BCD, which can then be sent to the 7447 decoders.

The integer value is placed in D1 and the remainder part is placed in D0. Finally, the result is displayed on the seven-segment displays.

The C language program for the voltmeter is provided below:

```
#include <P18F4321.h>
unsigned int FINAL,ADCONRESULT; //Initialize variables
unsigned char D1,D0;
void CONVERT(void);

void main ( )
{
TRISD = 0;              // Port D is Output
TRISC = 0;              // Port C is Output
ADCON0 = 0x01;          // Configure the ADC registers
ADCON1 = 0x0D;
ADCON2 = 0x29;
D0=0;                   // Data to display '0' on integer 7-seg display
D1=0;
while(1)                // Data to display '0' on fractional  7-seg display
{
ADCON0bits.GO = 1;      // Start the ADC

while(ADCON0bits.GO == 1); //Delay until conversion complete
```

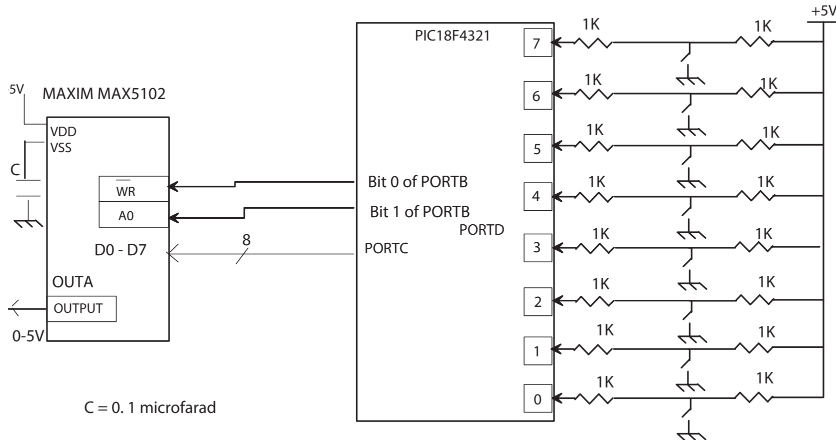


FIGURE 10.15 Figure for Example 10.12

```
        PORTC = D1;      // Output D1 to integer 7-segment display
        PORTD = D0;      // Output D0 to fractional 7-segment

ADCONRESULT = ADRESH;  // Move the ADC result into ADCONRESULT
FINAL = (ADCONRESULT*10)/51; // Conversion factor
CONVERT( );
}

}

void CONVERT( )
{
D1 = FINAL/10;
D0 = FINAL% 10;            //  D0 is remainder of  FINAL divided  by 10
}
```

10.13 Interfacing an External D/A (Digital-to-Analog) Converter Using C

As discussed in Chapter 9, most microcontrollers such as the PIC18F4321 do not have any on-chip D/A converter (or sometimes called DAC). Hence, an external D/A converter chip is interfaced to the PIC18F4321 to accomplish this function. Some microcontrollers such as the Intel/Analog Devices 8051 include an on-chip D/A converter. In order to illustrate the basic concepts associated with interfacing a typical D/A converter such as the Maxim MAX5102 was interfaced to the PIC18F4321 as discussed in Section 9.2.2 of Chapter 9.

Example 10.12 Assume the block diagram of Figure 10.15 . Write a C language program that will input eight switches via PORTD of the PIC18F4321, and output the byte to D0-D7 input pins of the MAX5102 D/A converter. The microcontroller will send appropriate signals to the $\overline{WR}$ and A0 pins so that the D/A converter will convert the input byte to an analog voltage between 0 and 5 V, and output the converted voltage on its OUTA pin.

```
        while (SSPSTATbits.BF == 0);       // Wait for transmission to finish
}
```

The following code is used on the slave PIC18F4321 device:

```
#include <p18f4321.h>

void main (void)
{
TRISCbits.TRISC4 = 1;           // RC4 is input
TRISCbits.TRISC3 =1;            // RC3 is input
TRISD=0x00;                     // PORTD is output
SSPSTAT= 0x40;                  // Transmission occurs on high to low clock
SSPCON1 = 0x25;                 // Enable serial functions and disable the  slave device
        while(1){
        while (SSPSTATbits.BF == 0);       // Wait for transmission to finish
        PORTD=SSPBUF;                      // Move serial buffer to PORTD
        }
}
```

Note that the above program is explained thoroughly in Example 9.7 of Chapter 9.

10.15 Programming the PIC18F4321 CCP Modules Using C

As mentioned in Chapter 9, the CCP module is implemented in the PIC18F4321 as an on-chip feature to provide measurement and control of time-based pulse signals. The basic concepts associated with the PIC18F CCP are explained in Chapter 9. Some of them will be repeated here for convenience.

Capture mode causes the contents of an internal 16-bit timer to be written in special function registers upon detecting an nth rising or falling edge of a pulse. Compare mode generates an interrupt or change on output pin when Timer1 matches a preset comparison value. PWM mode creates a re-configurable square wave duty cycle output at a user set frequency. The application software can change the duty cycle or period by modifying the value written to specific special function registers.

The PIC18F4321 contains two CCP modules, namely, CCP1 and CCP2. The CCP1 module of the PIC18F4321 is implemented as a standard CCP with enhanced PWM capabilities for better DC motor control. Hence, the CCP1 module in the PIC18F4321 is also called ECCP (Enhanced CCP). Note that the CCP2 module is provided with standard capture, compare, and PWM features.

The CCP module is describe in detail in Section 9.4 of Chapter 9. In this section, PIC18F assembly language programs (Examples 9.8 through 9.10 of Chapter 9) will be converted to C programs to illustrate how to program the PIC18F4321 CCP module using C.

Example 10.14 Assume PIC18F4321. Write a C language program to measure the period (in terms of the number of clock cycles) of an incoming periodic waveform connected at RC2/CCP1/P1A pin. Use Timer3, and capture mode of CCP1.

```
        TRISCbits.TRISC2=0;          // Configure CCP1 pin as output
        T3CON=0x40;                  // Select TIMER3 as clock for compare, 1:1 prescale
        CCPR1H=0x01;                 // Load CCPR1H with 0x01
        CCPR1L=0xF4;                 // Load CCPR1L  with 0xF4
        TMR3H=0;                     // Initialize TMR3H to 0
        TMR3L=0;                     // Initialize TMR3L to 0
        PIR1bits.CCP1IF=0;           // Clear CCP1IF
        T3CONbits.TMR3ON=1;          // Start Timer3

        while(PIR1bits.CCP1IF==0);   // Wait in a loop until CCP1IF is 1

        T3CONbits.TMR3ON=0;          // Stop Timer3

        while(1);                    // Halt
    }
```

Example 10.16 Write a C language program to generate a 4 KHz PWM with a 50% duty cycle on the RC2/CCP1/P1A pin of the PIC18F4321. Assume 4 MHz crystal.

Solution

PR2 = [(Fosc)/(4 x Fpwm x TMR2 Prescale Value)] - 1
PR2 = [(4 MHz)/(4 x 4 KHz x 1)] - 1 assuming Prescale value of 1
PR2 = 249. With 50% duty cycle, decimal value of the duty cycle = 0.5 x 249 = 124.5.Hence, the CCPR1L register will be loaded with 124, and bits DC1B1:DC0B0 (CCP1CON register) with 10 (binary).

The C language program is provided below:

```
#include<p18f4321.h>

    void main(void)
    {
        PR2=249;                      // Initialize PR2 register
        CCPR1L=124;                   // Initialize CCPR1L
        CCP1CON=0x20;                 // CCP1 OFF, DC1B1:DC0B0=10
        TRISCbits.TRISC2=0;           // Configure CCP1 pin as output
        CCP1CON=0x2C;                 // PWM mode
        TMR2=0;                       // Clear Timer2 to 0

        while(1)
        {
        PIR1bits.TMR2IF=0;            // Clear TMR2IF to 0
        T2CONbits.TMR2ON=1;           // Turn Timer2 ON
        while(PIR1bits.TMR2IF==0);    // Wait until TMR2IF is HIGH
        }
    }
```

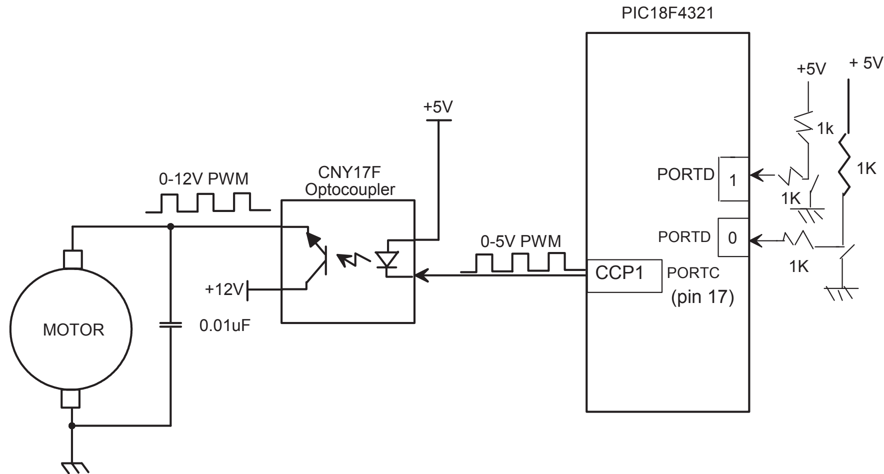


FIGURE 10.17 Figure for Example 10.17

10.16 DC Motor Control Using PWM Mode and C

As mentioned in Chapter 9, typical applications of the PWM mode include DC motor control. The speed of a DC motor is directly proportional to the driving voltage. The speed of a motor increases as the voltage is increased. In earlier days, voltage regulator circuits were used to control the speed of a DC motor. But voltage regulators dissipate lots of power. Hence, the PIC18F in the PWM mode is used to control the speed of a DC motor. In this scheme, power dissipation is significantly reduced by turning the driving voltage to the motor ON and OFF. The speed of the motor is a direct function of the ON time divided by the OFF time.

Microcontrollers such as the PIC18F4321 are not capable of outputting the required large current and voltage to control a typical DC motor. Hence, a driver such as the CNY17F Optocoupler is needed to amplify the current and voltage provided by the PIC18F's output, and provide appropriate levels for the DC motor. One of the many useful applications for using a PWM signal is its ability to control a mechanical device, such as a motor.

Note that the motor will run faster or slower based on the duty cycle of the PWM signal. The motor runs faster as the duty cycle of the PWM signal at the CCPx pin is increased. To illustrate this concept, two different duty cycles will be used in Example 10.17 converts the PIC18F assembly language program of Chapter 9 (Example 9.11) into C.

Example 10.17 Figure 10.17 shows a simplified diagram interfacing the PIC18F4321 to a DC motor via the CNY17F Optocoupler. The purpose of this example is to control the speed of a DC motor by inputting two switches connected at bit 0 and bit 1 of PORTD. The motor will run faster or slower based on the switch values (00 or 11), but will not provide any measure of the exact RPM of the motor.

When both switches are closed (00), a PWM signal at the CCP1 pin of the PIC18F4321 with 50% duty cycle will be generated. When both switches are open (11), a PWM signal at the CCP1 pin of the PIC18F4321 with 75% duty cycle will be generated. Otherwise, the motor will stop, and the program will wait in a loop.

Questions and Problems

10.1 Write a C language statement to configure

(a all bits of Port C as inputs

(b) all bits of Port D as outputs

(c) bits 0 through 4 of Port B as inputs

(d) all bits of Port A as outputs

10.2 The PIC18F4321 microcontroller is required to drive the LEDs connected to bit 0 of Ports C and D based on the input conditions set by switches connected to bit 1 of Ports C and D. The I/O conditions are as follows:

- If the input at bit 1 of Port C is HIGH and the input at bit 1 of Port D is LOW, the LED at Port C will be ON and the LED at Port D will be OFF.
- If the input at bit 1 of Port C is LOW and the input at bit 1 of Port D is HIGH, the LED at Port C will be OFF and the LED at Port D will be ON.
- If the inputs of both Ports C and D are the same (either both HIGH or both LOW), both LEDs of Ports C and D will be ON.

Write a C language program to accomplish this.

10.3 The PIC18F4321 microcontroller is required to test a NAND gate. Figure P10.3 shows the I/O hardware needed to test the NAND gate. The microcomputer is to be programmed to generate the various logic conditions for the NAND inputs, input the NAND output, and turn the LED ON connected to bit 3 of Port D if the NAND gate chip is found to be faulty. Otherwise, turn the LED ON connected to bit 4 of Port D. Write a C language program to accomplish this.

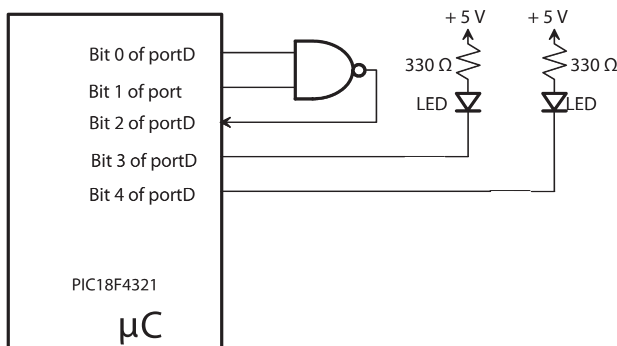


FIGURE P10.3

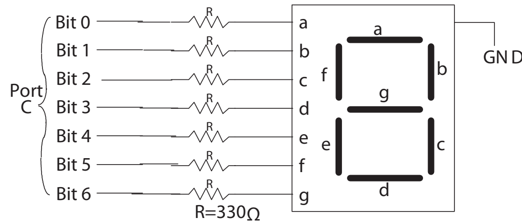


FIGURE P10.4

10.4 The PIC18F4321 microcontroller (Figure P10.4) is required to add two 3-bit numbers entered via DIP switches connected at bits 0-2 and bits 3-5 of Port D and output the sum (not to exceed 9) to a common-cathode seven-segment display connected to Port C as shown in Figure P10.4. Write a C language program to accomplish this by using a lookup table.

10.5 The PIC18F4321 microcontroller is required to input a number from 0 to 9 from an ASCII keyboard interfaced to it and output to an EBCDIC printer. Assume that the keyboard is connected to Port C and the printer is connected to Port D. Write a C language program using a lookup table for EBCDIC codes from 0 to 9 to accomplish this. Note that decimal numbers 0 through 9 are represented by F0H through F9H in EBCDIC code, and by 30H through 39H in ASCII code as mentioned in Chapter 1.

10.6 In Figure P10.6, the PIC18F4321 is required to turn on an LED connected to bit 1 of Port D if the comparator voltage Vx > Vy; otherwise, the LED will be turned off. Write a C language program to accomplish this using conditional or polled I/O.

10.7 Repeat Problem 10.6 using Interrupt I/O by connecting the comparator output to INT1. Note that RB1 is also multiplexed with INT1. Write main program and interrupt service routine in C language. The main program will configure the I/O ports, enable interrupt INT1, turn the LED OFF, and then wait for interrupt. The interrupt service routine will turn the LED ON and return to the main program at the appropriate location so that the LED is turned ON continuously until the next interrupt.

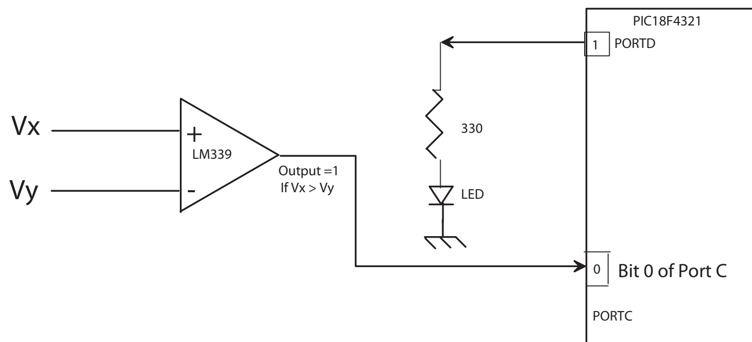


FIGURE P10.6

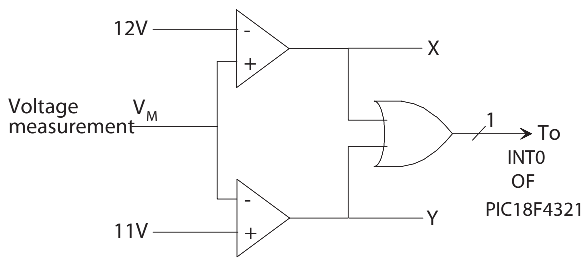


FIGURE P10. 8

10.8 In Figure P10.8, if V_M > 12 V, turn an LED ON connected to bit 3 of port A. If V_M < 11 V, turn the LED OFF. Using ports, registers, and memory locations as needed and INT0 interrupt:

(a) Draw a neat block diagram showing the PIC18F4321 microcontroller and the connections to ports in the diagram in Figure P10.8.

(b) Write the main program and the service routine in C language. The main program will initialize the ports and wait for an interrupt. The service routine will accomplish the task and stop.

10. 9 Write C language program to set interrupt priority of INT1 as the high level, and interrupt priority for INT2 level as low level.

10.10 Assume the PIC18F4321- DMC 16249 interface of Figure 10.10. Write a C program to display the phrase "PIC18F" on the LCD as soon as the four input switches connected to Port C are all HIGH.

10.11 Write a C program to initialize Timer0 as an 8-bit timer to provide a time delay with a count of 100. Assume 4 MHz internal clock with a prescaler value of 1:16.

10.12 Write a C language program to generate a square wave with a period of 4 ms on bit 0 of PORTC using a 4 MHz crystal. Use Timer0.

10.13 Write a C language program to generate a square wave with a period of 4 ms on bit 7 of PORTD using a 4 MHz crystal. Use Timer1.

10.14 Write a C language program to turn an LED ON connected at bit 0 of PORTC with a PR2 value of 200. Assume a 4 MHz crystal and TMR2 prescaler and postscaler values of 1:1.

10.15 Write a C language program to generate a square wave on pin 3 of PORTC with a 4 ms period using Timer3 in 16-bit mode with a prescaler value of 1:8. Use a 4 MHz crystal.

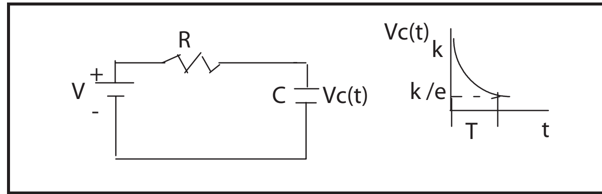


FIGURE P10.19

1. Sample a 5 V (zero-to-peak voltage), 60 Hz sinusoidal voltage 128 times.
2. Digitize the sampled value using the on-chip ADC of the PIC18F4321 along with its interrupt upon completion of conversion signal.
3. Compute the RMS value of the waveform using the formula,

   RMS Value = SQRT $[(\sum_{n=1}^{N}(X_n^2)/N]$, where X_n's are the samples and N is the total number of samples. Display the RMS value on two seven-segment displays (one for integer part, and the other for fractional part).

(a) Draw a hardware block diagram.

(b) Write C language program to accomplish the above.

10.19 *Capacitance meter.* Consider the RC circuit of Figure P10. 19. The voltage across the capacitor is Vc (t) = k $e^{-t/RC}$. In one-time constant RC, this voltage is discharged to the value k/e. For a specific value of R, value of the capacitor C = T/R, where T is the time constant that can be counted by the PIC18F4321. Design the hardware and software for the PIC18F4321 to charge a capacitor by using a pulse to a voltage of your choice via an amplifier. The PIC18F4321 will then stop charging the capacitor, measure the discharge time for one time constant, and compute the capacitor value.

(a) Draw a hardware schematic.

(b) Write a C program to accomplish the above.

10.20 Design a PIC18F4321-based digital clock. The clock will display time in hours, minutes, and seconds. Write a C program to accomplish this.

10.21 Design a PIC18F4321-based system to measure the power absorbed by a 2K resistor (Figure P10.21). The system will input the voltage (V) across the 2K resistor, convert it to an 8-bit input using the PIC18F4321's on-chip A/D converter, and then compute the power using V^2/R. Write a C language program to accomplish the above.

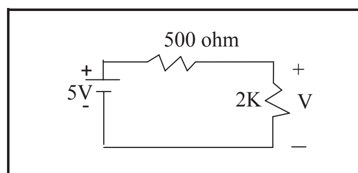


FIGURE P10.21

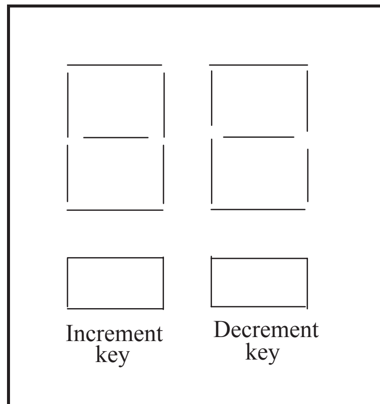


FIGURE P10.22

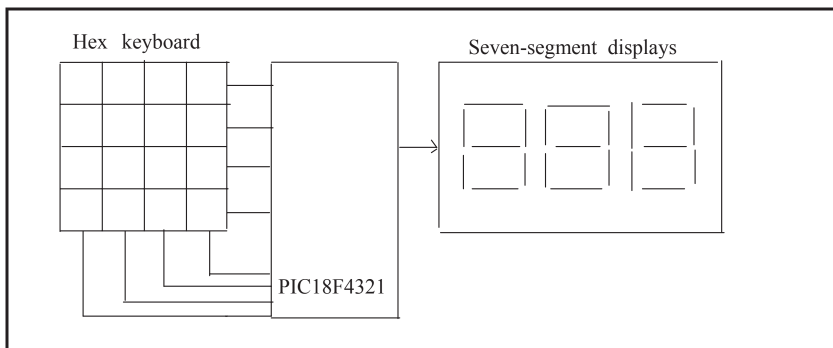


FIGURE P10.23

10.22 Design a PIC18F4321-based system (Figure P10.22) as follows: The system will drive two seven-segment digits, and monitor two key switches. The system will start displaying 00. If the increment key is pressed, it will increment the display by one. Similarly, if the decrement key is pressed, the display will be decremented by one. The display will go from 00 to 09, and vice versa.
Write a C language program to accomplish the above. Use ports of your choice. Draw a block diagram of your implementation.

10.23 It is desired to implement a PIC18F4321-based system as shown in Figure P10.23. The system will scan a hex keyboard with 16 keys, and drive three seven-segment displays. The PIC18F4321 will input each key pressed, scroll them in from the right side of the displays, and keep scrolling as each key is pressed. The leftmost digit is just discarded. The system continues indefinitely. Write a C language program to accomplish the above. Use ports of your choice.

10.24 Assume that two PIC18F4321s are interfaced in the SPI mode. A switch is connected to bit 0 of PORTD of the master PIC18F4321 and an LED is connected to bit 5 of PORTB of the slave PIC18F4321. Write C language programs to input the switch via the master, and output it to the LED of the slave PIC18F4321. If the

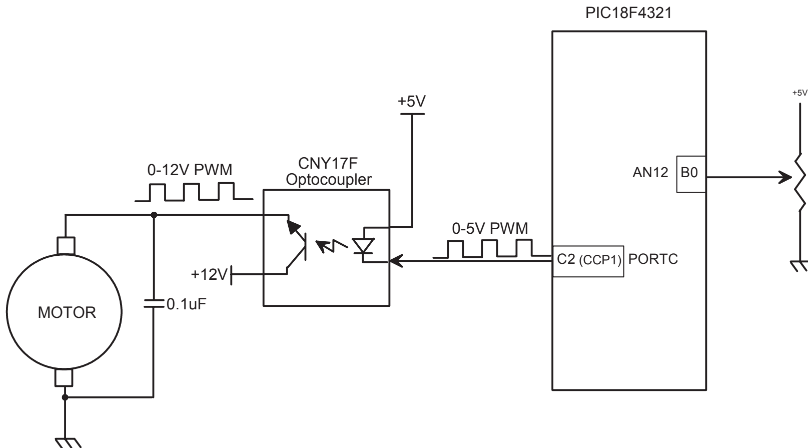


FIGURE P10.28

switch is open, the LED will be turned ON while the LED will be turned OFF if the switch is closed.

10.25 Assume PIC18F4321. Write a C language program that will measure the period of a periodic pulse train on the CCP1 pin using the capture mode. The 16-bit result will be performed in terms of the number of internal (Fosc/4) clock cycles, and will be available in the TMR1H:TMR1L register pair. Use 1:1 prescale value for Timer1. Store the 16-bit result in CCPR1H:CCPR1L.

10.26 Assume PIC18F4321. Write a C language program that will generate a square wave on the CCP1 pin using the Compare mode. The square wave will have a period of 20 ms with a 50% duty cycle. Use Timer1 internal clock (Fosc/4 from XTAL) with 1:2 prescale value.

10.27 Write a C language program to generate a 16 KHz PWM with a 75% duty cycle on the RC2/CCP1/P1A pin of the PIC18F4321. Assume 10 MHz crystal.

10.28 It is desired to change the speed of a DC motor by dynamically changing its pulse width using a potentiometer connected at bit 0 of PORTB (Figure P10.28). Note that the PWM duty cycle is controlled by the potentiometer. Write a C language program that will input the potentiometer voltage via the PIC18F4321's on-chip A/D converter using interrupts, generate an 800-Hz PWM waveform on the CCP1 pin, and then change the speed of the motor as the potentiometer voltage is varied. Assume 4 MHz crystal and a TMR2 prescaler value of 16. Ignore fractional part of the duty cycle.

10.29 It is desired to implement a traffic light controller using the PIC18F4321 as follows:

Step 1: Make North-South light Green and East-West light Red for 10 seconds. Check to see if any waiting car is trying to go from east to west and vice versa. If there is a waiting car, go to step 2; otherwise, repeat this step.

```
        MOVF    PORTC, F     ; Input PORTC
        MOVLW   0x07
        ANDWF   PORTC, F     ; Retain low three bits
        MOVLW   0x00         ; Check for no high switches, 0 is an even number
        SUBWF   PORTC, W
        BZ      LED          ; If no high switches, turn LED ON
        MOVLW   0x03         ; Check for two high switches
        SUBWF   PORTC, W
        BZ      LED          ; If  two high switches, turn LED ON
        MOVLW   0x05         ; Check for two high switches
        SUBWF   PORTC, W
        BZ      LED          ; If  two high switches, turn LED ON
        MOVLW   0x06         ; Check for two high switches
        SUBWF   PORTC, W
        BZ      LED          ; If  two high switches, turn LED ON
FINISH  SLEEP   ; Halt
LED     BSF     PORTD, 6     ; Turn LED  ON
        BRA     FINISH
        END
```

8.12

```
        INCLUDE   <P18F4321.INC>
        ORG       0x200
START   MOVLW     0x0F          ;Configure PORTB as input
        MOVWF     ADCON1
        BCF       TRISC, 1      ; Configure bit 1 of PORTC as an output
        BCF       PORTC, RC1    ;Turn LED  OFF
BACK    MOVF      PORTB, 0x20   ; Input PORTB into 0x20
        RRCF      0x20, W       ; Rotate right once to align output  data
        BNC       BACK          ;Wait for the COMP.output to be HIGH
        BSF       PORTC, RC1    ;Turn LED  ON
        BRA       START
        END
```

Chapter 9

9.1

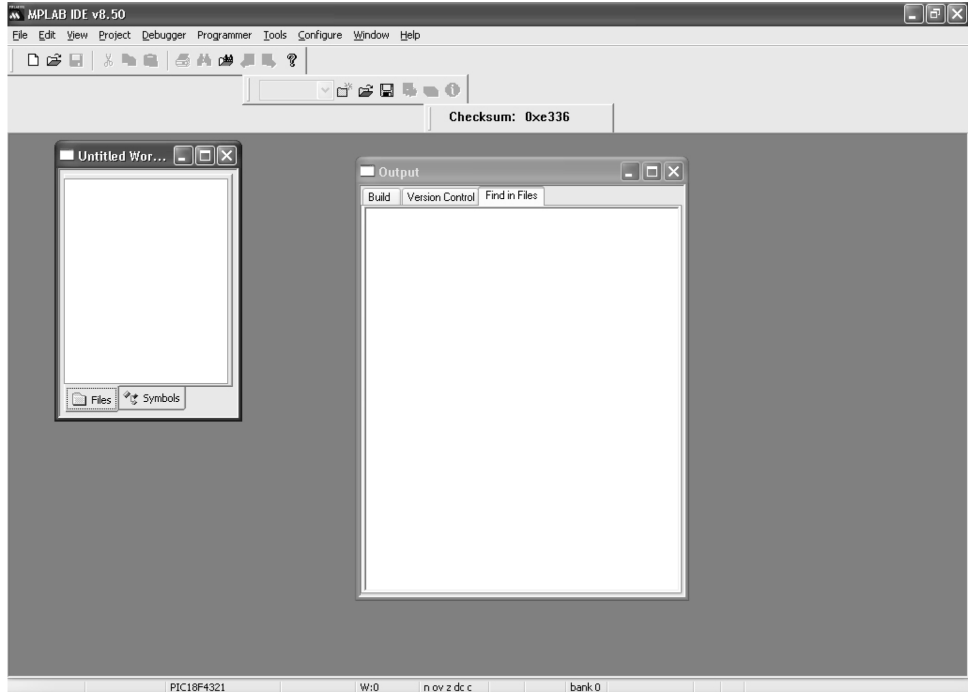
Bit 7: Set to 0 so that TMR0 is off
Bit 6: Set to 1 in order to enable the 8-bit mode of TMR0
Bit 5: Set to 1 so that an external crystal oscillator can be used
Bit 4: Set to 1 so the timer will increment when the clock is going from high to low (negative edge).
Bit 3: Set to 0 in order to enable the prescaler function
Bit 2-0: Set to 011 to enable a 1:16 prescaler
Hence, T0CON=0x73

APPENDIX G: TUTORIAL FOR COMPILING AND DEBUGGING A C PROGRAM USING THE MPLAB

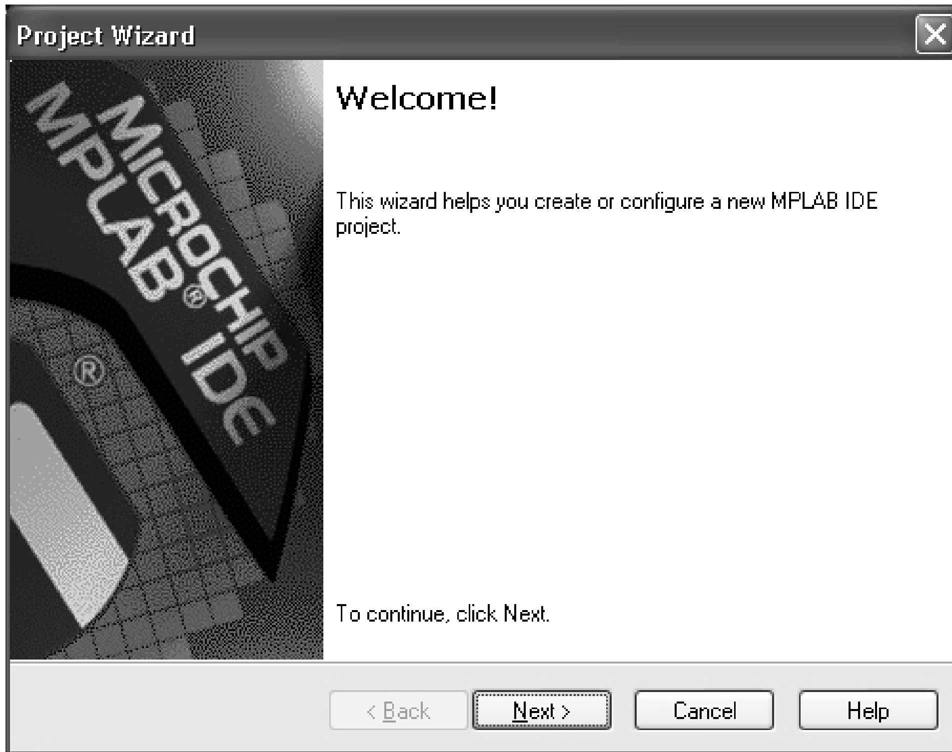
Compiling a C- language program using MPLAB

First download the latest versions of the MPLAB assembler and C18 compiler from the Microchip website www.microchip.com. After installing and downloading the program, you will see the following icon on your desktop:

Double click (right) on the MPLAB icon and wait until you see the following screen:

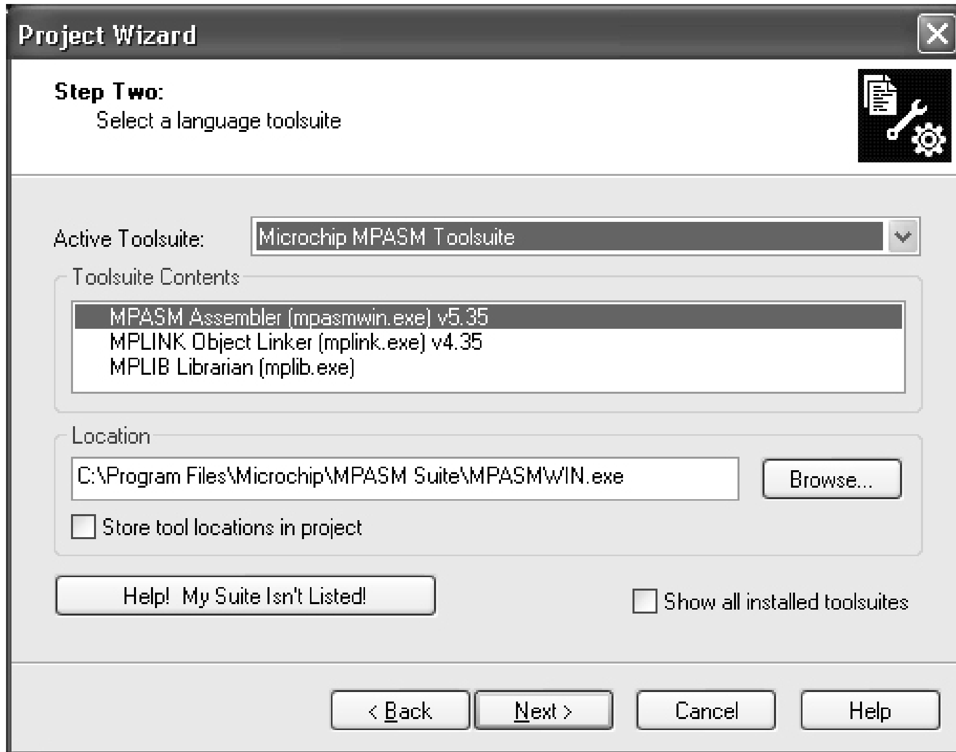


Next, click on ‘Project’ and then ‘Project Wizard’, the following screen will appear:

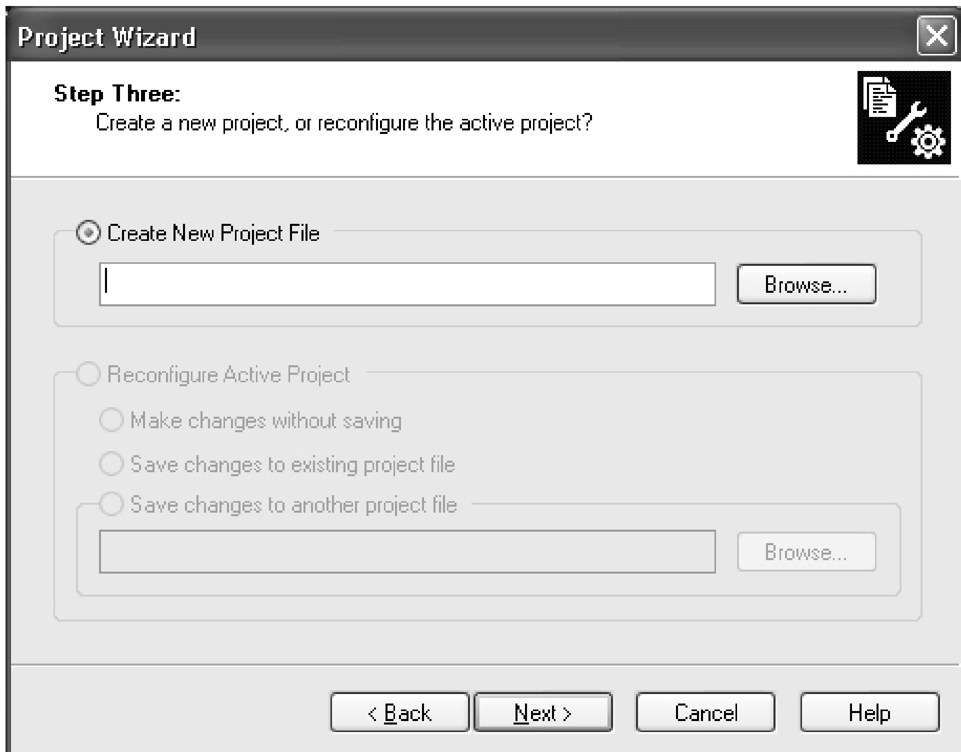


Click Next, the following screen shot will be displayed:

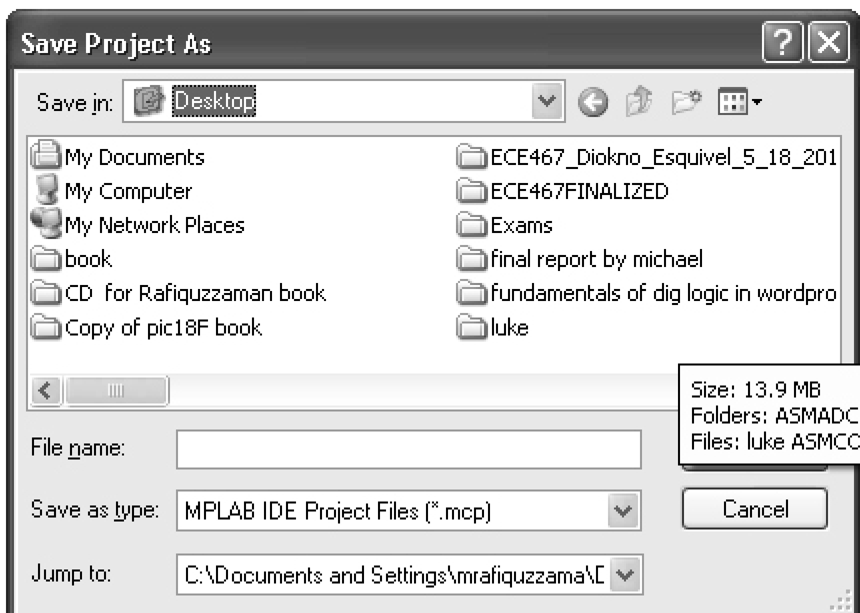
Select the device PIC18F4321, hit Next, and wait, the following will be displayed:



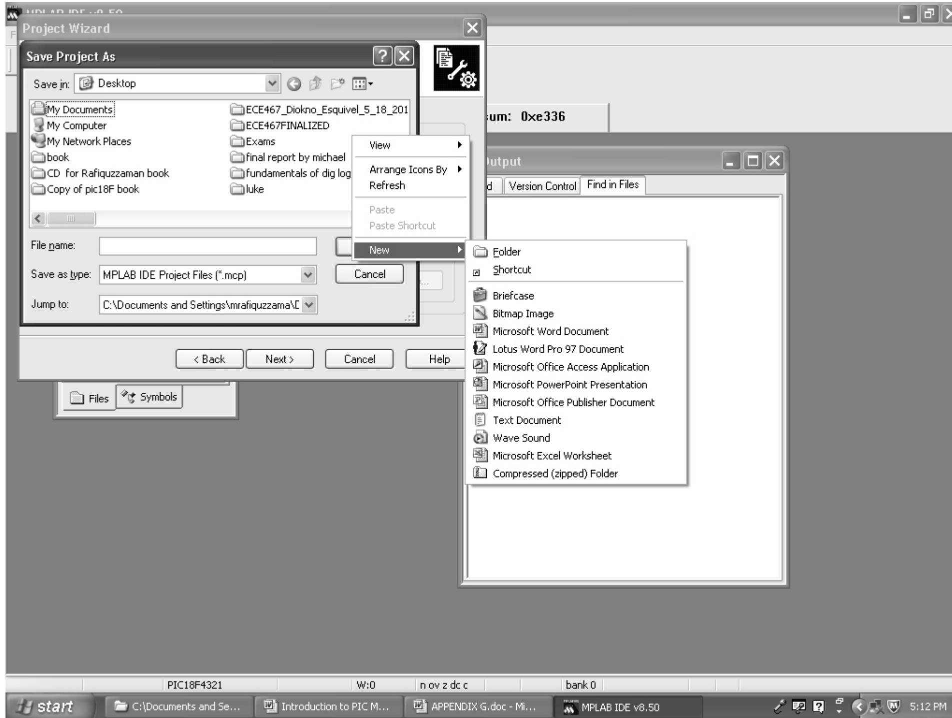
In the 'Active Toolsuite', select 'Microchip C18 Toolsuite', and click Next, the following will be displayed:



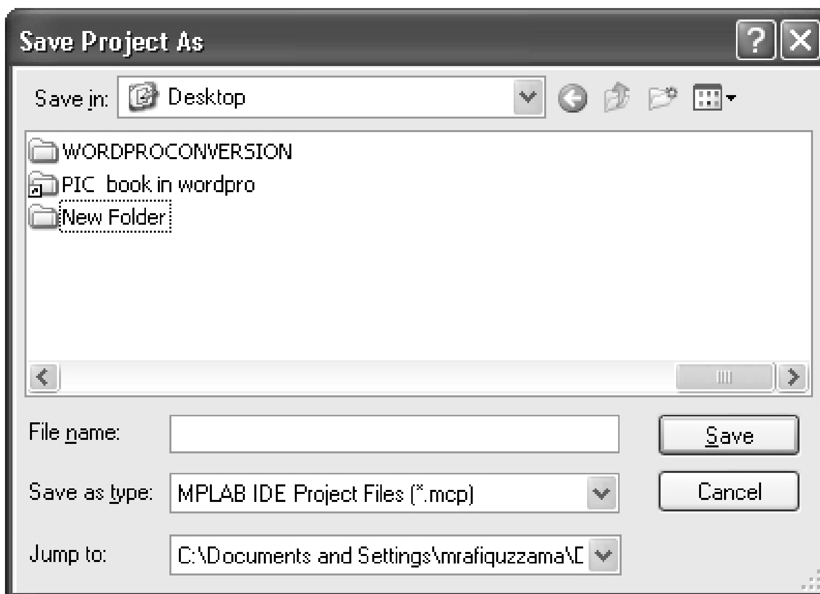
Select a location where all project contents will be placed. For this example, the folder will be placed on the desktop (arbitrarily chosen). Go to the desktop directory, make a new folder, and name the folder. In order to do this, Click on 'Browse', select desktop:



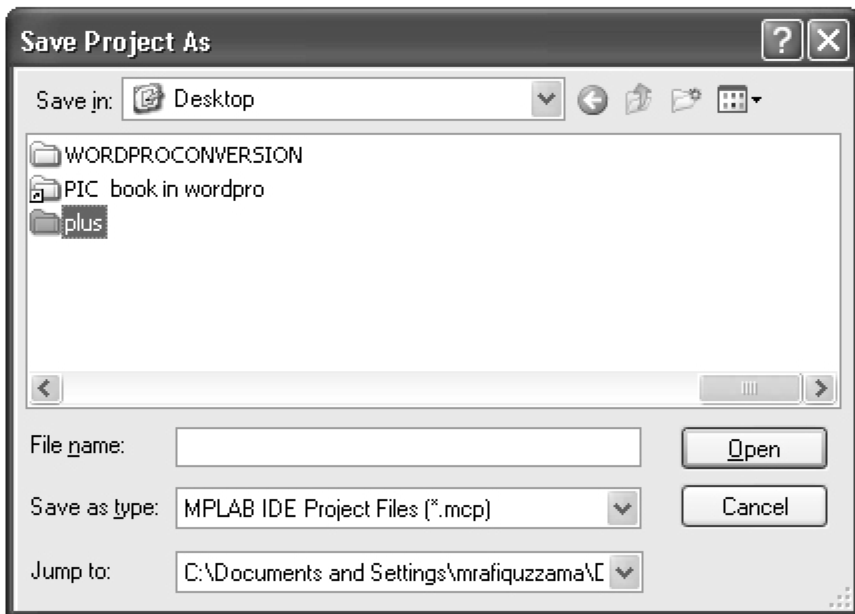
Next, create a new folder by clicking on the icon (second yellow icon from right on top row) or by right clicking on the mouse on the above window, and then go to New to see the following screen:



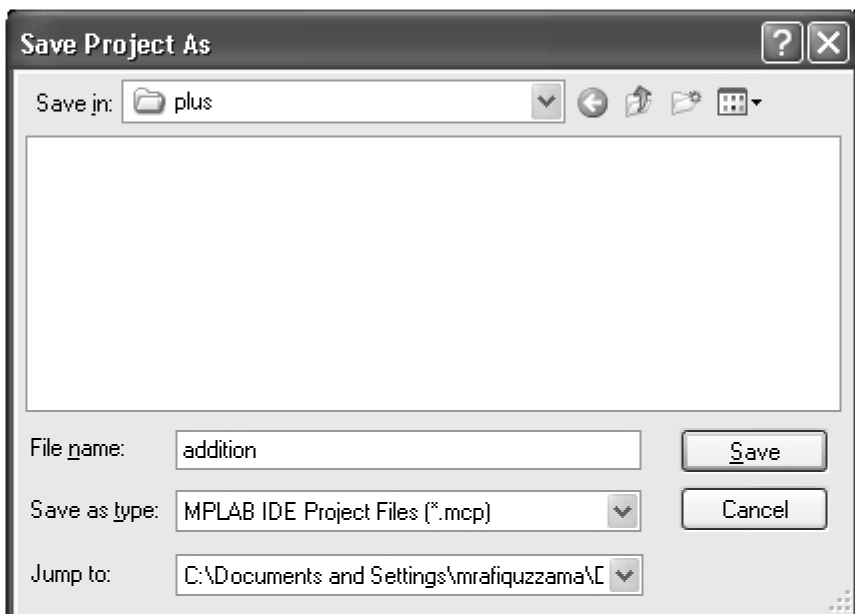
Click on Folder to see the following:



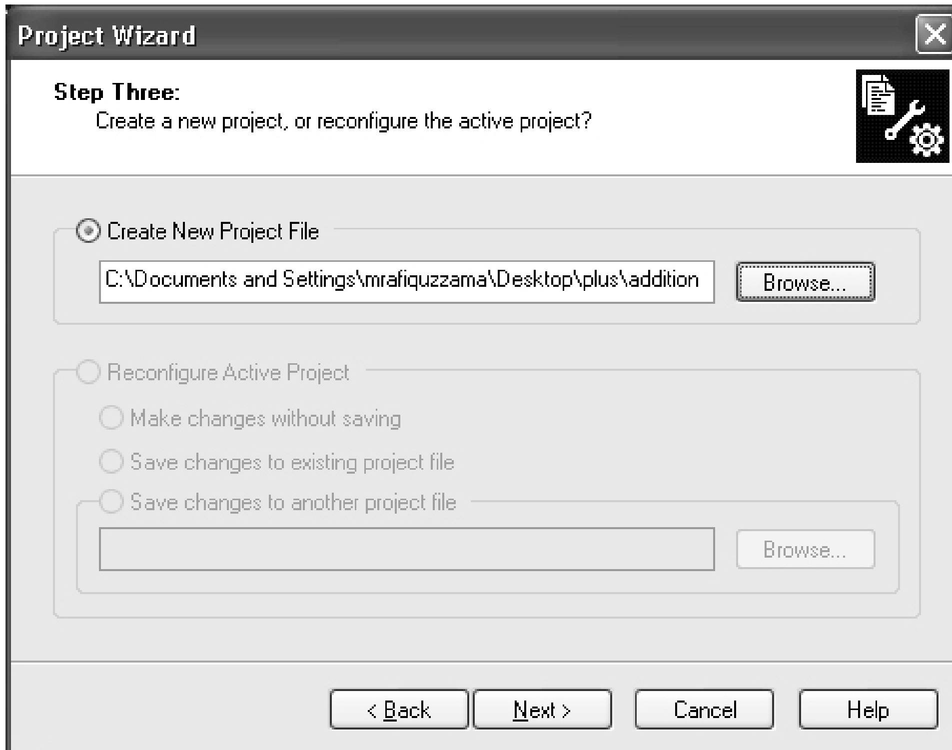
Click on folder, name it ‘plus’ (arbitrarily chosen name) and see the following :



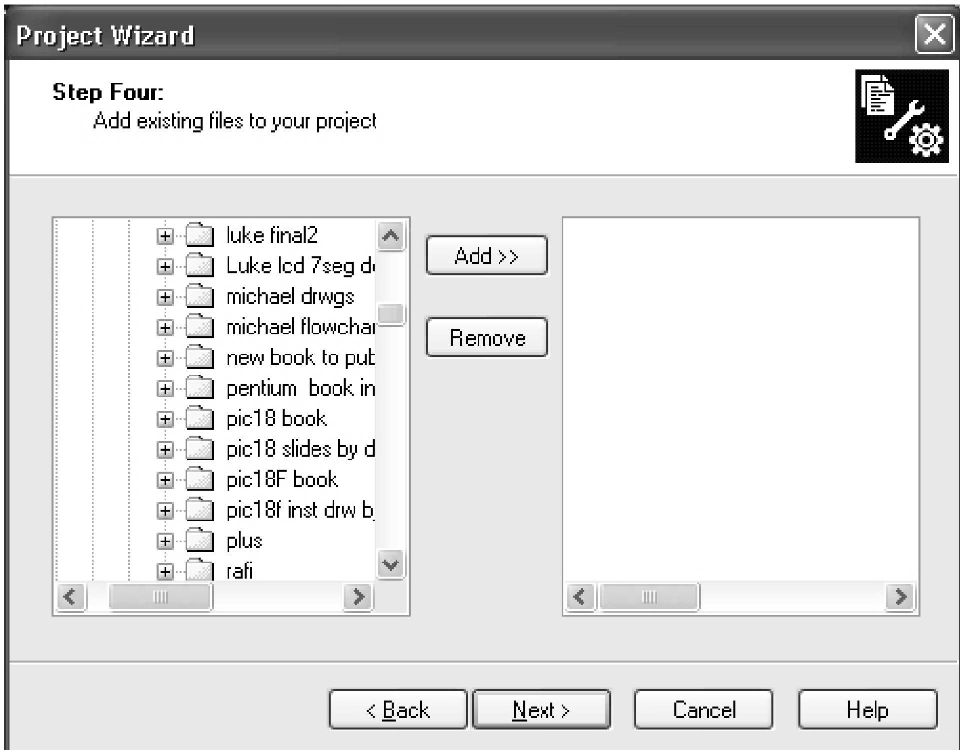
File name ‘addition’ is arbitrarily chosen . Type in the File name to see the following:



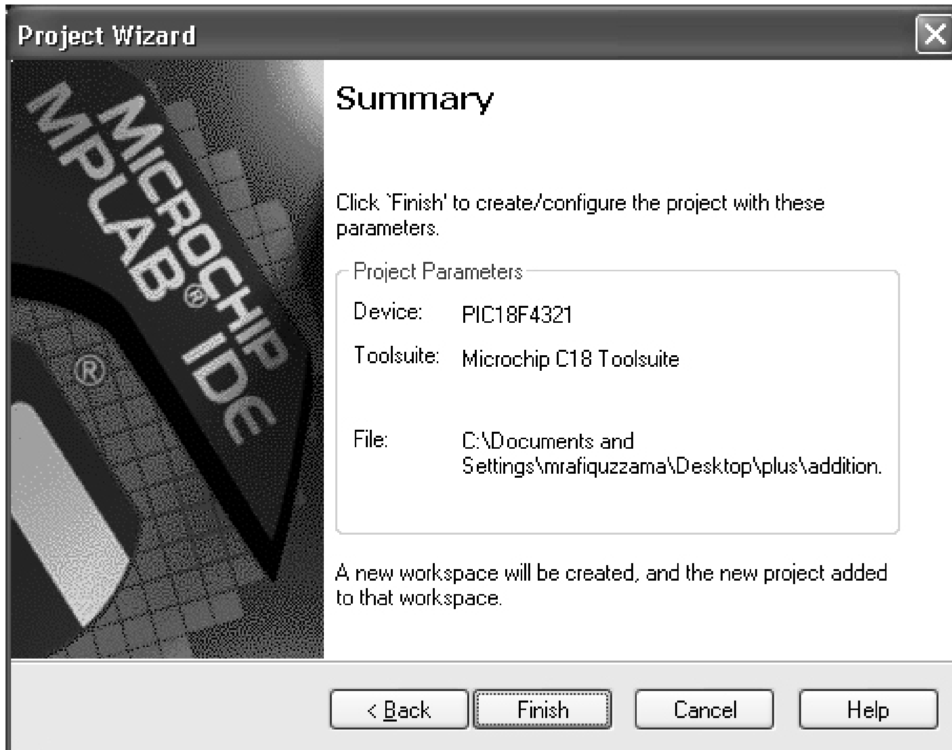
Next, click on Save, the following screen will appear:



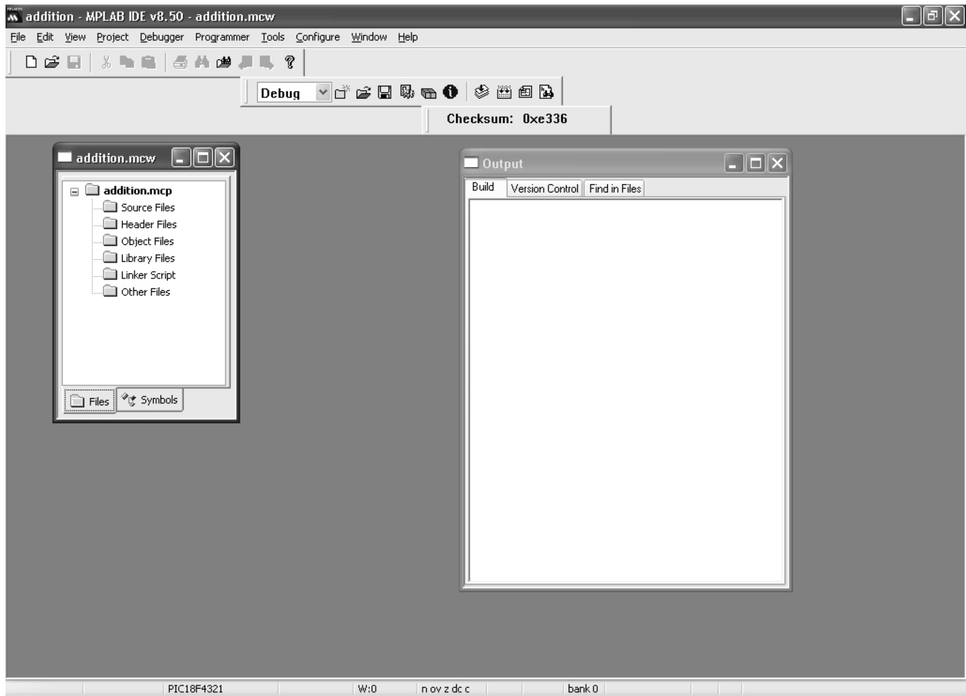
Click on Next, and see the following:



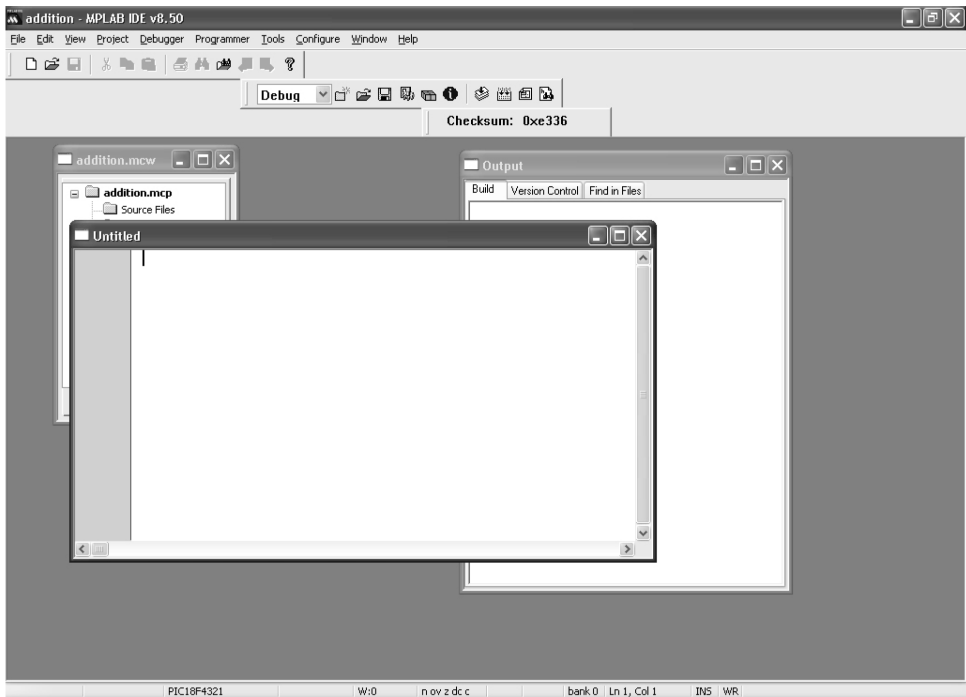
Click Next to see the following:



Click on Finish, and see the following:



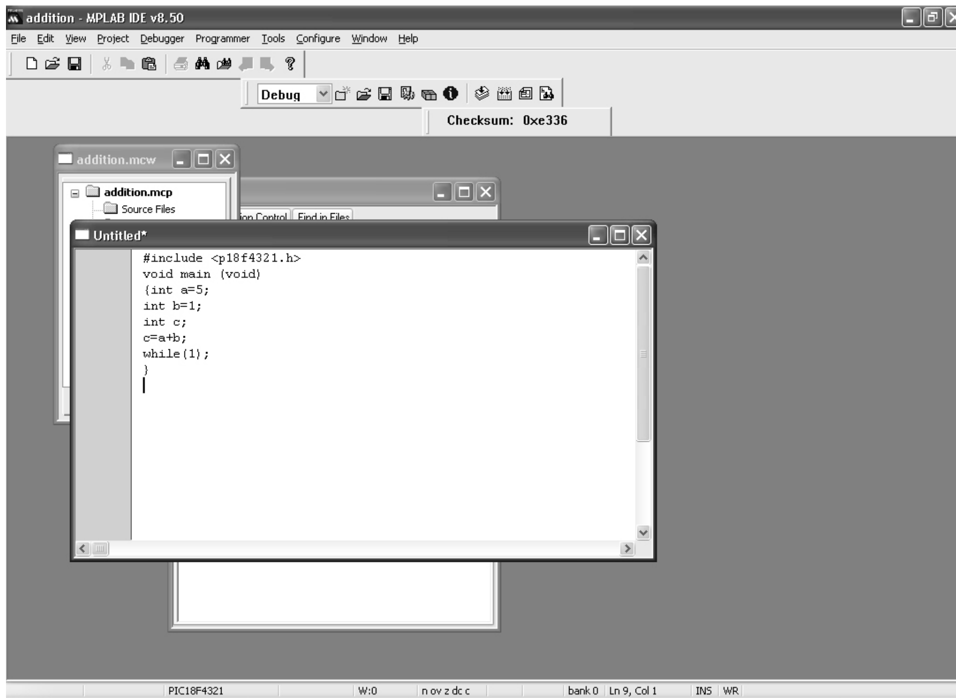
Click on File, and then New to see the following:



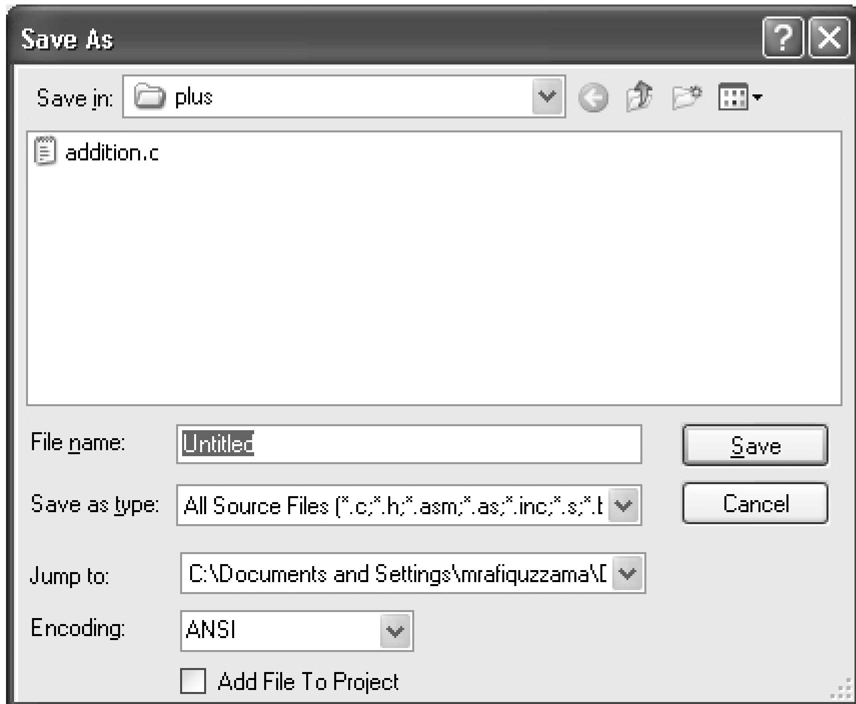
Type in the program you want to compile. The following addition program is entered:

```
#include <p18f4321.h>
void main (void)
{int a=5;
int b=1;
int c;
c=a+b;
while(1);
}
```

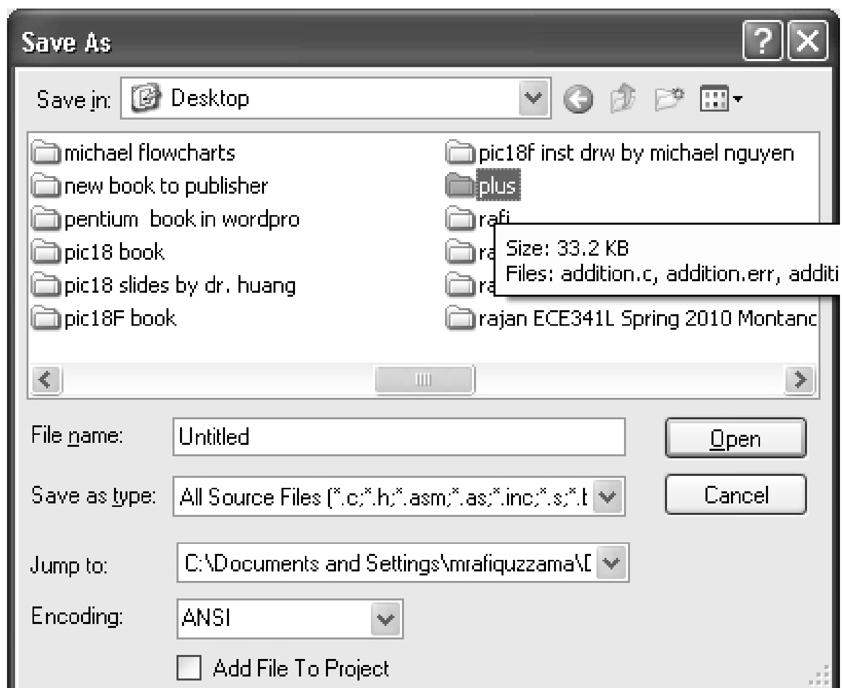
After entering the program, see the following:



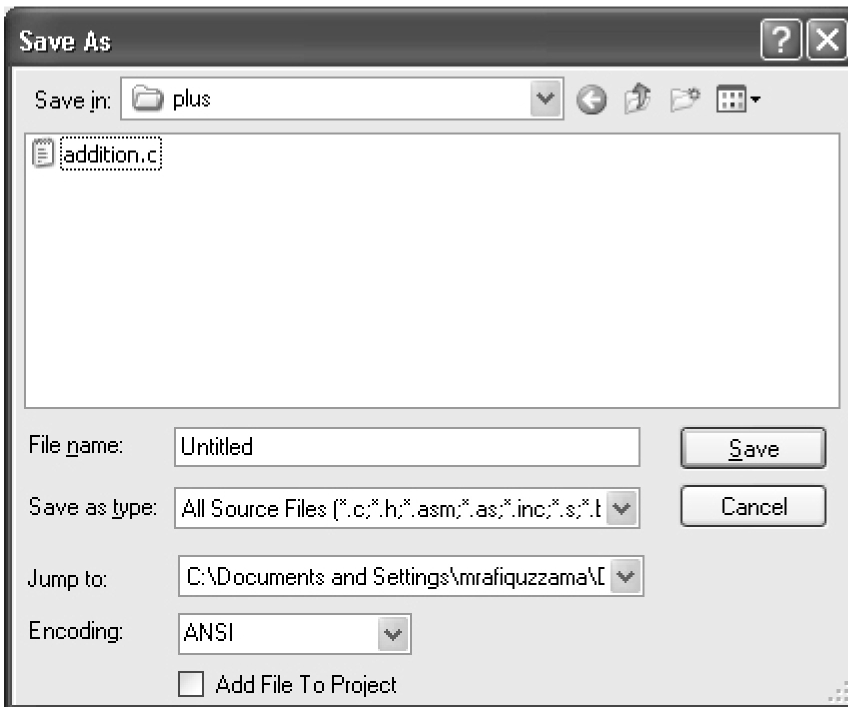
Next, click on File, and then Save as to see the following screen shot:



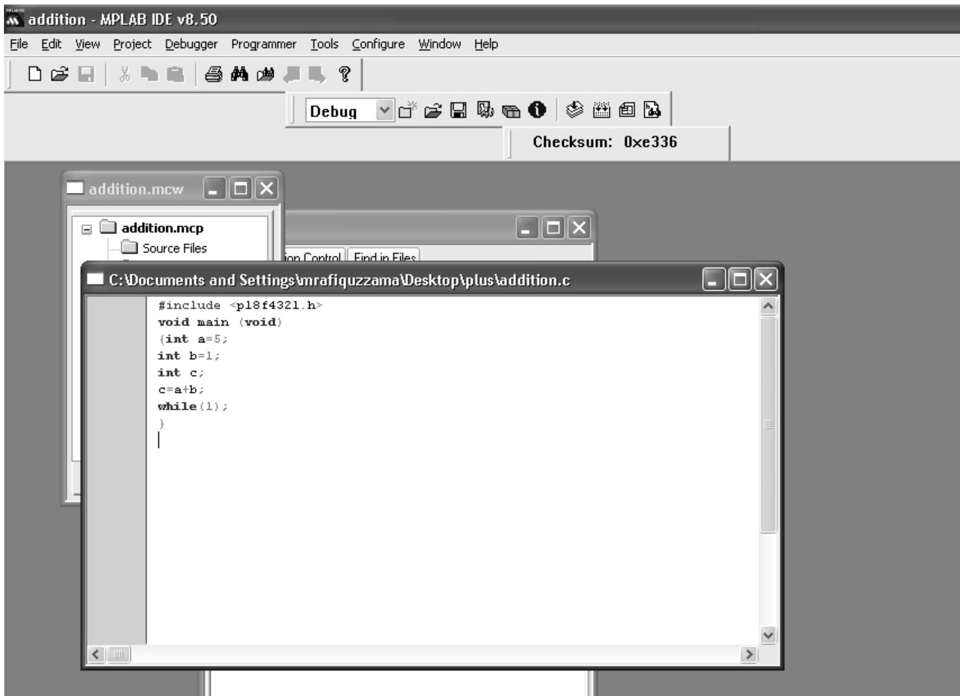
Make sure you scroll up to desktop, and then click on plus (the folder which was created before), and see the following:



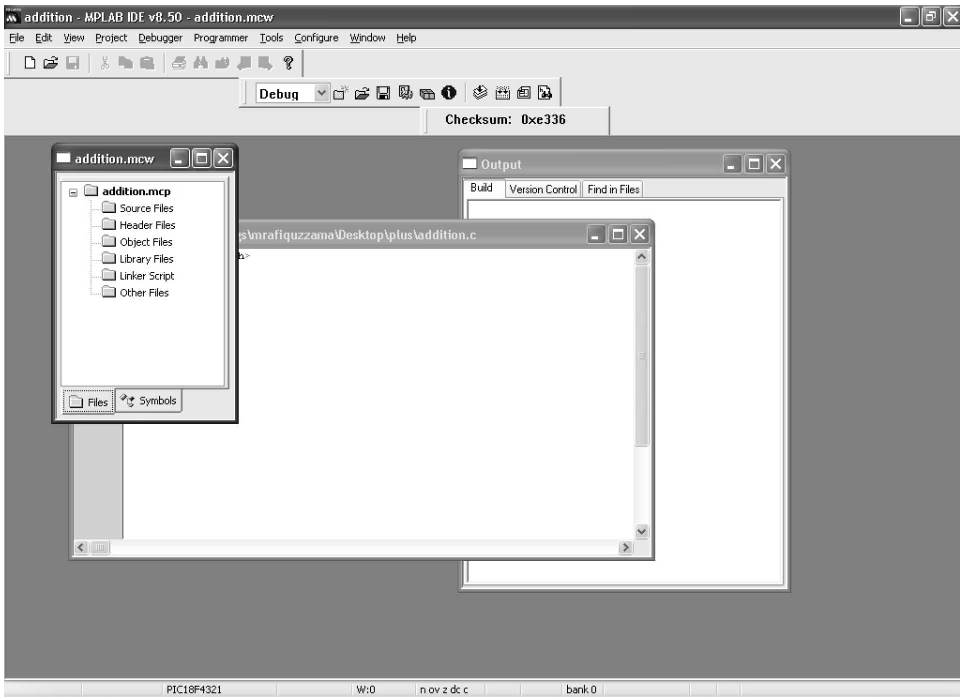
Next ,double click (left) on plus to see the following:



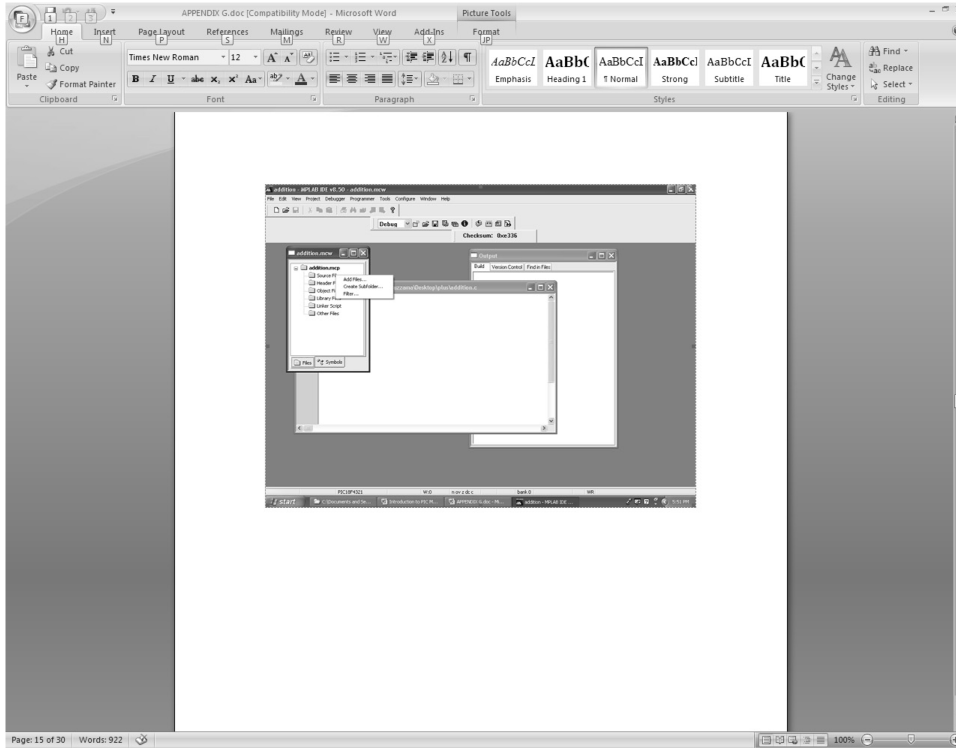
Delete Untitled, enter the same file name 'addition' with .c extension as File name. Click on save, and see the following screen shot (notice the display changes color) :



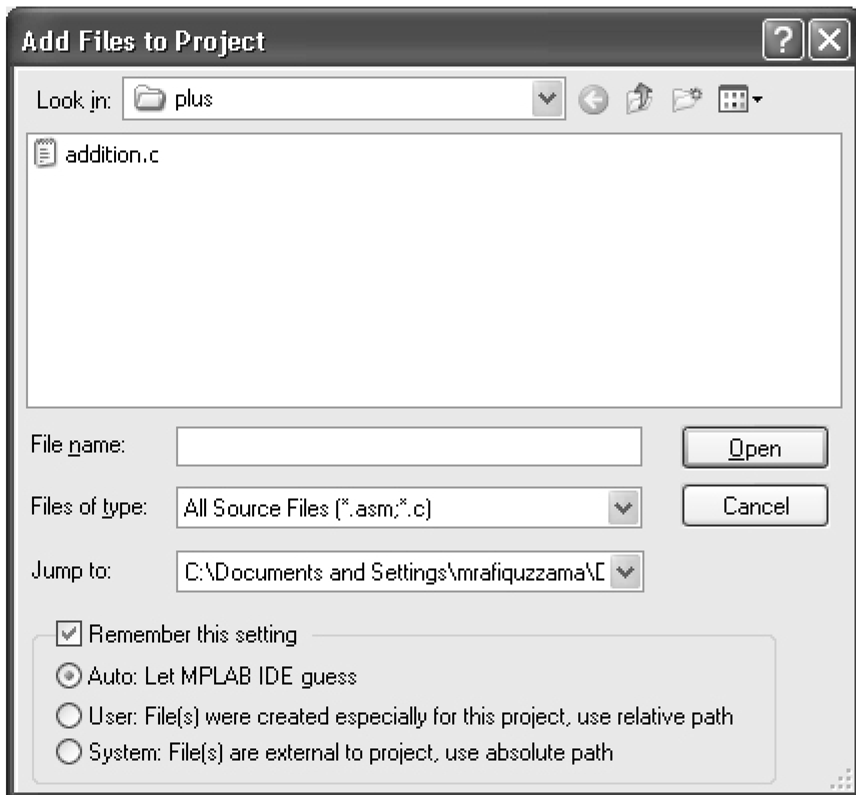
Next highlight by clicking on the top (blue) section of addition.mcw, and see the following:



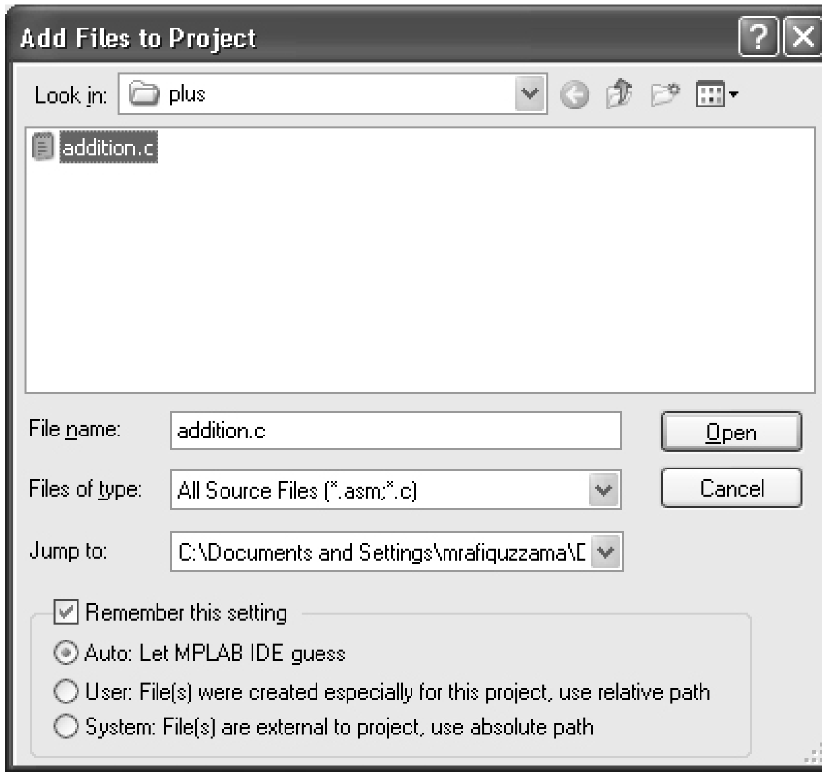
Right click on Source Files to see the following:



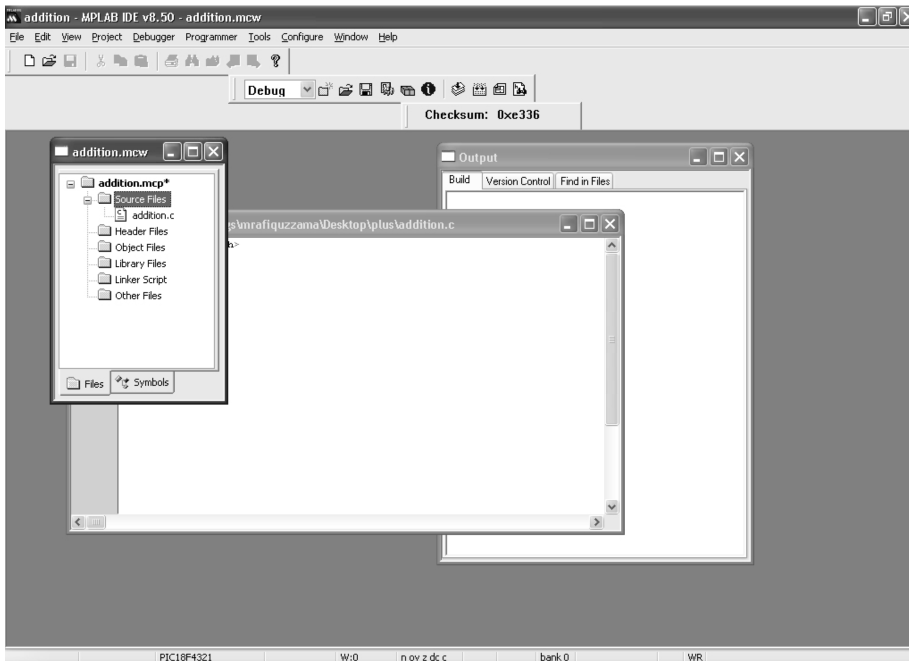
Click (left) on Add files to see the following:



Click once (left) on addition.c on the window to see the following:



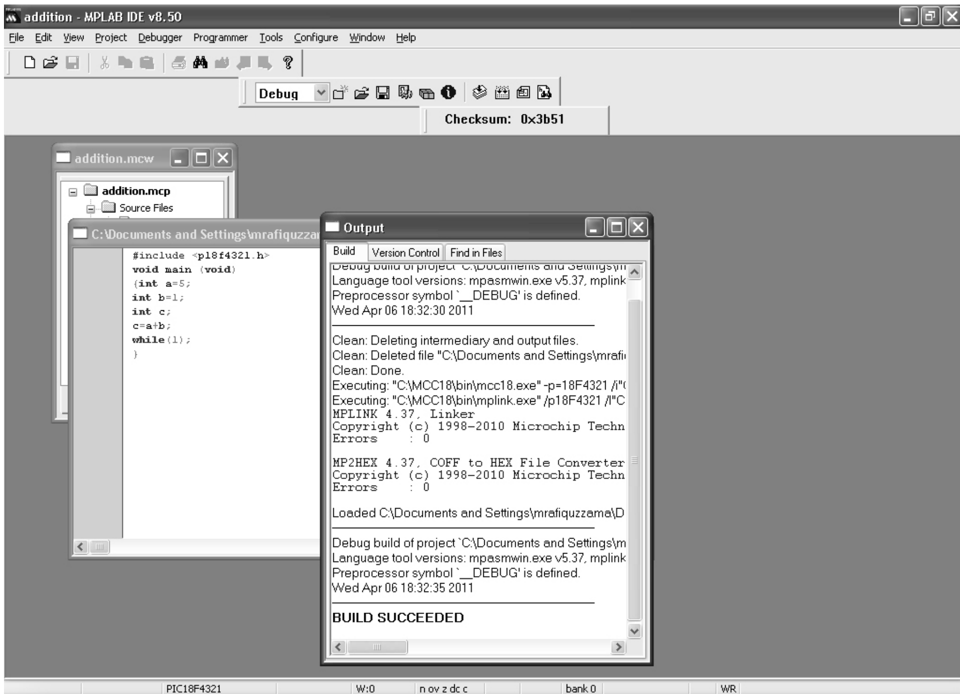
Click Open to see the following:



Next, do the following:

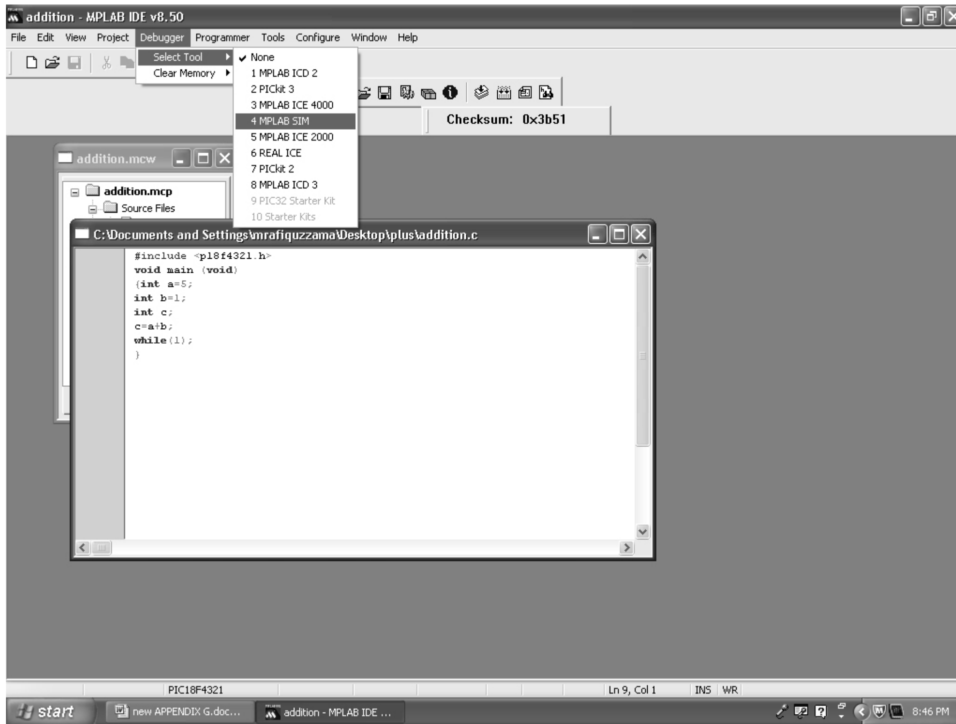
- Click on project, Click on Build options and then project
- Scroll down on output directory to Linker.Script Search Path, Select new
- Click on … (three dots on the extreme right)
- Select C:\MCC18, bin, and then LKR, Click OK
- Click on project, Click on Build options and then project
- Scroll down on output directory to Library Search Path, Select new
- Click on … (three dots on the extreme right)
- Select C:\MCC18, and then lib, Click OK
- Click on project, Click on Build options and then project
- Scroll down on output directory to Include Search Path, Select new
- Click on … (three dots on the extreme right)
- Select C:\MCC18, and then h, Click OK

Note that addition.c is listed under Source Files. Next, click on Project and then build all (or only the 'Build All' icon , third icon on top right of the Debug toolbar), and see the following:

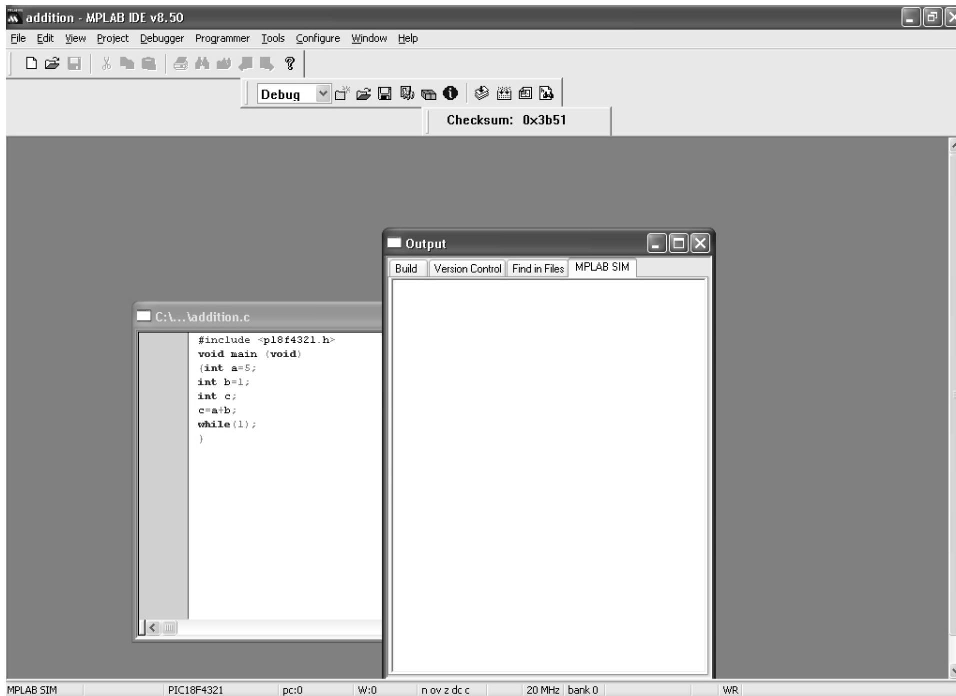


This means that the compiling the C program is successful. Next the result will be verified using the debugger.

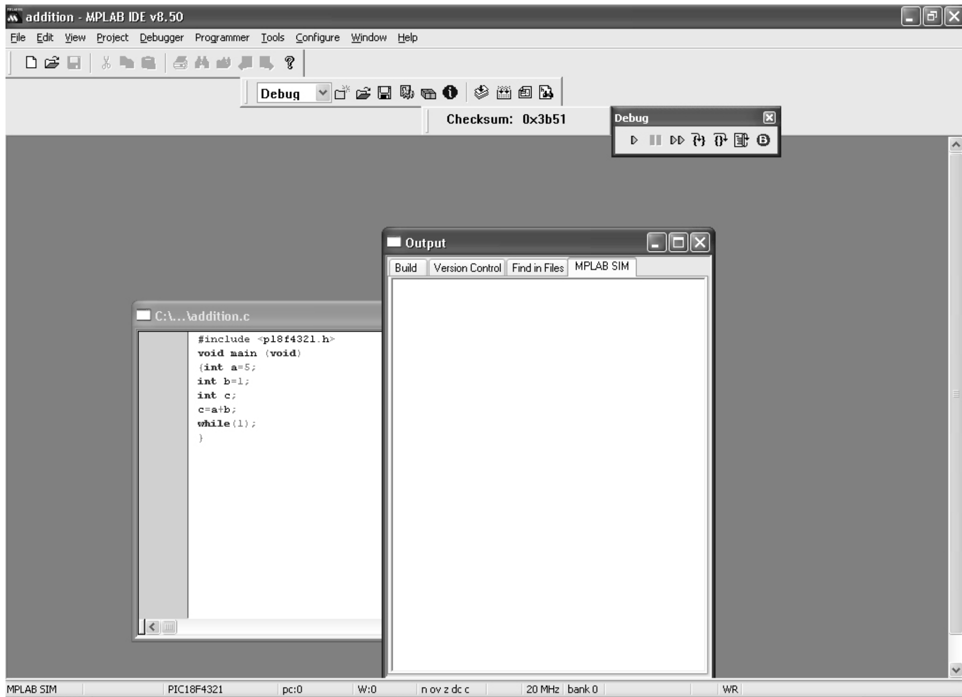
Click on Debugger, Select Tool, and then MPLAB SIM to see the following display:



Click on MPLAB SIM to see the following:

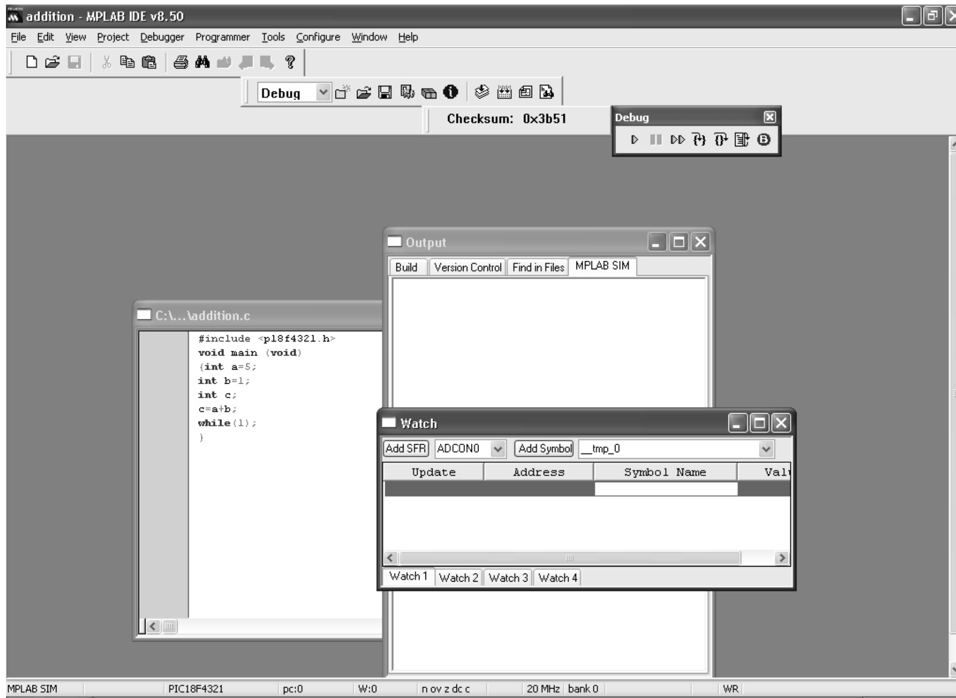


Click on View, toolbars, and Debug to see the following display with Debug toolbar:

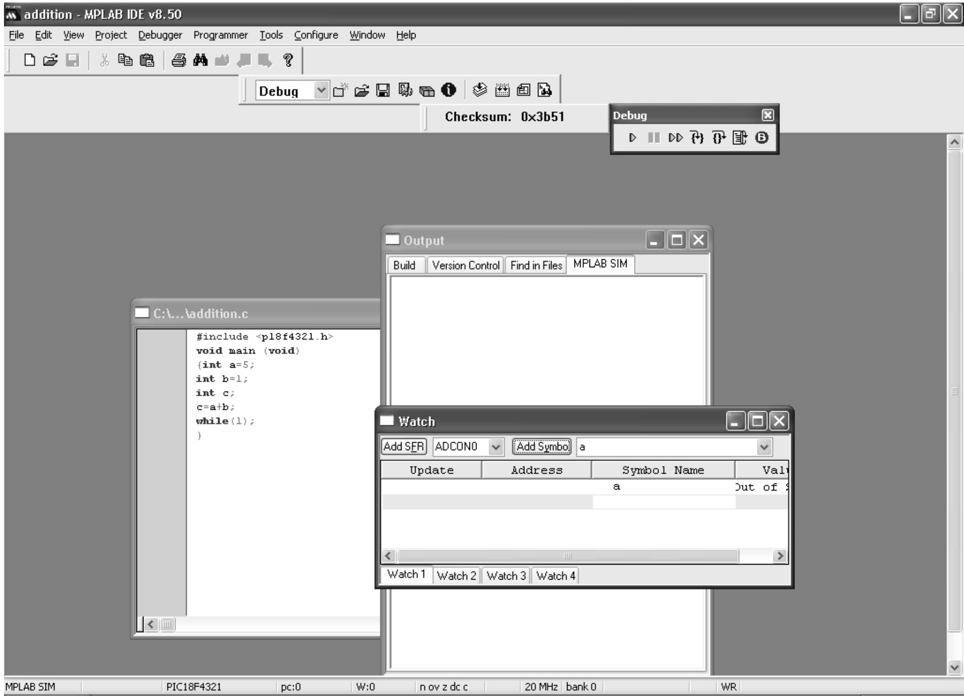


In the above, locate the Debug Toolbar. If, for some reason, Debug toolbar is missing, go to view, select Toolbars, click on Debug.

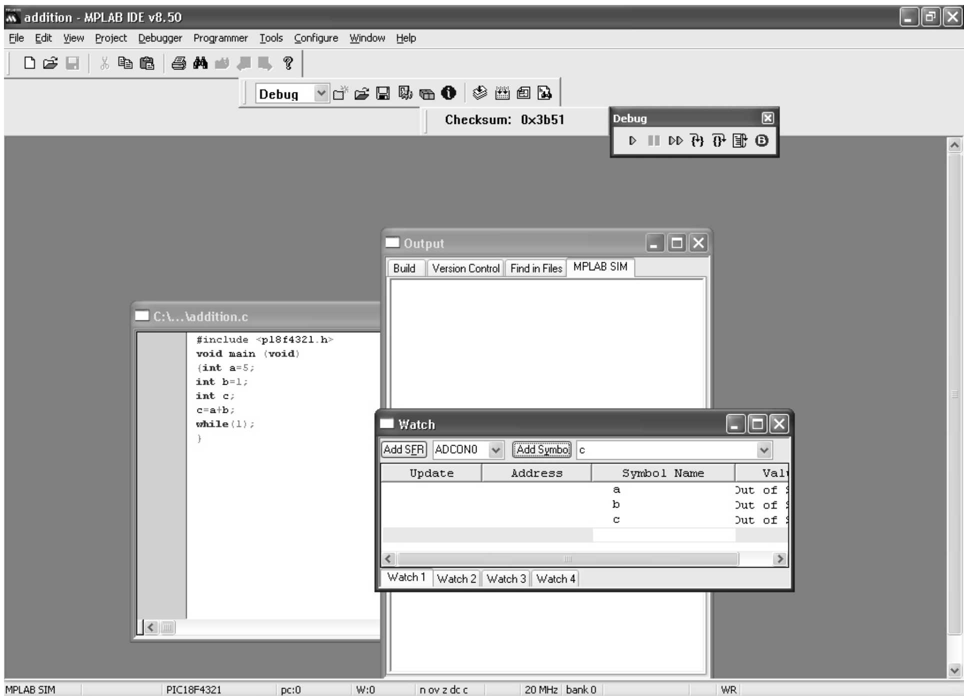
Next, click on View, and then watch to see the following:



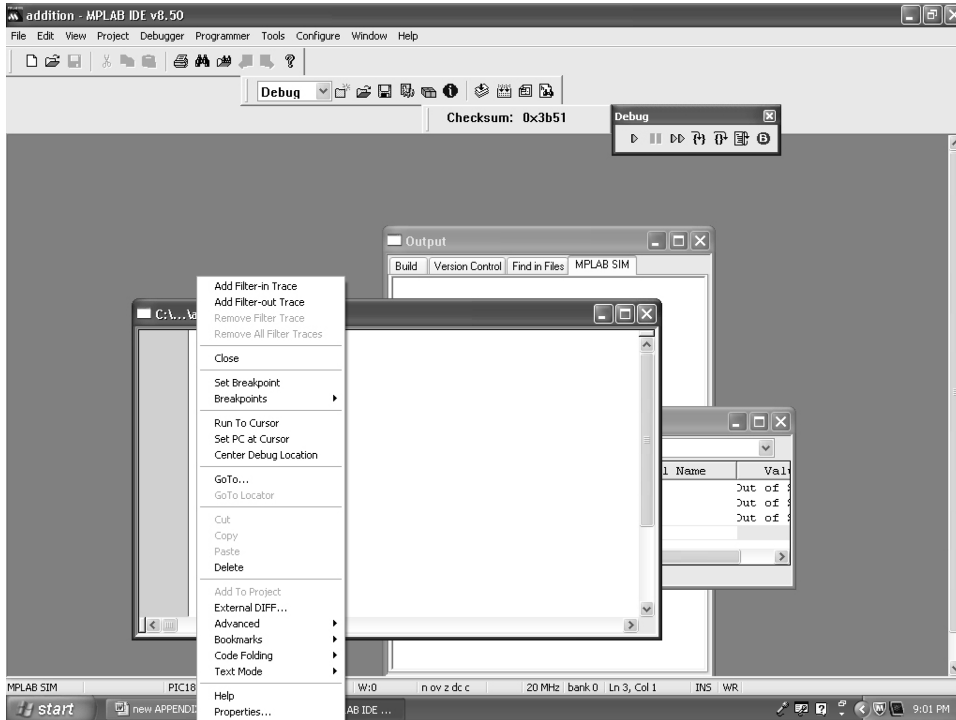
On the Watch list, you can now include locations a, b, c to monitor their contents. For example, to add 'a', simply select 'a' by scroll down using the arrow on the Add Symbol window , and then click on Add Symbol to see the following display:



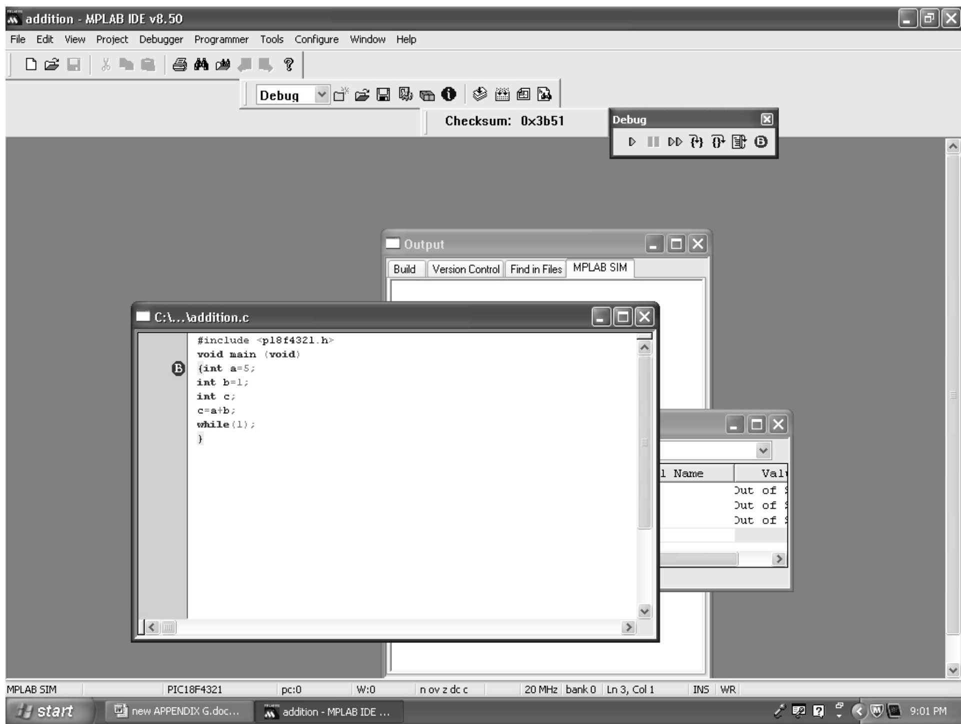
See that 'a' is displayed on the watch window. Similarly, display 'b' and 'c', and see the following screen shot:



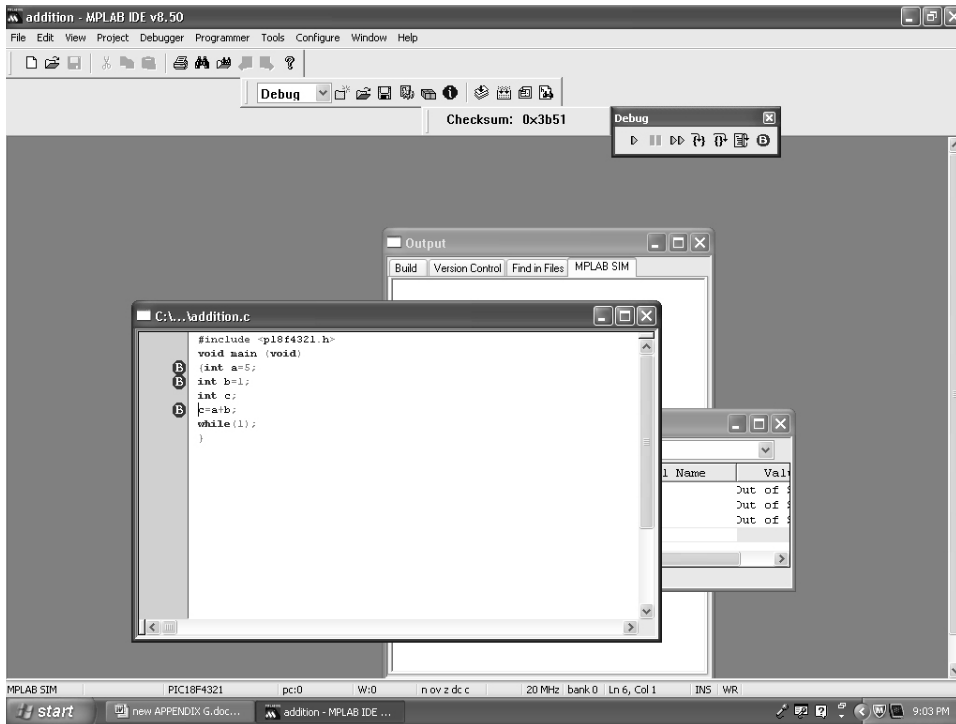
Next, insert breakpoints. Three breakpoints will be inserted for this program. One for int a = 5, one for int b = 2, and one for c = a+b. To insert a breakpoint, amove the cursor to the left of the line where breakpoint is to be inserted. For example, to insert a breakpoint at int a =5, move cursor to the left of the line, click (right) and see the following display:



Next, click on Set Breakpoint to see the following:

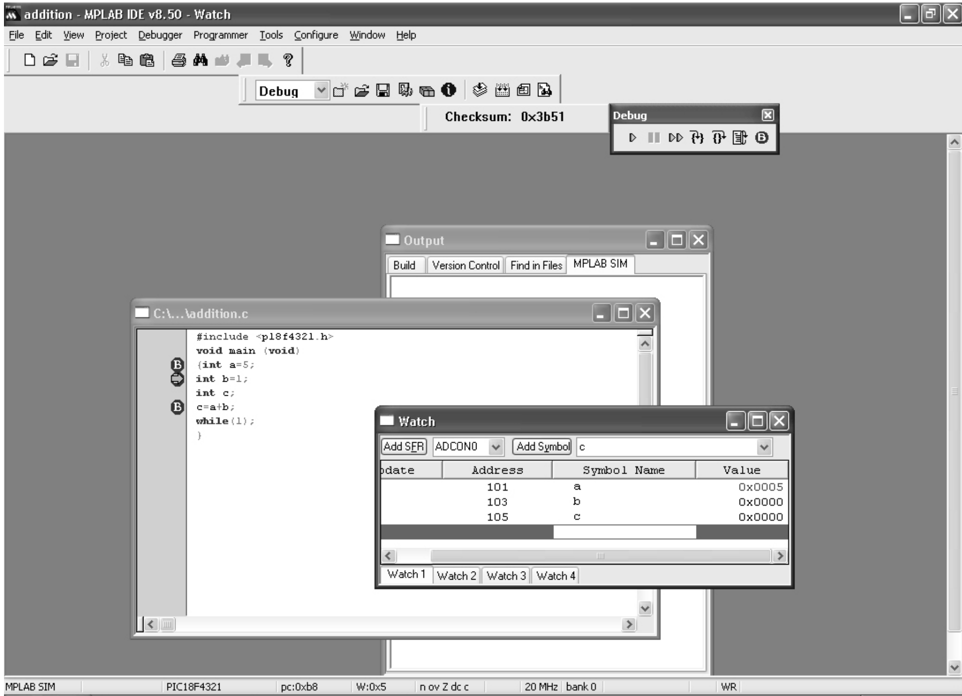


B in red on the left side of the line would indicate that the breakpoint is inserted. Similarly, insert the breakpoints for' 'b and 'c', and obtain the following display:

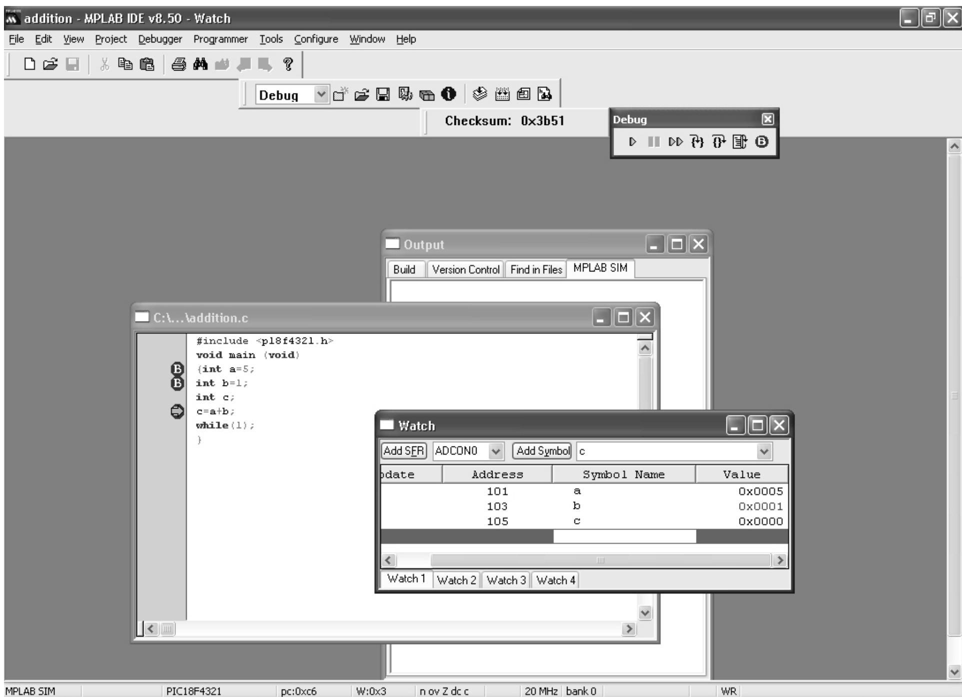


Next go to the Debug menu and Watch menu to see the contents of a, b, and c as each line is executed.

First go to Debug menu, and left click on reset (first symbol on right), and then click on the single arrow called the 'Run' arrow (left most arrow on the Debug menu), the code int a = 5; will be executed next. Click on single arrow again, the code is executed, and the following will be displayed:

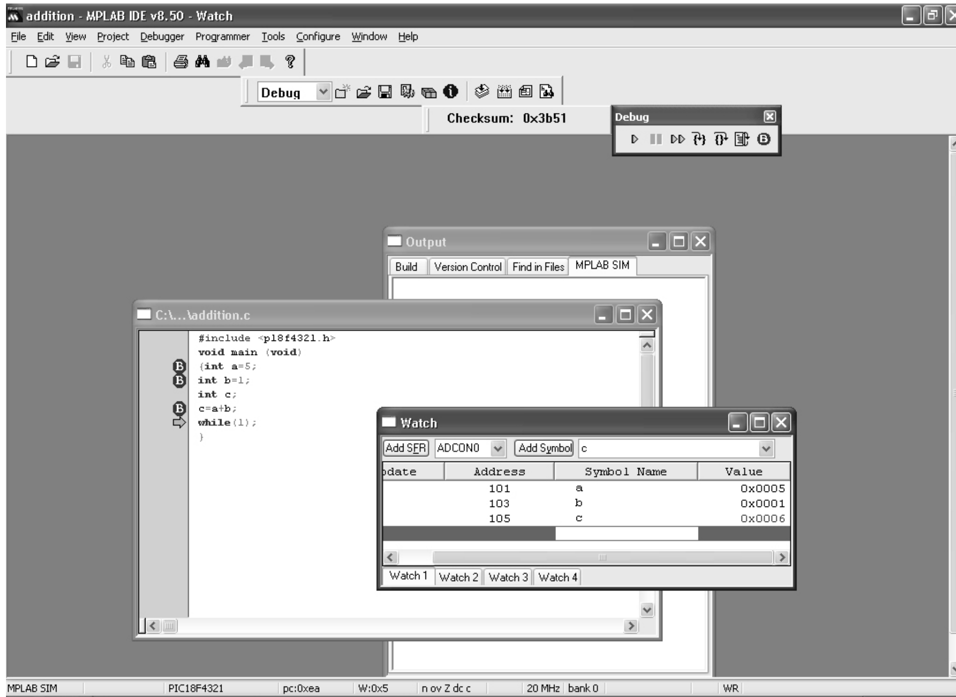


Note that 'a' contains 5. Next, left click on the single arrow, the following will be displayed:



Note that ‘b’ contains 1 after execution of int b = 1;

Next, left click on the single arrow, and then left click on Halt (icon with two vertical lines , second from left on the Debug menu) to see the final result after execution of the line
c = a +b ;
as follows:



In the above, see that ‘c’ contains 6 (final answer).

The debugging is now complete.

FIGURE H. 4 Pictorial view of connecting the PICkit™ 3 to the USB port

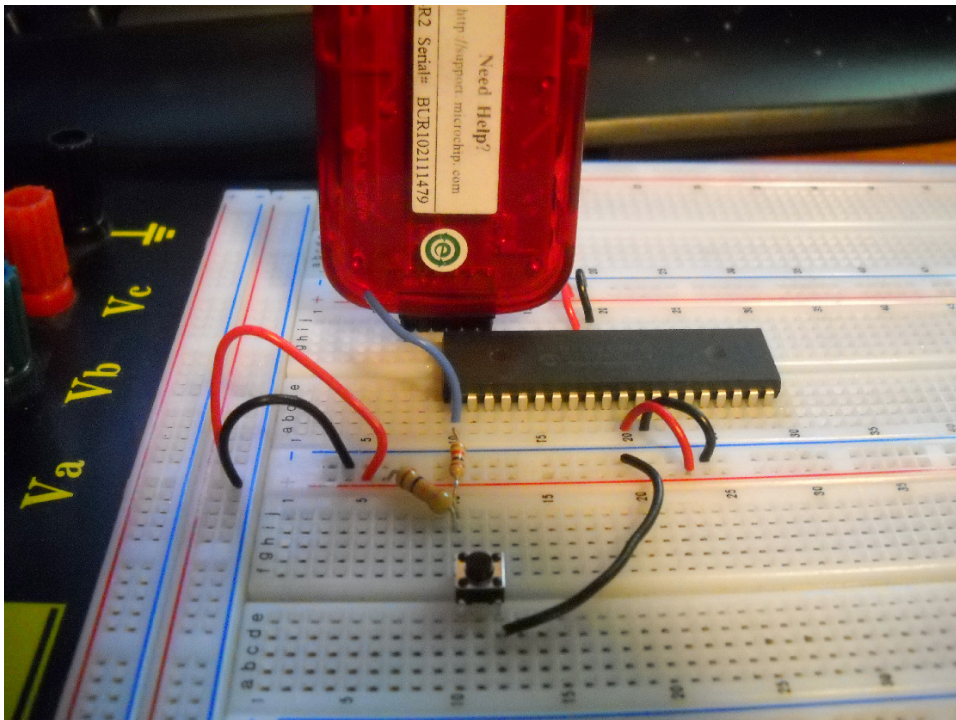
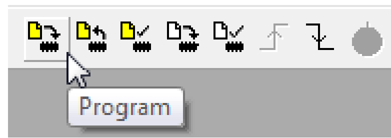


FIGURE H. 5 Connecting the PICkit™ 3 to the breadboard

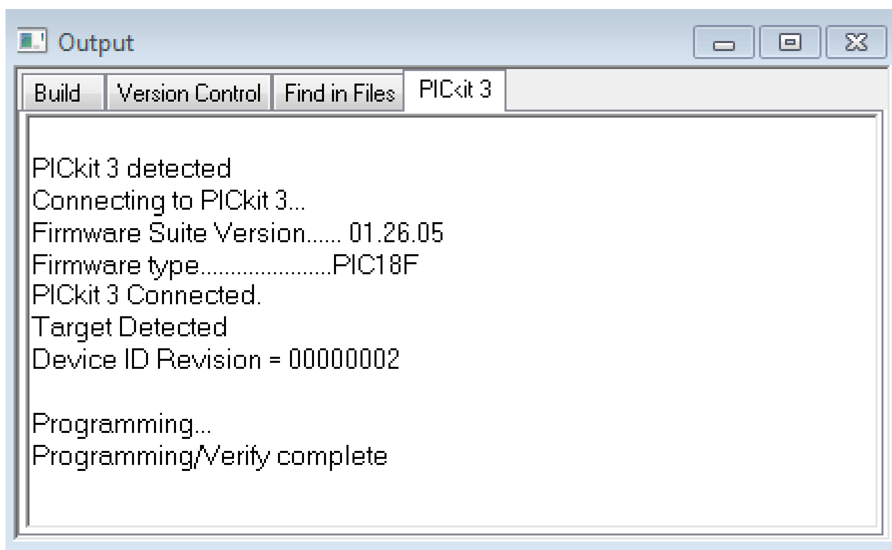
H.3 PROGRAMMING THE PIC18F4321 FROM PERSONAL COMPUTER USING THE PICkit3

In order to configure PICkit3 from the personal computer or laptop, The user needs to click on the ‘Programmer’, and then select PICkit3 as follows:

Several options will appear at the top menu to program the PIC18F4321. The screen shot is provided below:



After successfully assembling or compiling a program, click the “program” option and MPLAB will download the program into the microcontroller.
The following message will apear indicating that the code was successfully programmed and verified onto the PIC18F:



This will complete downloading the programs from the computer into the PIC18F4321 microcontroller.
Note that for PIC18F assembley language programs with I/O, the following four lines should be inserted after INCLUDE <P18F4321.INC>:

```
config OSC = INTIO2     ; Select internal clock
config WDT = OFF        ; Watch dog time OFF
config LVP = OFF        ; Low voltage programming OFF
config BOR = OFF        ; Brown-out reset OFF
```

For C programs with I/O, the following four lines should be inserted after # include <P18F4321.h>

```
# pragma config OSC = INTIO2     // Select internal clock
# pragma config WDT = OFF        // WDT OFF
# pragma config LVP = OFF        // LVP OFF
# pragma config BOR = OFF        // BOR OFF
```