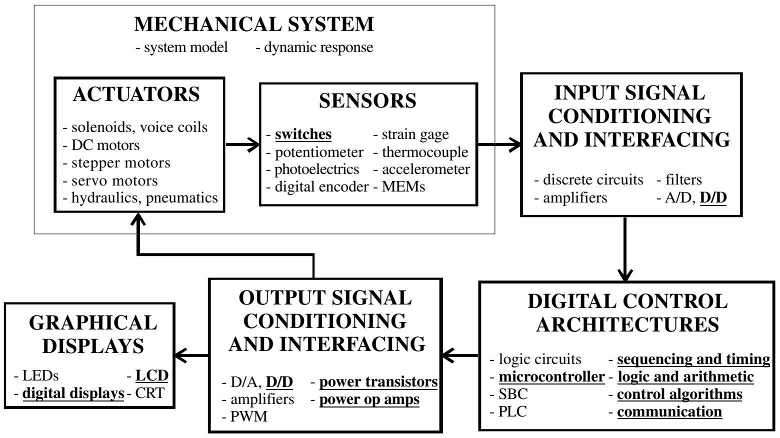
7 CHAPTER

Microcontroller Programming and Interfacing

This chapter describes how to program and interface a microcontroller. Various input and output devices are also presented. ■



CHAPTER OBJECTIVES

After you read, discuss, study, and apply ideas in this chapter, you will:

1. Understand the differences among microprocessors, microcomputers, and microcontrollers
2. Know the terminology associated with a microcomputer and microcontroller

3. Understand the architecture and principles of operation of a microcontroller
4. Understand the concepts of assembly language programming
5. Understand the basics of higher-level programming languages such as PicBasic Pro
6. Be able to write programs to control a PIC16F84 and other microcontrollers
7. Be able to interface a PIC microcontroller to input and output devices
8. Be able to design a PIC microcontroller-based mechatronic system

7.1 MICROPROCESSORS AND MICROCOMPUTERS

The digital circuits presented in Chapter 6 allow the implementation of combinational and sequential logic operations by interconnecting ICs containing gates and flip-flops. This is considered a **hardware** solution because it consists of a selection of specific ICs, which when hardwired on a circuit board, carry out predefined functions. To make a change in functionality, the hardware circuitry must be modified and may require a redesign. This is a satisfactory approach for simple design tasks (e.g., the security system presented in Section 6.6 and the digital tachometer presented in Design Example 6.1). However, in many mechatronic systems, the control tasks may involve complex relationships among many inputs and outputs, making a strictly hardware solution impractical. A more satisfactory approach in complex digital design involves the use of a microprocessor-based system to implement a **software** solution. Software is a procedural program consisting of a set of instructions to execute logic and arithmetic functions and to access input signals and control output signals. An advantage of a software solution is that, without making changes in hardware, the program can be easily modified to alter the mechatronic system's functionality.

A **microprocessor** is a single, very-large-scale-integration (VLSI) chip that contains many digital circuits that perform arithmetic, logic, communication, and control functions. When a microprocessor is packaged on a printed circuit board with other components, such as interface and memory chips, the resulting assembly is referred to as a **microcomputer** or **single-board computer.** The overall architecture of a typical microcomputer system using a microprocessor is illustrated in Figure 7.1.

The microprocessor, also called the **central processing unit** (CPU) or **microprocessor unit** (MPU), is where the primary computation and system control operations occur. The **arithmetic logic unit** (ALU) within the CPU executes mathematical functions on data structured as binary words. A **word** is an ordered set of bits, usually 8, 16, 32, or 64 bits long. The instruction decoder interprets instructions fetched sequentially from memory by the control unit and stored in the instruction register. Each instruction is a set of coded bits that commands the ALU to perform bit manipulation, such as binary addition and logic functions, on words stored in the

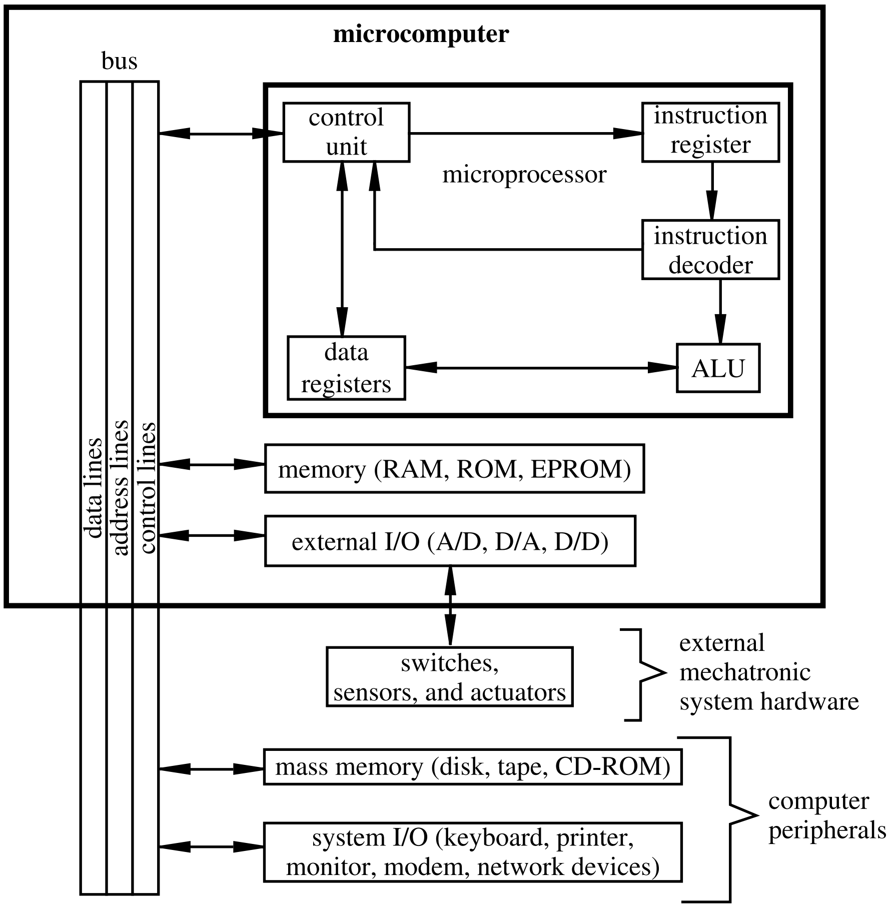

microcomputer
bus
data lines
address lines
control lines
control unit
instruction register
microprocessor
instruction decoder
data registers
ALU
memory (RAM, ROM, EPROM)
external I/O (A/D, D/A, D/D)
switches, sensors, and actuators
external mechatronic system hardware
mass memory (disk, tape, CD-ROM)
system I/O (keyboard, printer, monitor, modem, network devices)
computer peripherals

Figure 7.1 Microcomputer architecture.

CPU data registers. The ALU results are also stored in data registers and then transferred to memory by the control unit.

The **bus** is a set of shared communication lines that serves as the central nervous system of the computer. Data, address, and control signals are shared by all system components via the bus. Each component connected to the bus communicates information to and from the bus via its own bus controller. The data lines, address lines, and control lines allow a specific component to access data addressed to that component. The **data lines** are used to communicate words to and from data registers in the various system components such as memory, CPU, and input/output (I/O) peripherals. The **address lines** are used to select devices on the bus or specific data locations within memory. Devices usually have a combinational logic address decoder circuit that identifies the address code and activates the device. The **control lines** transmit read and write signals, the system clock signal, and other control signals such as system interrupts, which are described in subsequent sections.

A key to a CPU's operation is the storage and retrieval of data from a memory device. Different types of memory include **read-only memory** (ROM), **random-access memory** (RAM), and **erasable-programmable ROM** (EPROM). ROM is used for permanent storage of data that the CPU can read, but the CPU cannot write data to ROM. ROM does not require a power supply to retain its data and therefore is called nonvolatile memory. RAM can be read from or written to at any time, provided power is maintained. The data in RAM is considered volatile because it is lost when power is removed. There are two main types of RAM: **static RAM** (SRAM), which retains its data in flip-flops as long as the memory is powered, and **dynamic RAM** (DRAM), which consists of capacitor storage of data that must be refreshed (rewritten) periodically because of charge leakage. Data stored in an EPROM can be erased with ultraviolet light applied through a transparent quartz window on top of the EPROM IC. Then new data can be stored on the EPROM. Another type of EPROM is **electrically erasable (EEPROM)**. Data in EEPROM can be erased electrically and rewritten through its data lines without the need for ultraviolet light. Since data in RAM are volatile, ROM, EPROM, EEPROM, and peripheral mass memory storage devices such as magnetic disks and tapes and optical CD-ROMs are sometimes needed to provide permanent data storage.

Communication to and from the microprocessor occurs through I/O devices connected to the bus. External computer peripheral I/O devices include keyboards, printers, displays, modems, and network devices. For mechatronic applications, analog-to-digital (A/D), digital-to-analog (D/A) and digital I/O (D/D) devices provide interfaces to switches, sensors, and actuators.

The instructions that can be executed by the CPU are defined by a binary code called **machine code.** The instructions and corresponding codes are microprocessor dependent. Each instruction is represented by a unique binary string that causes the microprocessor to perform a low-level function (e.g., add a number to a register or move a register's value to a memory location). Microprocessors can be programmed using **assembly language,** which has a mnemonic command corresponding to each instruction (e.g., *ADD* to add a number to a register and *MOV* to move a register's value to a memory location). However, assembly language must be converted to machine code, using software called an **assembler,** before it can be executed on the microprocessor. When the set of instructions is small, the microprocessor is known as a **RISC** (reduced instruction-set computer) microprocessor. RISC microprocessors are cheaper to design and manufacture and usually faster. However, more programming steps may be required for complex algorithms, due to the limited set of instructions.

Programs can also be written in a higher-level language such as BASIC or C, provided that a compiler is available that can generate machine code for the specific microprocessor being used. The advantages of using a high-level language are that it is easier to learn and use, programs are easier to **debug** (the process of finding and removing errors), and programs are easier to comprehend. A disadvantage is that the resulting machine code may be less efficient (i.e., slower and require more memory) than a corresponding well-written assembly language program.

7.2 MICROCONTROLLERS

There are two branches in the ongoing evolution of the microprocessor. One branch supports CPUs for the personal computer and workstation industry, where the main constraints are high speed and large word size (32 and 64 bits). The other branch includes development of the **microcontroller,** which is a single IC containing specialized circuits and functions that are applicable to mechatronic system design. It contains a microprocessor, memory, I/O capabilities, and other on-chip resources. It is basically a microcomputer on a single IC. Examples of microcontrollers are Microchip's PIC, Motorola's 68HC11, and Intel's 8096. Factors that have driven development of the microcontroller are low cost, versatility, ease of programming, and small size. Microcontrollers are attractive in mechatronic system design since their small size and broad functionality allow them to be physically embedded in a system to perform all of the necessary control functions.

Microcontrollers are used in a wide array of applications including home appliances, entertainment equipment, telecommunication equipment, automobiles, trucks, airplanes, toys, and office equipment. All these products involve devices that require some sort of intelligent control based on various inputs. For example, the microcontroller in a microwave oven monitors the control panel for user input, updates the graphical displays when necessary, and controls the timing and cooking functions. In an automobile, there are many microcontrollers to control various subsystems, including cruise control, antilock braking, ignition control, keyless entry, environmental control, and air and fuel flow. An office fax machine controls actuators to feed paper, uses photo sensors to scan a page, sends or receives data on a phone line, and provides a user interface complete with menu-driven controls. A toy robot dog has various sensors to detect inputs from its environment (e.g., bumping into obstacles, being patted on the head, light and dark, voice commands), and an onboard microcontroller actuates motors to mimic actual dog behavior (e.g., bark, sit, and walk) based on this input. All of these powerful and interesting devices are controlled by microcontrollers and the software running on them.

Figure 7.2 is a block diagram for a typical full-featured microcontroller. Also included in the figure are lists of typical external devices that might interface to the microcontroller. The components of a microcontroller include the CPU, RAM, ROM, digital I/O ports, a serial communication interface, timers, A/D converters, and D/A converters. The CPU executes the software stored in ROM and controls all the microcontroller components. The RAM is used to store settings and values used by an executing program. The ROM is used to store the program and any permanent data. A designer can have a program and data permanently stored in ROM by the chip manufacturer, or the ROM can be in the form of EPROM or EEPROM, which can be reprogrammed by the user. Software permanently stored in ROM is referred to as **firmware.** Microcontroller manufacturers offer programming devices that can download a compiled machine code file from a PC directly to the EEPROM of the microcontroller, usually via the PC serial port and special-purpose pins on the microcontroller. These pins can usually be used for other purposes once the device is programmed. Additional EEPROM may also be available and used by the program

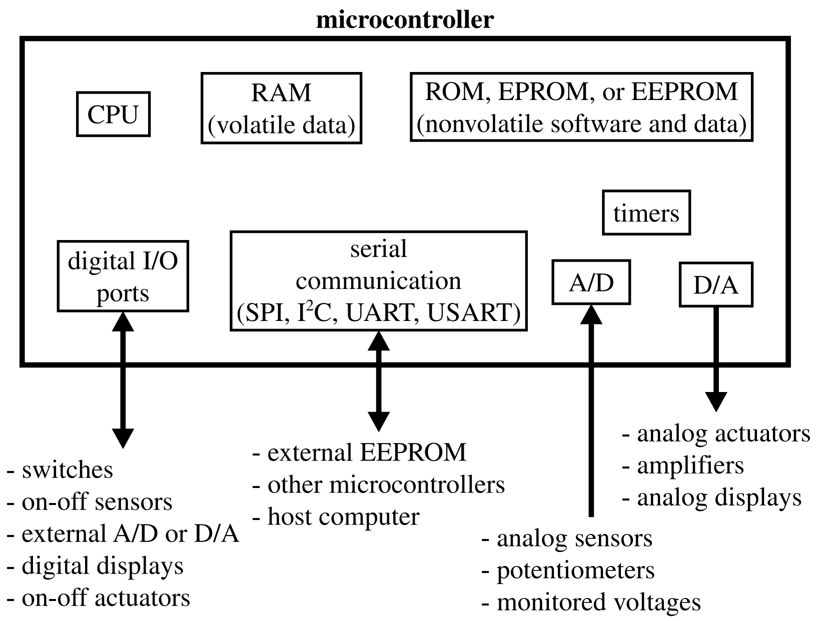


Figure 7.2 Components of a typical full-featured microcontroller.

to store settings and parameters generated or modified during execution. The data in EEPROM is nonvolatile, which means the program can access the data when the microcontroller power is turned off and back on again.

The digital I/O **ports** allow binary data to be transferred to and from the microcontroller using external pins on the IC. These pins can be used to read the state of switches and on-off sensors, to interface to external analog-to-digital and digital-to-analog converters, to control digital displays, and to control on-off actuators. The I/O ports can also be used to transmit signals to and from other microcontrollers to coordinate various functions. The microcontroller can also use a serial port to transmit data to and from external devices, provided these devices support the same serial communication protocol. Examples of such devices include external EEPROM memory ICs that might store a large block of data for the microcontroller, other microcontrollers that need to share data, and a host computer that might download a program into the microcontroller's onboard EEPROM. There are various standards or protocols for serial communication including SPI (serial peripheral interface), I^2C (interintegrated circuit), UART (universal asynchronous receiver-transmitter), and USART (universal synchronous-asynchronous receiver-transmitter).

The A/D converter allows the microcontroller to convert an external analog voltage (e.g., from a sensor) to a digital value that can be processed or stored by the CPU. The D/A converter allows the microcontroller to output an analog voltage to a nondigital device (e.g., a motor amplifier). A/D and D/A converters and their applications are discussed in Chapter 8. Onboard timers are usually provided to help create delays or ensure events occur at precise time intervals (e.g., reading the value of a sensor).

Microcontrollers typically have less than 1 kilobyte to several tens of kilobytes of program memory, compared with microcomputers whose RAM memory is measured in megabytes or gigabytes. Also, microcontroller clock speeds are slower than those used for microcomputers. For some applications, a selected microcontroller may not have enough speed or memory to satisfy the needs of the application. Fortunately, microcontroller manufacturers usually provide a wide range of products to accommodate different applications. Also, when more memory or I/O capability is required, the functionality of the microcontroller can be expanded with additional external components (e.g., RAM or EEPROM chips, external A/D and D/A converters, and other microcontrollers).

In the remainder of this chapter, we focus on the Microchip PIC microcontroller due to its wide acceptance in industry, abundant information resources, low cost, and ease of use. **PIC** is an acronym for peripheral interface controller, the phrase Microchip uses to refer to its line of microcontrollers. Microchip offers a large and diverse family of low-cost PIC products. They vary in footprint (physical size), the number of I/O pins available, the size of the EEPROM and RAM space for storing programs and data, and the availability of A/D and D/A converters. Obviously, the more features and capacity a microcontroller has, the higher the cost. Information for Microchip's entire line of products can be found on its website at *www.microchip.com*. We focus specifically on the **PIC16F84,** which is a low-cost 8-bit microcontroller with EEPROM flash memory for program and data storage. It has no built-in A/D, D/A or serial communication capability, but it supports 13 digital I/O lines and serves as a good learning platform because it is low cost and easy to program. Once you know how to interface and program one microcontroller, it is easy to extend that knowledge to other microcontrollers with different features and programming options.

■ **CLASS DISCUSSION ITEM 7.1**
Car Microcontrollers

List various automobile subsystems that you think are controlled by microcontrollers. In each case, identify all of the inputs to and outputs from the microcontroller and describe the function of the software.

7.3 THE PIC16F84 MICROCONTROLLER

The block diagram for the PIC16F84 microcontroller is shown in Figure 7.3. This diagram, along with complete documentation of all of the microcontroller's features and capabilities, can be found in the manufacturer's data sheets. The PIC16F8X data sheets are contained in a book available from Microchip and as a PDF file on its

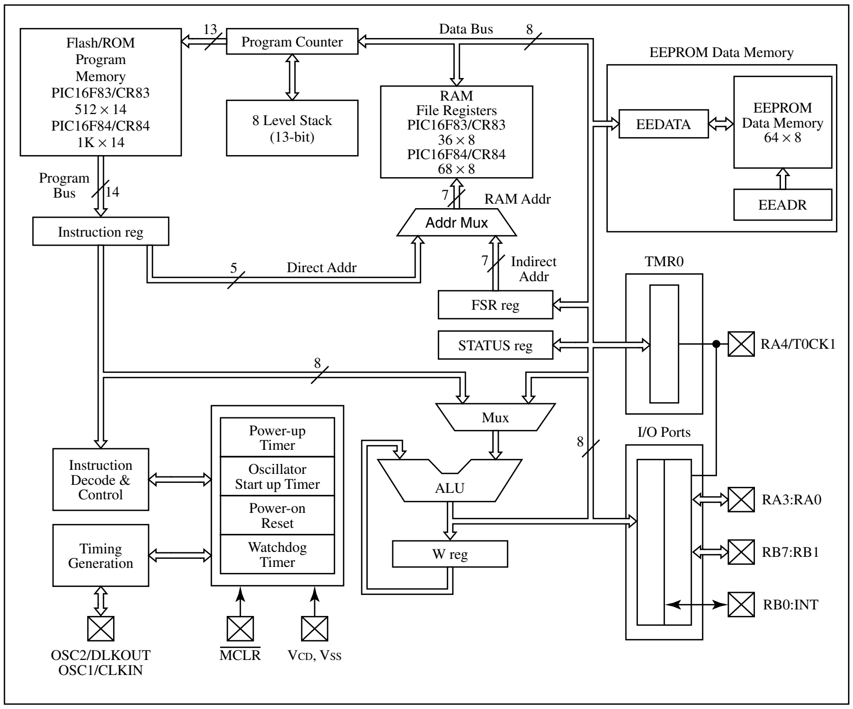


Figure 7.3 PIC16F84 block diagram. *(Courtesy of Microchip Technology Inc., Chandler, AZ)*

website. The PIC16F84 is an 8-bit CMOS microcontroller with 1792 bytes of flash EEPROM program memory, 68 bytes of RAM data memory, and 64 bytes of non-volatile EEPROM data memory. The 1792 bytes of program memory are subdivided into 14-bit words, because machine code instructions are 14 bits wide. Therefore, the EEPROM can hold up to 1024 (1 k) instructions. The PIC16F84 can be driven at a clock speed up to 10 MHz but is typically driven at 4 MHz.

When a program is compiled and downloaded to a PIC, it is stored as a set of binary machine code instructions in the flash program memory. These instructions are sequentially fetched from memory, placed in the instruction register, and executed. Each instruction corresponds to a low-level function implemented with logic circuits on the chip. For example, one instruction might load a number stored in RAM or EEPROM into the **working register,** which is also called the **W register** or **accumulator;** the next instruction might command the ALU to add a different number to the value in this register; and the next instruction might return this summed value to memory. Since an instruction is executed every four clock cycles, the PIC16F84 can do calculations, read input values, store and retrieve information from memory, and perform other functions very quickly. With a clock speed of 4 MHz an instruction is executed every microsecond and 1 million instructions can be executed every second. The microcontroller is referred to as 8-bit, because the data bus is 8 bits wide, and all data processing and storage and retrieval occur using bytes.

A useful special purpose timer, called a **watch-dog timer,** is included on PIC microcontrollers. This is a count-down timer that, when activated, needs to be continually reset by the running program. If the program fails to reset the watch-dog timer before it counts down to 0, the PIC will automatically reset itself. In a critical application, you might use this feature to have the microcontroller reset if the software gets caught in an unintentional endless loop.

The RAM, in addition to providing space for storing data, maintains a set of special purpose byte-wide locations called **file registers.** The bits in these registers are used to control the function and indicate the status of the microcontroller. Several of these registers are described below.

The PIC16F84 is packaged on an 18-pin DIP IC that has the pin schematic (pin-out) shown in Figure 7.4. The figure also shows the minimum set of external components recommended for the PIC to function properly. Table 7.1 lists the pin identifiers in natural groupings, along with their descriptions. The five pins RA0 through RA4 are digital I/O pins collectively referred to as **PORTA,** and the eight pins RB0

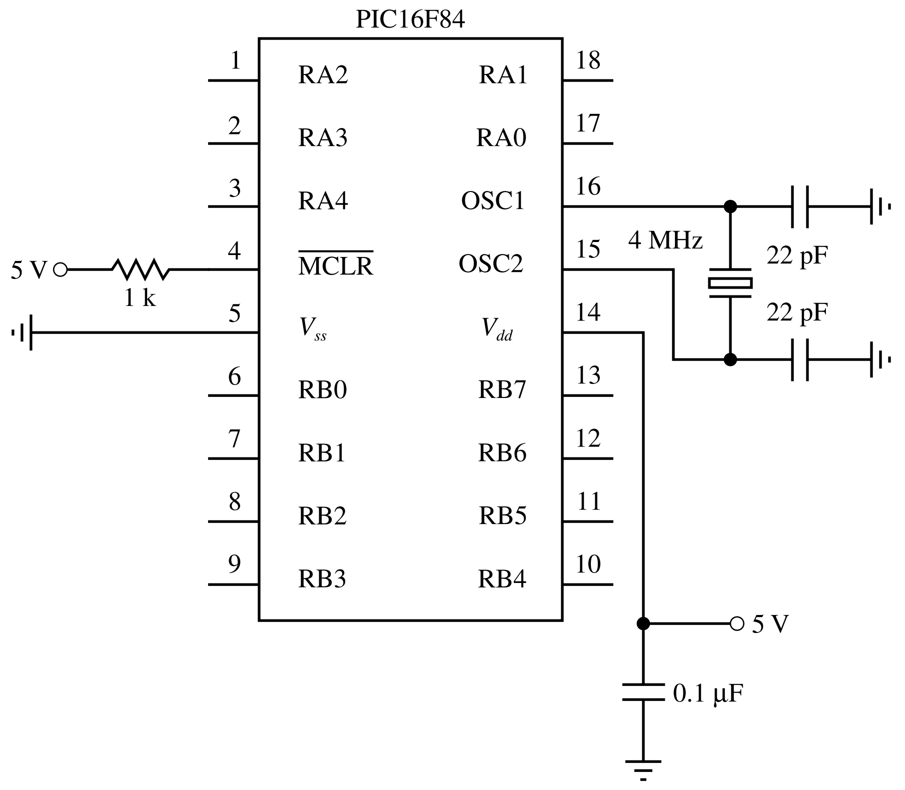


Figure 7.4 PIC16F84 pin-out and required external components.

Table 7.1 PIC16F84 pin name descriptions

Pin identifier	Description
RA[0–4]	5 bits of bidirectional I/O (PORTA)
RB[0–7]	8 bits of bidirectional I/O (PORTB)
V_{ss}, V_{dd}	Power supply ground reference (*ss*: source) and positive supply (*dd*: drain)
OSC1, OSC2	Oscillator crystal inputs
$\overline{\text{MCLR}}$	Master clear (active low)

through RB7 are digital I/O pins collectively referred to as **PORTB.** In total, there are 13 I/O lines, called **bidirectional** lines because each can be individually configured in software as an input or output. PORTA and PORTB are special purpose file registers on the PIC that provide the interface to the I/O pins. Although all PIC registers contain 8 bits, only the 5 least significant bits (LSBs) of PORTA are used.

An important feature of the PIC, available with most microcontrollers, is its ability to process interrupts. An **interrupt** occurs when a specially designated input changes state. When this happens, normal program execution is suspended while a special interrupt handling portion of the program is executed. This is discussed further in Section 7.6. On the PIC16F84, pins RB0 and RB4 through RB7 can be configured as interrupt inputs.

Power and ground are connected to the PIC through pins V_{dd} and V_{ss}. The *dd* and *ss* subscripts refer to the drain and source notation used for MOS transistors, since a PIC is a CMOS device. The voltage levels (e.g., $V_{dd} = 5$ V and $V_{ss} = 0$ V) can be provided using a DC power supply or batteries (e.g., four AA batteries in series or a 9-V battery connected through a voltage regulator). The master clear pin ($\overline{\text{MCLR}}$) is active low and provides a reset feature. Grounding this pin causes the PIC to reset and restart the program stored in EEPROM. This pin must be held high during normal program execution. This is accomplished with the pull-up resistor shown in Figure 7.4. If this pin were left unconnected (floating), the chip might spontaneously reset itself. To provide a manual reset feature to a PIC design, you can add a normally open (NO) pushbutton switch as shown in Figure 7.5. Closing the switch grounds the pin and causes the PIC to reset.

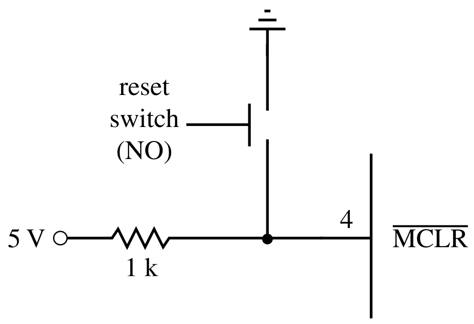


Figure 7.5 Reset switch circuit.

The PIC **clock** frequency can be controlled using different methods, including an external RC circuit, an external clock source, or a clock crystal. In Figure 7.4, we show the use of a clock crystal to provide an accurate and stable clock frequency at relatively low cost. The clock frequency is set by connecting a 4-MHz crystal across the OSC1 and OSC2 pins with the 22 pF capacitors grounded as shown in Figure 7.4.

7.4 PROGRAMMING A PIC

To use a microcontroller in mechatronic system design, software must be written, tested, and stored in the ROM of the microcontroller. Usually, the software is written and compiled using a personal computer (PC) and then downloaded to the microcontroller ROM as machine code. If the program is written in assembly language, the PC must have software called a **cross-assembler** that generates machine code for the microcontroller. An assembler is software that generates machine code for the microprocessor in the PC, whereas a cross-assembler generates machine code for a different microprocessor, in this case the microcontroller.

Various software development tools can assist in testing and debugging assembly language programs written for a microcontroller. One such tool is a **simulator,** which is software that runs on a PC and allows the microcontroller code to be simulated (run) on the PC. Most programming errors can be identified and corrected during simulation. Another tool is an **emulator,** which is hardware that connects a PC to the microcontroller in a prototype mechatronic system. It usually consists of a printed circuit board connected to the mechatronic system through ribbon cables. The emulator can be used to load and run a program on the actual microcontroller attached to the mechatronic system hardware (containing sensors, actuators, and control circuits). The emulator allows the PC to monitor and control the operation of the microcontroller while it is embedded in the mechatronic system.

The assembly language used to program a PIC16F84 consists of 35 commands that control all functions of the PIC. This set of commands is called the **instruction set** for the microcontroller. Every microcontroller brand and family has its own specific instruction set that provides access to the resources available on the chip. The complete instruction set and brief command descriptions for the PIC16F84 are listed in Table 7.2. Each command consists of a name called the **mnemonic** and, where appropriate, a list of operands. Values must be provided for each of these operands. The letters f, d, b, and k correspond, respectively, to a file register address (a valid RAM address), result destination (0: W register, 1: file register), bit number (0 through 7), and literal constant (a number between 0 and 255). Note that many of the commands refer to the working register W, also called the accumulator. This is a special CPU register used to temporarily store values (e.g., from memory) for calculations or comparisons. At first, the mnemonics and

Table 7.2 PIC16F84 instruction set

Mnemonic and operands	Description
ADDLW k	Add literal and W
ADDWF f, d	Add W and f
ANDLW k	AND literal with W
ANDWF f, d	AND W with f
BCF f, b	Bit clear f
BSF f, b	Bit set f
BTFSC f, b	Bit test f, skip if clear
BTFSS f, b	Bit test f, skip if set
CALL k	Call subroutine
CLRF f	Clear f
CLRW	Clear W
CLRWDT	Clear watch-dog timer
COMF f, d	Complement f
DECF f, d	Decrement f
DECFSZ f, d	Decrement f, Skip if 0
GOTO k	Go to address
INCF f, d	Increment f
INCFSZ f, d	Increment f, skip if 0
IORLW k	Inclusive OR literal with W
IORWF f, d	Inclusive OR W with f
MOVF f, d	Move f
MOVLW k	Move literal to W
MOVWF f	Move W to f
NOP	No operation
RETFIE	Return from interrupt
RETLW k	Return with literal in W
RETURN	Return from subroutine
RLF f, d	Rotate f left 1 bit
RRF f, d	Rotate f right 1 bit
SLEEP	Go into standby mode
SUBLW k	Subtract W from literal
SUBWF f, d	Subtract W from f
SWAPF f, d	Swap nibbles in f
XORLW k	Exclusive OR literal with W
XORWF f, d	Exclusive OR W with f

descriptions in the table may seem cryptic, but after you compare functionality with the terminology and naming conventions, it becomes much more understandable. In Example 7.1, we introduce a few of the statements and provide some examples. We illustrate how to write a complete assembly language program in Example 7.2.

For more information (e.g., detailed descriptions and examples of each assembly statement), refer to the PIC16F8X data sheet available on Microchip's website (*www.microchip.com*).

EXAMPLE 7.1 Assembly Language Instruction Details

Here, we provide more detailed descriptions and examples of a few of the assembly language instructions to help you better understand the terminology and the naming conventions.

BCF f, b
(read *BCF* as "bit clear f")
clears bit *b* in file register *f* to 0, where the bits are
numbered from 0 (LSB) to 7 (MSB)

For example, *BCF PORTB, 1* makes bit 1 in PORTB go low (where PORTB is a constant containing the address of the PORTB file register). If PORTB contained the hexadecimal (hex) value FF (binary 11111111) originally, the final value would be hex FC (binary 11111101). If PORTB contained the hex value A8 (binary 10101000) originally, the value would remain unchanged.

MOVLW k
(read *MOVLW* as "move literal to W")
stores the literal constant *k* in the accumulator (the W register)

For example, *MOVLW 0xA8* would store the hex value A8 in the W register. In assembly language, hexadecimal constants are identified with the *0x* prefix.

RLF f, d
(read *RLF* as "rotate f left")
shifts the bits in file register *f* to the left 1 bit, and stores the result in *f* if *d* is 1
or in the accumulator (the W register) if *d* is 0. The value of the LSB will become 0,
and the original value of the MSB is lost.

For example, if the current value in PORTB is hex 1F (binary 00011111), then *RLF PORTB, 1* would change the value to hex 3E (binary 00111110).

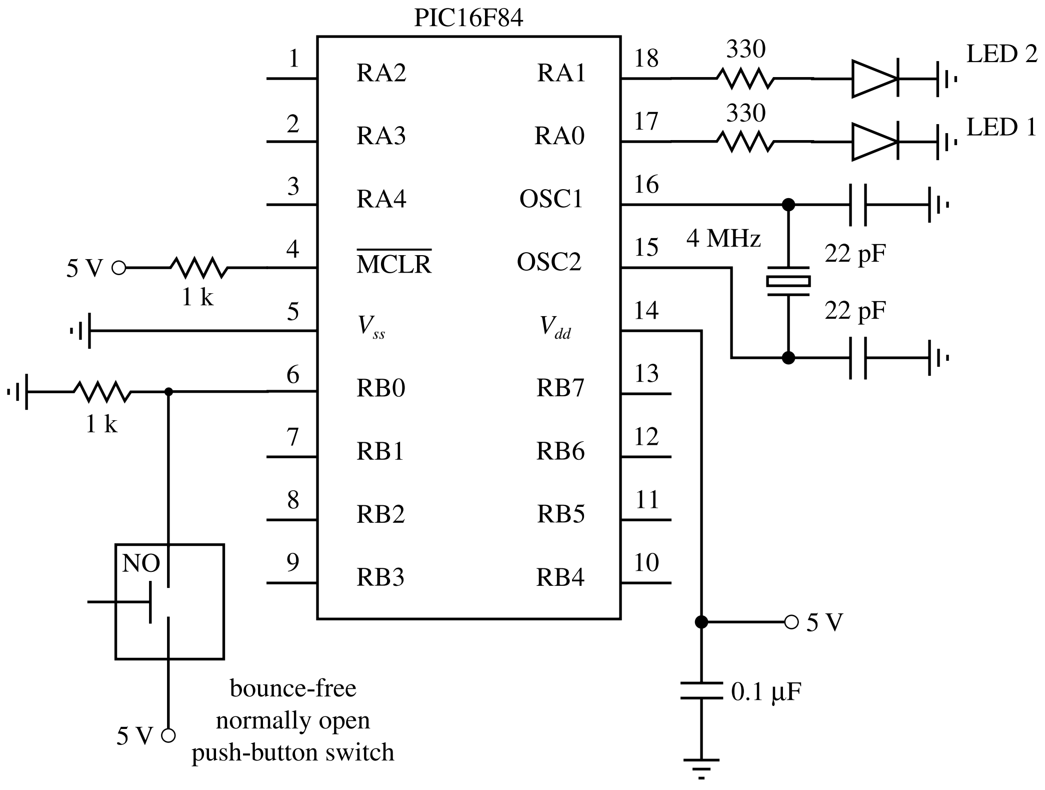
SWAPF f, d
(read *SWAPF* as "swap nibbles in f")
exchanges the upper and lower nibbles (a nibble is 4 bits or half a byte)
of file register *f* and stores the result in *f* if *d* is 1
or in the accumulator (the W register) if *d* is 0

For example, if the memory location at address hex 10 contains the value hex AB, then *SWAPF 0x10, 0* would store the value hex BA in the W register. *SWAPF 0x10, 1* would change the value at address hex 10 from hex AB to hex BA.

EXAMPLE 7.2 Assembly Language Programming Example

The purpose of this example is to write an assembly language program that will turn on an LED when the user presses a push-button switch. When the switch is released, the LED is to turn off. After the switch is pressed and released a specified number of times, a second

LED is to turn on and stay lit. The hardware required for this example is shown in the following figure:



The push-button switch is assumed to be bounce free, implying that when it is pressed and then released, a single pulse is produced (the signal goes high when it is pressed and goes low when it is released).

Assembly language code that will accomplish the desired task follows the text below. A remark or comment can be inserted anywhere in a program by preceding it with a semicolon (;). Comments are used to clarify the associated code. The assembler ignores comments when generating the hex machine code. The first four active lines (*list*. . . *target*) are assembler directives that designate the processor and define constants that can be used in the remaining code. Defining constants (with the *equ* directive) at the beginning of the program is a good idea because the names, rather than hex numbers, are easier to read and understand in the code and because the numbers can be conveniently located and edited later. Assembly language constants such as addresses and values are written in hexadecimal, denoted with a *0x* prefix.

The next two lines of code, starting with *movlw,* move the literal constant *target* into the W register and then from the W register into the *count* address location in memory. The target value (0x05) will be decremented until it reaches 0x00. The next section of code initializes the special function registers PORTA and TRISA to allow output to pins RA0 and RA1, which drive the LEDs. These registers are located in different banks of memory, hence the need for the *bsf* and *bcf* statements in the program. All capitalized words in the program are constant addresses or values predefined in the processor-dependent include file (p16f84.inc). The function of the TRISA register is discussed more later; but by clearing the bits in the register, the PORTA pins are configured as outputs.

The main loop uses the *btfss* (bit test in file register; skip the next instruction if the bit is set) and *btfsc* (bit test in file register; skip the next instruction if the bit is clear) statements to test the state of the signal on pin RB0. The tests are done continually within loops created by the *goto* statements. The words *begin* and *wait* are statement labels used as targets for the *goto* loops. When the switch is pressed, the state goes high and the statement *btfss* skips the *goto begin* instruction; then LED1 turns on. When the switch is released, pin RB0 goes low and the statement btfsc skips the *goto wait* instruction; then LED1 turns off.

After the switch is released and LED1 turns off, the statement *decfsz* (decrement file register; skip the next instruction if the count is 0) executes. The *decfsz* decrements the *count* value by 1. If the *count* value is not yet 0, *goto begin* executes and control shifts back to the label *begin*. This resumes execution at the beginning of the main loop, waiting for the next switch press. However, when the *count* value reaches 0, *decfsz* skips the *goto begin* statement and LED2 is turned on. The last *goto begin* statement causes the program to again jump back to the beginning of the main loop.

```
; bcount.asm (program file name)

; Program to turn on an LED every time a push-button switch is pressed and turn on
;  a second LED once it has been pressed a specified number of times

; I/O:
;  RB0: bounce-free push-button switch (1:pressed, 0:not pressed)
;  RA0: count LED (first LED)
;  RA1: target LED (second LED)

; Define the processor being used
list   p=16f84
include <p16F84.inc>

; Define the count variable location and the initial count-down value
count equ 0x0c                       ; address of count-down variable
target equu 0x05                     ; number of presses required

        ; Initialize the counter to the target number of presses
        movlw target                 ; move the count-down value into the
                                     ;  W register
        movwf count                  ; move the W register into the count memory
                                     ;  location

        ; Initialize PORTA for output and make sure the LEDs are off
        bcf      STATUS, RP0         ; select bank 0
        clrf     PORTA               ; initialize all pin values to 0
        bsf      STATUS, RP0         ; select bank 1
        clrf     TRISA              ; designate all PORTA pins as outputs
        bcf      STATUS, RP0        ; select bank 0

; Main program loop
        ; Wait for the push-button switch to be pressed
begin
        btfss  PORTB, 0
        goto   begin

        ; Turn on the count LED1
        bsf    PORTA, 0

        ; Wait for the push-button switch to be released
wait
        btfsc  PORTB, 0
        goto   wait

        ; Turn off the count LED1
        bcf    PORTA, 0

        ; Decrement the press counter and check for 0
        decfsz count, 1
        goto begin    ; continue if count-down is still > 0

        ; Turn on the target LED2
        bsf           PORTA, 1

        goto begin    ; return to the beginning of the main loop

        end           ; end of instructions
```

■ CLASS DISCUSSION ITEM 7.2
Decrement Past 0

In Example 7.2, the *decfsz* statement is used to count down from 5 to 0. When 0 (0x00) is decremented, the resulting value at address *count* will be 255 (0xFF), then 254 (0xFE), and so forth. What effect, if any, does this have on the operation of the code and the LED display?

Learning to program in assembly language can be very difficult at first and may result in errors that are difficult to debug. Fortunately, high-level language compilers are available that allow us to program a PIC at a more user-friendly level. The particular programming language we discuss in the remainder of the chapter is **PicBasic Pro.** The compiler for PicBasic Pro is available from microEngineering Labs, Inc. (*www.melabs.com*). PicBasic Pro is much easier to learn and use than assembly language. It provides easy access to all of the PIC capabilities and provides a rich set of advanced functions and features to support various applications. PicBasic Pro is also closely compatible with the BASIC language used to control Basic Stamp minicontrollers (Parallax, Inc., Rocklin, CA), which are popular development boards that utilize the PIC microcontroller.

For a wealth of additional information on the PIC and related products, refer to the “Microchip PIC Microcontroller Resources” link on the website for this textbook (*www.engr.colostate.edu/~dga/mechatronics.html*). This site contains many useful links to manufacturers and other useful Web pages that provide resources and information on support literature, useful accessories, and PicBasic Pro.

7.5 PICBASIC PRO

PIC programs can be written in a form of BASIC called **PicBasic Pro.** The PicBasic Pro complier can compile these programs, producing their assembly language equivalents, and this assembly code can then be converted to hexadecimal machine code (hex code) that can be downloaded directly to the PIC flash EEPROM through a programming device attached to a PC. Once loaded, the program begins to execute when power is applied to the PIC if the necessary additional components, such as those shown in Figure 7.4, are connected properly.

We do not intend to cover all aspects of PicBasic Pro programming. Instead, we present an introduction to some of the basic programming principles, provide a brief summary of the statements, and then provide some examples. The PicBasic Pro Complier manual available on microEngineering’s website is a necessary supplement to this chapter if you need to solve problems requiring more functionality than the examples we present here. If you have not used a programming language such as BASIC, C, C++, or FORTRAN, then Section 7.5.1 may be challenging for you. Even if this the case, as you read through the examples that follow, the concepts should become clearer.

7.5.1 PicBasic Pro Programming Fundamentals

To illustrate the fundamentals of PicBasic Pro, we start with a very simple example. The goal is to write a program to turn on an LED for a second, then turn it off for a second, repeating for as long as power is applied to the circuit. The code for this program, called *flash.bas,* follows. The hardware required is shown in Figure 7.6. Pin RA2 is used as an output to source current to an LED through a current limiting resistor. The first two lines in the program are comments that identify the program and its purpose. **Comment lines** must begin with an apostrophe. On any line, information on the right side of an apostrophe is treated as a comment and ignored by the compiler. The label *loop* allows the program to return control to this label at a later time using the *Goto* command. The statement *High PORTA.2* causes pin RA2 to go high and the LED turns on. The *Pause* command delays execution of the next line of code a given number of milliseconds (in this case, 1000, which corresponds to 1000 milliseconds or 1 second). The statement *Low PORTA.2* causes pin RA2 to go low, turning the LED off. The next *Pause* causes a 1 sec delay before executing the next line. The *Goto loop* statement returns control to the program line labeled *loop,* and the program continues indefinitely. The *End* statement on the last line of the

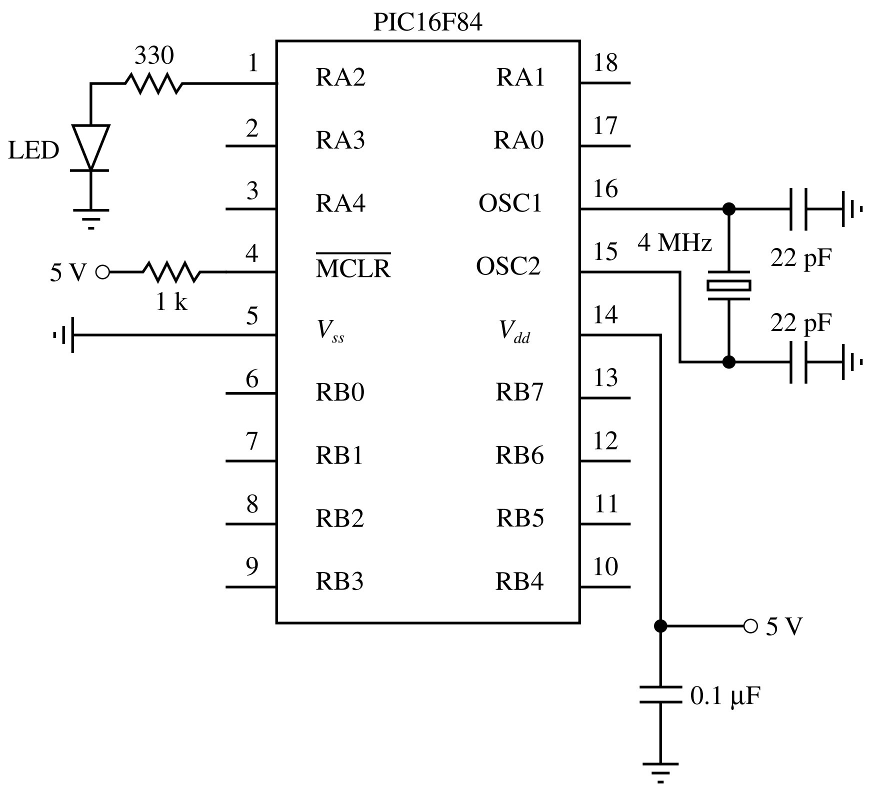


Figure 7.6 Circuit schematic for the flash.bas example.

program terminates execution. In this example, the loop continues until power is removed, and the *End* statement is never reached. However, to be safe you should always terminate a program with an *End* statement.

```
' flash.bas
' Example program to flash an LED once every two seconds

loop:  High PORTA.2                 ' turn on LED connected to pin RA2
       Pause 1000                   ' delay for one second (1000 ms)

       Low PORTA.2                  ' turn off LED connected to pin RA2
       Pause 1000                   ' delay for one second (1000 ms)

       Goto loop                    ' go back to label "loop" and repeat
                                    '  indefinitely
       End
```

As illustrated in the simple *flash.bas* example, PicBasic Pro programs consist of a sequence of program statements that are executed one after another. The programmer must be familiar with the syntax of PicBasic Pro, but it will be easier to learn and debug than assembly language programs. **Comments,** any text preceded by an apostrophe, can be placed anywhere in the program to help explain the purpose of specific lines of the code. Any user-defined labels, variable names, or constant names are called **identifiers** (e.g., *loop* in the preceding example). You can use any combination of characters for these identifiers, provided they do not start with a number. Also, identifiers must be different from all the words reserved by PicBasic Pro (e.g., keywords like *High* and *Low*). Identifiers may be any length, but PicBasic Pro ignores all the characters after the first 32. PicBasic Pro is not case sensitive so it does not matter whether or not letters are capitalized. Therefore, any combination of lower or upper case letters can be used for identifiers, including labels, variables, statements, and register or bit references. For example, to PicBasic Pro, *High* is equivalent to *HIGH* or *high*. However, when writing code, it is best to use a consistent pattern that helps make the program more readable. In the examples presented in this chapter, all variables and labels are written in lower case, all keywords in statements are written with an initial capital, and all registers and constants are written in upper case.

In some applications, you need to store a value for later use in the program (e.g., a counter that is incremented each time a push-button switch is pressed). PicBasic Pro lets you create variables for this purpose. The syntax for creating a **variable** is

```
name Var type
```
(7.1)

where *name* is the identifier to be used to refer to the variable and *type* describes the type and corresponding data storage size of the variable. The type can be BIT to store a single bit of information (0 or 1), BYTE to store an 8-bit positive integer that can range from 0 to 255 ($2^8 - 1$), or WORD to store a 2-byte (16-bit) positive integer that can range from 0 to 65,535 ($2^{16} - 1$). The following lines are examples of variable declarations and assignment statements that store values in variables:

```
my_bit Var BIT
my_byte Var BYTE

my_bit = 0
my_byte = 187
```

The *Var* keyword can also be used to give identifier names to I/O pins or to bits within a byte variable using the following syntax:

```
name Var byte.bit
```
(7.2)

For example,

```
led Var PORTB.0
lsb Var my_byte.0
```

would designate *led* as the state of pin RB0 and *lsb* as bit 0 of byte variable *my_byte*.

Another type of variable is an **array,** which can be used to store a set or vector of numbers. The syntax for declaring an array is

```
name Var type[size]
```
(7.3)

where *type* defines the storage type (BIT, BYTE, or WORD) and *size* indicates the number of elements in the array. A particular **element** in an array can be accessed or referenced with the following syntax:

$$\texttt{name[i]} \tag{7.4}$$

where *i* is the index of the element being referenced. The elements are numbered from 0 to size −1. For example, *values Var byte[5]* would define an array of 5 bytes, and the elements of the array would be *values[0], values[1], values[2], values[3],* and *values[4].*

Constants can be given names in a program using the same syntax as that used for variables (Equation 7.1) by replacing the *Var* keyword with *Con* and by replacing the *type* keyword by a constant value. When specifying values in a program, the prefix *$* denotes a hexadecimal value and the prefix *%* denotes a binary value. If there is no prefix, the number is assumed to be a decimal value. For example, with the following variable and constant definitions, all of the assignment statements that follow are equivalent:

```
number Var BYTE
CONSTANT Con 23

number = 23
number = CONSTANT
number = %10111
number = $17
```

Normally, constants and results of calculations are assumed to be unsigned (i.e., zero or positive), but certain functions, such as *Sin* and *Cos,* use a different byte format, where the MSB is used to represent the sign of the number. In this case, the byte can take on values between −127 and 127. Some of the fundamental expressions using **mathematical operators** and functions available in PicBasic Pro are listed in Table 7.3. See other operators and functions, more details, and examples in the PicBasic Pro Compiler manual.

Table 7.3 Selected PicBasic Pro math operators and functions

Math operator or function	Description
A + B	Add A and B
A – B	Subtract B from A
A * B	Multiply A and B
A / B	Divide A by B
A << n	Shift A n bits to the left
A >> n	Shift A n bits to the right
COS A	Return the cosine of A
A MAX B	Return the maximum of A and B
A MIN B	Return the minimum of A and B
SIN A	Return the sine of A
SQR A	Return the square root of A
A & B	Return the bitwise AND of A and B
A | B	Return the bitwise OR of A and B
A ^ B	Return the bitwise Exclusive OR of A and B
~A	Return the bitwise NOT of A

There is a collection of PicBasic Pro statements that allow you to read, write, and process inputs from and outputs to the I/O port pins. To refer to an I/O pin, you use the following syntax:

```
port_name.bit                    (7.5)
```

where *port_name* is the name of the port (PORTA or PORTB) and *bit* is the bit location specified as a number between 0 and 7. For example, to refer to pin RB1, you would use the expression PORTB.1. When a bit is configured as an output, the output value (0 or 1) on the pin can be set with a simple assignment statement (e.g., *PORTB.1 = 1*). When a bit is configured as an input, the value on the pin (0 or 1) can be read by referencing the bit directly (e.g., *value = PORTA.2*). All of the bits within a port can be set at one time using an assignment statement of the following form:

```
port_name = constant             (7.6)
```

where *constant* is a number between 0 and 255 expressed in binary, hexadecimal, or decimal. For example, *PORTA = %00010001* sets the PORTA.0 and PORTA.4 bits to 1, and sets all other bits to 0. Since the three most significant bits in PORTA are not used, *PORTA = %10001* is equivalent.

The I/O status of the PORTA and PORTB bits are configured in two special registers called **TRISA** and **TRISB.** The prefix *TRIS* is used to indicate that tristate gates control whether or not a particular pin provides an input or an output. The input and output circuits for PORTA and PROTB on the PIC16F84 are presented in Section 7.8, where we deal with interfacing. When a TRIS register bit is set high (1), the corresponding PORT bit is considered an input, and when the TRIS bit is low (0), the corresponding PORT bit is considered an output. For example, TRISB = %01110000 would designate pins RB4, RB5, and RB6 as inputs and the other PORTB pins as outputs. At power-up, all TRIS register bits are set to 1 (i.e., TRISA and TRISB are both set to $FF or %11111111), so all pins in PORTA and PORTB are treated as inputs by default. You must redefine them if necessary for your application, in the initialization statements in your program.

The port bit access syntax described by Equation 7.5 can also be used to access individual bits in byte variables. For example, given the following declarations,

```
my_byte Var byte
my_array Var byte[10]
```

my_byte.3 = 1 would set bit 3 in *my_byte* to 1 and *my_array[9].7 = 0* would set the MSB of the last element of *my_array* to 0. All bits within a variable can be set by assigning a value or expression to the variable with an **assignment statement:**

```
variable = expression            (7.7)
```

For example,

```
my_byte = 231
my_array[2] = my_byte - 12
```

Table 7.4 PicBasic Pro logical comparison operators

Operator	Description
= or ==	equal
<> or !=	not equal
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Two important features in any programming language are statements to perform logical comparisons and statements to branch, loop, and iterate. In PicBasic Pro, logic is executed within an *If* . . . *Then* . . . *Else* . . . statement construct, where a logical comparison is made and if the result of the comparison is true, then the statements after *Then* are executed; otherwise, the statements after *Else* are executed. PicBasic Pro supports the **logical comparison operators** listed in Table 7.4. The keywords **And, Or, Xor** (exclusive Or), and **Not** can also be used in conjunction with parentheses to create general Boolean expressions for use in logical comparisons. Example 7.3 and other examples to follow illustrate use of logical expressions.

A PicBasic Pro Boolean Expression — EXAMPLE 7.3

The following PicBasic Pro statement turns on a motor controlled by a transistor connected to pin RA0 when the state of a switch connected to pin RB0 is high or when the state of a switch connected to pin RB1 is low, and when a byte variable *count* has a value less than or equal to 10:

```
If (((PORTB.0 == 1) OR (PORTB.1 == 0)) And (count <= 10)) Then
  High PORTA.0
```

The simplest form of looping is to use a statement label with a *goto* statement as illustrated earlier in the *flash.bas* example. PicBasic Pro also provides *For* . . . *Next* and *While* . . . *Wend* statement structures to perform looping and iteration. These constructs are demonstrated in examples through the remainder of the chapter.

Table 7.5 lists all of the PicBasic Pro statements with corresponding descriptions. Complete descriptions and examples of the statements and their associated parameters and variables can be found in the PicBasic Pro Compiler manual available online at microEngineering Inc.'s website (*www.melabs.com*). In Table 7.5, the keywords are capitalized and the parameters or variables that follow the keywords are shown in lower case. Also, any statement parameters enclosed within curly brackets ({. . .}) are optional. All the features and operators just presented and all of the statements listed in the table are built from the limited set of assembly language instructions given in Table 7.2. PicBasic Pro eliminates the cryptic assembly language details for you and provides a high-level, more user-friendly language.

Table 7.5 PicBasic Pro statement summary

Statement	Description
@ assembly statement	Insert one line of assembly language code
ADCIN channel, var	Read the on-chip analog to digital converter (if there is one)
ASM . . . ENDASM	Insert an assembly language code section consisting of one or more statements
BRANCH index, [label1{, label2, . . .}]	Computed goto that jumps to a label based on index
BRANCHL index, [label1{, label2, . . .}]	Branch to a label that can be outside of the current page of code memory (for PICs with more than 2 k of program ROM)
BUTTON pin, down_state, auto_repeat_delay, auto_repeat_rate, countdown_variable, action_state, label	Read the state of a pin and perform debounce (by use of a delay) and autorepeat (if used within a loop)
CALL assembly_label	Call an assembly language subroutine
CLEAR	Zero all variables
CLEARWDT	Clear the watch-dog timer
COUNT pin, period, var	Count the number of pulses occurring on a pin during a period
DATA {@ location,} constant1{, constant2, . . .}	Define initial contents of the on-chip EEPROM (same as the EEPROM statement)
DEBUG item1{, item2, . . .}	Asynchronous serial output to a pin at a fixed baud rate
DEBUGIN {timeout, label,} [item1{,{item2, . . .}]	Asynchronous serial input from a pin at a fixed baud rate
DISABLE	Disable ON INTERRUPT and ON DEBUG processing
DISABLE DEBUG	Disable ON DEBUG processing
DISABLE INTERRUPT	Disable ON INTERRUPT processing
DTMFOUT pin, {on_ms, off_ms,} [tone1{, tone2, . . .}]	Produce touch tones on a pin
{EEPROM {@ location,} constant1{, constant2, . . .}}	Define initial contents of on-chip EEPROM (same as the DATA statement)
ENABLE	Enable ON INTERRUPT and ON DEBUG processing
ENABLE DEBUG	Enable ON DEBUG processing
ENABLE INTERRUPT	Enable ON INTERRUPT processing
END	Stop execution and enter low power mode

FOR count = start TO end {STEP {-} inc}
{body statements}
NEXT {count}
	Repeatedly execute statements as count goes from start to end in fixed increment

FREQOUT pin, on_ms, freq1{, freq2}	Produce up to two frequencies on a pin
GOSUB label	Call a PicBasic subroutine at the specified label
GOTO label	Continue execution at the specified label
HIGH pin	Make pin output high
HSERIN {parity_label,} {time_out, label,} [item1{, item2, . . .}]	Hardware asynchronous serial input (if there is a hardware serial port)
HSEROUT [item1{, item2, . . .}]	Hardware asynchronous serial output (if there is a hardware serial port)
I2CREAD data_pin, clock_pin, control,{ address,} [var1{, var2, . . .}]{, label}	Read bytes from an external I^2C serial EEPROM device
I2CWRITE data_pin, clock_pin, control,{ address,} [var1{, var2, . . .}]{, label}	Write bytes to an external I^2C serial EEPROM device
IF log_comp THEN label	Conditionally jump to a label

IF log_comp THEN
true_statements
ELSE
false_statements
ENDIF
	Conditional execution of statements

INPUT pin	Make pin an input
LCDIN {address,} [var1{, var2, . . .}]	Read RAM on a liquid crystal display (LCD)
LCDOUT item1{, item2, . . .}	Display characters on LCD
{LET} var = value	Assignment statement (assigns a value to a variable)

Statement	Description
LOOKDOWN value, [const1{, const2, . . .}], var	Search constant table for a value
LOOKDOWN2 value, {test} [value1{, value2, . . .}], var	Search constant/variable table for a value
LOOKUP index, [const1{, const2, . . .}], var	Fetch constant value from a table
LOOKUP2 index, [value1{, value2, . . .}], var	Fetch constant/variable value from a table
LOW pin	Make pin output low
NAP period	Power down processor for a selected period of time
ON DEBUG GOTO label	Execute PicBasic debug subroutine at label after every statement if debug is enabled
ON INTERRUPT GOTO label	Execute PicBasic subroutine at label when an interrupt is detected
OUTPUT pin	Make pin an output
PAUSE period	Delay a given number of milliseconds
PAUSEUS period	Delay a given number of microseconds
{PEEK address, var}	Read byte from a register
{POKE address, var}	Write byte to a register
POT pin, scale, var	Read resistance of a potentiometer, or other variable resistance device, connected to a pin with a series capacitor to ground
PULSIN pin, state, var	Measure the width of a pulse on a pin
PULSOUT pin, period	Generate a pulse on a pin
PWM pin, duty, cycles	Output a pulse width modulated (PWM) pulse train to pin
RANDOM var	Generate a pseudo-random number
RCTIME pin, state, var	Measure pulse width on a pin
READ address, var	Read a byte from on-chip EEPROM
READCODE address, var	Read a word from code memory
RESUME {label}	Continue execution after interrupt handling
RETURN	Continue execution at the statement following last executed GOSUB
REVERSE pin	Make output pin an input or an input pin an output
SERIN pin, mode,{ timeout, label,} {[qual1, qual2, . . .],}{ item1{, item2, . . .}}	Asynchronous serial input (Basic Stamp 1 style)
SERIN2 data_pin{\flow_pin}, mode, {parity_label,} {timeout, label,} [item1{, item2, . . .}]	Asynchronous serial input (Basic Stamp 2 style)
SEROUT pin, mode, [item1{, item2, . . .}]	Asynchronous serial output (Basic Stamp 1 style)
SEROUT2 data_pin{\flow_pin}, mode, {pace,} {timeout, label,} [item1{, item2, . . .}]	Asynchronous serial output (Basic Stamp 2 style)
SHIFTIN data_pin, clock_pin, mode, [var1{\bits1} {, var2{\bits2}, . . .}]	Synchronous serial input
SHIFTOUT data_pin, clock_pin, mode, [var1{\bits1} {, var2{\bits2}, . . .}]	Synchronous serial output
SLEEP period	Power down the processor for a given number of seconds
SOUND pin, [note1, duration1{, note2, duration2, . . .}]	Generate a tone or white noise on a specified pin
STOP	Stop program execution
SWAP var1, var2	Exchange the values of two variables
TOGGLE pin	Change the state of an output pin
WHILE logical_comp statements WEND	Execute code while condition is true
WRITE address, value	Write a byte to on-chip EEPROM
WRITECODE address, value	Write a word to code memory
XIN data_pin, zero_pin, {timeout, label,} [var1{, var2, . . .}]]	Receive data from an external X-10 type device
XOUT data_pin, zero_pin, [house_code1\key_code1{\repeat1}{, house_code2\key_code2{\repeat2, . . .}]	Send data to an external X-10 type device

7.5.2 PicBasic Pro Programming Examples

This section presents a series of problems that can be solved with a PIC16F84. The examples illustrate the application of PicBasic Pro. In Sections 7.7 and 7.8 we present more details on how to interface the PIC to a variety of input and output devices. In Section 7.9, we present a methodical design procedure that will help you create software and associated hardware when challenged to design a new microcontroller-based mechatronic system.

EXAMPLE 7.4 PicBasic Pro Alternative to the Assembly Language Program in Example 7.2

As in Example 7.2, the objective is to turn on an LED when the user presses a switch, and turn it off when the switch is released. After the switch is pressed and released a specified number of times, a second LED is to turn on and stay lit. Example 7.2 presented an assembly language solution to this problem. A corresponding PicBasic Pro solution follows. The comments included throughout the code help explain the function of the various parts of the program.

The *While . . . Wend* construct allows the program to wait for the first switch to be pressed or released. The *While* loop continually cycles and does nothing while the switch remains at a particular state. In most applications, there would be statements between the *While* and *Wend* lines that are executed each time through the loop, but none are required here.

The prefix *my_* is included as part of the identifiers *my_count* and *my_button* because the words *count* and *button* are **reserved words.** Reserved words are those used by PicBasic Pro as keywords in statements, predefined constants, and mathematical and logical functions. These words cannot be used as identifiers.

The only fundamental difference in the assembly language and PicBasic Pro solutions is the way the count is handled. Here, we are able to count up and detect when the count reaches the target value. In assembly language, this is not easily done, and we chose to count down instead. Another difference is that PicBasic Pro simplifies how memory is handled. In PicBasic Pro, you do not need to identify addresses for variables, specify memory banks, or move values through the accumulator. PicBasic Pro does all of this for you implicitly.

```
' bcount.bas
' Program to turn on an LED every time a push-button switch is pressed, and
'  turn on a second LED once it has been pressed a specified number of times

' Define variables and constants
my_count       Var   BYTE          ' number of times switch has been pressed
TARGET         Con   5             ' number of switch presses required

' Define variable names for the I/O pins
my_button      Var   PORTB.0
led_count      Var   PORTA.0
led_target     Var   PORTA.1

' Initialize the counter and guarantee the LEDs are off
my_count = 0
Low led_count
Low led_target

begin:
      ' Wait for the switch to be pressed
      While (my_button == 0)         ' wait as long as switch is not pressed (0)
      Wend

      ' Turn on the count LED now that the switch has been pressed
      High led_count

      ' Wait for the switch to be released
      While (my_button == 1)         ' wait as long as switch is pressed (1)
      Wend

      ' Turn off the count LED and increment the counter now that the switch
      '  has been released
      Low led_count
      my_count = my_count + 1

      ' Check if the target has been reached; and if so, turn on the target LED
      If (my_count >= TARGET) Then
             High led_target
      Endif

Goto begin

End
```

If you compare the two solutions, you will see that PicBasic Pro code is easier to write and comprehend. This would be even more evident if the problem were more complex. Things that are easy to do using PicBasic Pro but very difficult using assembly language include variable and array management, assignment statements with complex calculations, logical comparison expressions, iteration, interrupts, pauses, and special purpose functions.

One disadvantage to using PicBasic Pro, however, is that it consumes more program EEPROM space. For this example, the assembly language version requires 17 words of program memory and the PicBasic Pro version requires 39 words, even though the function is identical. This is a consequence of using a high-level language such as PicBasic Pro. Fortunately, the inexpensive PIC16F84 allows up to 1024 words, which is adequate for rather complex programs. Also, many other PIC chips have larger memory capacities for longer programs. Furthermore, the cost of microcontrollers continues to fall and memory capacities continue to rise.

■ CLASS DISCUSSION ITEM 7.3
PicBasic Pro and Assembly Language Comparison

Compare the assembly language code in Example 7.2 to the PicBasic Pro code in Example 7.4. Comment on any differences in the way each program functions.

■ CLASS DISCUSSION ITEM 7.4
PicBasic Pro Equivalents of Assembly Language Statements

For each of the assembly language statement examples presented in Example 7.1, write corresponding PicBasic Pro code.

EXAMPLE 7.5 PicBasic Pro Program for Security System Example

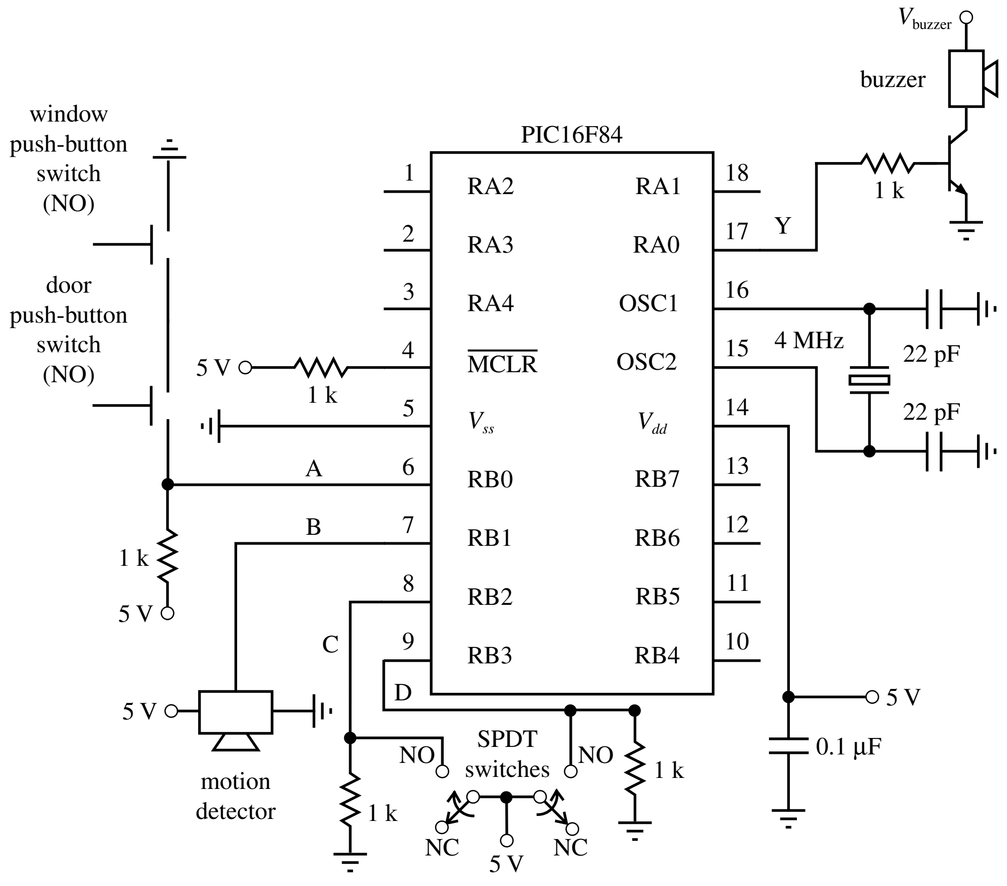
A PicBasic Pro program used to control the security system described in Section 6.6 follows the text. Please refer back to Section 6.6 to review the problem statement. The program comments that follow will help you understand how the code functions. Note how tabs, spaces, blank lines, parentheses, comments, and variable definitions all help make the program more readable. If all these formatting features and definitions were left out, the program would still run, but it would not be as easy for you to understand. The hardware required for PIC implementation is shown in the next figure.

The door and window sensors are assumed to be normally open (NO) switches that are closed when the door and window are closed. They are wired in series and connected to 5 V through a pull-up resistor; therefore, if either switch is open, then signal A will be high. Both the door and window must be closed for signal A to be low. This is called a **wired-AND** configuration since it is a hardwired solution providing the functionality of an AND gate.

The motion detector produces a high on line B when it detects motion. Single-pole, double-throw (SPDT) switches are used to set the 2-bit code C D. In the figure, the switches are both in the normally closed (NC) position; therefore, code C D is 0 0. The alarm buzzer sounds when signal Y goes high, forward biasing the transistor. When Y is high, the 1 k base resistor limits the output current to approximately 5 mA (5 V / 1 kΩ), which is well within the output current specification for a PORTA pin (20 mA as listed in Section 7.8.2).

```
' security.bas

' PicBasic Pro program to perform the control functions of the security
'  system presented in Section 6.6

' Define variables for I/O port pins
door_or_window      Var        PORTB.0    ' signal A
motion              Var        PORTB.1    ' signal B
c                   Var        PORTB.2    ' signal C
d                   Var        PORTB.3    ' signal D
alarm               Var        PORTA.0    ' signal Y

' Define constants for use in IF comparisons
OPEN           Con     1      ' to indicate that a door OR window is open
DETECTED       Con     1      ' to indicate that motion is detected

' Make sure the alarm is off to begin with
Low alarm

' Main polling loop
always:
       If ((c == 0) And (d == 1)) Then     ' operating state 1 (occupants
                                           '  sleeping)
```

```
            If (door_or_window == OPEN) Then
                    High alarm
            Else
                    Low alarm
            Endif
     Else
     If ((c == 1) And (d == 0)) Then      ' operating state 2 (occupants away)
            If((door_or_window == OPEN) Or (motion == DETECTED)) Then
                    High alarm
            Else
                    Low alarm
            Endif
     Else                                    ' operating state 3 or NA (alarm
                                             '  disabled)
                    Low alarm
     Endif
  Endif

Goto always                                     ' continue to poll the inputs

End
```



If all the variable and constant definitions, formatting, and comments were left out, the program would still run properly but the resulting code would be much more difficult to comprehend. This is what the resulting code would look like:

```
Low PORTA.0
always: If PORTB.2==0 And PORTB.3==1 Then
If PORTB.0==1 Then
High PORTA.0
Else
Low PORTA.0
Endif
Else
If PORTB.2==1 And PORTB.3==0 Then
If PORTB.0==1 Or PORTB.1==1 Then
High PORTA.0
Else
Low PORTA.0
Endif
Else
Low PORTA.0
Endif
Endif
Goto always
End
```

There is no advantage to leaving the formatting out, since it is ignored by the compiler. Also, the variable and constant declarations make no difference in the size of the compiled machine code.

■ CLASS DISCUSSION ITEM 7.5
Multiple Door and Window Security System

If you had more than one door and one window, how would modify the hardware design? Would you have to modify the software?

■ CLASS DISCUSSION ITEM 7.6
PIC vs. Logic Gates

In Example 7.5, a PIC solution to the security system problem was presented as an alternative to the logic gate solution presented in Section 6.6. What are the pros and cons of each approach? Which implementation do you think is the best choice in general and in this problem specifically?

Graphically Displaying the Value of a Potentiometer

EXAMPLE 7.6

This example presents code designed to sample the resistance of a potentiometer and display a scaled value in binary form using a set of LEDs. The code uses the PicBasic Pro statement *Pot,* which can indirectly sample the resistance of a potentiometer or other variable resistance. The code for the program, *pot.bas,* and the necessary hardware are included below.

The wiper of a potentiometer is connected to pin RA3 and one end of the potentiometer is in series with a capacitor to ground. Note that the third lead of the potentiometer is unconnected.

Each of pins RB0 through RB7 in PORTB is connected to an LED in series with a current-limiting resistor to ground. When any of these pins goes high, the corresponding LED is on. The eight LEDs display a binary number corresponding to the current position of the potentiometer. The value displayed can range from 0 to 255. This program uses the assignment statement *PORTB = value* to update the display, where *value* is a byte variable (8 bits) that contains the current sample from the potentiometer. The assignment statement drives the outputs of PORTB such that RB0 represents the least significant bit (LSB) and RB7 represents the most significant bit (MSB) of the scaled resistance sample.

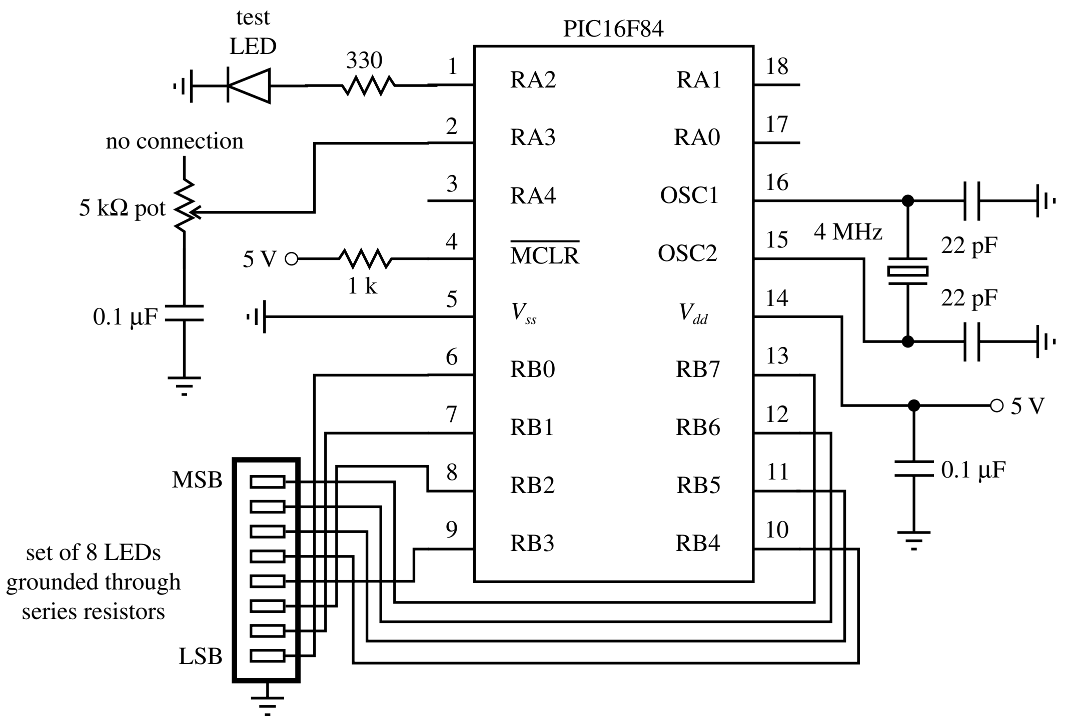
The test LED attached to pin RA2 is used to indicate that the program is running. When the program is running, the test LED blinks. It is good practice to include some sort of program execution indicator, especially in the debugging stages of a project. The blinking LED signals that the PIC has power and the necessary support components and that the program is loaded, running, and looping properly. This is a simple example, and not much can go wrong with the program logic and sequencing. However, complicated programs containing complex logic, branching, looping, and interrupts may hang up unexpectedly or terminate prematurely, especially before they are fully debugged. A nonblinking LED would indicate a problem with the program.

The syntax for the Pot statement is

```
Pot pin, scale, var
```

where *pin* is the input pin identifier, *scale* is a number between 1 and 255 to adjust for the maximum time constant (RC) of the potentiometer and series capacitor, and *var* is the name of a byte variable used to store the value returned by the *Pot* statement. Refer to the PicBasic Pro manual for details on how to choose an appropriate value for *scale*. When the potentiometer is at minimum resistance, the value of *var* is minimum (0 for 0 Ω), and when the resistance is maximum the value is maximum (255 if *scale* is selected appropriately). For a 5 kΩ potentiometer and a 0.1 μF series capacitor, an appropriate value for scale is 200.

In this example, the *TRISB = %00000000* assignment statement designates all PORTB pins as outputs and is required because the scaled potentiometer value is written directly to PORTB with the *PORTB = value* assignment statement. In previous examples, when statements like *High* and *Low* were used, the TRIS registers did not need to be set explicitly because the statements themselves automatically designate the pins as outputs. *PORTB = value* sets all the PORTB outputs to the corresponding bit values in the byte variable *value.*



```
' pot.bas

' Graphically displays the scaled resistance of a potentiometer using a set of
'  LEDs corresponding to a binary number ranging from 0 to 255.

' Define variables, pin assignments, and constants
value    Var     BYTE           ' define an 8 bit (byte) variable capable of
                                '  storing numbers between 0 and 255
test_led Var     PORTA.2        ' pin to which a test LED is attached (RA2)
pot_pin  Var     PORTA.3        ' pin to which the potentiometer and series
                                '  capacitor are attached (RA3)
SCALE    Con     200            ' value for Pot statement scale factor

' Define the input/output status of the I/O pins
TRISB = %00000000               ' designate all PORTB pins as outputs

loop:
      High test_led                   ' turn on the test LED

      Pot pot_pin, SCALE, value       ' read the potentiometer value
      PORTB = value                   ' display the binary value graphically
                                      '  with the 8 PORTB LEDs (RB0 through RB7)

      Pause 100                       ' wait one tenth of a second
      Low test_led                    ' turn off the test LED as an indication
                                      '  that the program and loop are running
```

```
    Pause 100                        ' wait one tenth of a second

Goto loop     ' continue to sample and display the potentiometer value and blink
              '  the test LED

End
```

This example illustrates how to sample the value of a resistance using the special PicBasic Pro statement *Pot*. PicBasic Pro provides an assortment of other high-level statements (e.g., *Button, Freqout, Lcdout, Lookdown, Lookup, Pwm, Serin, Serout,* and *Sound*) that help you create sophisticated functionality with only a few lines of code.

■ CLASS DISCUSSION ITEM 7.7
How Does Pot Work?

The PicBasic Pro statement *Pot* applied in Example 7.6 uses a digital I/O pin to measure the resistance of a potentiometer. *Pot* effectively converts the analog resistance value into a digital number, appearing to function as an A/D converter. How do you think PicBasic Pro accomplishes this? Hint: Consider step response of an RC circuit and the use of a single pin as an output and then an input.

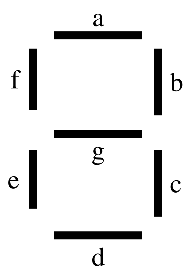
■ CLASS DISCUSSION ITEM 7.8
Software Debounce

Section 6.10.1 presented how to debounce a single-pole, double-throw switch using a NAND gate or flip-flop circuit. This is called a *hardware solution,* since it requires extra components wired together. If a switch is used to input data to a PIC design, debounce can be done in software instead. Assuming that a switch is connected to a PIC by only a single line, write PicBasic Pro code to perform the debounce. Note that the PicBasic Pro statement *Button* can be used for this purpose, but here we want you to think about how you would do it using more fundamental statements.

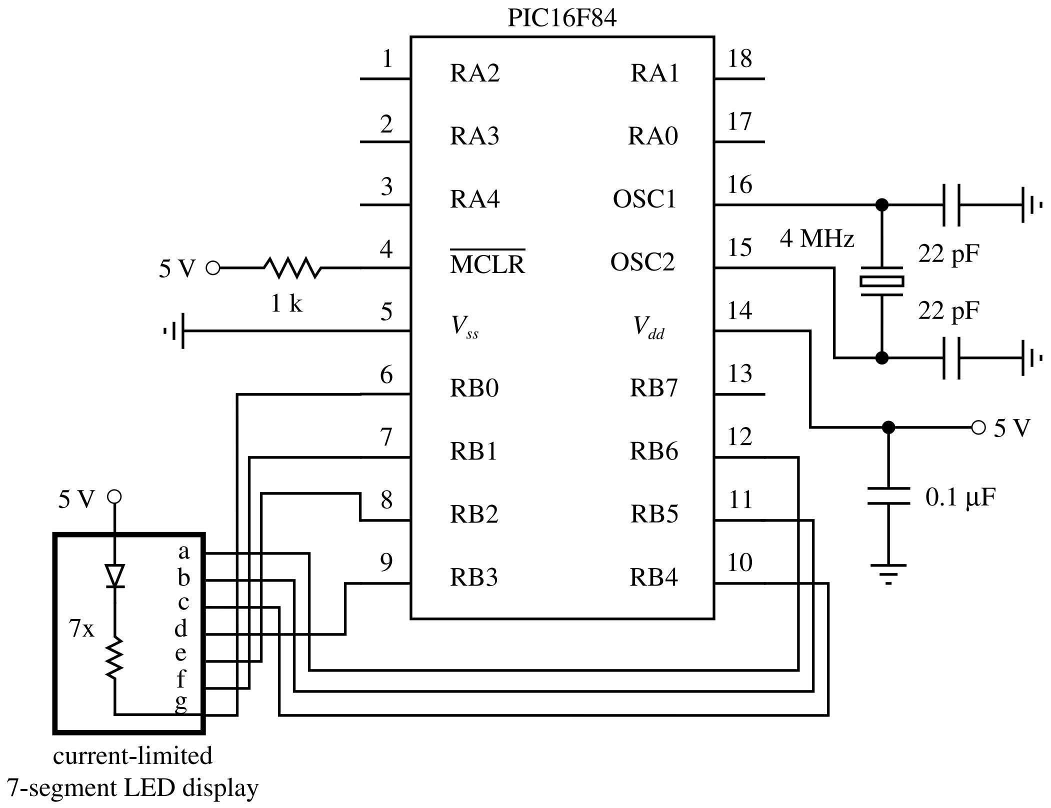
Options for Driving a Seven-Segment Digital Display with a PIC

DESIGN EXAMPLE 7.1

There will be PIC applications where you need to display a decimal digit using a seven-segment LED display. The display could represent some calculated or counted value (e.g., the number of times a switch was pressed). One approach is to drive the seven LED segments directly from seven output pins of a PIC. This would involve decoding in software to determine which segments need to be on or off to display the digit properly. If we label the segments as shown in the following figure and if the PORTB pins are wired to the segments of the LED display, where the segments are connected to 5 V through a set of current-limiting resistors, the following initialization code must appear at the top of your program:



```
' Declare variables
number   var   BYTE              ' digit to be displayed (value assumed to be from 0
                                 '  to 9)
pins     var   BYTE[10]          ' an array of 10 bytes used to store the 7-segment
                                 '  display codes for each digit

' Initialize I/O pins
TRISB = %00000000              ' designate all PORTB pins as outputs (although, pin 7 is
                               '  not used)

' Segment codes for each digit where a 0 implies the segment is on and a 1 implies
'  it is off, because the PIC sinks current from the LED display
'          %gfedcba                     display
pins[0] =  %1000000                      ' 0
pins[1] =  %1111001                      ' 1
pins[2] =  %0100100                      ' 2
pins[3] =  %0110000                      ' 3
pins[4] =  %0011001                      ' 4
pins[5] =  %0010010                      ' 5
pins[6] =  %0000011                      ' 6
pins[7] =  %1111000                      ' 7
pins[8] =  %0000000                      ' 8
pins[9] =  %0011000                      ' 9
```



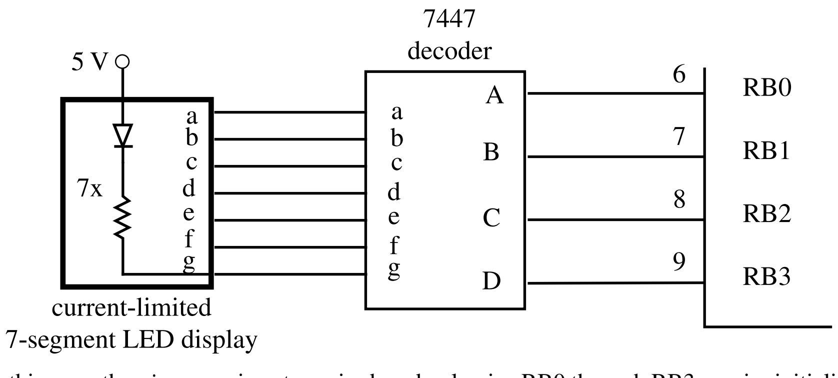
The remainder of your code might consist of a polling loop that needs to update the digital display periodically. A subroutine can be used to accomplish this. **Subroutines** are blocks of code to perform specialized functions that may need to be executed in numerous places within your program. Using the byte variable *number* declared in the code, the following subroutine could be used to display the value stored in the variable:

```
' Subroutine to display a digit on a 7-segment LED display. The value of the
'  digit must be stored in a byte variable called "number." The value is assumed
'  to be less than 10; otherwise, all segments are turned off to indicate an error.
display_digit:
      If (number < 10) Then
             PORTB = pins[number]        ' display the digit
      Else
             PORTB = %1111111            ' turn off all 7 segments
      Endif
Return
```

A number can be displayed at any point in your program by assigning the value to the variable *number* and calling the subroutine. For example, the following statements would display the digit 8:

```
number = 8
Gosub display_digit
```

The solution just presented above requires seven output pins. Since the PIC16F84 has a total of only 13 I/O pins, this could limit the addition of other I/O functions in your design. An alternative design that requires fewer output pins uses a seven-segment decoder IC (e.g., 7447). Here, only four I/O pins are required as shown next:



For this case, the *pins* array is not required, and only pins RB0 through RB3 require initialization as outputs. The subroutine would change to

```
display_digit:
       If (number < 10) Then
              PORTB = (PORTB & $F0) | number   ' display the digit
       Else
              PORTB = (PORTB & $F0) | $F       ' turn off all segments
       Endif
Return
```

The assignment statement for PORTB uses a **logic mask** to retain the four MSBs of PORTB, which may have been independently set by other program statements, and to assign the binary equivalent of *number* to the four LSBs that are output to the seven-segment display driver. A logic mask is a bit string used to protect selected bits in a binary number from change while allowing others to change. The bitwise AND (&) and OR (|) operators are used to help accomplish the mask operation. The term *PORTB & $F0* retains the four MSBs while clearing the four LSBs with zeros. For example, if PORTB's current value is %11011001, then *PORTB & $F0* yields the following result:

```
%11011001      (PORTB)
    &
%11110000      ($F0)
    =
%11010000      (PORTB & $F0)
```

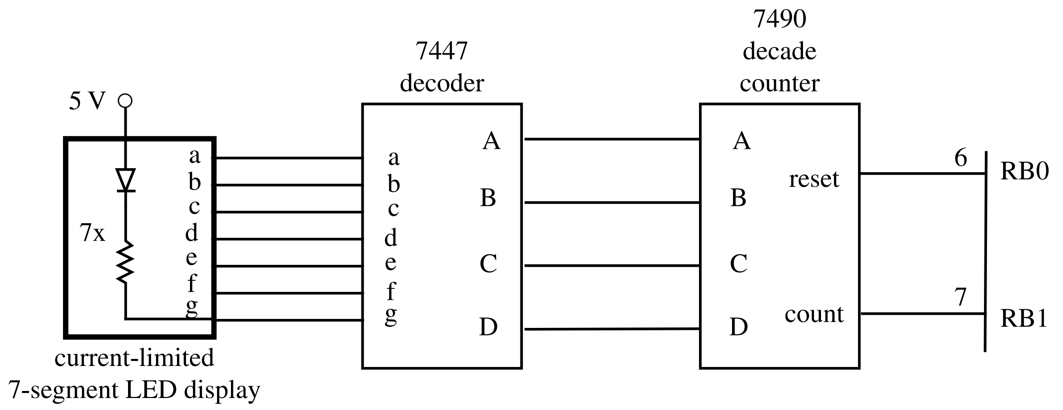
By OR-ing this result with *number,* the four LSBs are replaced by the corresponding binary value of *number*. For example, if the current value of *number* is 7 (%0111), then *(PORTB & $F0)* | *number* would yield the following result:

```
%11010000      (PORTB & $F0)
    |
%00000111      (number)
    =
%11010111      ((PORTB & $F0) | number)
```

As you can see, the four MSBs of PORTB remain unchanged and the four LSBs have changed to the binary value of *number*.

The $F (15) in the *Else* clause is the input value required by the decoder IC to blank all seven segments.

If you lack the luxury of even four I/O pins in your design but still wish to display a digit, another alternative is to use a 7490 decade counter IC with reset and count inputs. The reset input is assumed to have positive logic, so when the line goes high, the counter is reset to 0. The count input is edge triggered, and in this example it does not matter if it is positive or negative edge triggered. Only two PIC I/O pins are required to drive this alternative as shown next:



In this case, only pins RB0 and RB1 need be initialized as outputs and a new counter variable (*i*) has to be declared for use in the subroutine. Also, two pin assignments can be made as follows:

```
i       Var    BYTE                      ' counter variable used in FOR loop
reset   Var    PORTB.0                   ' signal to reset the counter to 0
count   Var    PORTB.1                   ' signal to increment the counter by 1
```

The subroutine would change to

```
display_digit:
        Pulsout reset, 1                 ' send a full pulse to reset the counter
                                         '  to zero
        If (number < 10) Then
              ' Increment the counter "number" times to display the appropriate
              '  digit
              For i = 1 To number
                      Pulsout count, 1   ' send a full pulse to increment the
                                         '  counter
              Next i
        Else
              ' Increment the counter 15 times to clear the display (all segments
              '  off)
              For i = 1 To 15
                      Pulsout count, 1      ' send a full pulse to increment the
                                            '  counter
              Next i
        Endif
Return
```

Here, the PicBasic Pro statement *Pulsout* is used to send a pulse to each of the control pins. The syntax of this command is

```
Pulsout pin, period
```

where *pin* is the pin identifier (e.g., PORTB.0) and *period* is the length of the pulse in tens of microseconds. The pulses generated by the code above are 10 microseconds wide.

In this example, the last two alternatives require additional components (the decoder and counter ICs). If the physical size of the design is no constraint and an objective is to minimize cost by not having to use an additional PIC16F84 or an alternative PIC with more I/O pins, then one of the latter alternatives might be attractive.

■ CLASS DISCUSSION ITEM 7.9
Fast Counting

In the third option presented in Design Example 7.1, the counter was incremented the appropriate number of times to display the desired decimal value on the display. Do you think that this counting will be detectable on the display? Why or why not?

7.6 USING INTERRUPTS

The program to solve the security system problem presented in Example 7.5 uses a method called **polling,** where the program includes a check of the sensor inputs within a loop to update the output accordingly. Program flow is easy to understand, since all processing takes place within the main program loop. The loop repeats as long as the microcontroller is powered. For more complex applications, polling may not be suitable, since the loop may take too long to execute. In a long loop, the inputs may not be checked often enough. An alternative approach is to use an interrupt. In an interrupt-driven program, some inputs are connected to special input lines, designated as interrupts. When one or more of these lines changes level, the microcontroller temporarily suspends normal program execution while the change is acted on by a subprogram or function called an **interrupt service routine.** At the end of the service routine, control is returned to the main program at the point where the interrupt occurred. Because polling is easier to implement, it is preferred over interrupts, as long as the polling loop can run fast enough.

To detect interrupts, two specific registers on the PIC must be initialized correctly. These are the option register (**OPTION_REG**) and the interrupt control register (**INTCON**). The definition for each bit in the first register (OPTION_REG) follows. Recall that the least significant bit is on the right and designated as bit 0 (b_0), while the most significant bit is on the left and designated as bit 7 (b_7):

$$\text{OPTION\_REG} = \%b_7b_6b_5b_4b_3b_2b_1b_0$$

```
bit 7:   RBPU: PORTB pull-up enable bit
         1 = PORTB pull-ups are disabled
         0 = PORTB pull-ups are enabled
bit 6:   Interrupt edge select bit
         1 = Interrupt on rising edge of signal on pin RB0
         0 = Interrupt on falling edge of signal on pin RB0
bit 5:   T0CS: TMR0 clock source select bit
         1 = External signal on pin RA4
         0 = Internal instruction cycle clock (CLKOUT)
bit 4:   T0SE: TMR0 source edge select bit
         1 = Increment on high-to-low transition of signal on pin RA4
         0 = Increment on low-to-high transition of signal on pin RA4
bit 3:   PSA: prescaler assignment bit
         1 = Prescaler assigned to the Watchdog timer (WDT)
         0 = Prescaler assigned to TMR0
bits 2, 1, and 0:
         3-bit value used to define the prescaler rate for the timer features
```

Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

In the *onint.bas* example presented below, OPTION_REG is set to $7F, which is %01111111. Setting bit 7 low enables PORTB pull-ups and setting bit 6 high causes interrupts to occur on the positive edge of a signal on pin RB0. When pull-ups are enabled, the PORTB inputs are held high until they are pulled low by the external input circuit (e.g., a switch to ground wired to pin RB0). Bits 0 through 5 are important only when using special purpose timers.

The definition for each bit in the second register (INTCON) follows:

```
bit 7:   GIE: global interrupt enable bit
         1 = Enables all unmasked interrupts
         0 = Disables all interrupts
bit 6:   EEIE: EE write complete interrupt enable bit
         1 = Enables the EE write complete interrupt
         0 = Disables the EE write complete interrupt
bit 5:   T0IE: TMR0 overflow interrupt enable bit
         1 = Enables the TMR0 interrupt
         0 = Disables the TMR0 interrupt
bit 4:   INTE: RB0 interrupt enable bit
         1 = Enables the RB0/INT interrupt
         0 = Disables the RB0/INT interrupt
bit 3:   RBIE: RB port change interrupt enable bit (for pins RB4 through RB7)
         1 = Enables the RB port change interrupt
         0 = Disables the RB port change interrupt
bit 2:   T0IF: TMR0 overflow interrupt flag bit
         1 = TMR0 has overflowed (must be cleared in software)
         0 = TMR0 did not overflow
bit 1:   INTF: RB0 interrupt flag bit
         1 = The RB0 interrupt occurred
         0 = The RB0 interrupt did not occur
bit 0:   RBIF: RB port change interrupt flag bit
        1 = At least one of the signals on pins RB4 through RB7 has changed state
            (must be cleared in software)
        0 = None of the signals on pins RB4 through RB7 has changed state
```

In the *onint.bas* example that follows, INTCON is set to $90, which is %10010000. For interrupts to be enabled, bit 7 must be set to 1. Bit 4 is set to 1 to check for interrupts on pin RB0. Bits 0 and 1 are used to indicate interrupt status during program execution. If more than one interrupt signal were required, bit 3 would be set to 1, which would enable interrupts on pins RB4 through RB7. In that case, INTCON would be set to $88 (%10001000). To check for interrupts on RB0 and RB4–7, INTCON would be set to $98 (%10011000). PORTA has no interrupt capability, and PORTB has interrupt capability only on pin RB0 and pins RB4 through RB7. Bits 6, 5, and 2 are for advanced features not used in this example.

A simple example, called *onint.bas,* that illustrates the use of interrupts follows. The corresponding schematic is shown in Figure 7.7. The program's function is described in detail in the following paragraphs.

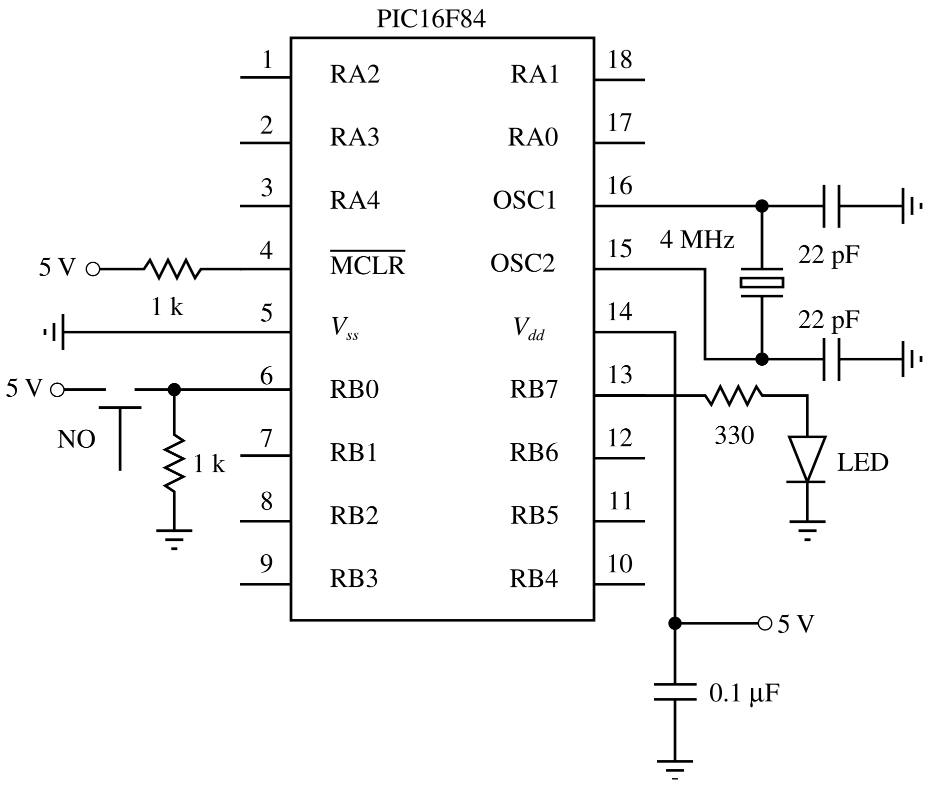


Figure 7.7 Interrupt example schematic.

```
' onint.bas

' This program turns on an LED and waits for an interrupt on PORTB.0. When RB0
'  changes state, the program turns the LED off for 0.5 seconds and then resumes
'  normal execution.

led    var     PORTB.7                ' designate pin RB7 as "led"

       OPTION_REG = $7F               ' enable PORTB pull-ups and detect positive
                                      '  edges on interrupt
       On Interrupt Goto myint        ' define interrupt service routine location
       INTCON = $90                   ' enable interrupts on pin RB0

' Turn LED on and loop until there is an interrupt
loop:  High led
       Goto loop

' Interrupt handling routine
Disable                               ' disable interrupts until the Enable
                                      '  statement appears
myint: Low led                        ' turn LED off
```

```
        Pause   500                     ' wait 0.5 seconds
        INTCON.1 = 0                    ' clear interrupt flag
        Resume                          ' return to main program
Enable                                  ' allow interrupts again

End                                     ' end of program
```

The *onint.bas* program turns on an LED connected to pin RB7 until an external interrupt occurs. A normally open (NO) push-button switch connected to pin RB0 provides the source for the interrupt signal. When the signal transitions from low to high, the interrupt routine executes, causing the LED to turn off for half a second. Control then returns to the main loop, causing the LED to turn back on again.

The first active line creates the variable name *led* to denote the pin identifier PORTB.7. In the next line, the OPTION_REG is set to $7F (or %01111111) to enable PORTB pull-ups and configure the interrupt to be triggered when the positive edge of a signal occurs on pin RB0. The PicBasic Pro statement *On Interrupt Goto* designates the label *myint* as the location to which program control jumps when an interrupt occurs. The value of the INTCON register is set to $90 (or %10010000) to properly enable RB0 interrupts.

The two lines starting with the label *loop* cause the program to loop continually, maintaining program execution while waiting for an interrupt. The LED connected to pin RB7 remains on during this loop. The loop is called an **infinite loop,** since it cycles as long as no interrupt occurs. Note that an active statement (such as *High led*) must exist between the label and the *Goto* of the loop for the interrupt to function, because PicBasic Pro checks for interrupts only after a statement is completed.

The final section of the program contains the interrupt service routine. *Disable* must precede the label and *Enable* must follow the *Resume* to prevent checking for interrupts until control is returned to the main program. The interrupt service routine executes when control of the program is directed to the beginning of this routine, labeled by *myint,* when an interrupt occurs on pin RB0. At the identifier label *myint,* the statement *Low led* clears pin RB7, turning off the LED. The *Pause* statement causes a 500 millisecond delay, during which time the LED remains off. The next line sets the INTCON.1 bit to 0 to clear the interrupt flag. The interrupt flag was set internally to 1 when the interrupt signal was received on pin RB0, and this bit must be reset to 0 before exiting the interrupt routine, so that subsequent interrupts can be serviced. At the end of the *myint* routine, control returns to the main program loop where the interrupt occurred.

7.7 INTERFACING COMMON PIC PERIPHERALS

This section introduces interfacing a PIC to two common peripheral devices. The first is a 12-button **keypad** that can be used to input numeric data. The second is a **liquid crystal display (LCD)** that can be used to output messages and numeric information to the user. More information on these and other useful peripheral devices can be found at *www.engr.colostate.edu/~dga/mechatronics.html* under “Microchip PIC Microcontroller Resources.”

7.7.1 Numeric Keypad

Figure 7.8 illustrates a common three-row, four-column 12-button keypad. Each key is attached to a normally open (NO) pushbutton switch. When a key is pressed, the switch closes. Figure 7.9 illustrates the electrical schematic of the keypad with a

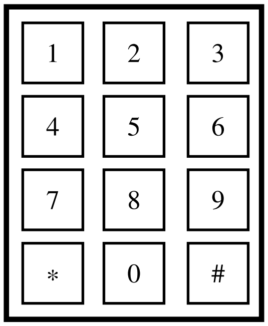


Figure 7.8 Numeric keypad.

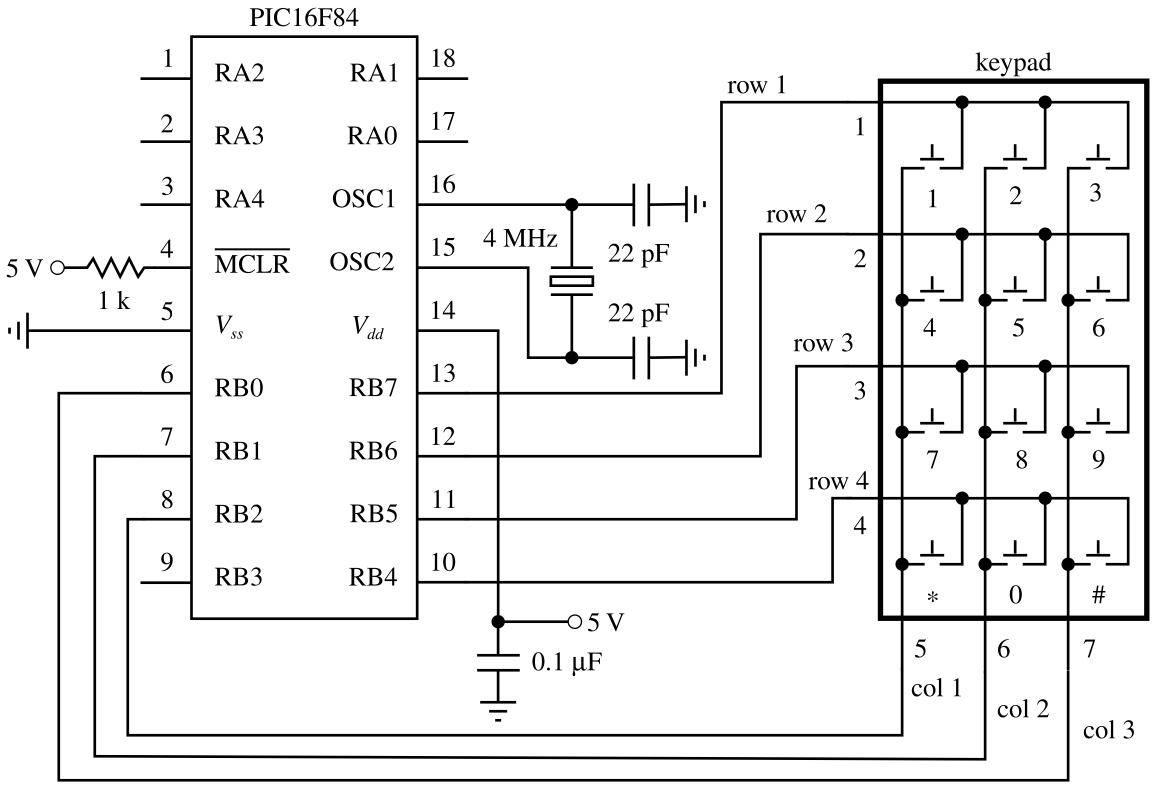


Figure 7.9 Numeric keypad schematic and PIC interface.

recommended interface to the PIC16F84. A standard keypad has a seven-pin header for connection to a ribbon cable socket. There is one pin for each row and one pin for each column as numbered in Figure 7.9.

The four rows (row 1, row 2, row 3, row 4) are connected to pins RB7 through RB4, which are configured as inputs. Internal pull-up resistance is available as a software option on these pins, so external pull-up resistors are not required (see details in Section 7.8.1). The three columns (col 1, col 2, col 3) are connected to pins RB0 through RB2, which are configured as outputs. The following PicBasic Pro code contains initializations and a framework for a polling loop that can be used to process input from the keypad. The column outputs are cleared low one at a time, and each row input is polled to determine if the key switch in that column is closed. For example, if col 1 is low while col 2 and col 3 are high, and only the 1 key is being held down, then row 1 will be low while the remaining row lines will be high, indicating that only the 1 key is down. Statements could be added to the *If* statement blocks, in place of the comments, to process the input.

```
' Pin assignments
row1 Var PORTB.7
row2 Var PORTB.6
row3 Var PORTB.5
row4 Var PORTB.4
col1 Var PORTB.2
col2 Var PORTB.1
col3 Var PORTB.0

' Enable PORTB pull-ups
OPTION_REG = $7f

' Initialize the I/O pins (RB7:RB4 and RB3 as inputs and RB2:RB0 as outputs)
TRISB = %11111000

' Keypad polling loop
loop:
        ' Check column 1
        Low col1  :  High col2  :  High col3
        If (row1 == 0) Then
                ' key 1 is down
        Endif
        If (row2 == 0) Then
                ' key 4 is down
        Endif
        If (row3 == 0) Then
                ' key 7 is down
        Endif
        If (row4 == 0) Then
                ' key * is down
        Endif
        ' Check column 2
        High col1  :  Low col2  :  High col3
        If (row1 == 0) Then
               ' key 2 is down
        Endif
        If (row2 == 0) Then
               ' key 5 is down
        Endif
        If (row3 == 0) Then
               ' key 8 is down
        Endif
        If (row4 == 0) Then
               ' key 0 is down
        Endif

        ' Check column 3
        High col1  :  High col2  :  Low col3
        If (row1 == 0) Then
               ' key 3 is down
        Endif
        If (row2 == 0) Then
               ' key 6 is down
        Endif
        If (row3 == 0) Then
               ' key 9 is down
        Endif
        If (row4 == 0) Then
               ' key # is down
        Endif
' Continue polling

Goto loop

End
```

7.7.2 LCD Display

The other common peripheral device we want to highlight is a standard Hitachi 44780-based liquid crystal display. LCDs come in different shapes and sizes that can support different numbers of rows of text and different numbers of characters per row. The standard choices for the number of characters and rows are 8×2, 16×1, 16×2, 16×4, 20×2, 24×2, 40×2, and 40×4. The commonly used 20×2 LCD is illustrated in the top of Figure 7.10. Applications of LCDs include displaying messages or information to the user (e.g., a home thermostat display, a microwave oven display, or a digital clock) and displaying a hierarchical input menu for changing settings and making selections (e.g., a fax machine display).

For an LCD display with 80 characters or less (all but the 40×4 just listed), the display is controlled via 14 pins. The names and descriptions of these pins are listed in Table 7.6. PicBasic Pro offers a simple statement called *Lcdout* to control an LCD

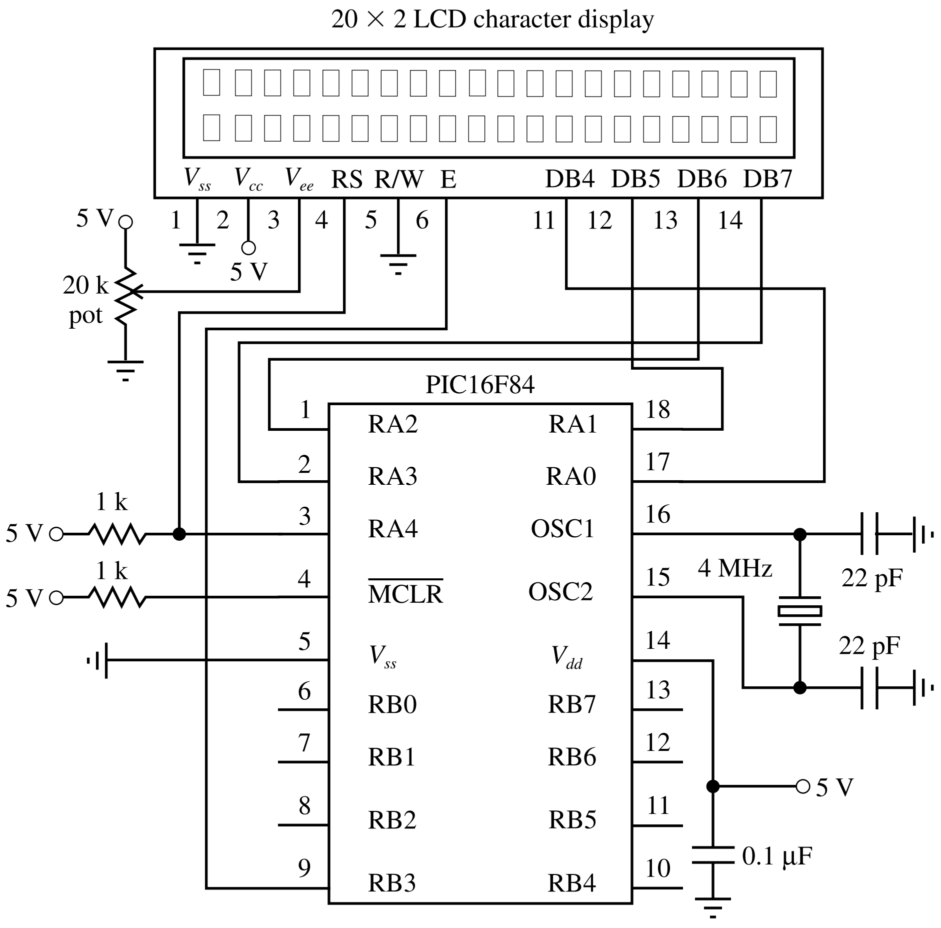

20 × 2 LCD character display
V_{ss} V_{cc} V_{ee} RS R/W E DB4 DB5 DB6 DB7
1 2 3 4 5 6 11 12 13 14
5 V
20 k pot
5 V
PIC16F84
RA2 RA3 RA4 MCLR V_{ss} RB0 RB1 RB2 RB3
RA1 RA0 OSC1 OSC2 V_{dd} RB7 RB6 RB5 RB4
1 k
1 k
4 MHz
22 pF
22 pF
0.1 μF

Figure 7.10 LCD PIC interface.

display. LCD displays with more than 80 characters (40 × 4) use a 16-pin header with different pin assignments not compatible with *Lcdout*. A 14-pin LCD can be controlled via four or eight data lines. PicBasic Pro supports both, but it is recommended that you use four lines to minimize the number of I/O pins required. Figure 7.10 shows the recommended interface to the PIC using a four-line data bus. Commands and data are sent to the display via lines DB4 through DB7, and lines DB0 through DB3 (pins 7 through 10) are not used. Pin RA4 is connected to 5 V through a pull-up resistor since it is an open drain output (see details in Section 7.8). The potentiometer connected to V_{ee} is used to adjust the contrast between the foreground and background shades of the display. The RS, R/W, and E lines are controlled automatically by PicBasic Pro when communicating with the display. Detailed information about LCD displays and how to write your own interface can be found at *www.engr.colostate.edu/~dga/mechatronics.html* under "Microchip PIC Microcontroller Resources."

Table 7.6 Liquid crystal display pin descriptions

Pin	Symbol	Description
1	V_{ss}	Ground reference
2	V_{cc}	Power supply (5 V)
3	V_{ee}	Contrast adjustment voltage
4	RS	Register select (0: instruction input; 1: data input)
5	R/W	Read/write status (0: write to LCD; 1: read from LCD RAM)
6	E	Enable signal
7–14	DB0–DB7	Data bus lines

With the hardware interface shown in Figure 7.10, the display can be controlled with the PicBasic Pro statement *Lcdout*. The simplest form of this statement is *Lcdout text* (e.g., *Lcdout "Hello world"*) where *text* is a string constant. The statement also supports various commands for controlling the display and cursor and for outputting numbers and data in different formats. Refer to the PicBasic Pro compiler manual for more information on these options. Here is a simple example to illustrate the use of these commands and format controls. If *x* is defined as a byte variable and currently contains the value 123, the following statement,

```
Lcdout $FE, 1, "Current value for x:", $FE, $C0, " ", DEC x
```

would clear the display and output the following two-line message:

```
    Current value for x:
    123
```

The code word *$FE* indicates to the display that the next item is a command. In the example above, command *1* clears the display and command *$C0* moves the cursor to the beginning of the next line. The prefix *DEC* is used to instruct the display to output the following number in its decimal digit form rather than its corresponding ASCII character.

7.8 INTERFACING TO THE PIC

In this section, we discuss interfacing the PIC to a variety of input and output devices. As we saw in Section 7.5.1, each pin in the I/O ports may be configured in software as an input or an output. In addition, the port pins may be multiplexed with other functions to use additional features of the PIC. In this section, we examine the electronic schematics of the different input and output ports of the PIC16F84. The ports are different combinations of TTL and CMOS devices and have voltage and current limitations that must be considered when interfacing other devices to the PIC. You should first refer to Section 6.11 to review details of TTL and CMOS equivalent output circuits and open drain outputs.

We begin by looking at the architecture and function of each of the ports individually. PORTA is a 5-bit-wide latch with the pins denoted by RA0 through RA4. The block diagram for pins RA0 through RA3 is shown in Figure 7.11 and the block diagram for pin RA4 is shown in Figure 7.12. The five LSBs of the TRISA register configure the 5-bit-wide latch for input or output. Setting a TRISA bit high causes the corresponding PORTA pin to function as an input, and the CMOS output driver

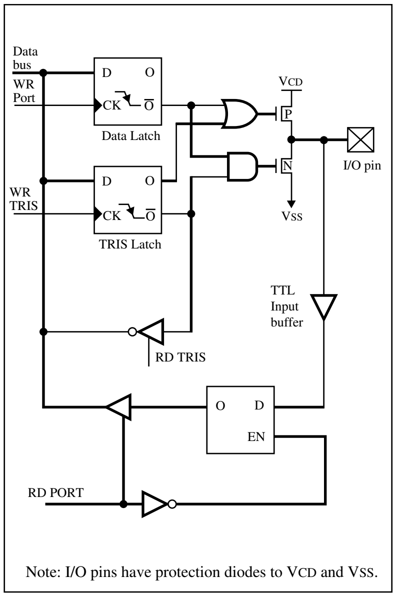


Figure 7.11 Block diagram for pins RA0 through RA3. *(Courtesy of Microchip Technology Inc., Chandler, AZ)*

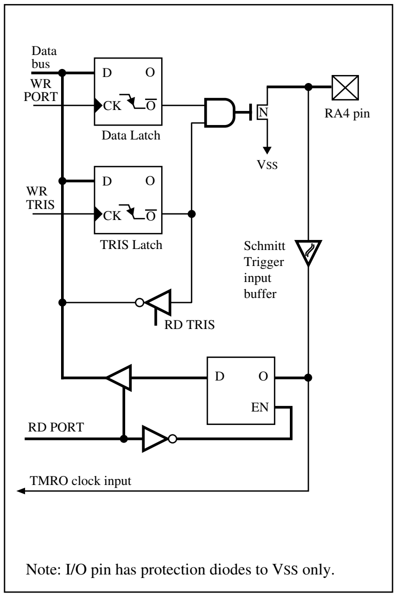


Figure 7.12 Block diagram for pin RA4. *(Courtesy of Microchip Technology Inc., Chandler, AZ)*

is in high impedance mode, essentially removing it from the circuit. Clearing a TRISA bit low causes the corresponding PORTA pin to serve as an output, and the data on the data latch appears on the pin. Reading the PORTA register accesses the pin values. RA4 is slightly different in that it has a Schmitt trigger input buffer that triggers sharply on the edge of a slowly changing input (see Section 6.12.2). Also, the output configuration of RA4 is open drain, and external components (e.g., a pull-up resistor to power) are required to complete the output circuit.

PORTB is also bidirectional but is 8 bits wide. Its data direction register is denoted by TRISB. Figure 7.13 shows the schematic for pins RB4 through RB7 and Figure 7.14 shows pins RB0 through RB3. A high on any bit of the TRISB register sets the tristate gate to the high impedance mode, which disables the output driver. A low on any bit of the TRISB register places the contents of the data latch on the selected output pin. Furthermore, all of the PORTB pins have **weak pull-up** FETs. These FETs are controlled by a single control bit called $\overline{\text{RBPU}}$ (active low register B pull-up). When this bit is cleared, the FET acts like a weak pull-up resistor. This pull-up is automatically disabled when the port pin is configured as an output. $\overline{\text{RBPU}}$ can be set in software through the OPTION_REG special purpose register (see Section 7.6).

7.8.1 Digital Input to the PIC

Figure 7.15 illustrates how to properly interface different types of components and digital families of devices as inputs to the PIC. All I/O pins of the PIC that are configured as inputs interface through a TTL input buffer (pins RA0 through RA3 and

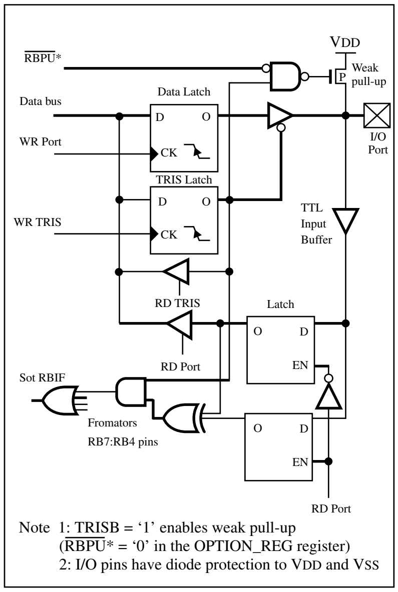


Figure 7.13 Block diagram for pins RB4 through RB7. *(Courtesy of Microchip Technology Inc., Chandler, AZ)*

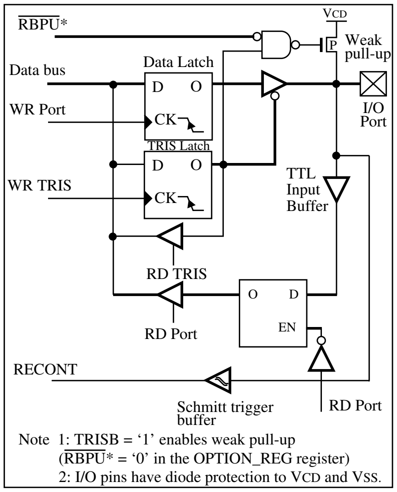


Figure 7.14 Block diagram for pins RB0 through RB3. *(Courtesy of Microchip Technology Inc., Chandler, AZ)*

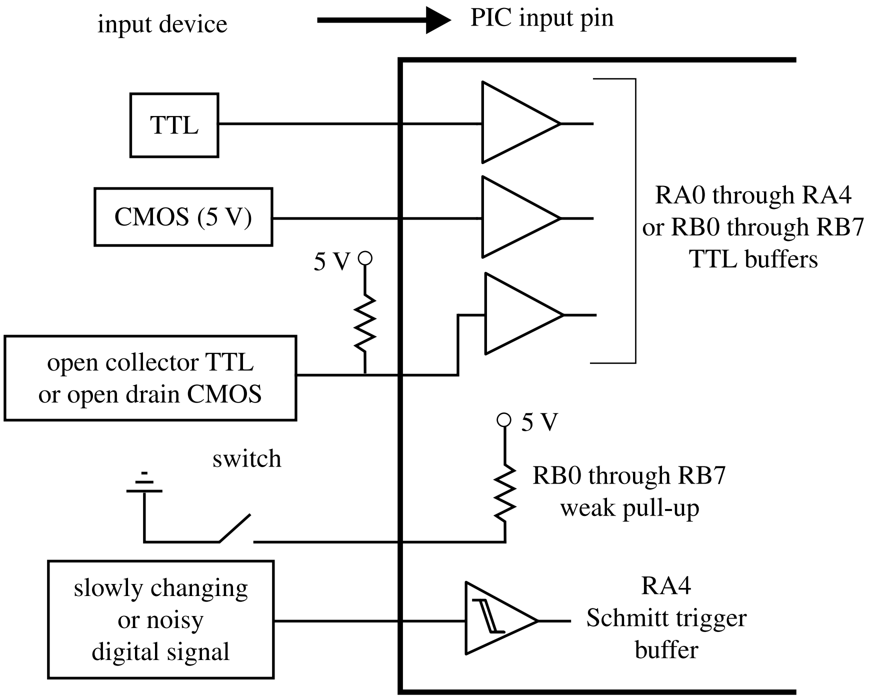

input device
PIC input pin
TTL
CMOS (5 V)
5 V
open collector TTL or open drain CMOS
RA0 through RA4 or RB0 through RB7 TTL buffers
switch
5 V
RB0 through RB7 weak pull-up
slowly changing or noisy digital signal
RA4 Schmitt trigger buffer

Figure 7.15 Interface circuits for input devices.

pins RB0 through RB7) or Schmitt trigger input buffer (RA4). The Schmitt trigger enhances noise immunity for a slowly changing input signal. Since an input pin is TTL buffered in the PIC, interfacing a TTL gate to the PIC can be done directly unless it is has an open collector output. In this case, an external pull-up resistor is

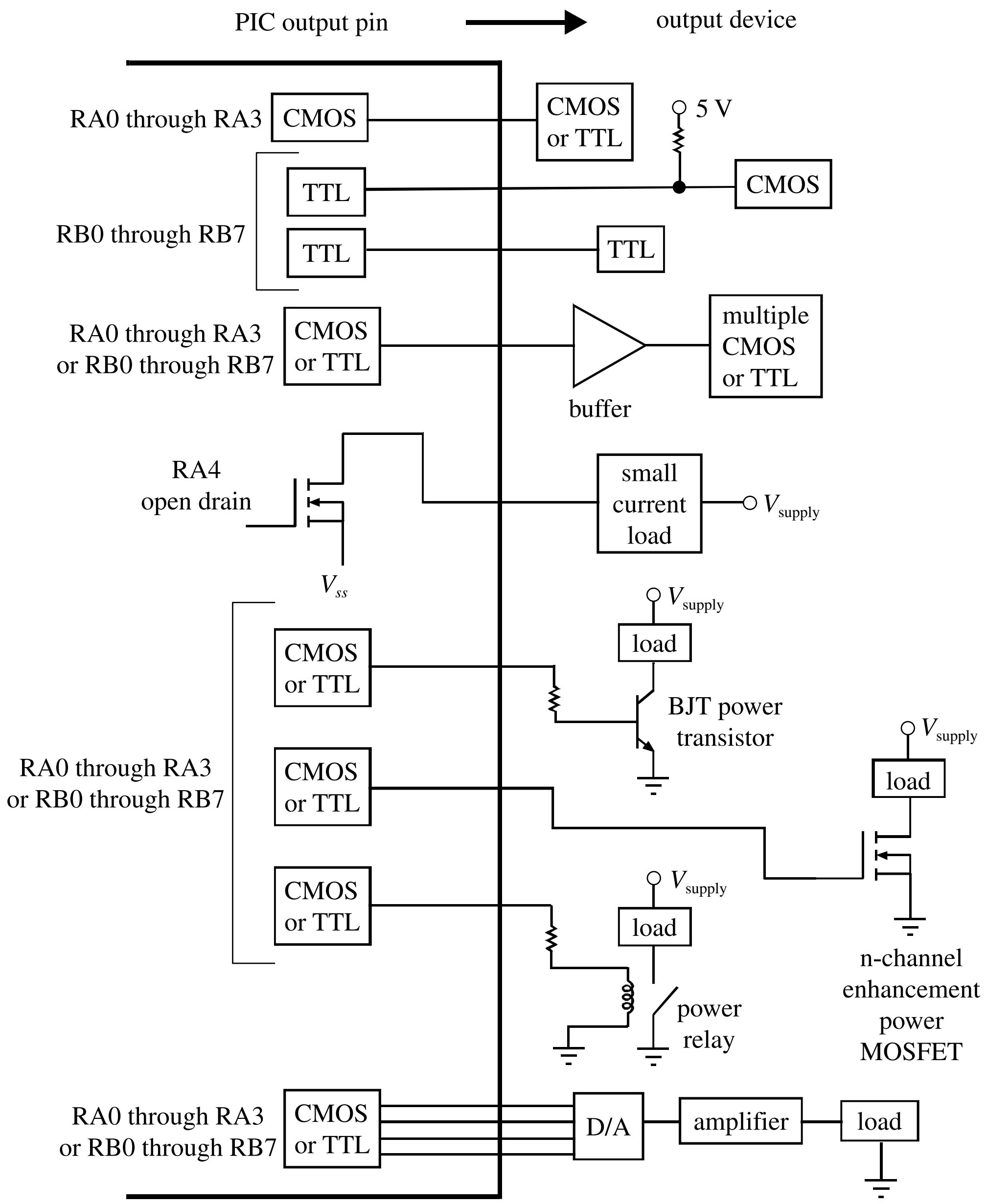


Figure 7.16 Interface circuits for output devices.

required. Since the output of a 5 V powered CMOS device swings nearly from 0 to 5 V, the device will drive a PIC input directly. The weak pull-up option on pins RB0 through RB7 is useful when using mechanical switches or keypads for input (see Section 7.7). The pull-up FET maintains a 5 V input until the switch is closed, bringing the input low. Although a TTL input usually floats high if it is open, the FET pull-up option is useful, since it simplifies the interface to external devices (e.g., keypad input). Finally, one must be aware of the current specifications of the PIC input and output pins. For the PIC16F84, there is a 25 mA sink maximum per pin with a 80 mA maximum for the entire PORTA and a 150 mA maximum for PORTB.

7.8.2 Digital Output from the PIC

Figure 7.16 illustrates how to properly interface different types of components and digital families of devices to outputs from the PIC. Pins RA0 through RA3 have full CMOS output drivers, and RA4 has an open drain output. RB0 through RB7 are TTL buffered output drivers. A 20 mA maximum current is sourced per pin with a 50 mA maximum current sourced by the entire PORTA and a 100 mA maximum for PORTB. CMOS outputs can drive single CMOS or TTL devices directly. TTL outputs can drive single TTL devices directly but require a pull-up resistor to provide an adequate high-level voltage to a CMOS device. To drive multiple TTL or CMOS devices, a buffer can be used to provide adequate current for the fan-out. Because pin RA4 is an open drain output, external power is required. When interfacing transistors, power transistors, thyristors, triacs, and SCRs, current requirements must be considered for a proper interface. If the PIC contains a D/A converter, it can be used with an amplifier to drive an analog load directly. Otherwise, as shown in the figure, an external D/A IC can be used with the digital I/O ports.

7.9 METHOD TO DESIGN A MICROCONTROLLER-BASED SYSTEM

In all the examples presented in this chapter, the problems were simple, yielding short solutions to illustrate fundamental coding structures. Also, many design decisions were included as part of the problem statements. In conceiving an altogether new design, it is advisable to follow a methodical design process that will take you from the initial problem statement to a programmed microcontroller that can be embedded in application hardware. A design procedure we recommend follows. To illustrate the application of the procedure, we apply it to the problem presented in Design Example 7.2.

1. *Define the problem.* State the problem in words to explain the desired functionality of the device (i.e., what is the device supposed to do?). Then list the types of inputs and outputs required and what functions need to be performed by the microcontroller. You need to identify the number of each type of I/O line you require, including digital inputs, digital outputs, A/D converters, D/A converters, and serial ports.
2. *Select an appropriate microcontroller model.* Based on the types and number of inputs and outputs identified in the previous step, choose a microcontroller

that has sufficient on-chip resources. Another factor that influences this choice is the anticipated amount of program and data memory required. If the program is very complex and the application requires significant data storage, then choose a microcontroller with ample memory capacity. Refer to manufacturer literature for a list of available models and capacities. Information for Microchip's entire line of products can be found at *www.microchip.com*.

3. *Identify necessary interface circuits.* Refer to the microcontroller input and output circuit specifications and use the information in Section 7.8 to design appropriate interface circuitry utilizing pull-up resistors, buffers, transistors, relays, and amplifiers where required. Also, in cases where many digital I/O lines are required, there are alternatives to driving all of the lines directly with PIC I/O pins. One is to use shift registers (e.g., the 74164, 74594, or 74595 for output, and the 74165 or 74597 for input), where a small set of PIC I/O pins (two for the nonlatched type, and three for the latched type) can be used to transmit bits serially to or from an 8-bit register, providing eight lines of I/O. Another alternative when expanding your I/O capability is to use a device providing multiplexed programmable I/O ports (e.g., the Intel 82C55A programmable peripheral interface, or PPI). This type of device allows one I/O port to switch access among several I/O ports. With Intel's 82C55A, 5 control lines and 8 data lines provide access to 24 lines of general purpose, user-configurable I/O.
4. *Decide on a programming language.* You can write the code in assembly language or in a high-level language such as C or PicBasic Pro. We recommend PicBasic Pro for most applications. Assembly language is a better option only when extremely fast execution speed is required or if memory capacity is a limiting factor.
5. *Draw the schematic.* Draw a detailed schematic showing required components, input and output interface circuitry, and wire connections. If using the PIC16F84, Figure 7.4 serves as a good starting point.
6. *Draw a program flowchart.* A **flowchart** is a graphical representation of the required functionality of your software. Figure 7.17 illustrates a set of building blocks that can be used to construct a flowchart. The flow control block is used as a destination label for a *goto* branch or a loop (e.g., *For . . . Next* or *While . . .*

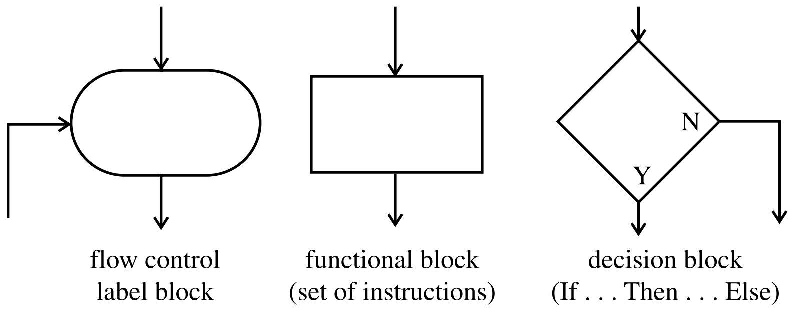

N
Y
flow control label block
functional block (set of instructions)
decision block (If . . . Then . . . Else)

Figure 7.17 Software flowchart building blocks.

Wend). The functional block represents one or more instructions that perform some task. The decision block is used to represent logic decisions. Design Example 7.2 illustrates how a typical flowchart is constructed.

7. *Write the code.* Implement the flowchart in software by writing code to create the desired functionality.

8. *Build and test the system.* Compile your code into machine code and download the resulting hex file to the microcontroller. This can be done using a programming device available from the manufacturer (e.g., the PicStart Plus serial programmer available from Microchip). Assemble the system hardware, including the microcontroller and interface circuitry. Fully test the system for the desired functionality. We recommend you do steps 7 and 8 incrementally as you build functionality, carefully testing each addition before continuing. For example, make sure you can read and process an input first. Then add and test additional inputs and incrementally add and test output functionality. In other words, do not try to write the complete program on the first attempt!

PIC Solution to an Actuated Security Device

DESIGN EXAMPLE 7.2

A few years ago, we assigned an interesting class project in our Introduction to Mechatronics and Measurement Systems course at Colorado State University. Detailed information can be found on the book website at *www.engr.colostate.edu/~dga/mechatronics.html*. We now illustrate the procedure just presented to generate a basic design. Then we describe some of the student solutions, because an interesting part of the project was the creativity that the class groups exhibited in solving this problem.

1. *Define the problem.* The goal of the project is to use the PIC16F84 microcontroller in the design of a combination security lock device. The device requirements include switches to enter a combination, a push-button switch to process the combination, LEDs and a buzzer to indicate the success of a combination attempt, a digital display to indicate the number of failed combination attempts, and an actuator to perform a useful output function. In the simple design presented here, we use three toggle switches to enter the combination allowing for eight possible combinations.

The inputs and outputs are all digital for this problem.

Inputs:

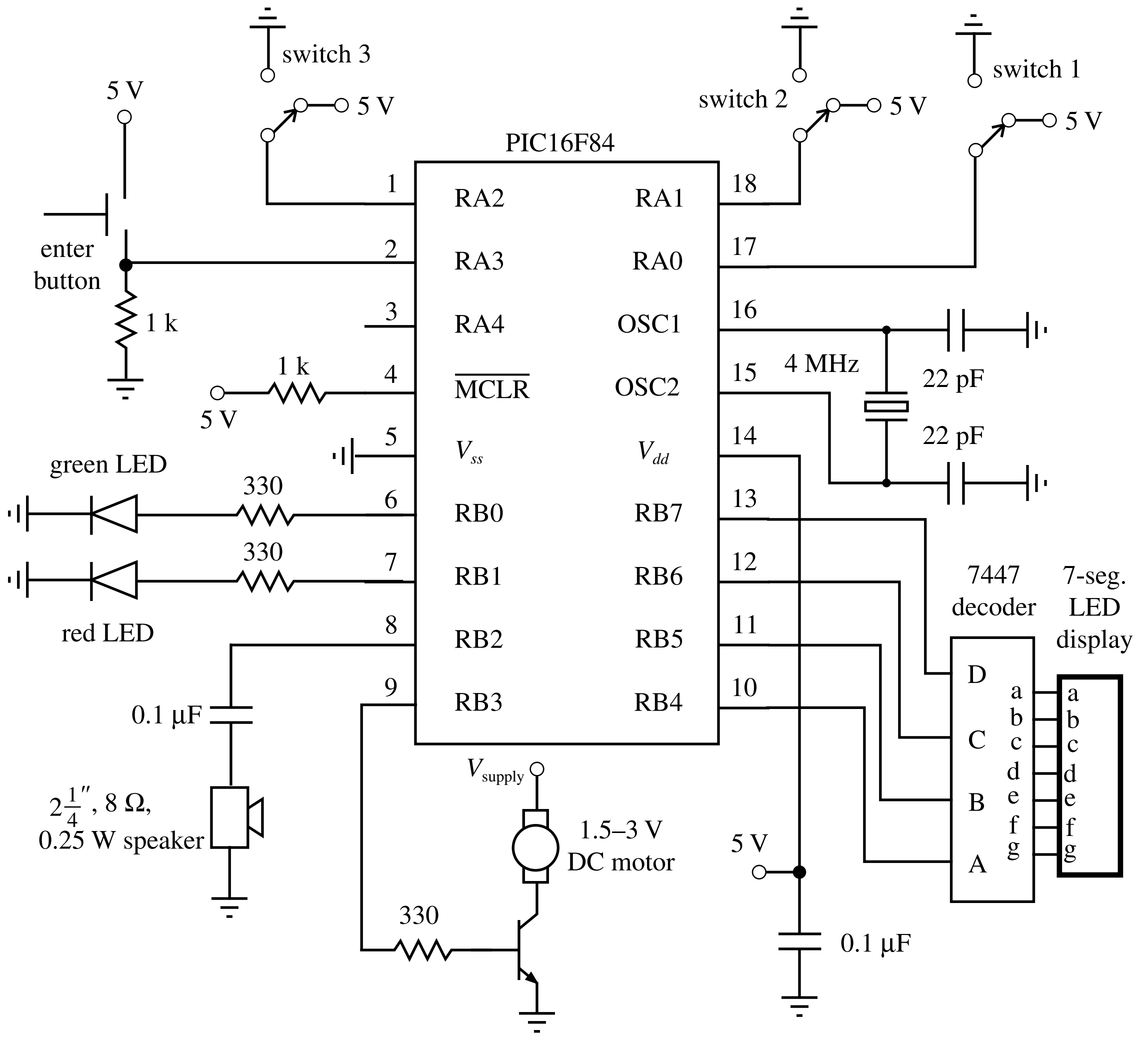
- three switches for the combination
- one push-button switch to serve as an enter key

Outputs:

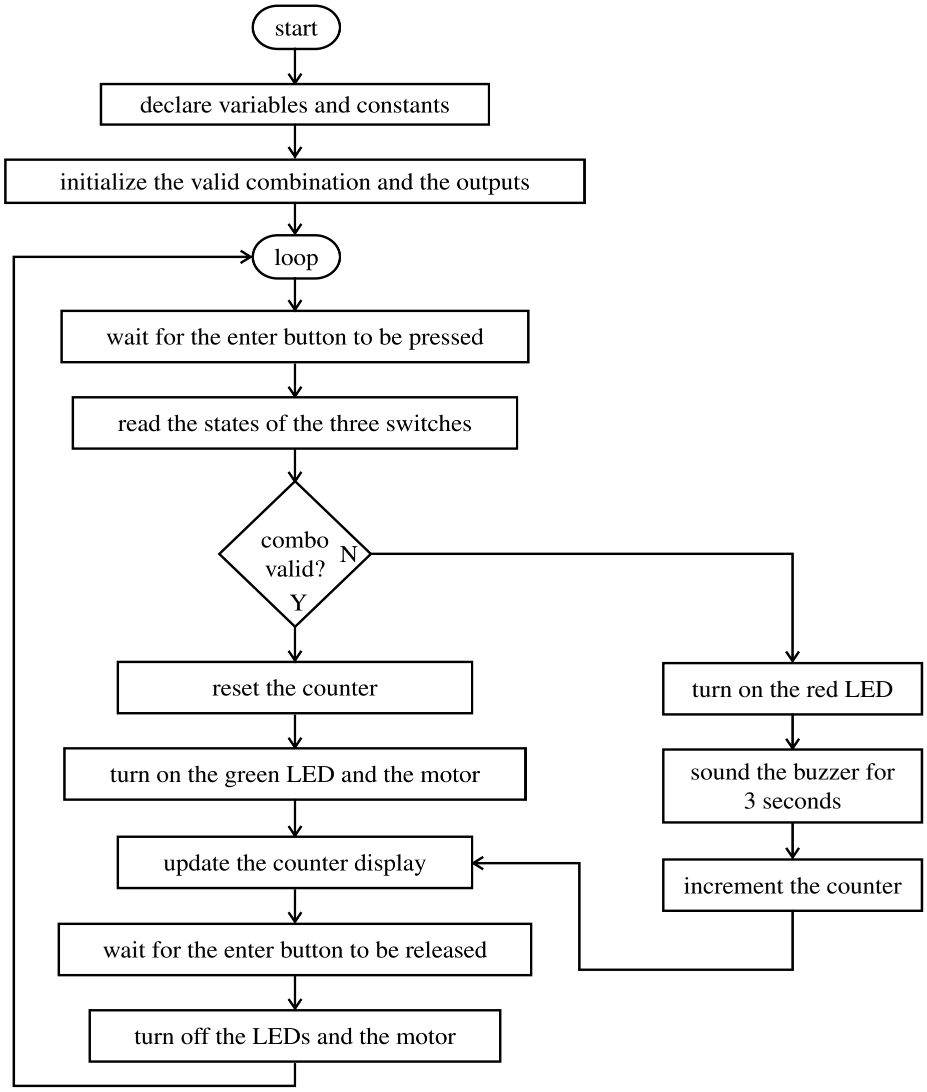
- two LEDs to indicate the combination status
- one seven-segment LED digit display
- one small speaker
- one small DC motor

2. *Select an appropriate microcontroller model.* If we drive the seven-segment display with four digital outputs (see Design Example 7.1), the required number of digital I/O lines is 12. This is the only I/O we require (i.e., we do not need A/D, D/A, or serial ports). The PIC16F84 is adequate for this problem because it provides 13 digital I/O pins.

3. *Identify necessary interface circuits.* A 7447 interface is required to decode the four digital outputs to control the seven-segment display (see Design Example 7.1). A small audio speaker can be driven directly through a series capacitor, as recommended in the description of the *Sound* statement in the PicBasic Pro compiler manual. The PIC cannot source enough current to drive the motor directly, so a digital output is used to bias a power transistor connected to the motor. The only I/O pin we will not use is RA4, the special purpose Schmitt trigger input and open drain output pin.
4. *Decide on a programming language.* We use PicBasic Pro. This or some other high-level language should always be your first choice, unless you have memory or speed constraints.
5. *Draw the schematic.* The following figure shows the necessary components. It does not matter which I/O pins are used for the different functions, but we kept them organized according to function.



6. *Draw a program flowchart.* The next figure illustrates the logic and flow necessary to perform the desired functions. The program checks the state of the switches when the push-button switch is pressed and compares the switch states to a prestored combination. If the combination is valid, a green LED and a DC motor turn on and stay on while the push-button switch is held down. If the combination is invalid, a red LED turns on, a buzzer sounds for 3 sec, and the digital display of failures is incremented by 1. When a valid combination is entered, the counter display resets to 0.


start
declare variables and constants
initialize the valid combination and the outputs
loop
wait for the enter button to be pressed
read the states of the three switches
combo valid?
N
Y
reset the counter
turn on the green LED and the motor
update the counter display
wait for the enter button to be released
turn off the LEDs and the motor
turn on the red LED
sound the buzzer for 3 seconds
increment the counter

7. *Write the code.* The PicBasic Pro code to implement the logic and flow illustrated by the flowchart follows. Note that multiple PicBasic Pro statements can be included on a single line if they are separated by colons (e.g., the multiple *Low* statements). Also, a

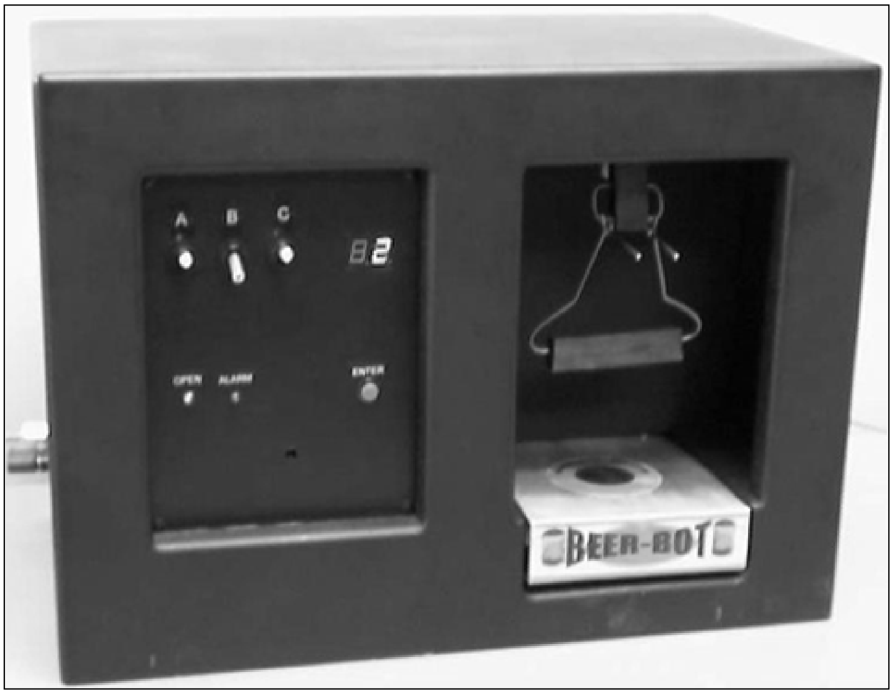
long statement can be continued on a second line by ending the first line with an underscore (e.g., the long *If* statement). Because the byte variable *number_invalid* is constrained to vary between 0 and 9, its four least significant bits represent the number in the binary coded decimal (BCD) form required by the display decoder.

```
' project.bas
' This program checks the state of three switches when a push-button switch is
'  pressed and compares the switch states to a prestored combination. If the
'  combination is valid, a green LED and a DC motor turn on and stay on while the
'  push-button switch is held down. If the combination is invalid, a red LED turns
'  on, a buzzer sounds for 3 seconds, and the displayed digit (representing the
'  number of failed attempts) increments by one. When a valid combination is
'  entered, the counter display resets to zero

' Declare all variables
switch_1 Var PORTA.0        ' first combination switch
switch_2 Var PORTA.1        ' second combination switch
switch_3 Var PORTA.2        ' third combination switch
enter_button Var PORTA.3    ' combination enter key
green_led Var PORTB.0       ' green LED indicating a valid combination
red_led Var PORTB.1         ' red LED indicating an invalid combination
speaker Var PORTB.2         ' speaker signal for sounding an alarm
motor Var PORTB.3           ' signal to bias the motor power transistor
a Var PORTB.4               ' bit 0 for the 7447 BCD input
b Var PORTB.5               ' bit 1 for the 7447 BCD input
c Var PORTB.6               ' bit 2 for the 7447 BCD input
d Var PORTB.7               ' bit 3 for the 7447 BCD input
combination Var BYTE        ' stores the valid combination in the 3 LSBs
number_invalid Var BYTE     ' counter used to keep track of the number of bad
                            '  combinations

' Initialize the valid combination and turn off all output functions
combination = %101                     ' valid combination (switch 3:on, switch
                                       '  2:off, switch 1:on)
Low green_led : Low red_led            ' make sure the LEDs are off
Low motor                              ' make sure the motor is off
Low a : Low b : Low c : Low d          ' display zero on the digit display
number_invalid = 0                     ' reset the number of invalid combinations
                                       '  to zero

' Beginning of the main polling loop
loop:
       ' Wait for the push-button switch to be pressed
       If (enter_button == 0) Then loop

       ' Read switches and compare their states to the valid combination
       If ((switch_1 == combination.0) AND (switch_2 == combination.1)_
            AND (switch_3 == combination.2)) Then
              ' Turn on the green LED
              High green_led
```

```
                ' Turn on the motor
                High motor

                ' Reset the combination attempt counter
                number_invalid = 0
        Else
                ' Turn on the red LED
                High red_led

                ' Sound the alarm
                Sound speaker, [80,100]

                ' Increment the combination attempt counter and check for overflow
                number_invalid = number_invalid + 1
                If (number_invalid > 9) Then
                        number_invalid = 0
                Endif
        Endif

        ' Update the invalid combination attempt counter digit display
         a = number_invalid.0 : b = number_invalid.1 : c = number_invalid.2 :
         d = number_invalid.3

        ' Wait for the push-button switch to be released
loop2:  If (enter_button == 1) Then loop2

        ' Turn off the LEDs and the motor
        Low green_led : Low red_led
        Low motor

        ' Loop back to the beginning of the polling loop to continue the process
        Goto loop

        End    ' end of program
```

8. *Build and test the system.* Now that we have a detailed schematic and a complete program, all that remains is to build and test the system. When first testing the system, comment out secondary parts of the code, in order to test the remaining parts. In the example, we could test the combination input and green LED but comment out the motor driver, the alarm, and the count and digital display. This way, we could ensure that the basic I/O and logic of the program function properly when the programmed PIC is inserted in the circuit. Then, additional functionality can be added a piece at a time to achieve the complete solution. We recommend you create a first prototype on a solderless breadboard until all of the bugs have been worked out. Then, a more permanent version can be created on a protoboard or printed circuit board.

When we assigned this design problem as a class project we had 30 groups of three or four students creating unique designs. Some of the more interesting designs included a wall safe, where the students fabricated a section of drywall with a face plate containing three light switches. Externally it appeared to be a set of switches to control lights in a room, but when the switches were set in the correct combination and a small push-button switch on the side was pressed, a solenoid released a spring-loaded door exposing a hidden wall safe. Another design was a rocket launcher. When the correct switch combination was entered, interesting sound effects were created (by using various *For . . . Next* loops and the PicBasic Pro *Sound* statement), and then the digital display performed a countdown. When the count reached 0, the device used a relay to fire a model rocket, which rose several hundred feet and landed softy with parachute assist. This was demonstrated to the whole class and a curious crowd on the campus grounds outside our building. The most popular design was affectionately called the *Beer-Bot* shown in the following image. This device dispensed a glass of liquid to the user if he or she knew the correct combination. When the correct combination was entered, the platform (lower right) translated out of the device with the aid of a DC motor driving a rack and pinion mechanism. The end of travel was detected by a limit switch. The platter was spring loaded so a simple switch could detect when a glass of adequate weight had been placed on it. Then the platter retracted and a pump was turned on to draw fluid from a concealed reservoir. When the liquid reached a certain level, a circuit was completed between two metal leads (top right) that pivot into the glass when the platter is retracted. At this point, the pump was turned off and the platter extended to present the full glass to the user, accompanied by delightful sound effects.



QUESTIONS AND EXERCISES

Section 7.5 PicBasic Pro

7.1. Write an assembly language program to turn an LED on and off at 0.5 Hz. Draw the schematic required for your solution.

7.2. Write a PicBasic Pro program to turn an LED on and off at 1 Hz while a push-button switch is held down. Draw the schematic required for your solution.

7.3. Write a PicBasic Pro program to perform the functionality of the Pot statement (see Class Discussion Item 7.7).

7.4. Write a PicBasic Pro subroutine to provide a software debounce on pin RB0 (see Class Discussion Item 7.8). The code should wait for the push-button switch to be pressed and released while ignoring the switch bounce.

7.5. Microcontrollers usually do not include digital-to-analog (D/A) converters, but you can easily create a crude version of one using a single digital I/O pin. This can be done by outputting a variable-width pulse train to an RC circuit. See the documentation for the PWM statement in the PicBasic Pro compiler manual for more information. Write a PicBasic Pro subroutine that uses pin RA0 to output a constant voltage (between 0 and 5 V) that is proportional to the value of a byte variable called *digital_value*, whose value can range from 0 to 255. The subroutine should hold this voltage for approximately 1 sec.

Section 7.6 Using Interrupts

7.6. Write a PicBasic Pro program to implement an interrupt-driven solution to the security system presented in Example 7.5.

Section 7.7 Interfacing Common PIC Peripherals

7.7. Write a PicBasic Pro program to display the value of a potentiometer as a percentage on an LCD display. The message should have the form *pot value = X%,* where *X* is the percentage value ranging from 0 to 100. Draw the schematic required for your solution.

7.8. Write a PicBasic Pro program that allows the user to enter a set of a multidigit numbers (up to five digits) on a numeric keypad. Have the # key serve as an enter key; and when a number is entered, it should appear on a two-line LCD display. The first number should appear on the first line, the second number should appear on the second line, and subsequent numbers should appear on the second line with the previous number moving up to the first line. Draw the schematic required for your solution.

Section 7.9 Method to Design a Microcontroller-Based System

7.9. Using a PIC16F84 and two 7447 display decoders (see Design Example 7.1), write a PicBasic Pro program to implement a counter with a two-digit display controlled by three push-button switches: one to reset the count to 0, one to increment the count by 1, and the third to decrement the count by 1. If the count is less than 10, the first digit should be blank. Incrementing past 99 should reset the count to 0 and decrementing below 0 should not be allowed. Make sure you debounce the switch input where necessary and prevent repeats while the push-button switches are held down. Use the design procedure presented in Section 7.9 and show the results for each step.

BIBLIOGRAPHY

Gibson, G. and Liu, Y., *Microcomputers for Engineers and Scientists,* Prentice-Hall, Englewood Cliffs, NJ, 1980.

Herschede, R., "Microcontroller Foundations for Mechatronics Students," masters thesis, Colorado State University, summer 1999.

Horowitz, P. and Hill, W., *The Art of Electronics,* 2nd Edition, Cambridge University Press, New York, 1989.

Microchip Technology Inc., www.microchip.com, 2001.

Microchip Technology Inc., *PIC16F8X Data Sheet,* Chandler, AZ, 1998.

Microchip Technology Inc., *MPASM User's Guide,* Chandler, AZ, 1999.

Microchip Technology Inc., *MPLAB User's Guide,* Chandler, AZ, 2000.

microEngineering Labs, Inc., www.melabs.com, 2001.

microEngineering Labs, Inc., *PicBasic Pro Compiler,* Colorado Springs, CO, 2000.

Motorola Technical Summary, "MC68HC11EA9/MC68HC711EA9 8-bit Microcontrollers," document number MC68HC11EA9TS/D, Motorola Advanced Microcontroller Division, Austin, TX, 1994.

Peatman, J., *Design with Microcontrollers,* McGraw-Hill, New York, 1988.

Stiffler, A., *Design with Microprocessors for Mechanical Engineers,* McGraw-Hill, New York, 1992.

Texas Instruments, *TTL Linear Data Book,* Dallas, TX, 1992.

Texas Instruments, *TTL Logic Data Book,* Dallas, TX, 1988.