PIC Microcontrollers

An Introduction to Microelectronics

Third Edition

Martin Bates

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON • NEW YORK • OXFORD
PARIS • SAN DIEGO • SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Newnes is an imprint of Elsevier

Newnes is an imprint of Elsevier
The Boulevard, Langford Lane, Kidlington, Oxford, OX5 1GB
225 Wyman Street, Waltham, MA 02451, USA

First edition 2000 (published by Arnold)
Second edition 2004
Third edition 2011

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Number: 2011930131

ISBN: 978-0-08-096911-4

For information on all Newnes publications
visit our website at www.elsevierdirect.com

Printed and bound in the United Kingdom

11 12 13 14 10 9 8 7 6 5 4 3 2 1

Preface to the Third Edition

The first edition of this book concentrated on a single chip that was widely used in hobby electronics and education – the PIC® 16F84A. This has now been superseded by chips that are faster, cheaper, and more complex and powerful. This has created a dilemma – whether to still use this chip that is effectively obsolete or substitute a current chip that is more complicated. In the end, I have done both – sticking with the '84 due to its relative simplicity in the initial stages, and then moving on to more recent chips with extra features, such as the 16F690. At the same time, I have taken advantage of simulation software that is now available, which provides on-screen animated circuits and user-friendly microcontroller program debugging.

All the main points in this book are illustrated by simple examples, which are downloadable from the support website at www.picmicros.org.uk. Program source code can be modified, reassembled and retested using Microchip's MPLAB® IDE development system, downloadable free from www.microchip.com. This website also provides many of the technical references and data sheets used in the book. The schematic capture and simulation software is from Labcenter Electronics at www.labcenter.com. A demo version is available, but to create and test your own applications, a license will be needed. A low-cost package, including a model for the 16F84A, is currently available.

The book is aimed at beginners, so more experienced readers should skip over any parts that are already familiar. Some basic principles have been moved to the appendices in this edition, to make room for updated applications and examples. Again, these are aimed primarily at learners at college or university, or independent hobbyists. Nevertheless, I hope that more experienced readers will find some of the examples useful, and will perhaps see the advantages of some of the techniques described, particularly interactive simulations, which enrich the application development experience at all levels, and potentially enhance productivity for the professional electronic design engineer.

Martin Bates
Sussex, England, July 2011

Introduction to the Third Edition

The microcontroller is now at the heart of many electronic products. Mobile phones, microwave ovens, digital television, credit cards, the Internet and many other current technologies rely on these small, unobtrusive devices to make it all happen.

This book is an attempt to introduce the beginner to this ubiquitous yet complex technology. Starting with the standard PC (on the basis that most people are familiar with its operation), the basic concepts and terminology will be established: microprocessor systems, memory, input and output, and general digital systems ideas. We will then go on to study one of the biggest selling products the general public has never heard of: the PIC® microcontroller (MCU). It dominates the market for small-scale industrial applications, with the manufacturer Microchip Technology Inc. currently reporting annual sales over US $1 billion per annum.

We will start by studying a chip that is no longer commercially significant, but is relatively simple, with the minimum of advanced features: the PIC 16F84A. This was one of the first popular small microcontrollers with flash program memory, the kind found in memory sticks. This allows it to be easily reprogrammed and therefore made it ideal for learning and hobby electronics. We will learn how to connect up and program this chip, and design simple applications, such as flashing output LEDs. In addition, simulation software will be introduced, which makes the design process easier, and more fun. We will then move on to the PIC 16F690, which has more features and is representative of more recent products in the PIC range. Many microcontrollers used in real applications such as motor vehicle engine control or communications systems are more powerful, but the operating principles are just the same. Other types of control technology will be reviewed for comparison with microcontrollers.

The book uses numerous examples relating to motor control, because this is a very common application (disk drives, washing machines, conveyors, etc.). The small direct current motor is inexpensive and can be easily connected to the output of a PIC via a simple current driver interface. The response of the motor is easily observed, yet can be complex, which demonstrates the problems associated with real-time system control. The motor also provides a link to wider areas of engineering – mechatronics, robots, machine tools and industrial systems – that is useful for students and engineers in these disciplines.

The big problem with microprocessors and microcontrollers is that, to fully understand how they work, we have to understand both the hardware and the software at the same time. Therefore, we have to circle round the subject, looking at the system from different angles, until a reasonable level of understanding is built up. The book will cover basic hardware design, interfacing, program development, debugging, testing and analysis using a range of simple examples. This is supported by appendices, which introduce basic concepts to readers who do not have this essential background – number systems, digital principles and microprocessor system concepts, as well as system design exercises. Appendix E covers the whole design process using the Proteus VSM™ electronic design suite.

There is a summary at the start of each chapter, so that its content can be seen at a glance, as well as a set of questions at the end for self-assessment or formal testing of students (with full answers at the end of the book) and suggested activities which can be developed into practical assessments if required. The style of the book is also intended as a model for students who need to write technical reports for such practical assessments. The stages of application development should be clearly identified in this case: specification, design, implementation and testing. Another useful model can be seen in the application notes written by professional engineers, such as those available on the Microchip website.

The content of each chapter is a compromise between maintaining overall continuity and allowing each chapter to be read independently. There will therefore be a certain amount of repetition between chapters, which I hope the reader will not find too irritating, and may aid learning. It is always difficult to decide exactly what to include in this kind of book, where the subject is vast and complex. My intention is always to keep it simple, and I hope my selection will help the reader to begin to get to grips with the fascinating world of microcontrollers, develop a reasonable understanding of real applications, and perhaps progress to a career in microcontroller application design. However, an understanding of microcontrollers is essential for any electrical engineer, since the technology is now central to most electronic products and industrial systems.

CHAPTER 1

Computer Systems

Chapter Outline

Chapter Points

- A microprocessor system consists of data input, storage, processing and output devices, under the control of a CPU.
- The main unit of a desktop PC is a modular system, consisting of the motherboard, power supply and disk drives.
- The motherboard carries the microprocessor (CPU), RAM, BIOS ROM, bus controllers and I/O interfaces.
- The CPU communicates with the main system chips via a shared set of address and data bus lines.
- The microcontroller provides most of the features of a conventional microprocessor system on one chip.

PIC Microcontrollers. DOI: 10.1016/B978-0-08-096911-4.10001-1

In this chapter, we will start with something familiar, looking at how a personal computer (PC) works when running a word processor, to establish a few technical concepts that are used in microcontrollers (MCUs). Hopefully, most readers will be familiar with this, and will know how the application functions from the user's point of view. Some basic microcontroller system ideas will be introduced by analyzing how software interacts with computer hardware, allowing the user to enter, store and process documents. For example, we will see why different kinds of memory are needed to support the system operation. If you are familiar with these concepts, you can skip this chapter.

The PC also provides the hardware platform for the PIC® program development system. The programs for the PIC are written using a text editor, and the machine code program is created and downloaded to the PIC chip using the PC. The PIC development system hardware can be seen connected in Figure 1.1. We will see how this works later.

We will also have a quick look at a basic microcontroller system, set up to operate as a simple equivalent of the microprocessor-based PC system, to see how it compares. Here, the microcontroller has a keypad with only 12 keys instead of a keyboard, and a seven-segment display instead of a screen. Its memory is much smaller than the PC, yet it can carry out the same basic tasks. In fact, it is far more versatile; the Intel™ processors used in the PC are designed specifically for that system. The microcontroller can be used in a great variety of circuits, and it is much cheaper.

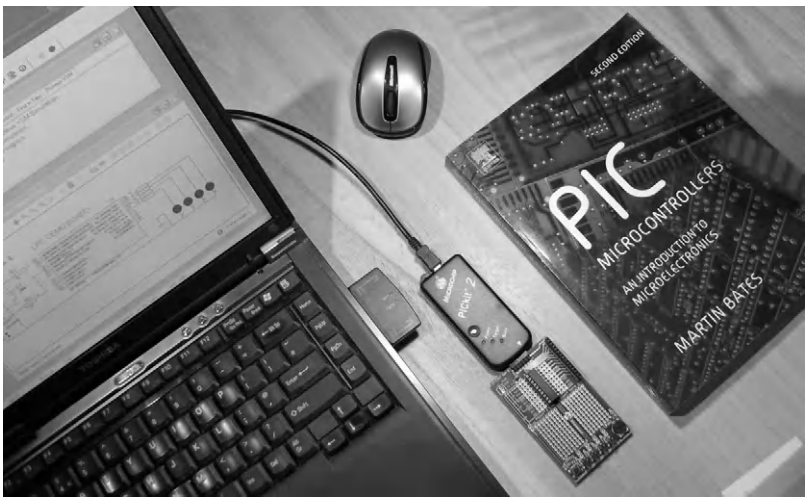


Figure 1.1
Laptop with PIC demo system attached

1.1. Personal Computer System

The conventional desktop system comprises a main unit, separate keyboard and mouse, and monitor. The main unit has connectors for these (when wireless peripherals are not available) and universal serial bus (USB) ports for memory sticks, printers, scanners, etc., as well as hardwired (Ethernet), or wireless (Wi-Fi) network interfaces. The circuit board (motherboard) in the main unit carries a group of chips which work together to provide digital processing of information and control of input and output devices. A power supply for the motherboard and the peripheral devices is included in the main unit.

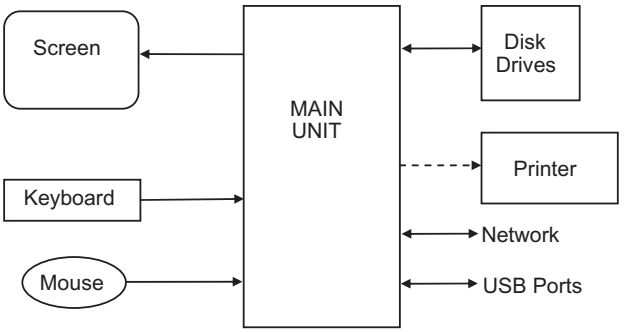
The laptop has the same components in a compact form, with integrated keyboard and screen, while tablet computers are even more compact with a touch-sensitive screen and no keyboard. The difference between a microprocessor and microcontroller system is illustrated quite well by comparing a desktop computer with a touch-screen game console or mobile phone. The facilities and applications are similar, they just differ in scale and complexity.

A block diagram (Figure 1.2a) is a good way to show such a system in simplified form, so we can identify the main components and how they connect. In the case of the disk drives and network, for example, the information flow is bidirectional, representing the process of saving data to, and retrieving data from, the hard disk or server. The internal architecture of a microcontroller is shown in its data sheet as a block diagram.

Any microprocessor or microcontroller system must have software to run on the hardware. In a desktop, this is stored on a hard disk inside the main unit; this can hold a large amount of data that is retained when the power is off. There are two main types of software required: the operating system (e.g. Microsoft® Windows) and the application (e.g. Microsoft® Word). As well as the operating system and application software, the hard disk stores the data created by the user, in this case, document files.

The keyboard is used for data input, and the screen displays the resulting document. The mouse provides an additional input device, allowing control operations to be selected from menus or by clicking on icons and buttons. This provides a much more user-friendly interface than earlier computers, which had a command-line interface. Then, actions were initiated by typing a text command such as ‘dir’ to show a directory (folder) of files. Network specialists still use this type of interface as it allows batch files (list of commands) to be created to control system operation. The network interface allows us to download data or applications from a local or remote server, or share resources such as printers over a local area network (LAN) and provide access to a wide area network (WAN), usually the Internet. In the domestic environment, a modem is currently needed to connect to the Internet via a telephone line or cable service. The network browser (e.g. Microsoft® Internet Explorer) is then another essential application.

(a)



(b)

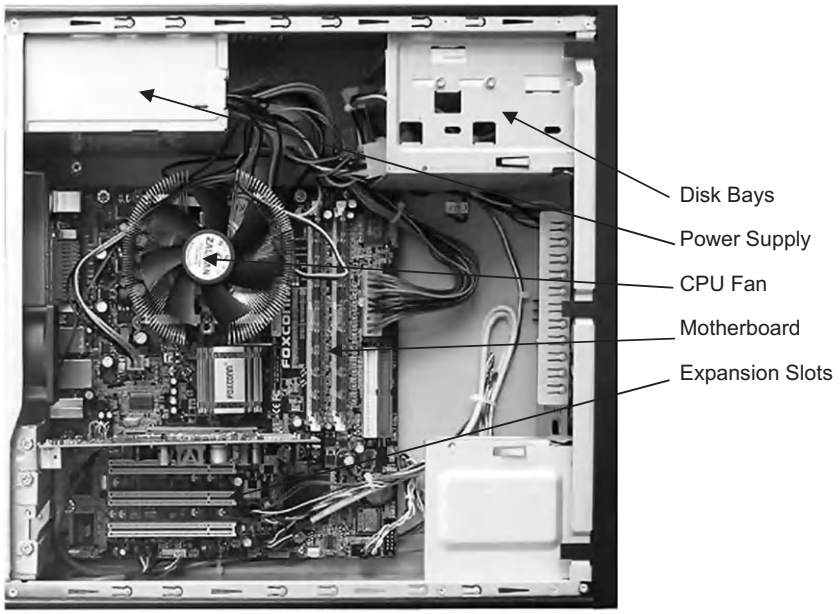


Figure 1.2
The PC system: (a) block diagram of PC system; (b) view of PC desktop main unit

1.1.1. PC Hardware

Inside the PC main unit (Figure 1.2b), the traditional motherboard has slots for expansion boards and memory modules to be added to the system. The power supply and disk drives are fitted separately into the main unit frame. The keyboard and mouse interface are integrated on the motherboard. In older designs, expansion boards carried interface circuits for the disk

drives and external peripherals such as the display and printer, but these functions are now increasingly incorporated into the motherboard itself. Peripherals are now usually connected via USB or wirelessly.

The desktop PC is a modular system, which allows the hardware to be put together to meet the individual user's requirements, with components sourced from different specialist suppliers, and allows subsystems, such as disk drives and keyboard, to be easily replaced if faulty. This also allows easy upgrading (e.g. fitting extra memory chips) and also makes the PC architecture well suited to industrial applications. In this case, the PC can be 'ruggedized' (put into a more robust casing) for use on the factory floor. This modular architecture is one of the reasons for the success of the desktop PC hardware, which has continued in the same basic form for many years, as a universal processor platform. The laptop is the main alternative for the general user, but it is not so flexible, and tends to be replaced rather than upgraded. Another reason for its success is the dominance of Microsoft operating systems, which have developed in conjunction with the Intel-based hardware, providing a standard platform for domestic, commercial and industrial computers.

1.1.2. PC Motherboard

The main features of typical motherboards are shown in Figure 1.3. The heart of the system is the microprocessor, a single chip, or central processing unit (CPU). The CPU controls all the other system components, and must have access to a suitable program in memory before it can do anything useful. The blocks of program required at any one time are provided by both the operating system and the application software, which are downloaded to random access memory (RAM) from the hard disk as required. The programs consist of lists of machine code instructions (binary code) that are executed in sequence by the CPU.

The Intel CPU has undergone rapid and continuous development since the introduction of the PC in the early 1980s. Intel processors are classified as complex instruction set computer (CISC) chips, which means they have a relatively large number of instructions that can be used in a number of different ways. This makes them powerful, but relatively slow compared with processors that have fewer instructions; these are classified as reduced instruction set computer (RISC) chips, of which the PIC microcontroller is an example.

The CPU needs memory and input/output devices for getting data in, storing it and sending it out again. The main memory block is made up of RAM chips, which are generally mounted in Dual In-line Memory Modules (DIMMs). As far as possible, input/output (I/O) interfacing hardware is fitted on the motherboard (keyboard, mouse, USB, etc., preferably wireless), but additional peripheral interfacing boards may be fitted in the expansion card slots to connect the main board to extra disk drives and other specialist peripherals, traditionally using the PCI bus, a parallel data highway 32 bits wide.

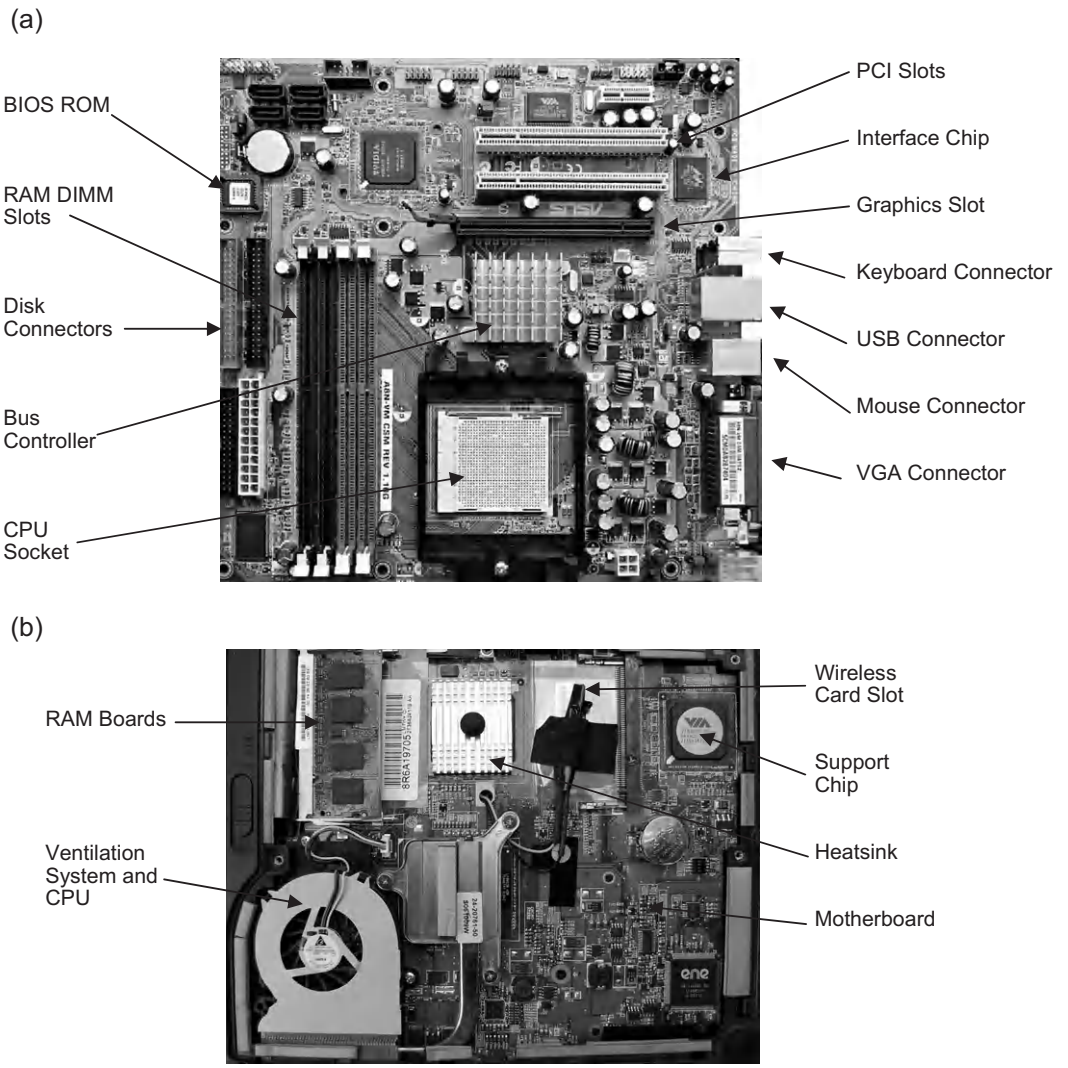


Figure 1.3
PC motherboards: (a) desktop motherboard; (b) laptop motherboard

All these parts are connected together via a pair of bus controller chips, which handle parallel data transfers between the CPU and the system. The 'northbridge' provides fast access to RAM and the graphics (screen) interface, while its partner, the 'southbridge', handles slower peripherals such as the disk drives, network and PCI bus. The motherboard itself can be represented as a block diagram (Figure 1.4) to show how the components are interconnected.

The block diagram shows that the CPU is connected to the peripheral interfaces by a set of bus lines. These are groups of connections on the motherboard, which work together to transfer the

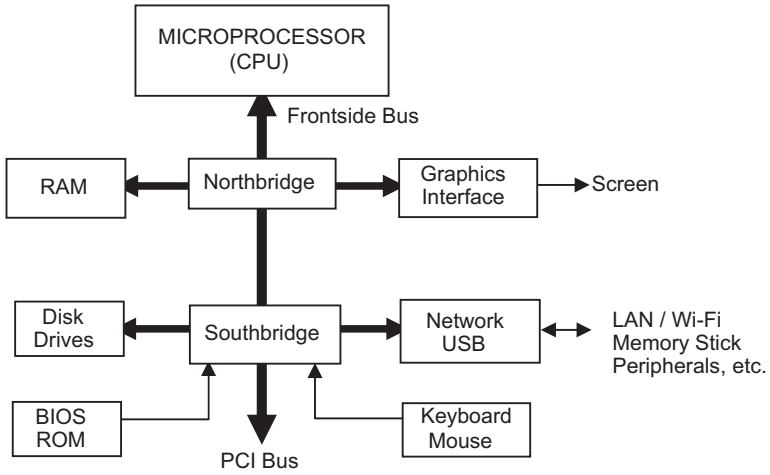


Figure 1.4
Block diagram of PC motherboard

data from the inputs, such as the keyboard, to the processor, and from the processor to memory. When the data has been processed and stored, it can be sent to an output peripheral, such as the screen.

Buses connect all the main chips in the system together, but, because they mainly operate as shared connections, can only pass data to or from one peripheral interface or memory location at a time. This arrangement is used because separate connections to all the main chips would need an impossible number of tracks on the motherboard. The disadvantage of bus connection is that it slows down the program execution speed, because all data transfers use the same set of lines, and only one data word can be present on the bus at any one time. To help compensate for this, the bus connections are as wide as possible. For example, a 64-bit bus, operating at 100 MHz (10^8 Hz), can transfer 6.4 gigabits (6.4×10^9 bits) per second. The current generation of Intel® CPUs also use multiple (typically 4) 64-bit cores in one chip to improve performance.

1.1.3. PC Memory

There are two principal types of memory in the PC system. The main memory block is RAM, where input data is stored before and after processing in the CPU. The operating system and application program are also copied to RAM from disk for execution, because access to data in RAM is faster. Unfortunately, RAM storage is 'volatile', which means that the data and application software disappear when the PC is switched off, and these have to be reloaded each time the computer is switched back on.

This means that some read-only memory (ROM), which is non-volatile, is needed to get the system started at switch on. The basic input/output system (BIOS) ROM chip contains enough

code to check the system hardware and load the main operating system software from disk. It also contains some basic hardware control routines so that the keyboard and screen can be used before the main system has been loaded.

The hard disk is a non-volatile, read and write storage device, consisting of a set of metal disks with a magnetic recording surface, read/write heads, motors and control hardware. It provides a large volume of data storage for the operating system, application and user files. The applications are stored on disk and then selected as required for loading into memory; because the disk is a read and write device, user files can be stored, applications added and software updates easily installed. Standard hard disk drives can now hold over 1 TB (1 terabyte = 10^{12} bytes) of data.

The PC system quickly becomes ever more elaborate, and this description may well already be out of date in some respects. However, the basic principles of microprocessor system operation are the same as established in the earliest digital computers, and these also apply to microcontrollers, as we will see.

1.2. Word-Processor Operation

In order to understand the operation of the PC microprocessor system, we will look at how the word-processor application uses the hardware and software resources. This will help us to understand the same basic processes that occur in microcontrollers.

1.2.1. Starting the Computer

When the PC is switched on, the RAM is empty. The operating system, application software and user files are all stored on the hard disk, so the elements needed to run the word processor must be transferred to RAM for quick access when using the application. The BIOS gets the system started. It checks that the hardware is working properly, loads (copies) the main operating system software (e.g. Windows) from hard disk into RAM, which then takes over. As you will probably have noticed, this all takes some time; this is because of the amount of data transfer required and the relatively slow access to the hard drive.

1.2.2. Starting the Application

Windows displays an initial screen with icons and menus, which allow the application to be selected by clicking on a shortcut. Windows converts this action into an operating system command which runs the executable file (WINWORD.EXE, etc.) stored on disk. The application program is transferred from disk to RAM, or as much of it as will fit in the available memory. The word-processor screen is displayed and a new document file can be created or an existing one loaded by the user from disk.

1.2.3. Data Input

The primary data input is from the keyboard, which consists of a grid of switches that are scanned by a dedicated microcontroller within the keyboard unit. This chip detects when a key has been pressed, and sends a corresponding code to the CPU via a serial data line in the keyboard cable, or wirelessly. The serial data is a sequence of voltage pulses on a single wire, which represent a binary code, each key generating a different code. The keyboard interface converts this serial code to parallel form for transfer to the CPU via the system data bus. It also signals separately to the CPU that a keycode is ready to be read into the CPU, by generating an 'interrupt' signal. This serial-to-parallel (or parallel-to-serial) data conversion process is required in all the interfaces that use serial data transfer, such as the keyboard, screen and network (see the appendices for more information on binary coding, and serial and parallel data).

The mouse is a convenient pointer controller for selecting options on screen and drawing graphics. The original mouse used two rollers set at right angles, with perforated disks attached. The holes were detected using an opto-sensor, sending pulses representing movement in two directions to the CPU. This mechanism has been replaced with direct optical sensing of variations in the surface under the mouse, using complex software to extract the direction and speed information. This also eliminates unreliable mechanical components.

Data input from a network or USB source is also in serial form, while the internal disk interface is traditionally in parallel, direct onto the peripheral bus. The parallel connection is inherently faster, since data bits are transferred simultaneously on all bus lines.

1.2.4. Data Storage

The character data is received by the CPU from the keyboard, or other interface, in parallel form, via the internal data bus. It is stored in a CPU register and then copied back to RAM. RAM locations are numbered and accessed via the system address bus, a set of lines that select a location as a binary number. This is why the CPU has so many pins: for speed of transfer, all data and address pins, and control lines, are separately connected to the northbridge controller via the frontside bus, and hence to the RAM. The data is stored in RAM as charge on the gate of an electronic switch, a field effect transistor (FET; see Appendix B). When charged, the FET is switched on, and this state can be read back at a later time. The addressing system accesses an array of these switches in rows and columns to store and retrieve bits of data. Each byte has a unique address.

1.2.5. Data Processing

The data processing in the CPU required by a simple text editor is minimal; the input characters are simply stored as binary code and displayed, with a separate graphics processor

converting the character code to a corresponding symbol on the screen. Nevertheless, the word-processor program has to handle different fonts, word wrapping at the end of lines and so on. It also has to handle text, page and document formatting, menu systems and the user interface. Editing embedded graphics is a bit more complex, since each pixel needs handling separately. The most demanding applications are those where the real world is simulated in a computer model in order to make predictions about the behavior of complex systems. Weather forecasting is an extreme example; the fact that we can still only forecast accurately a few days ahead illustrates the limitations of such system modeling, even on the most powerful computers.

The circuit simulation software used in this book, Proteus VSM, combining traditional circuit analysis with an interactive interface, is a good example of system modeling in a PC. It takes a circuit created as a schematic and applies network analysis (lots of simultaneous equations) to predict its operation when constructed. For digital elements, logic modeling is needed, and then the analogue and digital domains are co-simulated. Component characteristics and input variables are typically represented by 32-bit binary numbers, which correspond to decimal numbers in exponential form (as on a scientific calculator). The processor needs to be able to manipulate these circuit variables simultaneously to represent the circuit conditions at a series of points in time. The output is calculated and displayed via animated circuit components or virtual instruments, or graphically. Numerous examples are to follow!

1.2.6. Data Output

Going back to the word processor, the characters must be displayed on the screen as they are typed in, so the character codes stored in memory are also sent to the screen via the graphics interface. The display is made up of single colored dots (pixels) organized in lines across the screen, which are accessed in sequence, forming a scanned display. The shape of the character on screen must be generated from its code in memory, and sent out on the correct set of lines at the right time. The display must therefore be created as a two-dimensional image made up from a serial data stream which sets the color of each pixel on the screen in turn, line by line, where each line of text occupies a set of adjacent lines. The exact arrangement depends on the font type and size.

If a file is transferred on a network, it must also be sent in serial form. The characters (letters) in a text file are normally sent as ASCII code, along with formatting information and network control codes. ASCII code represents one character as one byte (8 bits) of binary code, and is therefore a very compact form of the data. The code for the letter ‘A’ (upper case), for example, is 01000001.

The printer works in a similar way to the screen, except that the output is generated as lines of dots of ink on a page. In an inkjet printer, you can see the scanning operation

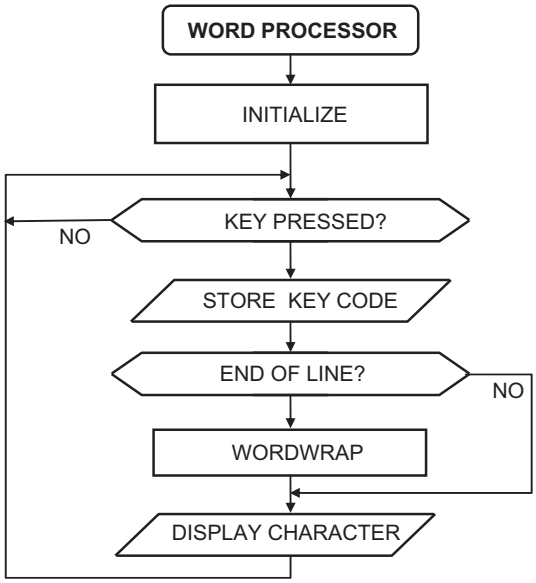


Figure 1.5
Word-processor flowchart

taking place. These days, printer data is usually transferred in serial form on a USB, wireless or network link. The printer itself is capable of formatting the final output; only the character code and any formatting codes are needed. A portable document format (PDF) file is a standard output format for the display and printing of documents containing text and graphics.

The operation of the word processor can be illustrated using a flowchart, which is a graphical method of describing a sequential process. Figure 1.5 shows the basic process of text input and word wrapping at the end of each line. Flowcharts will be used later to represent microcontroller program operation.

1.3. Microprocessor Systems

All microprocessor systems perform the same essential functions, that is, data or signal input, storage, processing and output. However, the PC is a relatively complex microprocessor system, with a hierarchical bus structure, which has developed to improve system performance by alleviating the bus bottleneck of earlier designs. The Intel PC processor itself also has many additional performance-enhancing features such as cache memory, multiple processing pipelines and multiple cores. To understand the microcontroller, we need to go back to a simpler system.

The basic microprocessor system needs a certain set of chips, with suitable interconnections, as follows:

- CPU
- RAM
- ROM
- I/O ports.

These devices must be interconnected by:

- Address bus
- Data bus
- Various control lines.

These buses and control lines originate from the CPU, which is in charge of the system. RAM and ROM chips are usually general purpose hardware, which can be used in any system. The I/O chips are sometimes designed to work with a particular processor, but all provide specific interfacing functions. In a basic system, this would be simple digital input and output, with perhaps a serial port providing an RS232 (see Chapter 12) type data link.

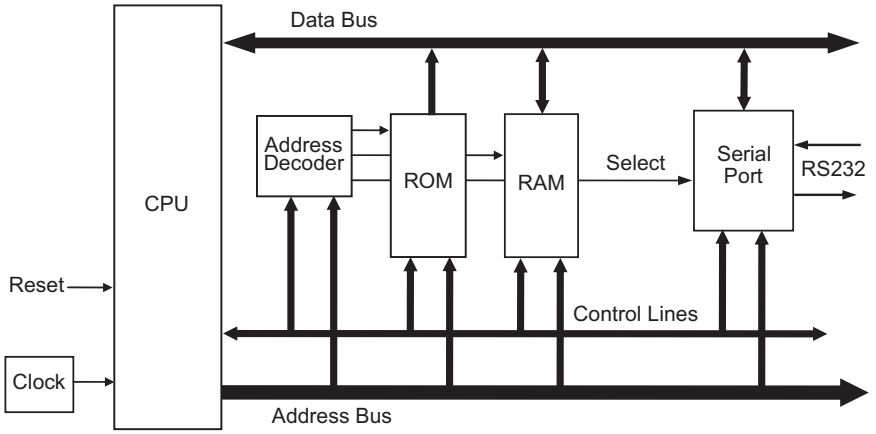
Additional support chips are needed to make a CPU system work. In a minimal system, an address decoder is needed to select the memory chip or I/O device required for a data transfer to or from the CPU. This system is illustrated in Figure 1.6a, and there is further information about microprocessor system operation in Chapter 14 and Appendix C.

1.3.1. System Operation

The CPU controls the system data transfers via the data and address buses and additional control lines. A clock circuit, usually containing a crystal oscillator (as found in digital watches), is required; this produces a precise fixed frequency signal that drives the microprocessor along. The CPU operations are triggered on the rising and falling edges of the clock signal, allowing their exact timing to be defined. This allows events in the CPU to be completed in the correct sequence, with sufficient time allowed for each step. The CPU generates all the main control signals based on the clock. A given CPU can be used in different system designs, depending on the type of application, the amount of memory needed, the I/O requirements and so on.

The address decoder controls access to memory and I/O registers for a particular design. Typically, a programmable logic device (PLD) is used to allocate each memory chip to a specific range of addresses. An input address code in a particular range generates a chip select output, which enables that device. The I/O port registers, which are set up to handle the data transfer in and out of the system, are also allocated particular addresses by the same mechanism, and accessed by the CPU in the same way as memory locations. The allocation of addresses to particular peripheral devices is called a memory map (Figure 1.6b).

(a)



(b)

Peripheral Device	Address Range	Memory Size
ROM	0000–03FF	1k
RAM	1000–8FFF	32k
Port	E000–E00F	16 registers

Figure 1.6
Microprocessor system: (a) block diagram; (b) typical memory map

1.3.2. Program Execution

The ROM and RAM will contain program code and data in numbered locations, that are selected by a binary code at its address inputs. If the program is in ROM, it can start immediately (as in the PC BIOS), but RAM must be loaded from a non-volatile program store, such as a hard disk.

A register is a temporary store for data within the CPU or port. In the port chip, it holds working data or a control code which sets up how the port will operate. For example, the bits in the data direction register control whether each port pin operates as an input or an output. The data being sent in or out is then stored temporarily in the port data register.

The program consists of a list of instructions in binary code stored in memory, with each instruction and any associated data (operands) being stored in sequential locations. The program instruction codes are fetched into the CPU and decoded. The CPU sets up the internal and external control lines as necessary and carries out the operation specified in the program, such as read a character code from the serial port into the CPU. The instructions are executed in order of their addresses, unless the instruction itself causes a jump to another point in the program, or an ‘interrupt’ (signal) is received from an internal or external source. The program counter keeps track of the current step.

1.3.3. Execution Cycle

Program execution is illustrated in Figure 1.7. Assuming that the application program code is in RAM, the program execution cycle proceeds as follows:

1. The CPU outputs (1) the address of the memory location containing the required instruction (this address is kept in the program counter). The sample address is shown in hexadecimal form (3A24) in Figure 1.7, but it is output in binary form on the address lines from the processor (for an explanation of hex numbering see Appendix A). The address decoder logic uses the address to select the RAM chip that has been allocated to this address. The address bus also connects directly to the RAM chip to select the individual location, giving a two-stage memory location select process.
2. The instruction code is returned to the CPU from the RAM chip via the data bus (2). The CPU reads the instruction from the data bus into an instruction register. The CPU then decodes and executes the instruction (3). The operands (code to be processed) are fetched (4) from the following locations in RAM via the data bus, in the same way as the instruction.
3. The instruction execution continues by feeding the operand(s) to the data processing logic (5). Additional data can be fetched from memory (6). The result of the operation is stored in a data register (7), and then, if necessary, in memory (8) for later use. In the meantime, the program counter has been incremented (increased) to the address of the next instruction code. The address of the next instruction is then output and the sequence repeats from step 2.

The operating system, the application program and the user data are stored in different parts of RAM during program execution, and the application program calls up operating system

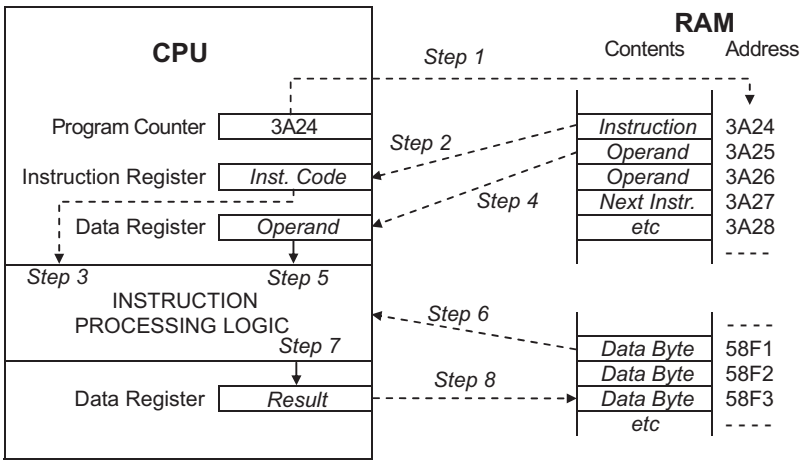


Figure 1.7
Program execution sequence

routines as required to read in, process and store the data. PC processors have multi-byte instructions, which are stored in multiple 8-bit locations, and use complex memory management techniques to speed up program execution.

1.4. Microcontroller Applications

We have now looked at some of the main ideas to be used in explaining microcontroller operation: hardware, software, how they interact and how the function of complex systems can be represented using block diagrams and flowcharts.

The microcontroller provides, in a simplified form, all the main elements of the microprocessor system on a single chip. As a result, less complex applications can be designed and built quickly and cheaply. A working system can consist of a microcontroller chip and just a few external components for feeding data and signals in and out. They tend to be used for control operations requiring limited amounts of memory but operating at high speed, with external hardware attached only as required by a specific application.

As an example of a typical microcontroller system, a digital camera, is shown in Figure 1.8(a), with the microcontroller clearly visible as the large black chip on the main board. A block diagram is a useful way of identifying the main components and the connections between them Figure 1.8(b). This is called a mechatronic application, because it has a lot of mechanical components as well as electronics.

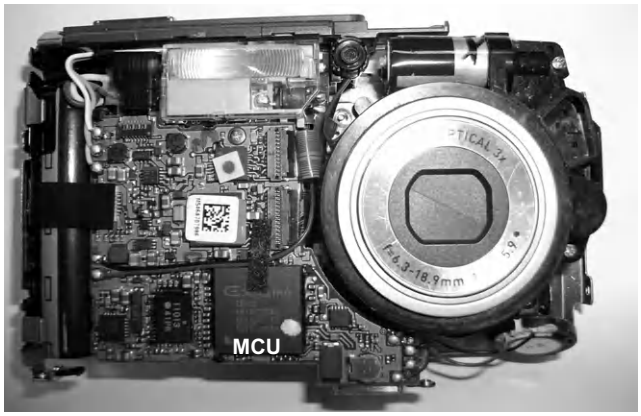
1.4.1. Microcontroller Application Design

A simple microcontroller-based equivalent of the word-processing application described above is shown in Figure 1.9. The purpose of the system is to store and display numbers that are input on the keypad. Four inputs and three outputs are required for keypad connection to the microcontroller, but to simplify the drawing, these parallel connections are represented by the block arrows. The operation of the keypad is explained in more detail in Chapter 13 (see Figure 13.3). The seven segment displays show the input numbers as they are stored in the microcontroller. Each display digit consists of seven light-emitting diodes (LEDs), such that each digit from 0 to 9 is displayed as a suitable pattern of lit segments.

The basic display program works as follows: when a key is pressed, the digit is displayed on the right (least significant) digit, and subsequent keystrokes will cause the previously entered digit to shift to the left, to allow decimal numbers up to 99 to be stored and displayed. Calculations could then be performed on the data, and the result displayed. Obviously, real calculators have more digits, but the principle is the same.

The block diagram can then be converted into a circuit diagram using schematic capture software. Labcenter ISIS, part of the Proteus VSM package, has been used to create

(a)



(b)

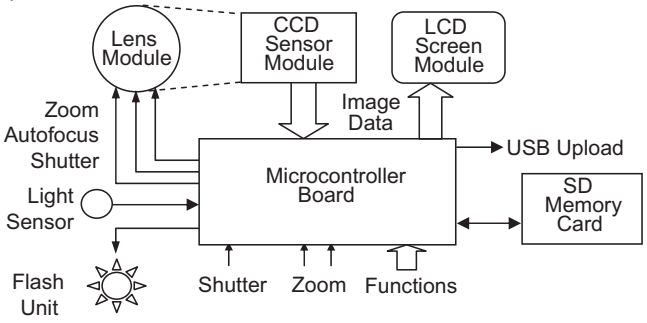


Figure 1.8
Typical microcontroller system: (a) digital camera (MCU labeled); (b) block diagram of digital camera

Figure 1.9(b). A provisional choice of microcontroller must be made (which can be changed later) and the connections worked out. The PIC 16F690 has been selected here because it has a suitable number of inputs and outputs available, and is used on the Microchip Technology Inc. (Microchip) demonstration board to be studied later. A programming connector is also needed to get the program into the MCU. It is not necessary to include this in the block diagram, as it is implicit in the PIC design.

The starting point for writing the program for the microcontroller is to convert the general specification such as that given above into a description of the operations, which can be programmed into the chip using the set of instructions that are available for that microcontroller. The instruction set is defined by the manufacturer of the device. The process by which the required function is implemented is called the program algorithm, which can be described using a flowchart (Figure 1.9c).

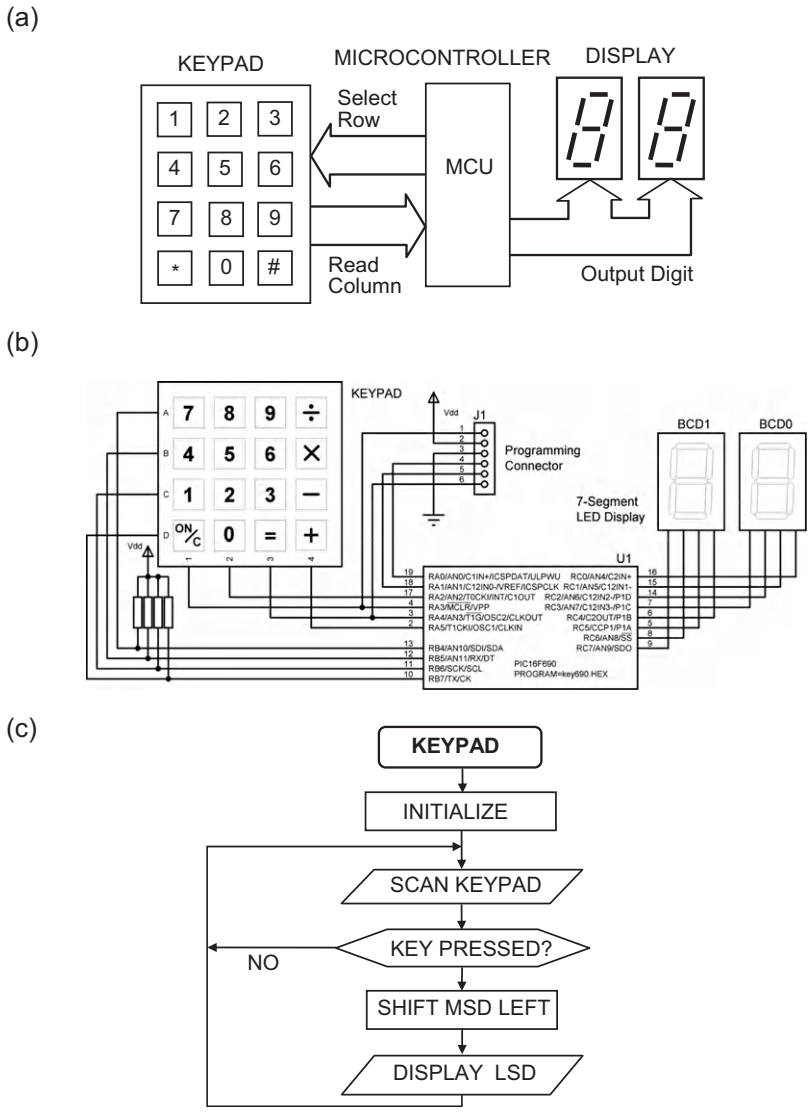
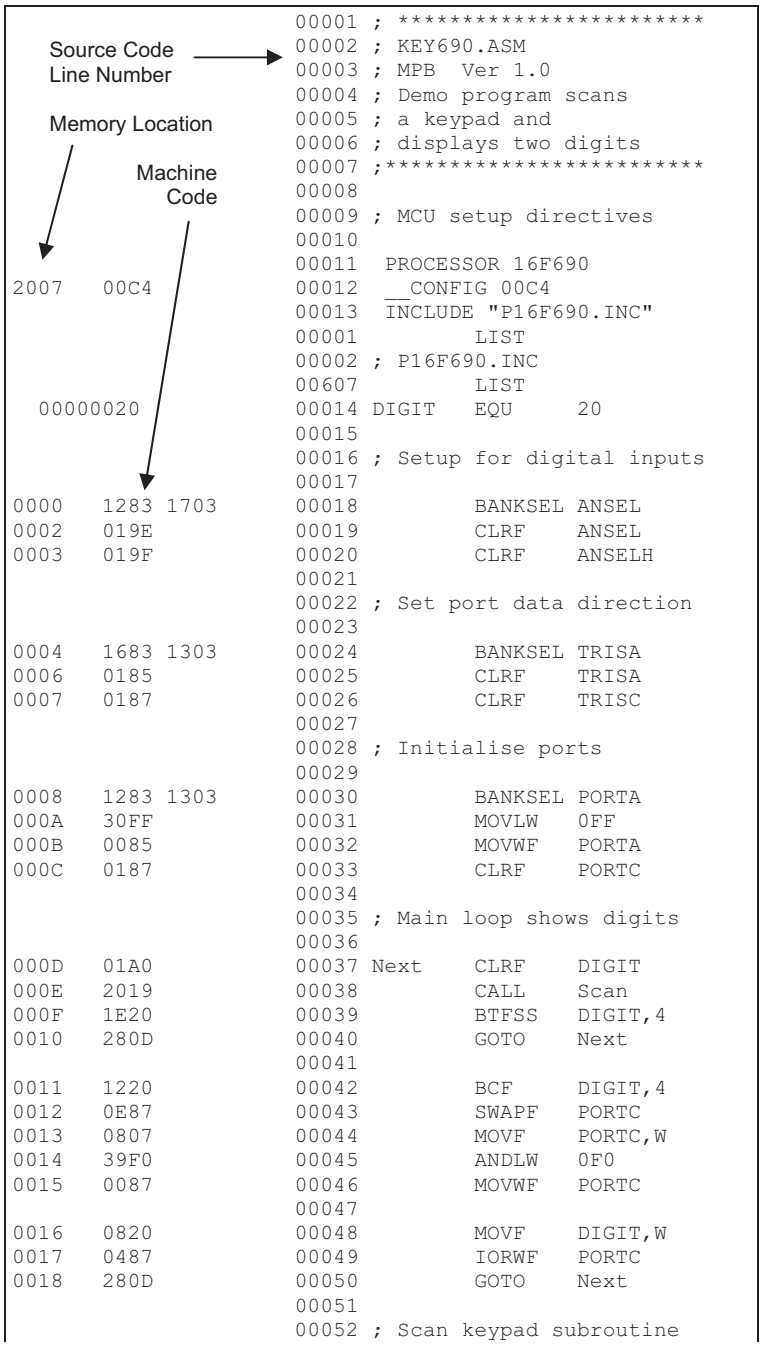


Figure 1.9
Keypad display system: (a) block diagram; (b) schematic; (c) flowchart (MSD: most significant digit; LSD: least significant digit)

This flowchart is now converted into a program, which is listed as Program 1.1. This source code is typed into a text editor and converted into a machine code program in the host PC, and downloaded to the chip via a programming module connected to the USB port (Figure 1.11). The main object of this book is to provide the reader with sufficient information to develop this kind of simple application, with a view to progressing to more complex projects. Proteus VSM allows the circuit to be tested on screen, with this program




```
                        00001 ; ************************
                        00002 ; KEY690.ASM
                        00003 ; MPB  Ver 1.0
                        00004 ; Demo program scans
                        00005 ; a keypad and
                        00006 ; displays two digits
                        00007 ;************************
                        00008
                        00009 ; MCU setup directives
                        00010
                        00011  PROCESSOR 16F690
2007   00C4             00012  __CONFIG 00C4
                        00013  INCLUDE "P16F690.INC"
                        00001         LIST
                        00002 ; P16F690.INC
                        00607         LIST
  00000020              00014 DIGIT   EQU     20
                        00015
                        00016 ; Setup for digital inputs
                        00017
0000   1283 1703        00018         BANKSEL ANSEL
0002   019E             00019         CLRF    ANSEL
0003   019F             00020         CLRF    ANSELH
                        00021
                        00022 ; Set port data direction
                        00023
0004   1683 1303        00024         BANKSEL TRISA
0006   0185             00025         CLRF    TRISA
0007   0187             00026         CLRF    TRISC
                        00027
                        00028 ; Initialise ports
                        00029
0008   1283 1303        00030         BANKSEL PORTA
000A   30FF             00031         MOVLW   0FF
000B   0085             00032         MOVWF   PORTA
000C   0187             00033         CLRF    PORTC
                        00034
                        00035 ; Main loop shows digits
                        00036
000D   01A0             00037 Next    CLRF    DIGIT
000E   2019             00038         CALL    Scan
000F   1E20             00039         BTFSS   DIGIT,4
0010   280D             00040         GOTO    Next
                        00041
0011   1220             00042         BCF     DIGIT,4
0012   0E87             00043         SWAPF   PORTC
0013   0807             00044         MOVF    PORTC,W
0014   39F0             00045         ANDLW   0F0
0015   0087             00046         MOVWF   PORTC
                        00047
0016   0820             00048         MOVF    DIGIT,W
0017   0487             00049         IORWF   PORTC
0018   280D             00050         GOTO    Next
                        00051
                        00052 ; Scan keypad subroutine
```

Program 1.1
Keypad program list file

```
                        00053
0019   1185             00054 Scan    BCF     PORTA,3
001A   1F06             00055         BTFSS   PORTB,6
001B   2839             00056         GOTO    one
001C   1E86             00057         BTFSS   PORTB,5
001D   2848             00058         GOTO    four
001E   1E06             00059         BTFSS   PORTB,4
001F   2857             00060         GOTO    seven
0020   1585             00061         BSF     PORTA,3
                        00062
0021   1105             00063         BCF     PORTA,2
0022   1F86             00064         BTFSS   PORTB,7
0023   2834             00065         GOTO    zero
0024   1F06             00066         BTFSS   PORTB,6
0025   283E             00067         GOTO    two
0026   1E86             00068         BTFSS   PORTB,5
0027   284D             00069         GOTO    five
0028   1E06             00070         BTFSS   PORTB,4
0029   285C             00071         GOTO    eight
002A   1505             00072         BSF     PORTA,2
                        00073
002B   1205             00074         BCF     PORTA,4
002C   1F06             00075         BTFSS   PORTB,6
002D   2843             00076         GOTO    three
002E   1E86             00077         BTFSS   PORTB,5
002F   2852             00078         GOTO    six
0030   1E06             00079         BTFSS   PORTB,4
0031   2861             00080         GOTO    nine
0032   1605             00081         BSF     PORTA,4
                        00082
0033   0008             00083         RETURN
                        00084
                        00085 ; Get number when key hit
                        00086
0034   3010             00087 zero    MOVLW   010
0035   00A0             00088         MOVWF   DIGIT
0036   1F86             00089         BTFSS   PORTB,7
0037   2834             00090         GOTO    zero
0038   2819             00091         GOTO    Scan
                        00092
0039   3011             00093 one     MOVLW   011
003A   00A0             00094         MOVWF   DIGIT
003B   1F06             00095         BTFSS   PORTB,6
003C   2839             00096         GOTO    one
003D   2819             00097         GOTO    Scan
003E   3012             00099 two     MOVLW   012
003F   00A0             00100         MOVWF   DIGIT
0040   1F06             00101         BTFSS   PORTB,6
0041   283E             00102         GOTO    two
0042   2819             00103         GOTO    Scan
                        00104
0043   3013             00105 three   MOVLW   013
0044   00A0             00106         MOVWF   DIGIT
0045   1F06             00107         BTFSS   PORTB,6
0046   2843             00108         GOTO    three
0047   2819             00109         GOTO    Scan
```

Program 1.1: (*continued*)

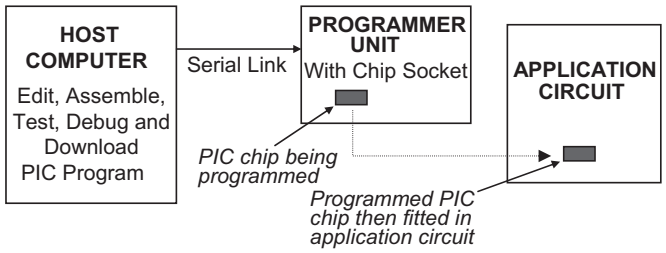
```
                        00110
0048   3014             00111 four    MOVLW   014
0049   00A0             00112         MOVWF   DIGIT
004A   1E86             00113         BTFSS   PORTB,5
004B   2848             00114         GOTO    four
004C   2819             00115         GOTO    Scan
                        00116
004D   3015             00117 five    MOVLW   015
004E   00A0             00118         MOVWF   DIGIT
004F   1E86             00119         BTFSS   PORTB,5
0050   284D             00120         GOTO    five
0051   2819             00121         GOTO    Scan
                        00122
0052   3016             00123 six     MOVLW   016
0053   00A0             00124         MOVWF   DIGIT
0054   1E86             00125         BTFSS   PORTB,5
0055   2852             00126         GOTO    six
0056   2819             00127         GOTO    Scan
                        00128
0057   3017             00129 seven   MOVLW   017
0058   00A0             00130         MOVWF   DIGIT
0059   1E06             00131         BTFSS   PORTB,4
005A   2857             00132         GOTO    seven
005B   2819             00133         GOTO    Scan
                        00134
005C   3018             00135 eight   MOVLW   018
005D   00A0             00136         MOVWF   DIGIT
005E   1E06             00137         BTFSS   PORTB,4
005F   285C             00138         GOTO    eight
0060   2819             00139         GOTO    Scan
                        00140
0061   3019             00141 nine    MOVLW   019
0062   00A0             00142         MOVWF   DIGIT
0063   1E06             00143         BTFSS   PORTB,4
0064   2861             00144         GOTO    nine
0065   2819             00145         GOTO    Scan
                        00146
                        00147         END ; of program
```

Program 1.1: (*continued*)

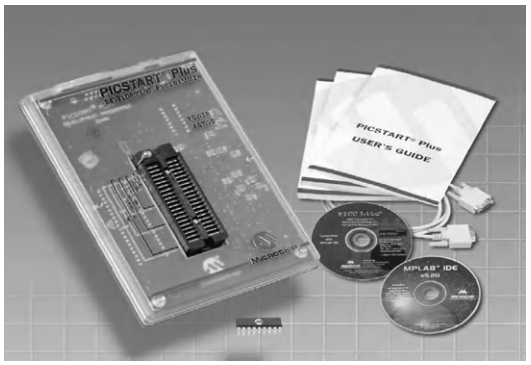
attached to the MCU. Animated inputs and outputs provide instant results, allowing the program to be developed and debugged quickly and easily (see Appendix E). The list file shown contains the source code and machine code, which will be explained in the next chapter.

With suitable development of the software and hardware, the system could be modified to work as a calculator, message display, electronic lock or similar application; for example, more digits could be added to the display. Keyboard scanning and display driving are standard operations for microcontrollers, and the techniques mentioned here to create a working application will be discussed fully in later chapters.

(a)



(b)



(c)

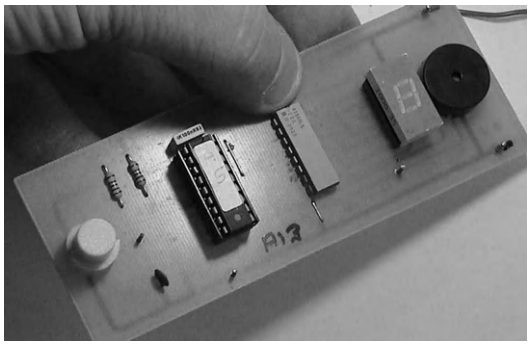


Figure 1.10
Preprogramming a PIC microcontroller: (a) block diagram; (b) programming unit (*courtesy of Microchip™ Technology Inc.*); (c) demo target board

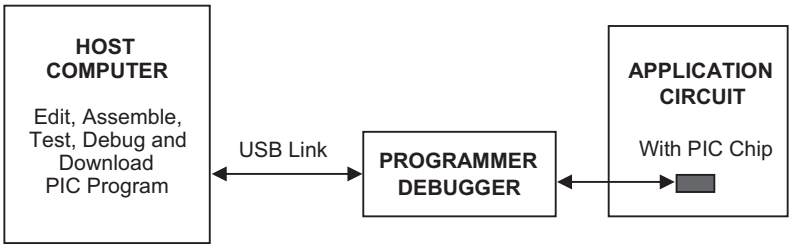
1.4.2. Programming a Microcontroller

For the examples in this book, we will be using PIC chips that have flash ROM program memory, which can be easily erased and reprogrammed. This is very useful when learning,

but also allows the firmware (microcontroller program) to be upgraded in any application, e.g. adding an app to a mobile phone, or upgrading its operating system. It is the same kind of memory used to store the image data in the SD card in the camera, and for general storage in a memory stick.

There are two ways of programming the PIC microcontroller. The preprogramming system is shown in Figure 1.10. The programming interface is the basic PICSTART Plus module, which accepts dual-in line (DIL) pin-out PIC chips up to 40 pins in a zero insertion force socket. The serial connection to the host PC COM port is made via an RS232 lead. This protocol is rather slow, and the COM port connector is not usually fitted to current PCs, so it is being replaced by USB in current programmers.

(a)



(b)

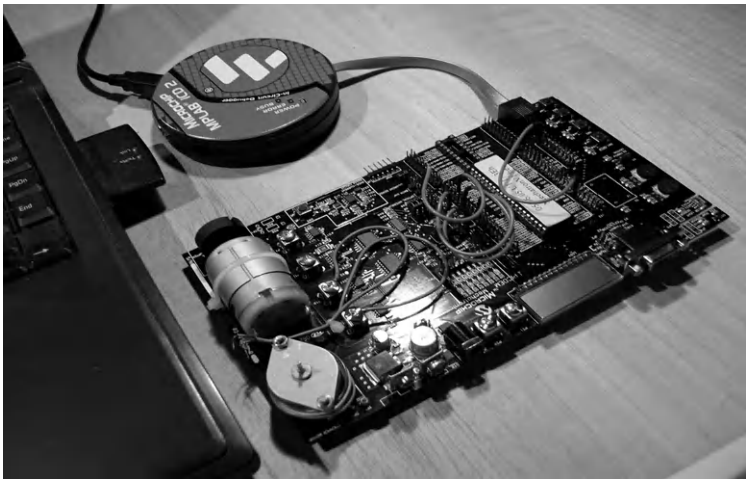


Figure 1.11
In-circuit programming: (a) block diagram; (b) ICSP/ICD programming interface and PIC mechatronics target board

Alternatively, the PIC can be programmed in circuit, that is, after it has been fitted into the finished circuit board. This is known as in-circuit serial programming (ICSP), and the same hardware can also support in-circuit testing and debugging (ICD), as seen in Figure 1.11. The in-circuit programming module is the Microchip ICD2, which connects to the host via USB and to the target system via a six-way RJ-45 connector. In this case, the application board is the PIC Mechatronics demo board, which is used to investigate control of brushed dc motors and stepper motors.

The program is written as a text file and converted (assembled) to machine code (hex) in the host computer, using suitable development system software, usually Microchip MPLAB integrated development environment (IDE). Mistakes in the source code must be corrected before a hex file can be successfully created. The program operation can then be tested in MPLAB and downloaded to the target system.

Electronic computer-aided design (ECAD) software such as Proteus VSM also allows us to simulate the circuit on screen, in order to debug the program before downloading. The complete circuit can then be checked for correct operation, ideally by running the microcontroller in debug (fault-finding) mode, if this is supported by that particular device. All these techniques will be explained later on.

The basic technology for implementing digital systems is described in appendices at the back of this book. If you are not familiar with any of these hardware concepts, please refer to these sections as necessary. Appendix A covers information coding and assembler programs, Appendix B describes the basic electronics of digital systems, and Appendix C show how these work together to provide data input, storage, processing and storage devices.

Questions 1

1. Name at least two PC user input devices, two user output devices and two storage devices. (6)
2. Why is the BIOS ROM needed to start the PC, and why does the start-up take some time? (4)
3. Why are shared bus connections used in a microprocessor system, even though it slows down program execution? (2)
4. State two advantages of the PC hardware modular design. (4)
5. State the differences between ROM and RAM and the significance for operation of the PC and typical MCU. (4)
6. State the function, in one sentence, of the following microprocessor system elements:
   (a) CPU (2)
   (b) ROM (2)
   (c) RAM (2)
   (d) Address bus (2)
   (e) Data bus (2)
   (f) Address decoder (2)

(g) Program counter (2)
(h) Instruction register. (2)

7. Explain the essential differences between a typical microprocessor system and microcontroller, and their applications. (8)
8. Outline the stages in the development of a microcontroller application. (6)

Answers on page 417–18. (Total 50 marks)

Activities 1

1. Open the system folder (control panel, system, device manager) on a PC or laptop and list the hardware features of the system, noting the characteristics of the CPU, memory and all the interfaces installed. Investigate why a 'software driver' is needed for each peripheral device, and report briefly on each, identifying the interface hardware, its function, driver name, version and other relevant information.
2. Under supervision if necessary, carry out the following investigation:
   Disconnect the power supply, remove the cover of the main unit of a desktop PC and identify the main hardware subsystems: power supply, motherboard and disk units. On the motherboard, identify the CPU, RAM modules, expansion slots, and the keyboard, graphics, disk and network interface. Photograph or sketch and identify the system main unit components. Compile an inventory of the system hardware, including relevant information from Activity 1.
3. Run a word processor and study the process of word wrapping, which occurs at the end of each line. Describe the algorithm that determines the word placement, and the significance of the space character in this process. Draw a flowchart to represent this process.
4. Select a typical microcontroller application, such as a mobile phone or coffee machine, write a description of how it works and devise a block diagram of the system, as shown in the digital camera in Figure 1.8.

CHAPTER 2

Microcontroller Operation

Chapter Outline

Chapter Points

- The PIC microcontroller contains a program execution section and a register processing section.
- The program is list of binary machine code instructions stored in flash memory.
- The program counter steps through the program addresses, and the instructions are decoded and executed.
- Data is transferred via port registers, stored in RAM/registers and processed in the ALU.
- Special function registers store control, setup and status information.
- Instructions move or process data, or control the execution sequence.
- The content of the data registers is manipulated as single data words or using register pairs.
- Program jumps can be unconditional or conditional, using bit testing or status bits to determine the sequence.
- Subroutines are distinct program blocks which operate using call, execute and return.

PIC Microcontrollers. DOI: 10.1016/B978-0-08-096911-4.10002-3

In Chapter 1, some basic ideas about microprocessor system operation were introduced. In order to understand the operation of a typical microcontroller (MCU), some knowledge of both the internal hardware arrangement and the instruction set is required, so in this chapter we will look at some basic elements of PIC® microcontroller architecture and essential features of machine code programs.

If necessary, the reader should refer to the appendices for details of number systems and assembler coding (Appendix A), logic circuit devices (Appendix B) and data system operation (Appendix C). These will also allow the PIC microcontroller data sheets, which form the primary technical reference, to be more readily understood. All PIC data sheets can be downloaded from the Microchip website www.microchip.com, by selecting MCUs, 8-bit, PIC 16 Family. A table of all 16 series chips, in numerical order, allows their features to be compared and their PDFs to be downloaded and stored locally for ease of access.

2.1. Microcontroller Architecture

The architecture (internal hardware arrangement) of a complex chip is best represented as a block diagram. This allows the overall operation to be described without having to analyze the internal circuit, which is extremely complex, in detail. PIC data sheets contain a definitive block diagram for each chip. Our starting point is the PIC 16F84A chip, because it has all the basic features but none of the more advanced elements that will be covered later. Also, the model for this chip is provided in the entry-level Proteus VSM microcontroller simulation package. Unfortunately, this chip is now effectively obsolete for new designs and is relatively expensive compared with more recently introduced chips, which actually have more features, such as the 16F690 that we will examine later on.

Simplified versions of the block diagrams from the data sheets will be used to help explain particular aspects of the chip operation. A general block diagram that shows some of the common features of PIC microcontrollers is seen in Figure 2.1. It shows that the MCU can be considered in two parts: the program execution section and the register processing section. Note that the program and data are accessed separately, and do not share the same data bus, as is the case within some processor systems. This arrangement, known as Harvard architecture, increases overall program execution speed. The timing and control block coordinates the operation of the two parts as determined by the program instructions, and responds to external control inputs, such as the reset and interrupts.

The program execution section contains the program memory, instruction register and control logic, which store, decode and execute the program. The register processing section has a block of random access memory (RAM), which starts with the special function registers (SFRs) that are used to control the processor operations, including the port registers, which are used for input and output. The rest of this RAM block provides general purpose registers (GPRs) for

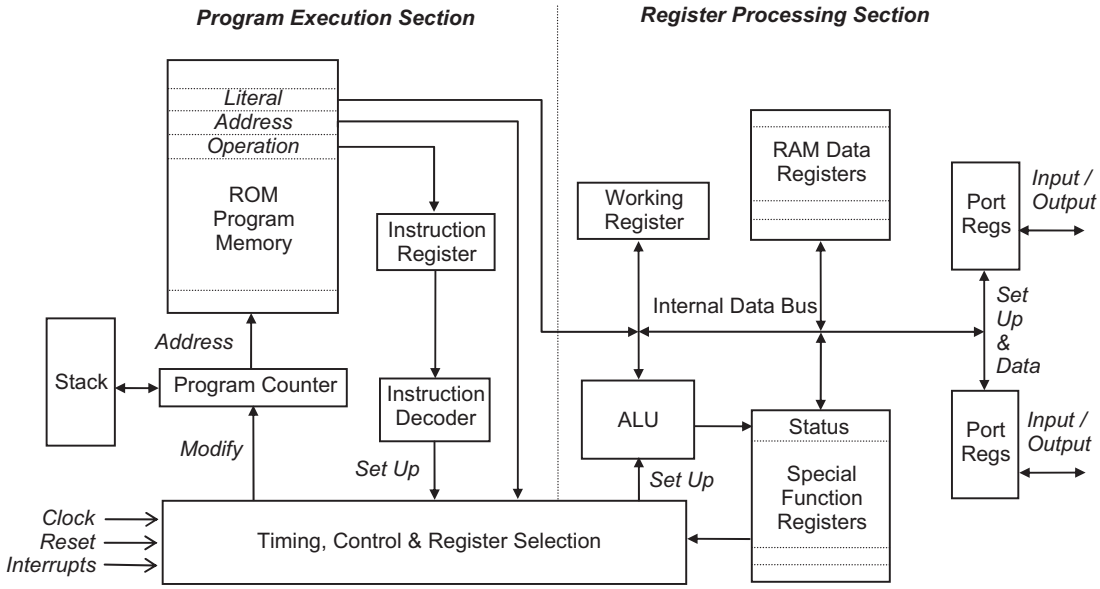


Figure 2.1
General microcontroller block diagram

data storage. The arithmetic and logic unit (ALU) processes the data, e.g. add, subtract and compare.

In some microcontrollers and microprocessors, the main data register is called the accumulator (A), but the name working register (W), used in the PIC system, is a better description. It holds the data that the processor is working on at the current time, and most data has to pass through it. For example, if a data byte is to be transferred from the port register to a RAM data register, it must be moved into W first. The working register works closely with the ALU in the data processing operations. Instructions may operate on W or on the RAM register, which includes the ports and SFRs.

2.1.1. Program Memory

Microcontrollers used for prototyping and short production runs use flash memory to store the program. The program can be downloaded while the chip is in the application circuit (in-circuit programming). Alternatively, the chip is placed in a programming unit attached to the host computer for program downloading, before fitting it in the application board. For longer production runs, preprogrammed chips can be ordered from the manufacturer, which use mask programmed ROM, where the program is incorporated during chip fabrication.

The PIC 16 program consists of a list of 14-bit binary codes, each containing the instruction and operand (data) in one operation code. The program starts at address zero and the

instructions are executed in turn unless a branch instruction or an external 'interrupt' occurs. Usually, the last instruction causes a loop back to repeat the control sequence. The capacity of the program memory block is one of the most significant features of each PIC, varying from 1024 to 8096 instructions in the 16 series. The higher power PIC 18, 24, 32 and dsPIC ranges offer more memory, input/output (I/O) and peripheral features.

2.1.2. Program Counter

The Program Counter (PC) is a register that keeps track of the program sequence, by storing the address of the instruction currently being executed. It is automatically loaded with zero when the chip is powered up or reset. The program counter is file register number 2 in the SFR set. As each instruction is executed, PC is incremented (increased by one) to point to the next instruction. Program jumps are achieved by reloading PC to point to an instruction other than the next in sequence. This new address is provided in the instruction.

Often, it is necessary to jump from address zero to the start of the actual program at a higher address, because special control words must be stored in particular low addresses. Specifically, PIC 16 devices use address 004 to store the 'interrupt vector' (start address of the interrupt routine). In this case, the main program must not be located at address zero; instead, a jump to a higher address should be placed there. An assembler directive is then needed to place the start of the program at a higher address. This problem can be ignored for programs that do not use interrupts, and such simple programs will be located by default at address zero. Interrupts will be explained in more detail later.

Associated with the program counter is the 'stack'. This is a temporary program counter store. When a subroutine is executed (see Chapter 5, Section 5.2.4), the stack register temporarily stores the current address, so that it can be recovered at a later point in the program. It is called a stack because the addresses are restored to the PC in the reverse order to which they were stored, that is, 'last in, first out' (LIFO), like a stack of plates.

2.1.3. Instruction Register and Decoder

To execute an instruction, the processor copies the instruction code from the program memory into the instruction register (IR). It can then be decoded (interpreted) by the instruction decoder, which is a combinational logic block which sets up the processor control lines as required. These control lines are not shown explicitly in the block diagram, as they go to all parts of the chip, and would make it too complicated. In the PIC, the instruction code includes the operand (working data), which may be a literal value or register address. For example, if a literal (a number) given in the instruction is to be loaded into the working register (W), it is placed on an internal data bus and the W register latch enable lines are activated by the timing and control logic. The internal data bus can be seen in the manufacturer's block diagram (Figure 1-1 in the PIC 16F84A data sheet).

2.1.4. Timing and Control

This sequential logic block provides overall control of the chip, sending signals to all parts of the chip to move the data around and carry out logical operations and calculations (see Appendix C). A clock signal is needed to drive the program sequence; it is traditionally derived from an external crystal oscillator, which provides an accurate, fixed frequency signal. More recent chips have an internal oscillator, which saves on external components.

A maximum frequency of operation is always specified; most current PIC 16 chips run at a maximum of 20 MHz, although newer designs reach 32 MHz. All can operate at any frequency below this maximum, down to 0 Hz. This is referred to as a static design: the clock can be stopped and the current MCU status will be retained. It takes four clock cycles to execute one instruction, unless it involves a jump, when it is eight. However, the execution of successive instructions overlaps to double the effective speed (pipelining; see data sheet).

The program starts automatically at program location zero, as long as the reset input is connected high (power-on reset). If required, a push button reset can be connected, which takes this input low for a manual reset. This should not normally be needed, but if there is a bug in the program or another system fault that causes the program to hang (get stuck in loop), a manual reset is useful, particularly during prototyping.

The only other way to stop or redirect a continuous loop is via an interrupt. Interrupts are signals generated externally or internally, which force a change in the sequence of operations. If an interrupt source goes active in the PIC 16, the program will restart at program address 004, where the sequence known as the 'interrupt service routine' (or a jump to it) must be stored. More details are provided in Chapter 6.

2.1.5. Arithmetic and Logic Unit

This is a combinational logic block that takes one or two input binary words and combines them to produce an arithmetic or logical result. In the PIC, it can operate directly on the contents of a register, but if a pair of data bytes is being processed (e.g. added together), one must be in W. The ALU is set up according to the requirements of the instruction being executed by the timing and control block. Typical ALU/register operations are detailed later in this chapter.

2.1.6. Port Registers

Input and output in a microcontroller are achieved by simply reading or writing a port data register. If a binary code is presented to the input pins of the chip by an external device (e.g. a set of switches), the data is latched into the register allocated to that port when it is read in the program. This input data can then be moved (or more accurately, copied) into another register for processing. If a port register is initialized for output, the code moved to its data register is

immediately available at the pins of the chip. It can then be displayed externally, for example, on a set of light-emitting diodes (LEDs).

Each port has a 'data direction' register associated with its data register. This allows each pin to be set individually as an input or output before the data is read from or written to the port data register. A '0' in the data direction register sets the port bit as an output, and a '1' sets it as an input. These port registers are mapped (addressed) as SFRs, starting from register 05 for port A, 06 for port B, and so on in the original PIC 16 specification. In more recently introduced chips (e.g. 16LF1826), which need more registers, the ports start at 0Ch (h is a suffix indicating a hexadecimal number; see Appendix A). The port data direction registers are mapped into a second register bank (bank 1) with addresses starting at 85h for port A, 86h for port B, and so on.

2.1.7. Special Function Registers

These numbered registers provide dedicated program control registers and processor status bits. In the PIC, the program counter, port registers and spare registers are all mapped as part of a block that starts at zero and ends at 0Bh in the 16F84A. For example, the program counter is register number 02. The working register is the only one that is not located in the main register block, and is accessed by specifying it in the instruction.

All processors contain control and status registers whose bits are used individually to set up the processor operating mode, or record significant results from those operations. In the PIC 16, the status register is located at SFR 03. The most frequently used bit is the zero flag. This is internally set to 1 if the result of any operation is zero in the destination register (the register that receives the result). The carry (C) flag is another bit in the status register; it is set if the result of an arithmetic operation produces a carry-out of the most significant bit of the destination register, that is, the register overflows.

The status register bits are often used to control program sequence by conditional branching. Alternate sections of code are executed depending on the condition of the status flag. In the PIC instruction set, this is achieved by an instruction that tests the bit and skips the next instruction if it is 0 or 1. The bit test and skip instruction is generally followed by a jump instruction to take the execution point to another part of the program, or not, as the case may be. This will be explained more fully in the next section.

The most important SFRs in the 16F84A are listed in Table 2.1. The RAM is divided into blocks, where bank 0 contains registers 00h to 7Fh, bank 1 registers 80h to FFh and so on, that is, 128 registers per bank. The SFRs are located at the bottom (lowest addresses) of each register bank (but the data sheet RAM block diagram shows them at the top). Some registers are duplicated in different banks (e.g. program counter, PCL), while others are unique (e.g. data direction register, TRISA). More complex chips that need more registers have extra banks of RAM. For example, the 16LF1826 has eight. The exact arrangement for each chip

Table 2.1: Selected PIC 16 special function registers

File Register Address	Name	Function
Bank 0		
01	TMR0	Timer/Counter allows external and internal clock pulses to be counted
02	PCL	Program Counter stores the current execution address
03	STATUS	Individual bits record results and control operational options
05	PORTA	Bidirectional input and output bits
06	PORTB	Bidirectional input and output bits
0B	INTCON	Interrupt control bits
Bank 1		
85	TRISA	Port A data direction bits
86	TRISB	Port B data direction bits

must be carefully studied in the data sheet before attempting any programming for that chip. A standard header file is available for all chips which assigns the data sheet labels to the SFRs.

2.2. Program Operations

We can see in Appendix A that a machine code program consists of a list of binary codes stored in the microcontroller memory. They are decoded in sequence by the processor block, which generates control signals that set up the microcontroller to carry out the instruction. Typical operations are:

- Load a register with a given number.
- Copy data from one register to another.
- Carry out an arithmetic or logic operation on a data word.
- Carry out an arithmetic or logic operation on a pair of data words.
- Jump to an alternative point in the program.
- Test a bit or word and jump, or not, depending on the result of the test.
- Jump to a subroutine, and return later to the same point.
- Carry out a special control operation.

The machine code program must be made up only from those binary codes that the instruction decoder will recognize. These codes can be read off from the instruction set given in the data sheet. When computers were first developed, this was exactly how the program was entered, in binary, using a set of switches. This is obviously time consuming and inefficient, and it was soon realized that it would be useful to have a software tool that would generate the machine code automatically from a program written in a more user-friendly form. Assembly language programming was therefore developed, when computer hardware had moved on enough to make it practicable.

Assembly language allows the program to be written using mnemonic (memorable) codes. Each processor has its own set of instruction codes and corresponding mnemonics. For example, a commonly used instruction mnemonic in PIC programs is 'MOVWF', which means move (actually copy) the contents of the working register (W) to a file register that is specified as the operand. The destination register is specified by number (file register address), such as 0Ch (the first general purpose register in the PIC 16F84A). The complete instruction is:

```
MOVWF    0C
```

This is converted by the assembler software (MPASM.EXE) to the hexadecimal code specified in the instruction set:

```
008C
```

The binary code stored in program memory is therefore

```
0000001 0001100
```

Note that the instruction is 14 bits in total, with the operand represented, in this case, by the last seven bits, and the operation code the first seven. The op-code bits are used by the instruction decoder to select the correct source and destination registers (W and SFR 0C) prior to the operation. A following clock edge will then trigger the copy operation on the internal data bus.

There are two main types of instruction, with four identifiable subgroups within each:

1. Data processing operations:
   - `MOVE:` copy data between registers
   - `REGISTER:` manipulate data in a single register
   - `ARITHMETIC:` combine register pairs arithmetically
   - `LOGIC:` combine register pairs logically.
2. Program sequence control operations:
   - `UNCONDITIONAL JUMP:` jump to a specified destination
   - `CONDITIONAL JUMP:` jump, or not, depending on a test
   - `CALL:` jump to a subroutine and return
   - `CONTROL:` miscellaneous operations.

Together, these types of operations allow inputs to be read and processed, and the results stored or output, or used to determine the subsequent program sequence.

A complete assembly language example is shown in the final section of Chapter 1. Program 1.1 is the list file KEY690.LST, whose function is to read a keypad and display the inputs. The source code mnemonics are on the right, with the machine code in column 2 and the memory location where each instruction is stored in column 1.

2.2.1. Single Register Operations

The processor operates on 8-bit data stored in RAM registers and W. The data can originate in three ways:

- A literal (numerical value) provided in the program
- An input via a port data register
- The result of a previous operation.

This data is processed using the set of instructions defined for that processor. Table 2.2 shows a typical set of operations that can be applied to a single register. The same binary number is shown before processing, and then after the operation has been applied to the register.

As an example of how these operations are specified in mnemonic form in the program, the hex and assembler code to increment a PIC register is:

```
0A86            INCF        06
```

Register number 06 happens to be port B data register, so the effect of this instruction can be seen immediately at I/O pins of the chip. The corresponding machine code instruction is 0A86h, or 00 1010 1000 0110 in binary (14 bits). As you can see, it is easier to recognize the mnemonic form. Bit 7 of the instruction code is significant in that it determines the destination of the result. The default is '1', which causes the result to be left in the RAM register. '0' places it in W, which helps to reduce the number of move instructions required.

An example of a single register operation appears at line 33 in the keypad program, CLRF PORTC, which sets all the output bits connected to the display to zero, switching it off.

Table 2.2: Single register operations

Operation	Before		After	Comment
CLEAR	0101 1101	→	0000 0000	Reset all bits to zero
INCREMENT	0101 1101	→	0101 1110	Increase value by one
DECREMENT	0101 1101	→	0101 1100	Decrease value by one
COMPLEMENT	0101 1101	→	1010 0010	Invert all bits
ROTATE LEFT	0101 1101	→	1011 1010	Move all bits left by one place*
ROTATE RIGHT	0101 1101	→	1010 1110	Move all bits right by one place*
CLEAR BIT	0101 1101	→	0101 0101	Clear bit (3) to 0
SET BIT	0101 1101	→	1101 1101	Set bit (7) to 1

*Carry bit included.

2.2.2. Register Pair Operations

Table 2.3 shows basic operations that can be applied to pairs of registers. The result is retained in one of them, the destination register. The data to be combined with the contents of the destination register is obtained from the source register, typically W or a literal (number supplied in the instruction). The source register contents generally remains unchanged after the operation.

The meaning of each type of instruction is explained below, with examples from the PIC instruction set. As noted above, there is an option to store the result in W, the working register, if that is the source. Note also that the PIC 16 instruction set does not provide moves directly between file registers; all data moves are via W.

Status bits are modified by specific register operations. The zero flag (Z) is invariably affected by arithmetic and logic instructions, and the carry flag (C) by arithmetic ones, including rotate. The effect on the source, destination and status registers of each instruction is specified in the instruction set, and this needs to be studied thoroughly before attempting to write assembler programs. The binary, hex and assembler code is given, together with the flag(s) affected, if any, in the following examples.

Table 2.3: Operations on register pairs

Operation		Before		After	Comment
MOVE					
	Source	0001 1100		0001 1100	Copy operation
	Destination	xxxx xxxx	→	0101 1100	Overwrite destination with source
ADD					
	Source	0001 1100		0001 1100	Arithmetic operation
	Destination	0001 0010	→	0010 1110	Add source to destination
SUB					
	Source	0001 0010		0001 0010	Arithmetic operation
	Destination	0101 1100	→	0100 1010	Subtract source from destination
AND					
	Source	0001 0010		0001 0010	Logical operation
	Destination	0101 1100	→	0001 0000	AND source & destination bits
OR					
	Source	0001 0010		0001 0010	Logical operation
	Destination	0101 1100	→	0101 1110	OR source & destination bits
XOR					
	Source	0001 0010		0001 0010	Logical operation
	Destination	0101 1100	→	0100 1110	Exclusive OR source & destination bits

Move

The most commonly used instruction in any program simply moves data from one register to another. It is actually a copy operation, as the data in the source register remains unchanged until overwritten.

```
00  1000  0000  1100        080C          MOVF  0C,W          (Z)
```

This instruction moves the contents of register 0Ch (12_{10}) into the working register. Note that bit 7, selecting the destination, is '0' for W, which has to be specified in the instruction.

```
00  0000  1000  1100        008C          MOVWF  0C
```

This instruction is the reverse move, from W to register 0Ch. Bit 7 is now '1', and the zero flag is not affected, even if the data is zero.

An example of the move instruction is seen at line 46 in the keypad program, MOVWF PORTC, which outputs a binary code to operate the display.

Arithmetic

Add and subtract are the basic arithmetic operations, carried out on binary numbers. Some processors also provide multiply and divide in their instruction set, but these can be created if necessary by using shift, add and subtract operations.

```
00  0111  1000  1100        078C        ADDWF    0C         (C,Z)
```

This instruction adds the contents of W to register 0C. The carry flag will store a carry-out of the most significant bit (MSB), if the result is greater than the maximum value, FFh (255_{10}), with the remainder left in the register. For example, if we add the decimal numbers 200 and 100, the result will be 300. The remainder will be $300 - 256 = 44$, with the carry flag representing 256 (result = 1 0010 1100 in binary). If the sum is exactly 256_{10}, the register result will be zero (Z flag set) and carry-out generated (C flag set).

The carry flag is also included when subtracting, so that numbers up to 511_{10} can be operated on. Rotate can be used to halve and double binary numbers, while increment and decrement are also available.

Logic

Logical operations act on the corresponding pairs of bits in a literal or source register, and destination. The result is normally retained in the destination, leaving the source unchanged. The result in each bit position is obtained as if the bits had been fed through the equivalent logical gate (see Appendix B).

```
11  1001  0000  0001        3901         ANDLW     01         (Z)
```

This instruction carries out an AND operation on the corresponding pairs of bits in the binary number in W and the binary number 00000001, leaving the result in W. In this

example, the result is zero if the LSB in W is zero, so it forms a check on the state of that bit alone.

This type of operation can be used for bit testing if the processor does not provide a specific instruction, or masking to select a portion of the source data. The AND operation gives a result 1 if BOTH source bits are 1. The OR operation gives the result 1 if EITHER bit is 1. XOR gives result 1 if ONE of the bits is 1. This covers all the options for logical processing.

An example of a logic instruction is seen at line 45 in the keypad program, ANDLW 0F0, which masks one of the digits for output to the display.

2.2.3. Program Control

As we have already seen, the microcontroller program is a list of binary codes in the program memory, which are executed in sequence. The sequence is controlled by the program counter, (PC). Most of the time, PC is simply incremented by one to proceed to the next instruction. However, if a program jump (branch) is needed, PC must be modified, that is, the address of the next instruction required loaded into PC, replacing the existing value.

The PC is cleared to zero when the chip is reset or powered up for the first time, so program execution starts at address 0000. The clock signal then drives the execution sequence forward. During the execution cycle of the first instruction, the PC is incremented to 0001, so that the processor is ready to execute the next instruction. This process is repeated unless there is a jump instruction.

The jump instructions must have a destination address as the operand. This can be given as a numerical address, but this would mean that the instructions would have to be counted up by the programmer to work out this address. So, as we can see in the program examples, a destination address is usually specified in the program source code by using a recognizable label, such as ‘again’, ‘start’ or ‘wait’. The assembler program then replaces the label with the actual address when the assembler code is converted to machine code.

Program sequence control operations are illustrated in Figures 2.2–2.4. The diagrams show the program memory from address zero, with different types of jump instruction at address 0002.

Jump

The unconditional jump (Figure 2.2) forces a jump to another point in the program every time it is executed. This is carried out by replacing the contents of the program counter with the address of the destination instruction, in this case, 005. Execution then continues from the new address. Note that the code for GOTO is 28 combined with the destination address 05, giving the instruction code 2805h.

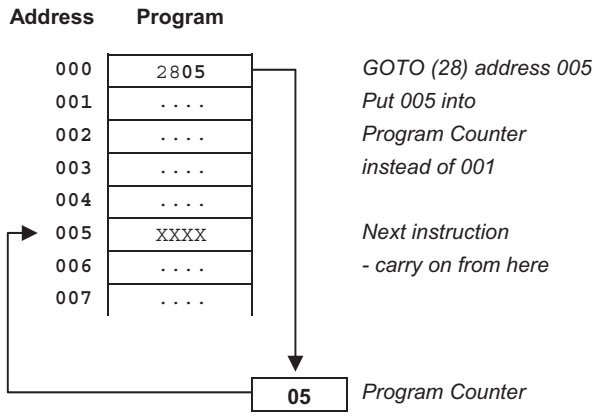


Figure 2.2
Unconditional jump

This example shows the sequence when jumping over the interrupt vector location 004. The unconditional jump is also frequently used at the very end of a program to go back to the beginning of the sequence, and keep repeating it, illustrated below as a program outline:

```
                Initialize
                ..................
start           first instruction
                ..................
                ..................
                GOTO start
```

The label 'start' is placed in the first column of the program code, to differentiate it from the instruction mnemonics, which must be placed in the second column, as we will see. The spelling of the label and its reference must match exactly, including upper and lower case letters. The label is replaced by the corresponding address by the assembler when creating the machine code for the GOTO instruction.

An example of an unconditional jump in the keypad program can be seen at line 50, GOTO Next, where Next is the label assigned to line 37.

Conditional Jump

The conditional jump instruction is required for making decisions in the program. Instructions to change the program sequence depending on, for instance, the result of a calculation or a test on an input, are an essential feature of any microprocessor instruction set.

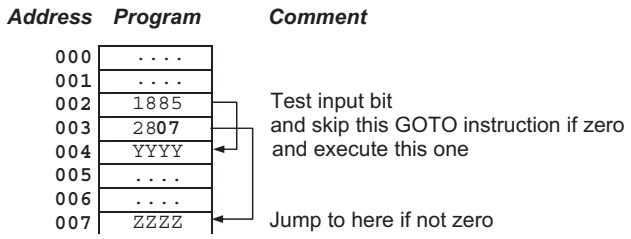


Figure 2.3
Conditional jump

In Figure 2.3, the code 1885 tests an input bit of the PIC and skips the next instruction if it is zero. Instruction YYYY (representing any valid instruction code) is then executed. If the input bit is high, the instruction 2807 is executed, which causes a jump to address 007, and instruction ZZZZ is executed next. This is called Bit Test and Skip, and is the way that conditional branches are achieved in the PIC.

In PIC assembly language, this program fragment looks like this:

```
            ....
            ....
            BTFSC     05,1          ; Test bit 1 of file register 5
            GOTO     dest1          ; Execute this jump if bit = 1
            ....                    ; otherwise carry on from here
            ....
            ....
dest1       ....                    ; branch destination
```

The PIC is designed with a minimal number of instructions, so the conditional branch has to be made up from two simpler instructions. The first instruction tests a bit in a register and then skips (misses out) the next instruction, or not, depending on the result. This next instruction is usually a jump instruction (GOTO or CALL). Thus, program execution continues either at the instruction following the jump, if the jump is skipped, or at the jump destination.

The program outline of a conditional jump used in a delay routine, would look like this:

```
                Allocate 'Count' register
                ....
                ....
                Load 'Count' register with literal XX
     Again      Decrement 'Count' register
                Test 'Count' register for zero
                If not zero yet, jump to label 'Again'
                When zero, execute this next instruction
                ....
```

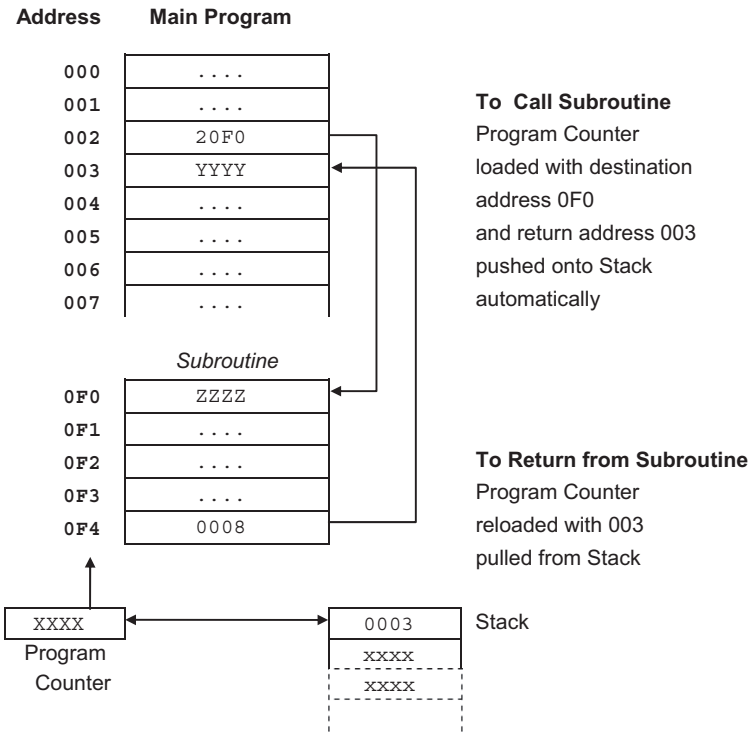


Figure 2.4
Subroutine call

This software timing loop simply causes a time delay in the program, which is useful, for instance, for outputting signals at specific intervals. A register is used as a down counter by loading it with a number, XX, and decrementing it repeatedly until it is zero. A test instruction then detects that the zero flag has gone active, and the loop is terminated. Each instruction takes a known time to execute, therefore the delay can be predicted.

An example of a conditional jump can be seen at lines 39 and 40 in the keypad program, where BTFSS DIGIT,4 is followed by GOTO Next, so that the jump back is only executed if bit 4 of the register labelled DIGIT (GPR 20) is zero.

Subroutine

Subroutines are used to carry out discrete program functions. They allow programs to be written in manageable, self-contained blocks, which are then executed as required, often more than once per program cycle. The instruction CALL is used to jump to a subroutine, which must be terminated with the instruction RETURN.

CALL has the address of the first instruction in the subroutine as its operand. When the CALL instruction is decoded, the destination address is copied to the PC, as in the GOTO instruction. In

addition, the address of the next instruction in the main program is saved in the 'stack', a special set of registers. The return address is 'pushed' onto the stack when the subroutine is called, and 'pulled' back into the program counter at the end of the routine, when the RETURN instruction is executed. These addresses are automatically stored in order and retrieved in reverse order.

In Figure 2.4, the subroutine is a block of code whose start address has been defined by label as 0F0. The CALL instruction at address 002 contains the destination address as its operand. When this instruction is encountered, the processor carries out the jump by copying the destination address (F0h) into the program counter. At the same time, the address of the next instruction in the main program (003) is pushed onto the stack, so that the program can come back to the original point after the subroutine has been executed.

One advantage of using subroutines is that the block of code can be used more than once in the program, but only needs to be typed in once. A delay loop can be written as a subroutine. In a program to generate an output pulse train, it can be 'called' twice within a loop, which sets an output high, delays, sets the output low, and delays again before repeating the whole process. The delay subroutine can be written such that it takes its delay count from W each time it is called, making it a variable delay routine, as shown in the outline below:

```
;  Program DELTWICE ******************************

           Allocate  'Count'  Register
           ....

           ....
           Load  'Count'  register  with  value XX
           CALL  'delay'
           Next  Instruction
           ....
           ....
           Load  'Count'  register  with  value  YY
           CALL  'delay'
           Next  Instruction
           ....
           ....
           END  of  Program

;  Subroutine DELAY ******************************

delay      Decrement 'Count' register
           Test 'Count' register for zero
           If not zero, jump to label 'delay'
            RETURN from subroutine

;   End of code ***********************************
```

An example of a subroutine call is included in the keypad program at line 38, CALL Scan, which causes a jump to the subroutine starting at line 54. RETURN is encountered at line 83, when the execution continues from line 39.

A simple application will be developed in the next chapter to illustrate the basic principles of assembly language programming.

Questions 2

1. Outline the sequence of program execution in a microcontroller, describing the role of the program memory, program counter, instruction register, instruction decoder, and timing and control block. (5)
2. A register is loaded with the binary code 01101010. The carry bit is set to zero. State the contents of the register after the following operations on this data (refer to PIC MCU data sheet for exact effects):
   (a) clear, (b) increment, (c) decrement, (d) complement, (e) rotate right, (f) shift left, (g) clear bit 5, (h) set bit 0. (8)
3. A source register is loaded with the binary code 01001011, and a destination register loaded with 01100010. State the contents of the destination register after the following operations: (a) MOVE, (b) ADD, (c) AND, (d) OR, (e) XOR. (5)
4. In a microcontroller program, a subroutine starts at address 016F and ends with a 'return' instruction at address 0172. A 'call subroutine' instruction is located at address 02F3. Assuming that the microcontroller has one complete instruction in each address, list the changes in the contents of the program counter and stack between the time of execution of the instruction before the call and the instruction following the call. Indicate an unknown value as XXXX. (5)
5. Write a program outline for the process by which two numbers, say 4 and 3, could be multiplied by successive addition. Use the register instructions Clear, Move, Add, Decrement, Test for Zero and Jump if Zero to Label. Load a register with zero, and add 4 to it three times by using a counter initially loaded with 3 and decremented to zero to control the loop. (7)

Answers on page 417 (Total 30 marks)

Activities 2

Download the PIC 16F84A data sheet from www.microchip.com.

1. Study the PIC 16F84A block diagram (data sheet Figure 1-1), and identify the features described in Section 2.1. Note the separate internal instruction and data buses, and summarize the function of each block. Describe how data is moved between registers and memory, and the function of the multiplexers (refer to Appendix C).
2. Study the PIC Instruction Set (data sheet Table 7-2). Note the format of the binary code for each instruction, and identify the meaning of the symbols f, b, k, d, x, C, DC and Z. Explain why some instructions take two cycles.

3. Study the list file generated for program BIN4 shown in Figure 4-4, noting the machine code at the lower left. The program memory addresses from 0000 to 000F appear in column 1, and the machine code instructions appear in column 2. Refer to the instruction set in the PIC 16F84A data sheet, and analyze the program by deducing the code for each instruction and operand, identifying SFR and GPR addresses, register bits, destination bit, address labels and literal values as appropriate. Complete the table below (for all addresses from 0004 through 000F), analyzing each instruction — 0000 to 0003 have been completed as an example:

Hex Address	Hex Inst.	Binary Inst. (14 bits)	Inst. Bits	Operand Bits	Operand Type	Instruction Mnemonic
0000	3000	11 0000 0000 0000	11 00	0000 0000	Literal 00	MOVLW 00
0001	0066	Do not include	—	—	—	TRIS 06
0002	2807	10 1000 0000 0111	10 1	000 0000 0111	Address label 0007	GOTO 0007
0003	008C	00 0000 1000 1100	00 0000 1	000 1100	File address 06	MOVWF 06
0004						
…						
000F	(last instruction)					

CHAPTER 3

A Simple PIC Application

Chapter Outline

Chapter Points

- A block diagram can be used to outline the hardware for an application.
- The PIC 16 chip has 14-bit instructions, containing both the operation code and operand.
- The program is written using assembler mnemonics and labels.
- Layout and comments are used to document the program functions.
- The program is converted into machine code instructions comprising op-codes and operands.
- The program is downloaded to the chip as a hex file.
- The PIC program is stored in flash ROM at addresses from 000.
- The instructions are decoded and executed by the processor control logic.
- The CPU registers and the execution sequence are modified according to the program instructions.

PIC Microcontrollers. DOI: 10.1016/B978-0-08-096911-4.10003-5

We will now develop a minimal machine code program for the PIC®, avoiding complicating factors as far as possible. A simplified internal architecture will be used to explain the execution of the program. Since the core architecture and programming methods are similar for all PIC microcontrollers (MCUs), this serves as an introduction to the whole PIC range, specifically the 16 series.

The specification for the application is as follows:

> *The circuit should output a binary count to eight LEDs, under the control of two push-button inputs. One input will start the output sequence when pressed. The sequence will stop when the button is released, retaining the current value on the display. The other input will clear the output (all LEDs off), allowing the count to resume from zero.*

3.1. Hardware Design

We need a microcontroller that will provide two inputs and eight outputs, which will drive the light-emitting diodes (LEDs) without additional interfacing, and has reprogrammable flash memory to allow the program to be developed in stages. An accurate clock is not required, so a crystal oscillator is not necessary. The PIC 16F84A meets these requirements; it is a basic device, so we will not be distracted by unused features. Later, we can replace it with a more recent processor, such as the PIC 16F690. The 16F84A should not be used for new designs.

3.1.1. PIC 16F84A Pin-Out

The PIC 16F84A microcontroller is supplied in an 18-pin dual in-line (DIL) chip. The pin labeling, taken from the data sheet (download from www.microchip.com), is shown in Figure 3.1. Some of the pins have dual functions, which will be discussed later.

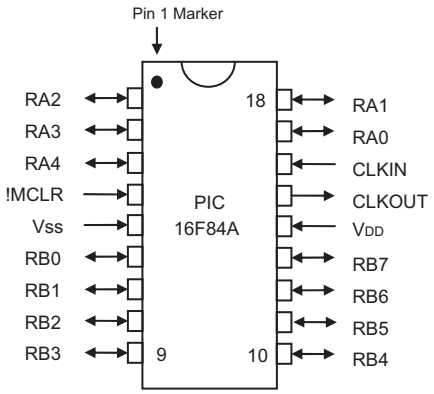


Figure 3.1
Pin-out of PIC 16F84A

The chip has two ports, A and B, consisting of five and eight pins, respectively. The port pins allow data to be input and output as digital signals, at the same voltage levels as the supply, which is connected between V_{DD} and V_{SS}, nominally 5 V. CLKIN and CLKOUT are used to connect clock circuit components, and the chip then generates a fixed frequency clock signal that drives all its operations along. !MCLR (NOT Master CLeaR) is a reset input, which can be used to restart the program. Note that the active low operation of this input is indicated by a bar over the pin label in the data sheet, or an exclamation mark here. In simple applications, this input does not need to be used, but it MUST be connected to the positive supply rail to allow the chip to run. If you construct a circuit and it does not work, check this point. A summary of the pin functions is provided in Table 3.1.

Port B has eight pins, so we will assign these pins to the LEDs and initialize them as outputs. Port A has five pins, two of which can be used for the input switches. A resistor and capacitor will be connected to the CLKIN pin to control the clock frequency. In this chip, an external crystal can be used for a more precise clock frequency, and in many current chips, an internal oscillator is provided, which means that no external clock components are needed.

3.1.2. BIN Hardware Block Diagram

The hardware arrangement can be represented in simplified form as a block diagram (Figure 3.2). This is not really necessary for such a trivial circuit, but is a useful system design technique for more complex applications. The main parts of the hardware and relevant inputs

Table 3.1: PIC 16F84A pins arranged by function

Pin	Label	Function	Comment
14	V_{DD}	Positive supply	+5 V nominal, 2 V to 5.5 V allowed
5	V_{SS}	Ground supply	0 V
4	!MCLR	Master clear	Active low reset input
16	CLKIN	Clock input	Connect RC clock components to 16
15	CLKOUT	Clock output	Connect crystal oscillator to 15 and 16
17	RA0	Port A, bit 0	Bidirectional input/output
18	RA1	Port A, bit 1	Bidirectional input/output
1	RA2	Port A, bit 2	Bidirectional input/output
2	RA3	Port A, bit 3	Bidirectional input/output
3	RA4	Port A, bit 4	Bidirectional input/output + TMR0 input
6	RB0	Port B, bit 0	Bidirectional input/output + interrupt input
7	RB1	Port B, bit 1	Bidirectional input/output
8	RB2	Port B, bit 2	Bidirectional input/output
9	RB3	Port B, bit 3	Bidirectional input/output
10	RB4	Port B, bit 4	Bidirectional input/output + interrupt input
11	RB5	Port B, bit 5	Bidirectional input/output + interrupt input
12	RB6	Port B, bit 6	Bidirectional input/output + interrupt input
13	RB7	Port B, bit 7	Bidirectional input/output + interrupt input

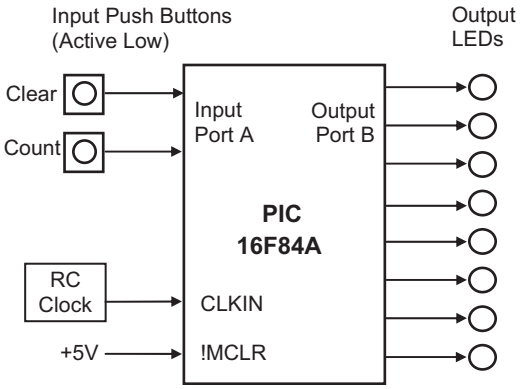


Figure 3.2
Block diagram of BIN hardware

and outputs should be identified, together with the direction of signal flow. The type of signal can be indicated, e.g. parallel or serial data, or analogue waveform. The power connections need not be shown; it is assumed that suitable supplies are available for the active components. The idea is to outline the basic hardware arrangement without having to design the circuit in detail at this stage.

Port A (5 bits) and port B (8 bits) give access to the data registers of the ports, the pins being labelled RA0 to RA4, and RB0 to RB7, respectively. The two push-button switches will be connected to RA0 and RA1, and a set of LEDs connected to RB0 to RB7. The switches will later be used to control the output sequence. However, these inputs will not be used in the first program, BIN1. The connections required are shown in Table 3.2.

The block diagram can now be converted into a circuit diagram. A drawing is created using electronic schematic capture software such as Proteus VSM (ISIS) or ORCAD. In Proteus, the design file can then be used to test the circuit by interactive simulation. When finalized, it can be converted into a printed circuit board and the hardware produced. The schematic can be inserted into other documentation as required, or printed separately.

The schematic for the BIN circuit design created in ISIS (BIN.DSN) is shown in Figure 3.3. This is available on the support website www.picmicros.org.uk and can be used to simulate the circuit operation described below. The process for editing the schematic and simulating the circuit operation is described in Appendix E. The operation of the chip itself can be simulated in MPLAB, the Microchip development system.

3.1.3. BIN Circuit Operation

Active low switch circuits, consisting of normally open push buttons and pull-up resistors, are connected to the control inputs; the resistors ensure that the inputs are high when the buttons

Table 3.2: PIC 16F84A pin allocation for BIN application

Pin	Connection
V_{SS}	0 V
V_{DD}	+5 V
!MCLR	+5 V
CLKIN	CR clock circuit
CLKOUT	Not connected (n/c)
RA0	Reset switch
RA1	Count switch
RA2	n/c
RA3	n/c
RA4	n/c
RB0	LED bit 0
RB1	LED bit 1
RB2	LED bit 2
RB3	LED bit 3
RB4	LED bit 4
RB5	LED bit 5
RB6	LED bit 6
RB7	LED bit 7

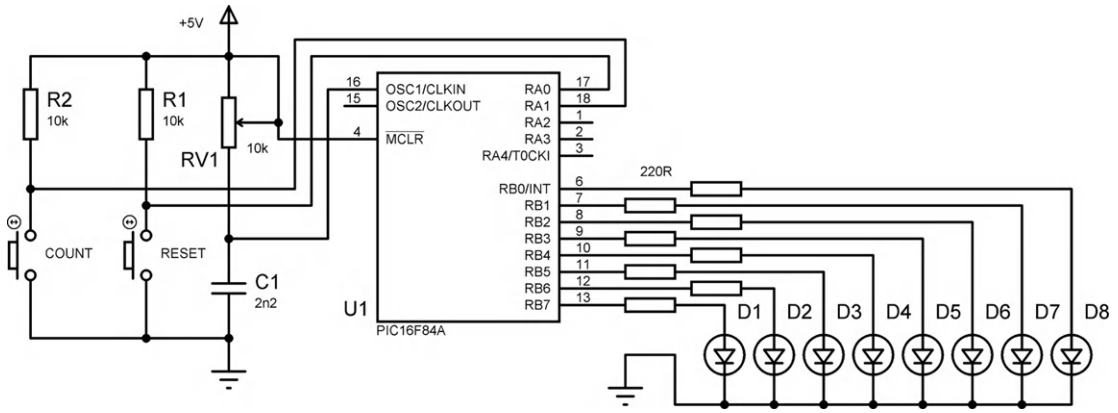


Figure 3.3
BIN ISIS circuit schematic

are not pressed. The outputs are connected to LEDs in series with current-limiting resistors. The PIC outputs are capable of supplying enough current (up to 20 mA) to drive LEDs directly, making the circuit relatively simple. The external clock circuit consists of a capacitor (C) and resistor (R) in series; the value of C and R multiplied together will determine the chip clock

rate. The resistance in this circuit has been made variable, and the values shown should allow the clock frequency to be adjusted to 100 kHz. The reset input (!MCLR) must be connected to the positive supply (+5 V). Other unused pins can be left open circuit, and unused input/output (I/O) pins will default to inputs.

3.2. Program Execution

Microcontroller circuits will not function without a program in the chip; this is created using the PIC development system software on a PC, and downloaded via a serial data link. This process has already been outlined, and will be described in more detail later, so for now we will assume that the program is already in memory.

Figure 3.4 is a block diagram showing a simplified program execution model for the PIC 16F84A. The main elements are the program memory, decoder, working register and file registers. The binary program, in hexadecimal, is stored in the program memory. The instructions are decoded one at a time by the instruction decoder, and the required operations set up in the registers by the control logic. The file registers are numbered from 00 to 4F, with the first 12 registers (00 to 0B) being reserved for specific purposes. These are called the special function registers (SFRs). The rest may be used for temporary data storage; these are called the general purpose registers (GPRs). Only selected registers are shown in this diagram. All addresses and register contents are in hexadecimal. Appendix A explains hexadecimal numbers.

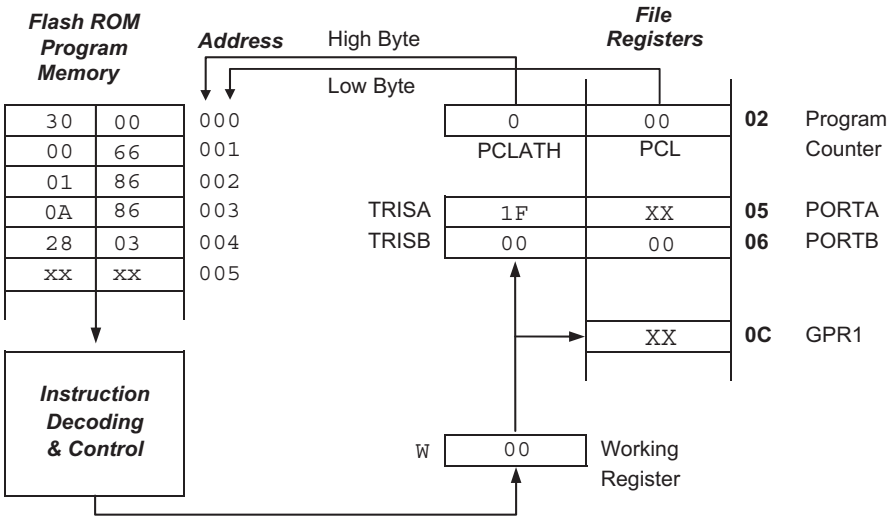


Figure 3.4
PIC 16F84A simple program execution model

3.2.1. Program Memory

The program memory is a block of flash read-only memory (ROM), which means it is non-volatile, but can be reprogrammed. The program created in the host computer is downloaded via port register pins RB6 and RB7. The methods for doing this will be described in more detail in Chapter 4, as will the assembler programming language required to create the program code.

The 14-bit codes are loaded into memory starting at address 000. When the chip is powered up, the program counter resets automatically to 000, and the first instruction is fetched from this address, copied to the instruction register in the control block, decoded and executed. The file registers are modified accordingly, and the resulting output seen at the ports.

3.2.2. Program Counter, PCL: File Register 02

The program counter keeps track of the program execution by holding the address of the current instruction. It is automatically incremented to point to the next instruction during the execution cycle. If there is a jump in the program, the program counter is modified by the jump instruction (e.g. the last one in this program), so that it then points to the required jump destination address. PCLATH stands for program counter latch high. This stores the most significant two bits of the 10-bit program counter, which also cannot be accessed directly.

3.2.3. Working Register, W

This is the main data register (8 bits), used for holding the data that is currently being worked on. It is separate from the file register set and is therefore referred to as W in the PIC program. Literals (values given in the program) must be loaded into W before being moved to another register or used in a calculation. Most data movements have to be via W, in two stages, since direct moves between file registers are not available in the basic PIC instruction set.

3.2.4. Port B Data Register, PORTB: File Register 06

The 8 bits stored in the port B data register will appear on the LEDs connected to pins RB0−RB7, if the port bits are initialized as outputs. The data direction for each pin is determined by placing a data direction code in the register TRISB. A ‘0’ in TRISB sets the corresponding pin in the port register as an output (0 = output). A ‘1’ sets it to input (1 = input). In this case, 00000000 (binary) will be placed in TRISB to set all bits as outputs, but any combination of inputs and outputs can be used.

3.2.5. Port A Data Register, PORTA: File Register 05

The least significant five bits of File Register 05 are connected to pins RA0—RA4, the other three being unused. Inputs RA0 and RA1 will be used later to read the push buttons. If not initialized as outputs, the PIC I/O pins automatically become inputs, i.e. TRISA = xxx11111. We will use this default setting for port A, so this port does not have to be explicitly initialized. The state of these inputs will have no effect unless the program actually uses them; the first program BIN1 will not use them.

3.2.6. General Purpose Register 1, GPR1: File Register 0C

The first GPR will be used later in a timing loop. It is the first of a block of 68 such registers, numbered 0C to 4F in the '84A chip. They may be allocated by the programmer as required for temporary data storage, counting and so on.

3.2.7. Bank 1 Registers

The main registers such as the program counter and port data registers are in a random access memory (RAM) block called register bank 0, while TRISA, TRISB and PCLATH are in a separate block, bank 1. Bank 0 can be directly addressed, meaning that data can be moved into it using a simple 'move' instruction.

Unfortunately, this is not the case with bank 1. There are two ways to write to these registers. The first way is a simple method, which we will use initially; it requires the required 8-bit code to be loaded into W first, and then moved into the bank 1 register using the TRIS instruction. Later, we will use the recommended method, using bank selection, but this is a little more complicated. TRIS does not now appear in the main instruction set, but continues to be recognized by the PIC assembler.

3.3. Program BIN1

The simple program called BIN1 is shown as Program 3.1. It consists of a list of 14-bit binary machine code instructions, represented as four-digit hex numbers (see Chapter 2). Bits 14 and 15 are assumed to be zero, so the codes are represented by hex numbers in the range 0000 to 3FFF. The program is stored at hex addresses 0000 to 0004 (five instructions) in program memory.

3.3.1. Program Analysis

The program instructions must be related to the PIC internal architecture, as outlined in Chapter 2, and specified in the data sheet. The instruction set in the data sheet explains the significance of each bit in each instruction.

Memory address	*Machine code instruction*	*Meaning*
0000	**3000**	Load working register (W) with number 00
0001	**0066**	Store W in port B direction code register
0002	**0186**	Clear port B data register
0003	**0A86**	Increment port B data register
0004	**2803**	Jump back to address 0003 above

Program 3.1
BIN1 Machine Code

`Address 0000:  Instruction = 3000  Meaning: MOVE zero into W`

The code 3000 means move (copy) a literal (number given in the program) into the working register (W). All literals must be placed initially in W before transfer to another register. The literal, which is zero in this case, can be seen in the code as the last two digits, 00.

`Address 0001:  Instruction = 0066  Meaning: MOVE W into TRISB`

This means copy the contents of W to the port B data direction register (TRISB). W contains 00 as a result of the first instruction. This code will set all 8 bits of register TRISB to zero, making all bits of port B operate as outputs. The file register address of port B (6) is given as the last digit of the code. These first two instructions are required to initialize port B for output, using the TRIS command to load the bank 1 register called TRISB, address 86 in the file register set.

`Address 0002:  Instruction = 0186  Meaning: CLEAR PORTB to zero`

This instruction will clear file register 6 (last digit), which sets all bits in the port B data register (PORTB) to zero. Operations can be carried out directly on the port data register, and the result will appear immediately on the LEDs in the BIN hardware or simulation.

`Address 0003:  Instruction = 0A86  Meaning: INCREMENT PORTB`

Port B data is now modified by this instruction. The binary value is increased by one, and this value will be seen on the LEDs. This operation will be repeated as a result of the next instruction (jump back), so the port LEDs will show a binary count sequence.

`Address 0004:  Instruction = 2803  Meaning: GOTO last address`

This is a jump instruction, which causes the program to go back and repeat the previous instruction. This is achieved by the instruction overwriting the current program counter contents with the value 03, the destination address, which is given as the last two digits of the instruction code. The execution point is thus forced back to the previous instruction, so the program keeps repeating indefinitely. Most control programs have the same basic structure as this simple example; an initialization sequence and an endless loop, which will read the inputs and modify the outputs.

3.3.2. Program Execution

BIN1 is a complete working program, which initializes and clears port B, and then keeps incrementing it. The last two instructions, increment port B and jump back, will repeat indefinitely, with the value being increased by one each time. In other words, port B data register will act as an 8-bit binary counter. When it reaches FF, it will roll over to 00 on the next increment operation.

If you study the binary count table seen in Appendix A (Table A.3), you can see that the least significant bit is inverted each time the binary count is incremented. The least significant bit (LSB), RB0, will thus be toggled (inverted) every time the increment operation is repeated. The next bit, RB1, will toggle at half this rate, and so on, with each bit toggling at half the frequency of the previous bit. The most significant bit (MSB) therefore toggles at 1/128 of the frequency of the LSB. The output pattern generated is shown in Figure 3.5.

A PIC instruction takes four clock cycles to complete, unless it causes a jump, in which case it will take eight clock cycles (or two instruction cycles). The repeated loop in BIN1 will therefore take $4 + 8 = 12$ clock cycles, and it will take 24 cycles for the RB0 to go low and high, the output period of the LSB. If the CR clock is set to run at 100 kHz, the clock period is $1/10^5$ s $= 10$ μs (frequency $= 1$/period), giving an instruction cycle time of 40 μs. The loop will take $12 \times 10 = 120$ μs, giving an output period of 240 μs, a frequency of 4167 Hz, and RB7 will then flash at $4167/128 = 32.5$ Hz. These outputs can be displayed on an oscilloscope or logic analyzer (virtual or real).

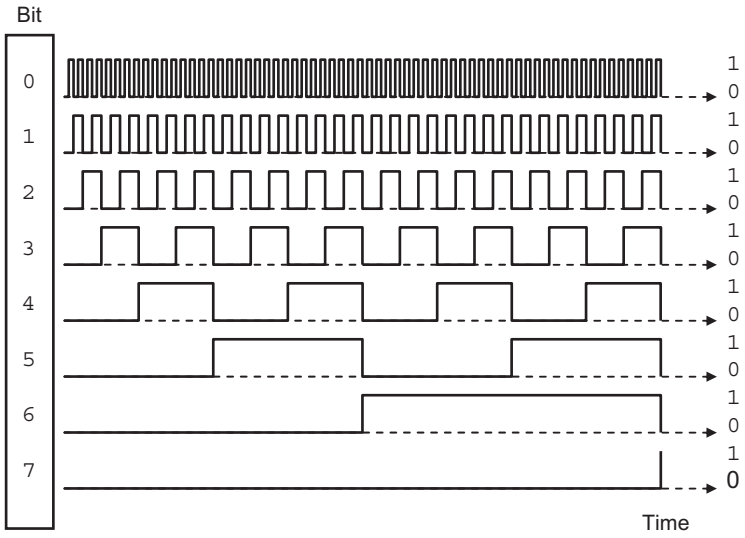


Figure 3.5
BIN1 output waveforms

In the real hardware, the output changes too quickly to see unaided, but it is possible to reduce the clock speed by increasing the value of the resistance and/or the capacitor in the clock circuit. We will see later how to slow the outputs down without changing the clock, by adding a delay routine. In simulation mode, as we will see later, the timing can also be checked using debugging tools such as the stopwatch in MPLAB or the timing display in Proteus VSM.

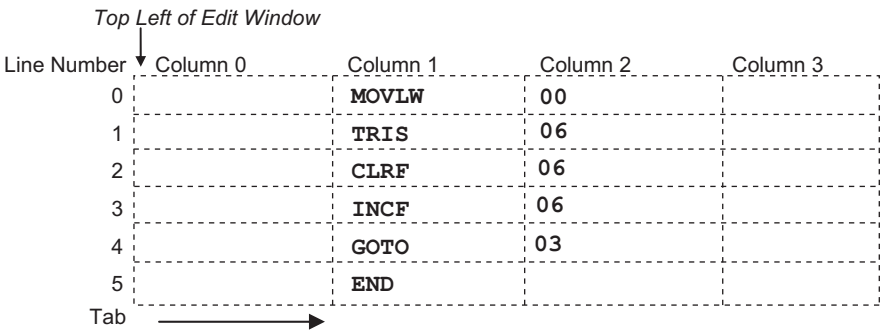
The frequencies generated are actually in the audio range, and they can be heard by passing them to a small loudspeaker or piezo buzzer. This is a handy way of checking quickly that the program is working, and also immediately suggests a range of PIC applications – generating signals and tones at known frequencies. We will come back to this idea later, and see how to generate audio outputs, or a tone sequence to make a tune, such as a mobile phone ring tone.

3.4. Assembly Language

It should be apparent that writing the machine code manually for any but the most trivial applications is going to be a bit tedious. Not only do the actual hex instruction codes have to be worked out, but so do jump destination addresses and so on. In addition, the codes are not easy to recognize or remember.

3.4.1. Mnemonics

For this reason, simple microcontroller programs are written in assembly language, not machine code. Each instruction has a corresponding mnemonic defined in the instruction set. The main task of the assembler program supplied with the chip is to convert a source code program written in mnemonic form into the equivalent machine code. The mnemonic form of the program BIN1 is shown in Program 3.2.



Program 3.2
Mnemonic form of program BIN1

The instructions can now written as recognizable (when you get used to them!) code words. The program is typed into a text editor, spaced out as shown, using the tab key to place the code in the correct columns. Note that the first column (column 0) is kept blank; we will see why later. The instruction mnemonics are placed in column 1, and the operands (data to be operated on) in column 2. The operand 00 is the data direction code for the port initialization, 06 is the file register number of the port data register, and 03 is the jump destination address, line 3 of the program. The PIC instructions are all 14 bits long, so each line of source code becomes a 14-bit code, which we have already seen. The meaning of the mnemonics is as follows:

Line	Mnemonic		Meaning
0	MOVLW	00	Move Literal 00 into W
1	TRIS	06	Move contents of W into TRISB to set port B (06) as outputs
2	CLRF	06	Clear file register 06 (port B) to zero
3	INCF	06	Increment file register 06 (port B)
4	GOTO	03	Jump back to address 03 (previous instruction)
	END		End of source code — this is not an instruction!

The END statement is an assembler directive; it tells the assembler that this is the end of the program, and is not converted into an actual instruction. When entering the program, there must be space before and after each instruction mnemonic, and it is advisable to lay out the program in columns as shown to improve its readability.

3.4.2. Assembly

The source code program could be created using a general purpose text editor, but is normally edited within a dedicated software package such as MPLAB, the PIC integrated development environment (IDE), which contains the assembler as well as a text editor. ISIS schematic capture also incorporates a suitable text editor, which can be opened after the circuit drawing has been completed, or it can be run in conjunction with MPLAB.

The source code text is entered in an edit window and the assembler invoked from the menu. The assembler program analyzes the source code, character by character, and works out the binary code required for each instruction. The terminology can be confusing here; the assembly language application program (user source code) is created in the text editor, while the software tool that performs the conversion is the assembler program or utility.

The source code is saved on disk as a text file called PROGNAME.ASM, where ‘progname’ represents any suitable filename. This is then converted by the assembler program MPASM.EXE, which creates the machine code file PROGNAME.HEX. This appears as

hexadecimal code when listed. At the same time, PROGNAME.LST, the list file, is created, containing both the source and hex code, which may be useful later on when debugging (fault finding) the program, we have already seen KEY690.LST in Chapter 1. This assembly and download process will be described in more detail in the next chapter.

3.4.3. Labels

The mnemonic form of the program with numerical operands can now be further improved. We want the operands to be specified in a more easily recognizable form, in the same way that the mnemonics represent the instruction codes. Therefore, the assembler is designed to recognize labels.

A label is a word that represents a number, which can be an address, register or literal. Examples used below are 'again', 'portb' and 'allout'. These are listed at the top of the program with the replacement value, and the assembler simply replaces any occurrence of the label with the corresponding number.

Jump destinations are similarly defined by label, by placing the label at the beginning of the destination line, and using a matching label as the jump instruction operand. When the program is assembled, the assembler notes the numerical address of the instruction where the label was found, and replaces the label, when found as an operand, with this address.

The program BIN1 can thus be rewritten using labels as shown in BIN2 source code (Program 3.3). The literal value 00 and the port register address 06 have been replaced with labels, which are assigned at the beginning of the program. The 'equate' statements define the numbers that are to be replaced in the source code. In this case, the label 'allout' will represent the port B data direction code, while the data register address itself, 06, will be

Edit Window

```
allout  EQU     00
portb   EQU     06

        MOVLW   allout
        TRIS    portb

        CLRF    portb
again   INCF    portb
        GOTO    again

        END
```

Program 3.3
BIN2 source code using labels

represented by the label 'portb'. 'EQU' is another example of an assembler directive, which is an instruction to the assembler program and will not be translated into code in the executable program.

Note that lower case is used for the labels, while upper case is used for the instruction mnemonics and assembler directives. Although this is not obligatory, this convention will be used because the instruction mnemonics are given in upper case in the instruction set. The labels can then be distinguished by using lower case. The jump destination label is simply defined by placing it in column 0 of the line containing the destination instruction. The 'GOTO label' instruction then uses a matching label. Initially, labels will be limited to six characters; they must start with a letter, but can contain numbers, e.g. 'loop1'. Longer labels may be used if preferred.

The programs BIN1 and BIN2 are functionally identical, and the machine code will be the same.

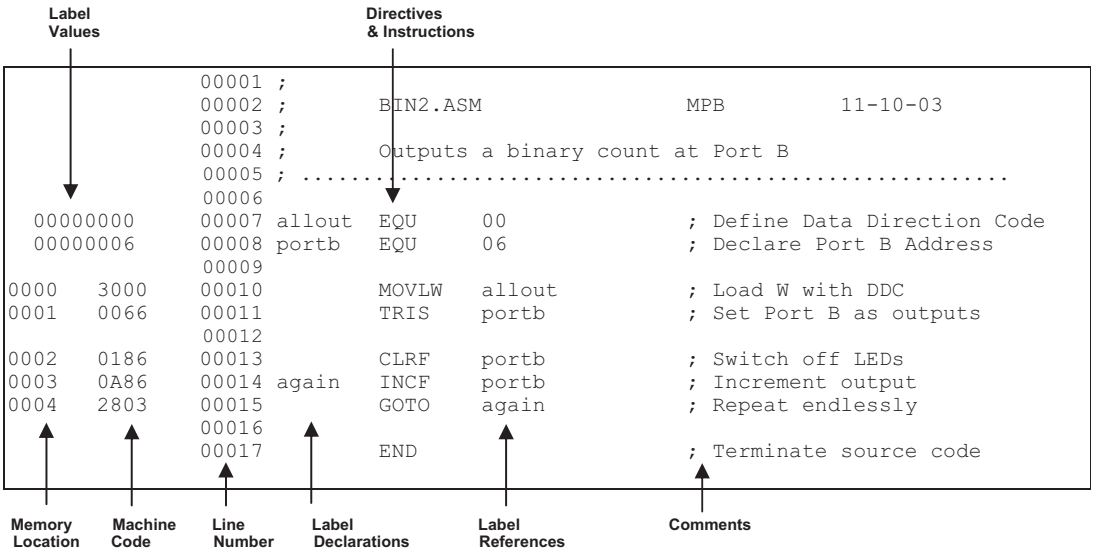
3.4.4. Layout and Comments

A final version of BIN2 (Program 3.4) includes comments in the program to explain the action of each line, and the overall program. As much information as possible should be provided. When learning programming, comments help the learner to retain information, and when developing real applications, it will help with future modifications and upgrading or software maintenance. Even if you have written the program yourself, you may have forgotten how it works later on!

Comments must be preceded by a semicolon (;), which tells the assembler to ignore the rest of that line. Comments and information can thus occupy a whole line, or can be added after

```
;       BIN2.ASM          MPB          11-10-03
;       Outputs a binary count at Port B
; ..............................................

allout  EQU     00          ; Data Direction Code
portb   EQU     06          ; Declare Port B Address

        MOVLW   allout      ; Load W with DDC
        TRIS    portb       ; Set Port B as outputs

        CLRF    portb       ; Switch off LEDs
again   INCF    portb       ; Increment output
        GOTO    again       ; Repeat endlessly

        END                 ; Terminate source code
```

Program 3.4
BIN2 source code with comments



Program 3.5
BIN2 list file (edited)

each instruction in column 3. A minimal header has been added to BIN2, with the source code file name, author and date, and a comment added to each line. Blank lines can be used without a comment 'delimiter' (the semicolon); these are used to break up the source code into functional sections, and thus make the structure of the program easier to understand. In BIN2.ASM, the first block contains the operand label equates, the second the port initialization and the third the output sequence. The layout of the source code is very important in showing how it works.

The list file for the program BIN2 is provided as Program 3.5. It contains the source code, machine code and program memory addresses in one file. It has been edited to remove extraneous detail; the original may be downloaded from www.picmicros.org.uk with all the other demo application filesets. A complete list file may be seen in Table 4.4 (Chapter 4).

We now have a program that can be entered into a text editor, assembled and downloaded to the PIC chip. The exact method will vary with the development system you are using. Next, we will look in more detail at developing the program.

Questions 3

1. State the four-digit hex code for the instruction INCF 06. (2)
2. State the two-digit hex code for the instruction MOVLW. (2)
3. What is the meaning of the least significant two digits in the PIC machine code instruction 2803? (2)
4. Why must the instruction mnemonic be in the second column of the source code? (2)

5. Give two examples of a PIC assembler directive. Why are they not represented in the machine code? (3)
6. What are the numerical values of the labels 'allout' and 'again' in BIN2? (2)
7. A line from the list file for BIN2 is shown below. Explain the significance of each item. (6)

```
0003 0A86 00014 again INCF portb
```

8. State the function and origin of program files with the extension: (a) ASM, (b) HEX, (c) LST. (6)

Answers on page 419. (Total 25 marks)

Activities 3

1. Check the machine code for BIN1 against the information given in the PIC instruction set in the data sheet, so that you could, if necessary, write a program entirely in machine code. Modify the machine code program by deleting the 'Clear Port B' operation and changing the 'Increment Port B' to 'Decrement Port B'. What would be the effect at the output when the program was run? Suggest an alternative to the instruction MOVLW 00 which would have the same effect.
2. Refer to Appendix E for a guide to using Proteus VSM to simulate the circuit. You will need a version of Proteus that includes the working model of the 16F84A chip. Enter or download the schematic BIN.DSN into ISIS and attach the program BIN1.ASM. Check that it assembles and runs correctly. Display the SFRs and source code. Set the MCU simulated clock to 100 kHz and single step the program. Observe the execution sequence, and check that the time taken to complete one loop is 120 µs.
3. Enter the program BIN2, using labels, into the text editor, assemble and test as above. Display or print out the list file BIN2.LST and check that the machine code generated is the same as BIN1. Note that there is no machine code generated for comment lines or assembler directives.

If necessary, refer forward to relevant sections to complete these activities.

CHAPTER 4

PIC Program Development

Chapter Outline

PIC Microcontrollers. DOI: 10.1016/B978-0-08-096911-4.10004-7

Chapter Points

- The development process consists of specification, hardware selection and design, program development and testing.
- The program is converted to assembler source code, FILENAME.ASM, using the instruction format defined for the assembler.
- The assembler converts the source code text into object code, FILENAME.HEX.
- Syntax errors detected must be corrected at this stage.
- A list file, FILENAME.LST, is created which lists the source code, object code, label and memory allocation.
- The simulator allows the machine code to be tested without downloading to the actual target system.
- Logical errors detected must be corrected at this stage.
- The program can then be downloaded and tested in the target hardware.

We have seen how to get started with developing PIC® application hardware and software in Chapter 3, and can take a closer look at some of the software tools available, and how each is used in the program development process. The program BIN2 will be further developed using the same hardware design.

This chapter will describe some features of the standard PIC development tools that are currently available, but hardware and software to support application developers are being continuously developed by Microchip and third-party suppliers, so it is impossible to be completely up to date. These tools are downloadable from www.microchip.com. We will also look at how to use Proteus VSM (www.labcenter.com) for schematic capture and simulation, since this provides the most user-friendly method of testing the design prior to developing real hardware. A tutorial for using this software is provided in Appendix E.

4.1. Program Development

The primary development system toolset for PIC applications is Microchip MPLAB IDE (integrated development environment). At the time of writing, MPLAB version 8.60 is the most recently released version. The basic principles of use do not change greatly as it is updated, rather extra facilities and support for an ever-expanding range of devices are added, and the reader will need to refer to the manufacturer's documentation for details concerning the use of any particular version. The intention at this stage is to outline how to assemble and test demonstration programs BIN3 and BIN4. More advice on debugging programs is provided in Chapter 9.

The flowchart in Figure 4.1 gives an overview of the program development process. The starting point is the specification for the program, which describes how the application will function when complete. This must then be analyzed by the software designer so that the required program can be derived from it, taking into account the features of the instruction set of the microcontroller (MCU). The program algorithm describes the process whereby the required outputs are obtained from the given inputs. Various software design techniques are available to outline the program, including flowcharts and pseudocode, which we will use here. These represent the program processes and their sequence in a logically consistent way, such that the assembler (or other language) source code can be derived from them.

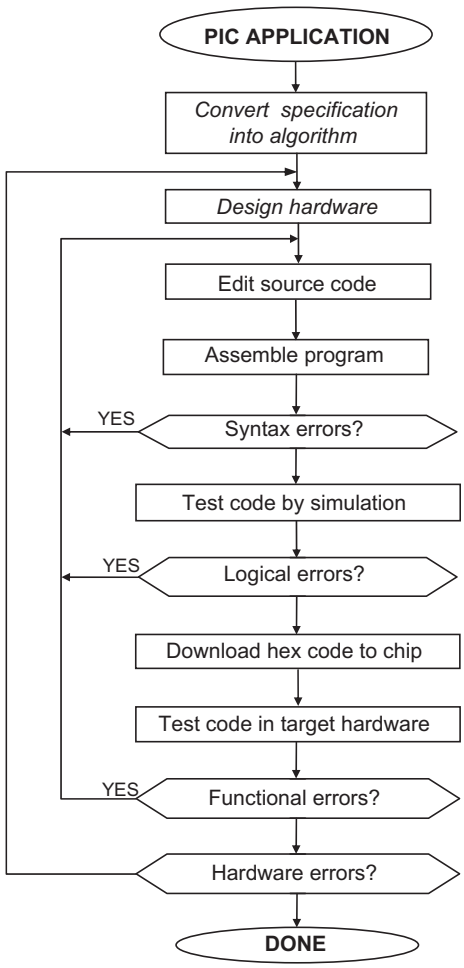


Figure 4.1
Program development flowchart

The source code is developed from the program algorithm by filling in the details and converting each program block to assembler code. The program must be saved regularly as it is developed; it is a good idea to always have backup copies on different disks (memory stick, hard disk or network drive) in case of disk, memory or network failure. The source code text file is called PROGNAME.ASM, where PROGNAME represents the application name, such as BIN1. Successive versions of a program can be numbered BIN1, BIN2, etc., so that we can revert to an earlier version if new code does not work properly.

After initial text entry, the program can be assembled by calling up the assembler utility, MPASMWIN.EXE. It converts the source code into machine code, and creates additional files to help with debugging (fault finding) the program. If a mistake has been made in the individual instruction (e.g. misspelling a mnemonic), it will be reported in an error message window and an entry added to the error file on disk. This must then be corrected in the source code and the program reassembled until it is free of syntax errors.

The program can then be tested for correct operation by simulation, using the Microchip MPLAB simulator MPSIM or Proteus VSM (ISIS). This means running the program in the host computer as though it were running in the chip itself. The program is loaded and executed on screen, and the outputs are observed. It can then be checked step by step for the correct logical operation, by monitoring the changes in the registers, and checking the timing if necessary. Simulated inputs are needed to represent all the likely input sequences and combinations. If a logical error is found, the source code must be re-edited and reassembled, and the simulation repeated.

This process is simpler in ISIS, since inputs can be generated interactively on the schematic (see Figure 3.3) from animated manual inputs or simulated signals. Outputs can be observed directly on light-emitting diodes (LEDs), seven-segment displays or liquid crystal displays (LCDs), or on virtual instruments such as oscilloscopes or logic analyzers. However, MPLAB has the advantage that inputs and outputs can be generated from a predefined stimulus file and outputs recorded in a trace file, which provides a form of automated testing using standard sequences and a permanent record of the test results. Thus, MPSIM has a more extensive range of debugging tools, but Proteus can test the whole circuit and is more intuitive.

When the logical errors have been removed, the program can be downloaded to the chip. Final testing can then allow the finished circuit function to be compared with the original specification. If necessary, in-circuit debugging can be used to detect any remaining errors that may arise from interaction with the real peripheral circuits. Most current PIC chips have on-board circuitry to support this option, while smaller chips need a dedicated header connector (see Chapter 7). Only MPSIM provides this final debugging stage.

The main software tools and the files created and used by MPLAB during the development process are listed in Table 4.1. The most significant ones are the source code (.ASM) and

Table 4.1: Components of MPLAB development system (version 8.60)

Software Tool	Tool Function	Files Produced or Used	File Description
Text Editor: MPLAB.EXE + MPEditor.dll, etc.	Used to create and modify source code text file	PROGNAME.ASM	Source code text file
Assembler: MPASMWIN.EXE (stands alone)	Generates machine code from source code, reports syntax errors, generates list and symbol files	PROGNAME.HEX PROGNAME.ERR PROGNAME.LST	Executable machine code Error messages List file with source & machine code
Simulator: MPLAB.EXE + MPSim.dll, etc.	Allows program to be tested in software before downloading	PROGNAME.HEX PROGNAME.COF	Executable machine code Linker output file
Programmer: MPLAB.EXE + PICkit2.dll, etc.	Downloads machine code to chip	PROGNAME.HEX	Executable machine code

machine code (.HEX) for simulation and downloading. The error file (.ERR) stores the assembler error messages that are displayed automatically when generated. The list file (.LST) contains the source text, machine code, memory allocation and label values in one text file. In more complex applications, multiple relocatable machine code object files (.O) that have been assembled separately as application components are combined by a linker utility to produce a COF file which contains the hex and list files (see Microchip 'MPASM User's Guide').

4.2. Program Design

There are international standards for specifying engineering designs, which should be applied in commercial work. The design standards for different types of products will vary; for instance, a military application will typically be designed to a higher standard of reliability and more rigorously tested and documented than a commercial one. Our designs here are somewhat artificial in that they are intended to illustrate features of the PIC microcontroller, rather than meet a genuine user requirement. Nevertheless, we can follow the design process through the main steps.

4.2.1. Application Specification

The first step in the design process is to specify the functions and performance required by the application. In the real world, this needs to be done in some detail so that the overall design, development and production costings and timescales can be predicted as far as possible, as well

as establishing the market or customer requirements. For our purposes, the minimal specification given in Chapter 3 will suffice:

> *The circuit should output a binary count to eight LEDs, under the control of two push-button inputs. One input will start the output sequence when pressed. The sequence will stop when the button is released, retaining the current value on the display. The other input will clear the output (all LEDs off), allowing the count to resume from zero.*

The next step is to design the hardware in which the application program will run, unless it already exists. A block diagram, which shows the user interface requirements, is a good starting point. The interfacing of the microcontroller is often implemented using certain standard devices, such as push buttons, keypad, LED indicators, LCD, relays and so on. The circuit design techniques required will not be covered in any detail here; the most common interfacing techniques and devices are described in *Interfacing PIC Microcontrollers: Embedded Design by Interactive Simulation* by the author (Newnes 2006). The microcontroller must be selected by specifying the requirements such as:

- Number and type of inputs and outputs (based on chip pin-out)
- Program memory size (maximum number of instructions)
- Data memory size (number of spare file registers)
- Program execution speed (clock speed)
- Availability of special interfaces (e.g. analogue inputs, serial ports).

The hardware configuration for the BINx applications has already been described in Chapter 3 (Figure 3.3). We have established that the instruction set and programming features of the microcontroller selected are suitable. If further features were required, the existing hardware design could be modified. If the microcontroller selected was then found to be lacking in some respect, for example, not enough input/output (I/O) pins, another microcontroller, or other types of hardware such as a conventional microprocessor system, may be considered. However, it is easier to stay within one family of processors, since it is normally based on a shared architecture and instruction set, from which the most suitable can be selected. The PIC range is the most extensive available for small and mid-range applications.

Microcontrollers are normally used in so-called real-time applications, typically a control system where inputs are measured and outputs modified as rapidly as possible, such as a motor vehicle engine controller. Complex data processing may be needed, but data storage is minimal. For simpler applications, assembly language provides maximum speed and minimum memory requirements, but for more complex software, a higher level language may be needed. This will provide a greater range of more user-friendly statements and constructs, such as mathematical functions and display drivers. The more powerful PIC microcontrollers are therefore generally programmed in the language C, the next level up from assembler. The basic syntax is more like English than assembler, and is therefore easier to learn and does not depend on an intimate knowledge of the MCU architecture. The downside is that each C statement

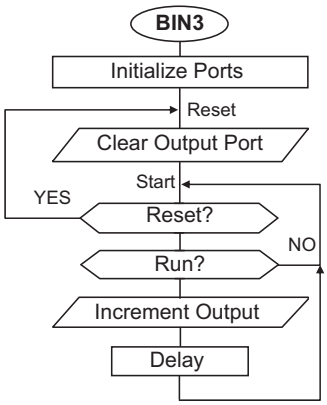


Figure 4.2
Flowchart for program BIN3

translates into several assembler instructions, so the program is longer and therefore slower to execute. It also needs more program memory. Programming PIC microcontrollers in C is introduced in *Programming 8-bit PIC Microcontrollers in C with Interactive Hardware Simulation* by the author (Newnes 2008).

4.2.2. Program Algorithm

A flowchart can be useful for illustrating the overall algorithm (process) graphically, particularly when learning programming. A flowchart for BIN3 is shown in Figure 4.2. The program title is placed in the start symbol at the top of the flowchart, and the processes required are defined as a sequence of blocks. Each flowchart box will contain a description of the action at each stage, using different shaped boxes for processes (rectangle), input and output (sloping) and decisions (pointed). The decision box has two outputs, to represent a conditional branch in the program. This decision box should contain a question with the answer yes or no, and the active selection labeled YES or NO as appropriate; only output one needs to be labeled. The jump destinations can also be labeled, and these same labels can be used in the program as address labels.

Programming packages exist that allow a flowchart to be converted directly into code, which are most useful in education and training environments. Software design techniques, including flowcharts, will be discussed in more detail later.

4.3. Program Editing

The program is written using the instruction set of the processor selected, as specified in the MCU data sheet. The instruction set is essentially the same for all 16 series PIC chips. The source code, that is, the assembly code program, must be entered into a suitable text editor,

which is normally provided with the development system. We will not go into the details of using a text editor, as it is assumed that the reader is familiar with using a word processor.

The MPLAB programming window has limited editing features, because it is only used for creating plain text files. The typeface Courier is used because each character occupies the same space, unlike proportionally spaced typefaces such as Arial and Times Roman. Displayed in this way, the text lines up vertically as well as horizontally, so the program can be laid out consistently in columns using tab stops, making it easier to understand. The tab spacing should be set to eight characters for the programs in this book. When printing, also use Courier to maintain the correct layout.

When a new application is started, a separate folder should be created to contain the source code file, and all the other files that will be generated by the assembler. Name the folder with the application name, e.g. BIN3. When the source code file has been opened, enter the source code filename (e.g. BIN3.ASM) at the top of the file, and immediately save it in the folder. This ensures that the required filepath is checked for correct operation before any further source text is entered. Always keep a backup copy of your file on a different drive before quitting the development system.

4.3.1. Instruction Set

The PIC 16 data sheets and the mid-range manual show the instruction set arranged by byte, bit, literal and control operations. Table 4.2 shows the same instruction set organized by function with an example given with each instruction so that the typical syntax can be seen. A detailed description of each instruction is provided in the data sheet, with more information in Chapter 6.

The grouping of the instructions in Table 4.2 reflects the different types of instruction explained in Chapter 2: data movement, single and register pair arithmetic and logical operations, sequence control and miscellaneous instructions. In the example instructions, the register used is always 0C, the first general purpose register in the smaller PIC 16 chips. The register being operated upon will normally be referred to by label (see Program BIN3 below). Literals and register bit numbers may be referred to by number or label, depending on the context.

4.3.2. BIN3 Source Code

Program BIN3 uses the same instructions as BIN2 (Chapter 3), with additional statements to read the switches and control the output. Program 4.1 is the result.

First, note the general layout and punctuation. The program header block contains as much information as is necessary at this stage. These comments are preceded by a semicolon on each line to indicate to the assembler that this text is not part of the program. Assembler

Table 4.2: PIC mid-range instruction set

PIC 16 Instruction Set by Functional Groups			
MOVE	Move data from F to W	`MOVF`	`0C,W`
	Move data from W to F	`MOVWF`	`0C`
	Move literal into W	`MOVLW`	`0F9`
REGISTER	Clear W (reset all bits and value to 0)	`CLRW`	
	Clear F (reset all bits and value to 0)	`CLRF`	`0C`
	Decrement F (reduce by 1)	`DECF`	`0C`
	Increment F (increase by 1)	`INCF`	`0C`
	Swap the upper and lower four bits in F	`SWAPF`	`0C`
	Complement F value (invert all bits)	`COMF`	`0C`
	Rotate bits Left through Carry Flag	`RLF`	`0C`
	Rotate bits Right through Carry Flag	`RRF`	`0C`
	Clear (reset to zero) the bit specified (e.g. bit 3)	`BCF`	`0C,3`
	Set (to 1) the bit specified (e.g. bit 3)	`BSF`	`0C,3`
ARITHMETIC	Add W to F (with carry out)	`ADDWF`	`0C`
	Add F to W (with carry out)	`ADDWF`	`0C,W`
	Add L to W (with carry out)	`ADDLW`	`0F9`
	Subtract W from F (with carry in)	`SUBWF`	`0C`
	Subtract W from F, result in W	`SUBWF`	`0C,W`
	Subtract W from L, result in W	`SUBLW`	`0F9`
LOGIC	AND the bits of W and F, result in F	`ANDWF`	`0C`
	AND the bits of W and F, result in W	`ANDWF`	`0C,W`
	AND the bits of L and W, result in W	`ANDLW`	`0F9`
	OR the bits of W and F, result in F	`IORWF`	`0C`
	OR the bits of W and F, result in W	`IORWF`	`0C,W`
	OR the bits of L and W, result in W	`IORLW`	`0F9`
	Exclusive OR the bits of W and F, result in F	`XORWF`	`0C`
	Exclusive OR the bits of W and F, result in W	`XORWF`	`0C,W`
	Exclusive OR the bits of L and W	`XORLW`	`0F9`
TEST & SKIP	Test a bit in F and Skip next instruction if it is Clear (= 0)	`BTFSC`	`0C,3`
	Test a bit in F and Skip next instruction if it is Set (= 1)	`BTFSS`	`0C,3`
	Decrement F and Skip next instruction if it is now Zero	`DECFSZ`	`0C`
	Increment F and Skip next instruction if it is now Zero	`INCFSZ`	`0C`
JUMP	Go To a labelled line in the program (e.g. start)	`GOTO`	`start`
	Jump to the Label at the start of a Subroutine (e.g. delay)	`CALL`	`delay`
	Return at the end of a Subroutine to the next instruction	`RETURN`	
	Return at the end of a Subroutine with L in W	`RETLW`	`0F9`
	Return from Interrupt Service Routine to next instruction	`RETFIE`	
CONTROL	No Operation – delay for 1 cycle	`NOP`	
	Go into Standby Mode to save power	`SLEEP`	
	Clear Watchdog Timer to prevent automatic reset	`CLRWDT`	
	Load Port Data Direction Register from W*	`TRIS`	`06`
	Load Option Control Register from W*	`OPTION`	

Each instruction is given as an example, with explicit register and literal values.
F: Any file register (specified by number or label), e.g. 0C; W: working register; L: literal value (given in instruction), e.g. 0F9. The result of arithmetic and logic operations can generally be stored in W instead of the file register by adding ' ,W ' to the instruction. General purpose register 1, address 0C, represents all file registers. Literal value 0F9 represents all values 00–FF. Bit 3 is used to represent file register bits 0–7. For MOVE instructions data is copied to the destination and retained in the source register. Some register pair operations are duplicated for result in F or W. Total number of instructions=35, not including TRIS and OPTION. For full details see Microchip's 'PIC Mid-Range MCU Family Reference Manual', available online.
*Use of these special instructions is not recommended by the manufacturer.

```
;
;       BIN3.ASM                                    MPB  12-10-03
; ..............................................................
;
;       Slow output binary count is stopped, started
;       and reset with push buttons.
;
;       Processor = 16F84A      Clock = CR, 100kHz
;       Inputs: RA0, RA1        Outputs: RB0 - RB7
;
; ************************************************************

; Register Label Equates.....................................

porta   EQU     05              ; Port A Data Register
portb   EQU     06              ; Port B Data Register
timer   EQU     0C              ; Spare register for delay

; Input Bit Label Equates ...................................

inres   EQU     0               ; 'Reset' input button = RA0
inrun   EQU     1               ; 'Run' input button = RA1

; ************************************************************

; Initialise Port B (Port A defaults to inputs)..............

        MOVLW   00              ; Port B Data Direction Code
        TRIS    portb           ; Load the DDR code into F86
        GOTO    reset

; Start main loop ...........................................

reset   CLRF    portb           ; Clear Port B

start   BTFSS   porta,inres     ; Test RA0 input button
        GOTO    reset           ; and reset Port B if pressed
        BTFSC   porta,inrun     ; Test RA1 input button
        GOTO    start           ; and run count if pressed

        INCF    portb           ; Increment count at Port B

        MOVLW   0FF             ; Delay count literal
        MOVWF   timer           ; Copy W to timer register
down    DECFSZ  timer           ; Decrement timer register
        GOTO    down            ; and repeat until zero

        GOTO    start           ; Repeat main loop always
        END                     ; Terminate source code
```

Program 4.1
BIN3 source code

directives such as EQU and END are also not part of the program proper, but used to define labels and the end of the program source code. The labels 'porta', 'portb' and 'timer' refer to file registers 05, 06 and 0C, respectively; 'inres' and 'inrun' are input bit labels representing the push buttons. The program uses 'Bit Test and Skip' instructions followed by 'GOTO label' for conditional jumping.

At this stage, the learner can type the source code into the editor without full analysis in order to practice use of the editor. The instructions are placed in the first three columns, and the comments can be left out to save time. Labels go in the first column, instruction mnemonics in the second and the instruction operands in the third. The source code text file should be saved as BIN3.ASM in a suitably named directory or folder on disk. Alternatively, the source code file can be downloaded from the support website www.picmicros.org.uk. It can then be tested in the free MPLAB simulator (see Section 4.7) or in Proteus VSM (see Appendix E).

4.3.3. Syntax

'Syntax' refers to the way that words are put together to create meaningful statements, or a series of statements. In programming, the syntax rules are determined by the assembler, which will be used to create the machine code. The assembler must be provided with source code that it can convert into the required machine code without any ambiguity, that is, only one meaning is possible. This is why the assembler syntax rules are very strict.

4.3.4. Layout

The program layout should be in four columns, as described in Table 4.3. Each character then occupies the same space, and the columns are correctly aligned. The label, command and operand columns are set to a width of eight characters, with the maximum label length of six characters, leaving a minimum of two clear spaces between columns (longer labels can be used, but a different form of the program layout must then be used). The tab key is normally used to place the text in columns, and the tab spacing can be adjusted if necessary.

Table 4.3: Layout of assembler source code

Column 1 Label	Column 2 Command	Column 3 Operand(s)	Column 4 Comment
Label equated to a value, or to indicate a program destination address for jumps	Mnemonic form of the instruction for the processor to carry out a specific operation. Only mnemonics specified in the instruction set may be used	The data or register contents to be used in the instruction. Registers are usually represented by a label. Some instructions do not need an operand	Explanatory text to the right of a semicolon on any line of code helps the programmer and user to understand the program. It has no effect on the operation of the program. Full line comments may also be used between program blocks

4.3.5. Comments

Comments are not part of the actual program, but are included to help the programmer and user understand how the program works. Comments are preceded by a semicolon (;), which can be placed at the beginning of a line to indicate a comment which relates to a whole program block (functional set of statements), or at the start of column 4 for line comment. The comment and line are terminated with a line return ('Enter' key).

A standard header block is recommended (see Program 4.1). For simple programs, the first line should at least contain the source code file name, the author and date, and/or version number. A program description should also be provided in the header, and for programs that are more complex, the processor type, target hardware details and other relevant information. In general, the bigger the program, the more information would be expected in the header comments.

4.4. Program Structure

Structured programming means constructing the program, as far as possible, from discrete blocks. This makes the program easier to write and understand, more reliable and easier to modify at a later date. Program BIN3 is unstructured, in that the program instructions are essentially executed in the order given in the source code. An equivalent 'structured' program, BIN4, is listed as Program 4.2.

The main difference between BIN3 and BIN4 is that the program now has the delay sequence as a 'subroutine'. The subroutine is inserted before the main program block, and assembled first. It is then 'called' from the main program by label. The subroutine can be created as a self-contained program block, and reused in the program as necessary. It can be called as many times as required, which means that the block of code needs to be written only once. It can also be converted to a separate file and reused in another program. In addition, the delay time is loaded before the subroutine execution, so the same delay routine could be used to provide different delay times.

A program flowchart has been given for BIN3 (Figure 4.2). The same flowchart describes BIN4, but the delay routine can now be expanded as a separate subroutine flowchart (Figure 4.3). The use of flowcharts in program design will be more fully examined in Chapter 8.

4.5. Program Analysis

The program BIN4 will now be analyzed in some detail as it was designed to contain examples of common PIC syntax. A sample instruction of each type will be examined.

```
;
;       Source File:    BIN4.ASM
;       Author:         M. Bates
;       Date:           15-10-03
;       ..................................................
;       Program Description:
;
;       Slow output binary count is stopped, started
;       and reset with push buttons. This version uses a
;       subroutine for the delay....
;
;       Processor:      PIC 16F84A
;
;       Hardware:       PIC Demo System
;       Clock:          CR ~100kHz
;       Inputs:         Push Buttons RA0, RA1 (active low)
;       Outputs:        LEDs (active high)
;
;       WDTimer:        Disabled
;       PUTimer:        Enabled
;       Interrupts:     Disabled
;       Code Protect:   Disabled
;
; ********************************************************

; Register Label Equates.....................................

porta   EQU     05              ; Port A Data Register
portb   EQU     06              ; Port B Data Register
timer   EQU     0C              ; Spare register for delay

; Input Bit Label Equates ...................................

inres   EQU     0               ; 'Reset' input button = RA0
inrun   EQU     1               ; 'Run' input button = RA1

; ********************************************************

; Initialise Port B (Port A defaults to inputs).............

        MOVLW   b'00000000'     ; Port B Data Direction Code
        TRIS    portb           ; Load the DDR code into F86
        GOTO    reset


; 'delay' subroutine ........................................

delay   MOVWF   timer           ; Copy W to timer register
down    DECFSZ  timer           ; Decrement timer register
        GOTO    down            ; and repeat until zero
        RETURN                  ; Jump back to main program
```

Program 4.2
BIN4 source code

```
; Start main loop ......................................

reset   CLRF    portb           ; Clear Port B Data

start   BTFSS   porta,inres     ; Test RA0 input button
        GOTO    reset           ; and reset Port B if pressed
        BTFSC   porta,inrun     ; Test RA1 input button
        GOTO    start           ; and run count if pressed

        INCF    portb           ; Increment count at Port B
        MOVLW   0FF             ; Delay count literal
        CALL    delay           ; Jump to subroutine 'delay'

        GOTO    start           ; Repeat main loop always
        END                     ; Terminate source code
```

Program 4.2: (*continued*)

4.5.1. Label Equates

```
timer  EQU   0C
```

The use of labels in place of numbers makes programs easier to write and understand, but we have to 'declare' those labels at the beginning of the program. In assembly code, the

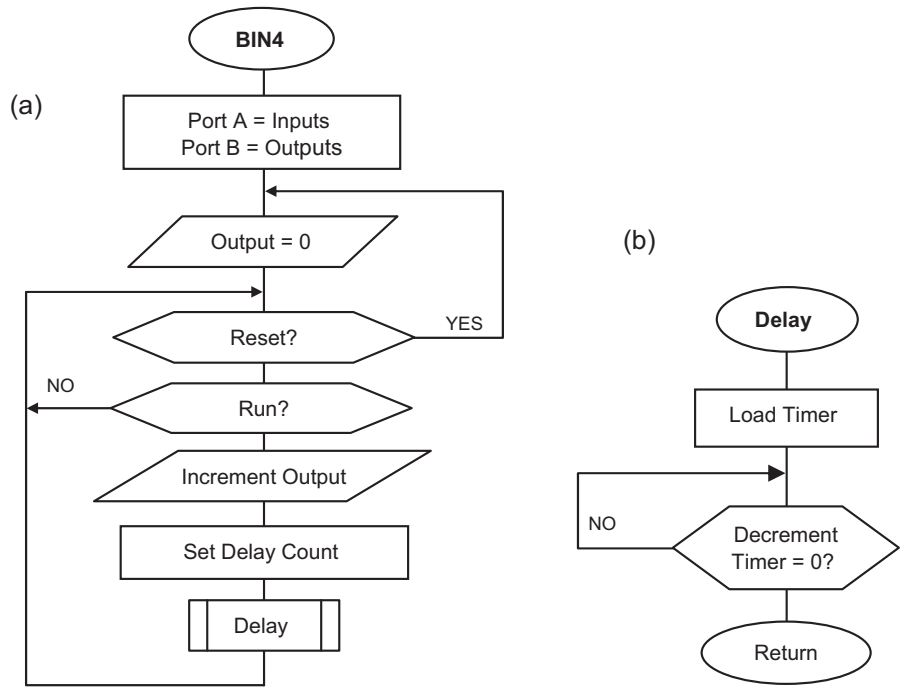


Figure 4.3
Flowcharts for program BIN4: (a) main routine; (b) subroutine

assembler directive EQU is used to assign a label to a number, which can be a literal, a file register number or an individual register bit. In BIN4, 'porta' and 'portb' are the port data registers (05 and 06) and 'timer' is the first spare register (0C), which will be used as a counter register. The labels 'inres' and 'inrun' will represent bit 0 and bit 1 of port A; they are simply given the numerical values 0 and 1. The label is replaced by the number when the program is assembled.

4.5.2. Port Initialization

```
TRIS  portb
```

Port B is used as the output for the 8-bit binary count. The data direction must be set up using the TRIS command, which loads the port data direction register with the data direction code. In this example, the code is given in binary, b'00000000'. This is useful, especially if the port bits are to be set as a mixture of inputs and outputs; the binary code identifies the data direction for each bit individually. This code is loaded into W using MOVLW, and the TRIS command follows.

The TRIS instruction is still available as a simple way of initializing the ports, but the manufacturers recommend an alternative method, which involves bank selection, and will be covered later. Hopefully, TRIS will continue to be supported in by the MPASM assembler, as it is easier for beginners.

4.5.3. Program Jumps

```
GOTO  start
```

The 'GOTO label' command is used to make the program jump to a line other than the one following. In BIN4, 'GOTO reset' skips over the following DELAY routine, to start the main loop. We will come back to the reason for this in a moment. There is another unconditional jump at the end of the program, 'GOTO start', which makes the main loop repeat endlessly. Other 'GOTO label' instructions are used with 'Test and Skip' instructions to create conditional branches. In this program, the input buttons are checked using this type of instruction and the program branches, or not, depending on whether a button has been pressed.

4.5.4. Bit Test and Skip if Set/Clear

```
BTFSS  porta,inres
```

The input button connected to port A, bit 0 is tested using the above instruction, which means 'Bit Test File (register bit) and Skip the next instruction if it is Set (=1)'. Without labels, the instruction 'BTFSS 05,0' would have the same effect. The buttons are connected 'active low', meaning that the input goes from '1' to '0' when the button is pressed. If the button connected to RA0 is not pressed, the input will be high, that is, set. The following

instruction, 'GOTO reset' will therefore be skipped, and the next executed. When the button is pressed, the 'GOTO reset' is executed, and the CLRF instruction repeated, clearing the previous count.

BTFSC means 'Bit Test and Skip if Clear'; it works in the same way as BTFSS, except that the logic is reversed. Thus, 'BTFSC porta,inrun' tests bit 1 of port A register and skips the following 'GOTO start' if the 'run' button has been pressed. The program will then proceed to increment the output count. If the button is not pressed, the program waits by jumping back to the 'start' line. The combined effect of the input buttons is that the count runs when the 'run' button is pressed, and the count is reset to zero if the 'reset' button is pressed.

4.5.5. Decrement/Increment Register and Skip If Zero

```
DECFSZ  timer
```

The other instructions for conditional branching allow a register to be incremented or decremented and then checked for a zero result, all in one instruction. This is a common requirement for counting and timing applications, and in the delay routine in BIN3, a register 'timer' is loaded with the maximum value FF and decremented. If the result is not yet zero, the jump 'GOTO down' is executed. When the register reaches zero, the GOTO is skipped and the subroutine ends. In BIN4, the timer value is set up before the delay subroutine is called.

4.5.6. Subroutine Call and Return

The main elements of the subroutine call structure are shown below:

```
start ....              ; start main program
      ....
      CALL   delay      ; jump to subroutine
      ....              ; return to here
      GOTO   start      ; end of main loop
delay ....              ; subroutine start
      ....
      ....
      RETURN            ; subroutine ends
```

In this program, the subroutine provides a delay by loading a register and counting down to zero. The delay is started using the 'CALL delay' instruction, when the program jumps to the label 'delay' and runs from there. CALL means 'jump and come back to the same place later', so the return address has to be stored for later recall in a special memory block called the 'stack'.

The address of the instruction following (in this case 'GOTO start') is saved automatically on the stack as part of the execution of the CALL instruction. The subroutine is terminated with the instruction 'RETURN', which does not require an operand because the return destination address is automatically pulled from the stack and replaced in the program counter. This takes the program back to the original place in the main program. The PIC 16 stack can store up to eight return addresses, so multiple levels of subroutine can be used. The return addresses are pushed onto and pulled from the stack in order, so if a CALL or RETURN is missed out of the program, a stack error will occur. Unfortunately, this mistake will not be detected by the assembler, but will cause a run-time error message.

4.5.7. End of Source Code

```
END
```

The source code must be terminated with assembler directive END so that the assembly process can be stopped in an orderly way, and control returned to the host operating system. It is the only assembler directive that is essential.

4.6. Program Assembly

To create the PIC program, the MPLAB IDE development system must be downloaded and installed from the Microchip website www.microchip.com. After starting the software, clicking on the new file button opens a source code edit window. The code for the demo programs can be entered and saved in a suitable folder, using the same name as the fileset, e.g. BIN4. The source code is saved as APPNAME.ASM. If the source code already exists, it can be reopened in the usual way. The sample files may be downloaded from www.picmicros.org.uk. A workspace file will be found, BIN4.MCW, which will open all the relevant windows automatically (File, Open Workspace).

Note that in previous versions of MPASMWIN there was a limited file path length, so a folder near the root of the drive is desirable. If the file path is too long, an error message may be generated by the assembler, but it will not state the cause explicitly. This is simply a historical limitation of the assembler, but can prevent successful assembly for no obvious reason. This problem seems to have been fixed in the current version.

Once entered or opened in the edit window, the source code can be assembled in MPLAB by selecting 'Quickbuild sourcefile.asm' from the Project menu. The correct processor type must first be selected via the configuration menu, 'select device'. The assembler program (MPASM) takes the source code text and decodes it character by character, line by line, starting at the top left. The corresponding machine code for each line in the source code is generated until the END directive is detected. The binary code created is automatically saved as a file called

BIN4.HEX in the same folder as the source code. At the same time, several other files are also created, some of which are needed for debugging.

In Proteus VSM, the circuit schematic must be created first, and then the program attached to the MCU, by selecting Source, Add/Remove Source Files from the menu. The processor type and assembler must be selected and a New source file created or attached. The program is assembled by selecting Build All in the Source menu, and is automatically reassembled after editing when the simulator is set to run, which makes source code debugging quick and easy. Application creation in Proteus VSM is detailed in Appendix E.

4.6.1. Syntax Errors

If there are any syntax errors in the source code, such as spelling, layout, punctuation or failure to define labels properly, error messages will be generated by the assembler. These will be displayed in a separate window, indicating the type of error and line number. The messages and line numbers must be noted, or the error file, BIN4.ERR, printed out then the necessary changes made to the source code. The error is sometimes on a previous line to the one indicated, and sometimes a single error can generate more than one message. Warnings and information messages can usually be ignored, and can be disabled. There are more details about error messages in Chapter 9.

You may receive the following messages:

```
  Warning[224] C:\MPLAB\BOOKPRGS\BIN4.ASM 65 : Use of this instruction
is not recommended.

  Message[305] C:\MPLAB\BOOKPRGS\BIN4.ASM 81 : Using default
destination of 1 (file).
```

The first warning is caused by the special instruction TRIS, which is not part of the main instruction set. It is a simple way of initializing the port, and there is an alternative method using register bank selection, which is preferred in real applications. This will be introduced later.

The message about the ‘default destination’ is caused by the simplified syntax used in these programs, where the file register is not explicitly specified as the destination in instructions where the result can be placed in either the file register or the working register. The assembler assumes that the file register is the destination by default, and we are taking advantage of this to simplify the source code.

When all errors have been eliminated and the program has been successfully assembled, the machine code can be inspected by selecting ‘View’, ‘Program Memory’. Note that the source code labels are not reproduced, as the program code has been ‘disassembled’ (recreated) from the machine code. That is, the hex file has been converted back to mnemonic form so that it can be checked against the original.

4.6.2. List File

A program 'list file' BIN4.LST is produced by the assembler, which contains the source code, the machine code, error messages and other information all in one listing (Table 4.4). This is useful for analyzing the program and assembler operations, and debugging the source code.

The list file header shows the assembler version used and source file details. The column headings are:

`LOC:`	memory location addresses at which the machine code will be stored
`VALUE:`	the numerical value with which equate labels will be replaced
`OBJECT CODE:`	machine code produced for each instruction
`LINE:`	line number of list file
`SOURCE TEXT:`	source code including comments.

At the end of the list file, additional information is provided:

`SYMBOL TABLE:`	lists all the equate and address labels allocated
`MEMORY USAGE MAP:`	shows the locations occupied by the object code.

Note that no machine code is produced by lines that are occupied by a full line comment. The actual program starts to be produced at line 00041. The machine code for the first instruction is shown in column 2 (3000), and the address where it will be stored in the chip when downloaded is shown in column 1 (0000). The whole program will occupy locations 0000 to 000F (16 instructions).

If we study the machine code, we can see how the labeling works; for example, the last instruction 'GOTO start' is encoded as 2808, and the 08 refers to address 0008 in column 1, the location with the label 'start'. The assembler program has replaced the label with the corresponding numerical address for the jump destination. Similarly, the label 'porta' is replaced with its file register number 05 in the instruction code to test the input, 1C05.

The label values are listed again in the symbol table. These values will be used by the simulator to allow the user to display the simulated registers by label. The amount of program memory used, 16 locations, 0000 to 000F, is shown in graphical format in the memory usage map, and finally a total of errors, warnings and messages given. If there are fatal errors, which prevent successful assembly of the program, the list file will not be produced.

4.7. Program Simulation

The BIN4.HEX file could now be downloaded to the PIC chip and the program executed in hardware. It should run correctly because the program given here is known to be good. However, when a program is first developed, it is quite likely that logical errors will be present.

Table 4.4: BIN4 list file

```
MPASM  5.36                          BIN4.ASM   9-12-2010  16:17:57         PAGE  1


LOC  OBJECT CODE     LINE SOURCE TEXT
  VALUE

                      00001 ;
                      00002 ;       Source File:    BIN4.ASM
                      00003 ;       Author:         M. Bates
                      00004 ;       Date:           15-10-03
                      00005 ;       ................................................
                      00006 ;       Program Description:
                      00007 ;
                      00008 ;       Slow output binary count is stopped, started
                      00009 ;       and reset with push buttons. This version uses a
                      00010 ;       subroutine for the delay....
                      00011 ;
                      00012 ;       Processor:      PIC 16F84
                      00013 ;
                      00014 ;       Hardware:       PIC Demo System
                      00015 ;       Clock:          CR ~100kHz
                      00016 ;       Inputs:         Push Buttons RA0, RA1 (active low)
                      00017 ;       Outputs:        LEDs (active high)
                      00018 ;
                      00019 ;       WDTimer:        Disabled
                      00020 ;       PUTimer:        Enabled
                      00021 ;       Interrupts:     Disabled
                      00022 ;       Code Protect:   Disabled
                      00023 ;
                      00024 ; ************************************************************
                      00025
                      00026 ; Register Label Equates..............................
                      00027
  00000005            00028 porta   EQU     05              ; Port A Data Register
  00000006            00029 portb   EQU     06              ; Port B Data Register
  0000000C            00030 timer   EQU     0C              ; Spare register for delay
                      00031
                      00032 ; Input Bit Label Equates ..............................
                      00033
  00000000            00034 inres   EQU     0               ; 'Reset' input button = RA0
  00000001            00035 inrun   EQU     1               ; 'Run' input button = RA1
                      00036
                      00037 ; ************************************************************
                      00038
                      00039 ; Initialize Port B (Port A defaults to inputs).........
                      00040
0000   3000           00041         MOVLW   b'00000000'     ; Port B Data Direction Code
0001   0066           00042         TRIS    portb           ; Load the DDR code into F86
0002   2807           00043         GOTO    reset
                      00044
                      00045
                      00046 ; 'delay' subroutine .....................................
                      00047
0003   008C           00048 delay   MOVWF   timer           ; Copy W to timer register
0004   0B8C           00049 down    DECFSZ  timer           ; Decrement timer register
0005   2804           00050         GOTO    down            ; and repeat until zero
0006   0008           00051         RETURN                  ; Jump back to main program
                      00052
                      00053
                      00054 ; Start main loop ........................................
                      00055
```

Table 4.4: continued

```
0007   0186           00056 reset   CLRF    portb           ; Clear Port B Data
                      00057
0008   1C05           00058 start   BTFSS   porta,inres     ; Test RA0 input button
0009   2807           00059         GOTO    reset           ; and reset Port B if pressed
000A   1885           00060         BTFSC   porta,inrun     ; Test RA1 input button
000B   2808           00061         GOTO    start           ; and run count if pressed
                      00062
000C   0A86           00063         INCF    portb           ; Increment count at Port B
000D   30FF           00064         MOVLW   0FF             ; Delay count literal
000E   2003           00065         CALL    delay           ; Jump to subroutine 'delay'
                      00066
000F   2808           00067         GOTO    start           ; Repeat main loop always
                      00068         END                     ; Terminate source code


SYMBOL TABLE
  LABEL                             VALUE

__16F84A                          00000001
delay                             00000003
down                              00000004
inres                             00000000
inrun                             00000001
porta                             00000005
portb                             00000006
reset                             00000007
start                             00000008
timer                             0000000C


MEMORY USAGE MAP ('X' = Used,  '-' = Unused)

0000 : XXXXXXXXXXXXXXXX ---------------- ---------------- ----------------

All other memory blocks unused.

Program Memory Words Used:    16
Program Memory Words Free:  1008


Errors   :     0
Warnings :     1 reported,     0 suppressed
Messages :     2 reported,     0 suppressed
```

This means that the program executes but it does not necessarily carry out the right operations in the right order. For example, if the wrong input pin is specified for testing, the button will not be detected. Logical errors can be detected by running the program through a test sequence that exercises all the features of the application. Any operational errors then have to be traced back to the relevant source code section. This process may have to be repeated many times, making it very time consuming.

This is where simulation is useful: it allows the program to be 'run' in a virtual environment on the host PC, as if it were being executed in the chip, but without having to download to the actual hardware. It can then be checked for logical errors and the source code changed and re-tested much more quickly and easily. This will allow most of the logical errors to be

eliminated, leaving maybe a few issues related to the real hardware to be resolved, e.g. input/output timing.

MPLAB provides a source code simulation and debugging for the MCU alone, while Proteus VSM provides a much more user-friendly interactive method, with an animated schematic and virtual signal analysis for the whole circuit (see Appendix E). Simulation allows the effect of the program on the registers and outputs to be checked at critical points. For example, in BIN4 we would check to see that port B has been incremented after the execution of the main loop, because this is the primary function of the program. Figure 4.4 shows a screenshot from MPLAB version 8.60 when simulating BIN4.

MPLAB now provides a logic analyzer, which allows the outputs to be viewed on a time axis, as seen in Figure 3.5 by selecting View, Simulator Logic Analyzer. The 'Channels' button opens a dialogue that allows the output pins RB0 to RB7 to be added to the display.

4.7.1. Single Stepping

Source code debugging allows the program to be tested in the edit window. First, load and assemble the source code file BIN4.ASM. Then enable simulation mode by selecting 'Debugger, Select Tool, MPLAB SIM'. A control panel should appear in the toolbar. Operate

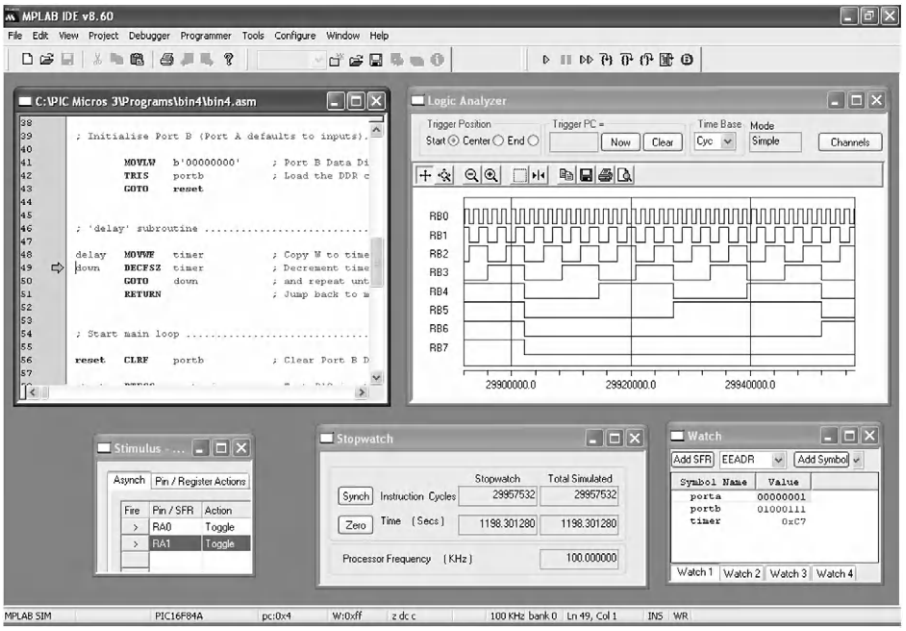


Figure 4.4
MPLAB (Version 8.60) simulation workspace for debugging BIN4

the 'Run' button: nothing appears to happen, but when the 'Halt' button is operated, the current execution point is indicated in the source code window. You may get a message about the 'Watchdog Timer', which is an automatic interrupt that prevents the program getting stuck in a loop. If necessary, open the Configure, Configuration Bits dialogue and turn the watchdog timer off, unchecking the 'Configuration Bits set in code' option. At the same time, ensure that the Power-Up Timer and Code Protection are off. These will be explained later. The program can now be executed one instruction at a time using the 'Step Into' button in the control panel, and the sequence examined. With no inputs, the program should loop through the reset sequence. The program can be restarted from the top at any time by clicking on the 'Reset' button. If Animate is selected from the Debugger menu, execution steps automatically.

4.7.2. Input Simulation

We now need to simulate the action of the push buttons in the hardware that are used to start and stop the output sequence. Select Debugger, Stimulus, New Workbook (Asynch tab). Clicking the first cell in the PIN/SFR column allows an input to be selected from a drop-down menu. Select RA0 in the first row, and RA1 in the second. In the Action column, set both to toggle mode: this will make the input change each time the Fire button is pressed. The inputs can now be operated to allow the program to proceed from the reset loop. Both inputs should be set high initially, simulating the default (inactive) condition in the hardware. Taking RA1 low will allow the main loop to proceed, and toggling RA0 will execute the reset loop. Unfortunately, the state of the input is not indicated in the stimulus table, so file register 05, port A, must be displayed (preferably in binary) to confirm the changes on the inputs.

4.7.3. Register Display

The special function registers can be viewed to check the effect of the program on the output register 06, port B. The changes at the inputs can also be checked, and any intermediate changes in internal registers tracked. Registers can also be displayed selectively using a watch window. The display format can be changed from hex to binary by right-clicking for the register properties, so individual bit status can be checked. Port A bits 0 and 1 should change with the asynchronous stimulus inputs, and port B should display a random binary number after the increment loop has been executed for an arbitrary time.

4.7.4. Step Out, Step Over

The Step Into control will step through all subroutines. Once the program has entered the delay loop, the same simple sequence is being repeated, so single stepping is not so useful. We need either to drop out of the subroutine (Step Out) or to bypass it altogether (Step Over). These commands cause the loop to be executed at full speed, with single stepping being resumed after

the RETURN instruction. Once a subroutine is correct, it can be stepped over when debugging the rest of the program.

4.7.5. Breakpoints

Another technique for executing selected parts of the program at full speed is the breakpoint. For example, if part of a large program is known to be correct, we will want to run through that section at full speed and start single stepping at a later point in the program. In BIN4, a breakpoint can be set at the start of the main loop so that it executes once, then stops so we can inspect the registers. A breakpoint can be set by simply double-clicking in the line number margin of the source code window. The program can then be run from the start, and it will stop at the breakpoint. Run again, and a complete loop will be executed at full speed and port B should be seen to increment by one.

4.7.6. Stopwatch

The program timing can be checked using the stopwatch feature. This displays the total number of instructions executed and the time elapsed, calculated from the simulated processor clock frequency. For BIN4, RC oscillator should be selected in the configuration dialogue. The clock speed is set in the Debugger, Settings dialogue, Osc/Trace tab. In this case, set the processor frequency to 100 kHz. Then run the program to the breakpoint at 'start', zero the clock and run again. The stopwatch will display the total time for one cycle. The frequency of the output can be predicted from this measurement. Two program loop cycles will cause the low-output bit RB0 to be toggled up and down once, giving one full output cycle. Therefore, we can double the loop time to give the output period, and calculate the reciprocal to give the frequency at RB0.

From the stopwatch readings:

Number of instructions executed per loop = 777
Processor frequency = 100 kHz
Loop time = 31.08 ms

Therefore:

Output period at RB0 $= 2 \times 31.08$ = 62.16 ms
Output frequency at RB0 = 1/0.06216 = 16.1 Hz

This shows that changes in the higher order output bits will be clearly visible using this clock frequency with the maximum delay loop count (FF). The frequency at RB1 will be about 8 Hz, RB2 4 Hz, RB3 2 Hz and so on, with RB7 flashing about once every 8 s. By adjusting the count value, and inserting NOP (no operation) instructions, an exact set of frequencies can

be obtained. Using a crystal clock or calibrated internal clock will make the timing more accurate. More information about using MPLAB for debugging is given in Chapter 9.

4.8. Program Downloading

After testing in the simulator for correct operation, the machine code program can be blown into the flash memory on the chip. The program is downloaded via a serial link into a specific pin, RB7 in the case of the 16F84A. There are two methods for program downloading, outlined below.

4.8.1. Programming Unit

The original programming method for PIC chips required the chip to programmed before fitting in the finished hardware circuit. A programming unit was plugged into the serial port of the PC (COM1 or COM2) and the chip inserted into the zero insertion force (ZIF) socket on the programmer (Figure 4.5). The chip orientation had to be carefully checked, as inversion would reverse the supply pins and burn out the chip. Antistatic precautions had to be observed, since the PIC is a complementary metal oxide semiconductor (CMOS) device, and static discharge on a pin can break down the field effect transistor (FET) gate insulation in the internal circuitry. Before downloading, the correct device must be selected in MPLAB under device specifications, and the configurations bits set as described below. These can also be set up using an assembler directive in the source code.

Oscillator (Clock): RC

The main options in the earlier chips were 'RC' and 'XT'. RC must be selected for the oscillator configuration used in the BIN hardware. XT will be selected in later applications

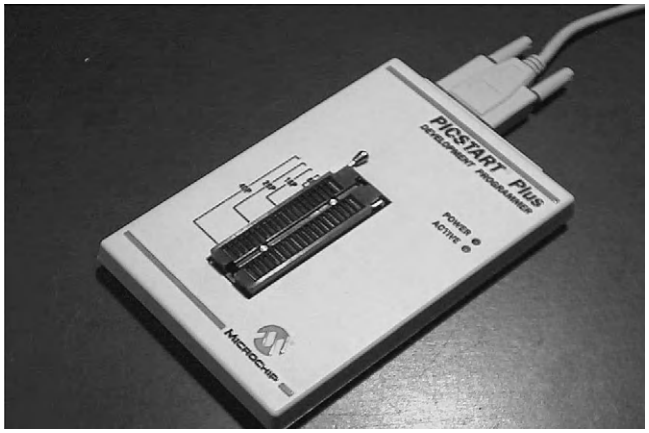


Figure 4.5
PIC programming unit

using an external crystal oscillator. The program will not run in the hardware if the wrong type of oscillator has been selected, so check this carefully. More recently introduced chips now have a third option, an internal oscillator that eliminates the need for external components to control the clock. This is now generally the preferred option.

Watchdog Timer: Off

The watchdog timer (WDT) is an internal timer, which automatically restarts the program if it is not cleared back to zero within 18 ms, using the instruction CLRWDT. This can be used to stop the controller hanging, owing to an undetected program bug or an input condition that has not been predicted in testing. For applications not using this feature, WDT must be switched off, or the program will reset repeatedly, preventing normal operation.

Power-up Timer: On

Mains-derived power supplies may take some time to reach the correct value (+5 V) when first switched on. The power-up timer (PuT) is an internal timer which delays the start of program execution until the power supply is at the correct voltage and is stable. This helps to ensure that the program starts correctly every time. It is not relevant in simulation mode, but should be enabled when the program is downloaded for hardware operation.

Code Protect: Off

If the code protect (CP) bit is enabled, the program cannot be read back into MPLAB and copied or manipulated. This is normally only necessary for commercial applications to prevent software piracy, so we can switch off code protection for our simple examples.

For program downloading, PICSTART would be selected from the Programmer menu. If the programming unit had been correctly connected, a programming dialogue was displayed, with the hex code to be downloaded visible (Figure 4.6; note that this illustration dates back to version 5 of MPLAB). When the configuration bits had been checked, the Program operation could be selected to download the machine code. When complete, confirmation of successful programming should be received, and the chip manually transferred to the application circuit, again observing antistatic precautions.

4.8.2. In-Circuit Programming and Debugging

In-circuit programming is now usually the preferred downloading method, where the chip is left in-circuit after assembly of the printed circuit board (PCB), allowing programming and final debugging in the final hardware. When a board is produced in volume, the programming can be done as the final stage when the hardware is complete. This is also very useful at the prototyping stage, as program modifications can be more rapidly and safely implemented and tested.

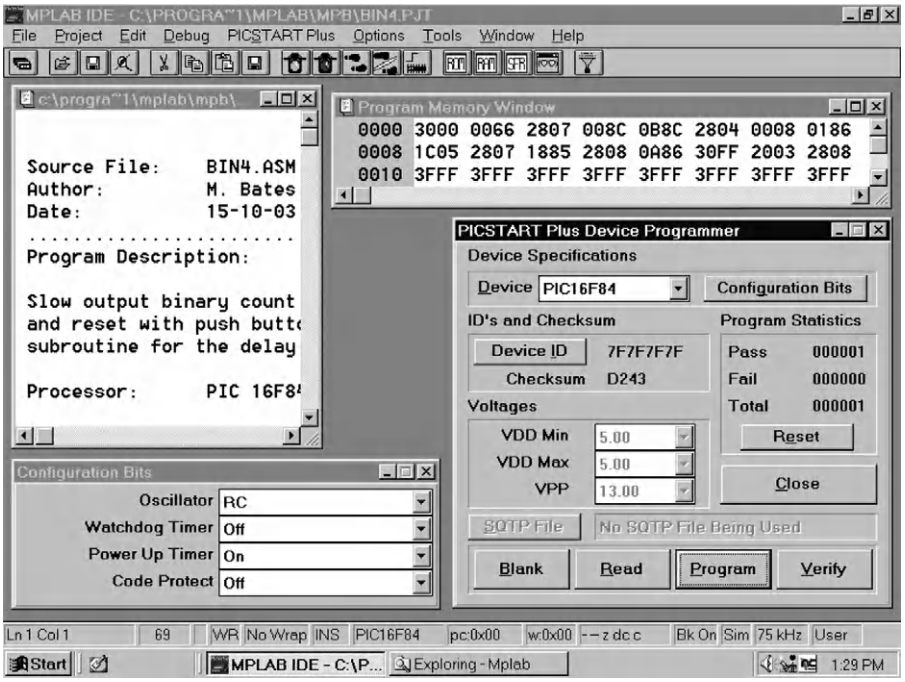


Figure 4.6
MPLAB (Version 5) program downloading windows

To facilitate in-circuit programming and debugging (ICD), the board is designed with a six-pin connector, which connects the programming pins on the chip to a programmer/debugger module that is in turn connected to the universal serial bus (USB) output on the host PC. This system has already been illustrated in Chapter 1 (Figure 1.11), and will explained in more detail in Chapter 7 by examining some Microchip demonstration systems.

When the hardware has been connected, the appropriate programmer/debugger is selected from the programmer menu. PICkit2/3 is the low-cost option, which nevertheless offers some very useful features. With the connection confirmed, the program can be downloaded. The same module can then be selected from the debugger menu and the program run and debugged using the same tools as used in software simulation (single step, breakpoints, etc.). However, program execution now takes place in the chip itself. The interaction with the real hardware can be monitored, and hence the hardware and software verified at the same time. Note that not all chips support in-circuit debugging, especially smaller and older chips. Neither the 16F84A nor the 16F690 supports ICD without a special header, which must be connected between the debugger and the chip. The PIC 16F887 used later on the Microchip 44-pin demo board does support ICD.

When debugging is complete, the final step is to configure the chip to run independently by selecting the Release option in the drop-down menu in the project toolbar. This allows the board to run when disconnected from the debugging module.

Thus, the software can be tested and debugged in stages: initially in MPSIM or Proteus VSM, and finally in the actual hardware. Proteus also provides fully featured PCB design as well as circuit simulation, so the finished hardware can be produced using the same package.

4.9. Program Testing

Preliminary hardware inspection and testing is important if the circuit is newly constructed, and essential if it is a new design. First, have a good look at the board to check that the correct components have been fitted in the correct places and orientation, there are no dry joints or solder bridges and so on. Connect the power *before* fitting the chip(s) if they are in sockets, and check that the supply voltages are correct, and on the right pins. When the hardware has been thoroughly checked, switch on and check that the MCU is not overheating.

In a commercial product, a test schedule must be devised and correct operation confirmed and recorded. The test procedure should check all possible input sequences, not just the correct ones, if the design is to be completely reliable. It is quite difficult to be sure that complex programs are 100% testable, as it is often not easy to predict every possible operating sequence. An outline test procedure for BIN4 is suggested in Table 4.5.

The program should start immediately on power-up. If it does not function correctly when tested against the original specification, a fault-finding process needs to be followed, as outlined below.

Table 4.5: Basic test schedule for BIN4

	Test	Correct operation	Checked
1	Inspection: check PCB and components	Visual: correct value, orientation and connections	
2	Connect 5 V power supply	Visual + multimeter: chip power supplies OK	
3	Check and adjust clock frequency	Oscilloscope or frequency meter: 100 kHz	
4	Press RUN	Count on LEDs	
5	Release RUN	LED count halted	
6	Press and release RESET	LEDs off	
7	Press RUN	Count on LEDs from zero	
	Signed and dated	Name	Date

1. *Hardware checks:*
   (a) *+5 V on MCLR, V_{DD}, 0 V on V_{SS}*
   (b) *Clock signal on CLKIN*
   (c) *Input changes on RB0, RB1.*
2. *Software checks:*
   (a) *Simulation correct*
   (b) *Correct clock selected*
   (c) *WDT off, PuT on, CP off*
   (d) *Program verified.*

More suggestions on system testing are given in Chapter 9.

Questions 4

1. Place the following program development steps in the correct order: (a) Download hex file; (b) Assemble source code; (c) Edit source code; (d) Test in hardware; (e) Design program; (f) Simulate in software. (6)
2. State the file extension for the following files, and describe their function: Source code; Machine code; List file. (6)
3. State two advantages of using subroutines. (4)
4. State an instruction for making a conditional jump in PIC assembler code. (2)
5. State an instruction which must precede a TRIS instruction to make the port pins all outputs. (2)
6. How could you halve the delay time in BIN4? (2)
7. Explain how a switched input to the PIC 16F84A can be simulated in MPLAB asynchronously. (4)
8. State the configuration settings that should be selected when downloading to the BIN hardware. (4)

Answers on page 419–20. (Total 30 marks)

Activities 4

1. Download the supporting documentation for MPLAB from www.microchip.com. Study the tutorial in the User's Guide and the help files supplied with MPLAB as necessary to familiarize yourself with editing, assembling and simulating an application program. Start up MPLAB, create a source code file for BIN3, and enter the assembler code program, leaving out the comments. Assemble, correct any errors and simulate. Check that the port B (F6) file register operates as required.
2. Modify the program as BIN4, and confirm that its operation is essentially the same.
3. Modify the program to scan the output, that is, move one lit LED along the display, repeating indefinitely, using a rotate instruction.

4. Construct a prototype circuit, download BIN4 and test to the schedule given in Table 4.5. Refer forward to Chapter 10 if necessary.
5. If Proteus VSM is available, enter or download the schematic and source code for application BIN4 from www.picmicros.org.uk. Assemble the source code, attach the hex file to the MCU and check that the simulation works (LEDs flash). Adjust the program to give an output period of exactly 50 ms at RB0. Use NOPs in the code if necessary. Use the virtual oscilloscope and timer counter to check the output period, and the logic analyzer to display all the outputs simultaneously.

CHAPTER 5

PIC Architecture

Chapter Outline

Chapter Points

- The 16XFXXX mid-range microcontroller family is represented by a block diagram.
- Its Harvard architecture uses a RISC instruction set, execution pipeline and flash program memory.
- A fixed-length 14-bit instruction contains the operation code and operands.
- The MCU has a working register, program counter, stack and instruction register, with decoding, timing and control logic.
- Special function registers have dedicated functions.
- RAM general purpose registers provide temporary data storage.

PIC Microcontrollers. DOI: 10.1016/B978-0-08-096911-4.10005-9

An overview of microcontroller (MCU) principles has been provided in Part 1. We now need to look at the PIC® internal hardware in more detail. We will use the 16F84A chip as a reference, despite its partial obsolescence, since it has all the essential elements found on all the current chips in the family without complicating features such as analogue inputs and serial ports. It is also currently included in the low-cost Proteus VSM Starter Kit for microcontroller circuit simulation. All members of the PIC family are based on the same core architecture, with each having a different combination of memory and peripheral features.

An essential reference is the PIC 16F84A data sheet, especially Figure 5.1, the block diagram of the internal architecture. A more complete picture is provided in the 'PIC Mid Range Reference Manual' (Figure 4-2), which includes all the peripheral options for the 16 series chips. For an explanation of the main blocks in the internal architecture, refer to Appendix C,

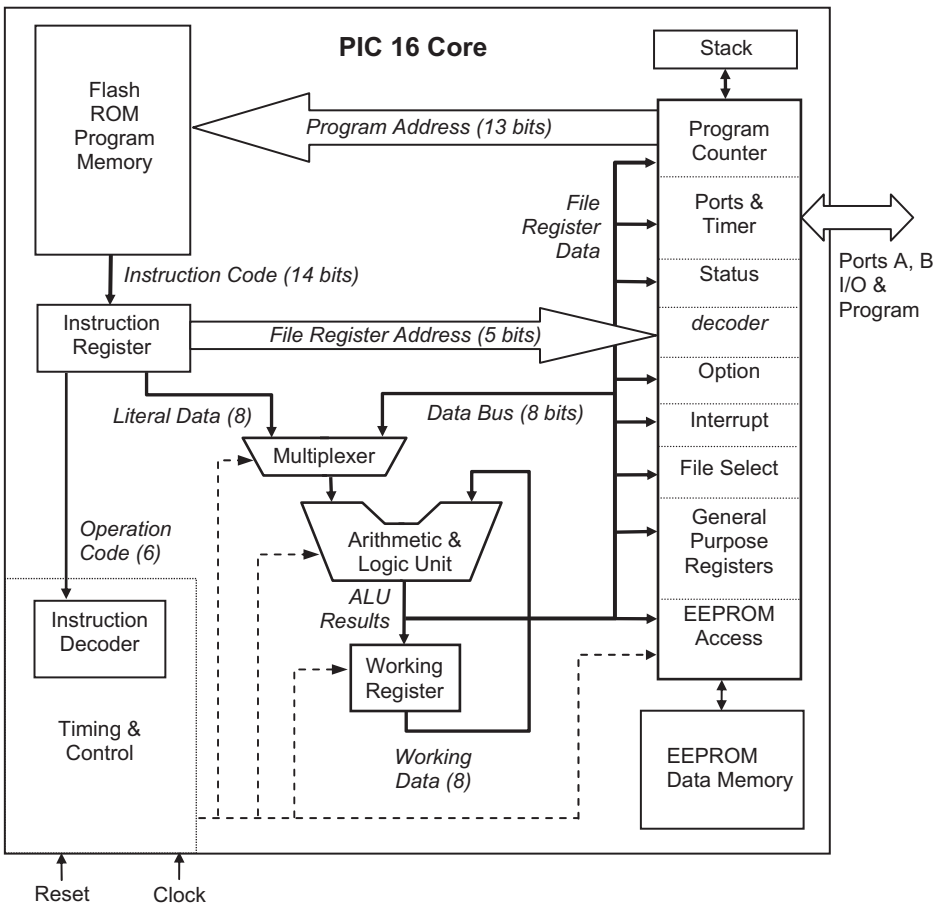


Figure 5.1
PIC 16F84A simplified internal architecture

which describes how the registers, arithmetic and logic unit, multiplexer, decoder, program counter and memory work. Later, we will move on to the 16F690, which has a good range of peripherals and is used in the Microchip LPC demo board. All data sheets and reference manuals can be downloaded from www.microchip.com.

5.1. Block Diagram

A somewhat simplified internal architecture (Figure 5.1) has been derived from the block diagram given in the data sheet. Some features seen in the manufacturer's diagram have been left out because they are not important at this stage. The functional blocks of the chip are shown, with the main address paths identified. The bit width of each parallel path is indicated, with the 8-bit data bus connected to all the main registers. The timing and control block has control connections to all other blocks, which determine the processor operation at any point in time, but they are not all shown explicitly in order to keep the diagram as clear as possible.

The file register set contains various control and status registers, as well as the port registers and the program counter. The most commonly used are the ports (PORTA, PORTB), status register (STATUS), real-time clock counter (TMR0) and interrupt control (INTCON). There is also a number of spare general purpose registers (GPRx), which can be used as data registers, counters and so on. The file registers are numbered 00 to 4F, but are usually given labels in the program source code. File registers also give access to a block of EEPROM, which provides non-volatile data memory.

5.1.1. Clock and Reset

A clock circuit is connected to the timing and control block to drive all the operations of the chip. For applications where precise timing is not required, a simple external resistor and capacitor network controls the frequency of the internal oscillator. Relatively low frequencies are generated (< 1 MHz) with an RC clock. For more precise timing, a crystal oscillator is used (see data sheet Figure 6-7); a convenient frequency is 4 MHz, because each instruction takes four clock cycles to execute, that is, 1 μs. The exact program execution timing can then be more easily calculated, and the hardware timer used for accurate signal generation and measurement. With the high-speed oscillator option selected, the processor can be clocked at up to 20 MHz, giving a minimum 200 ns instruction execution period, and a maximum instruction execution rate of 5 mips (millions of instructions per second). Most current chips also include an internal oscillator that runs at frequencies between 32 kHz and 32 MHz; this is now the default option as it eliminates the external clock components.

In earlier processors, an external reset circuit was often needed to ensure a smooth start-up. Now, the timing and control circuits contain start-up timers, which means that the reset input !MCLR can simply be connected to V_{DD}, the positive supply (normally via a precautionary

resistor), to enable the processor. An external reset button (with a pull-up resistor) or control signal can still be connected to !MCLR if an external restart might be required. The MCU program can then be restarted by pulsing the reset input low. In most of our simple applications, the power-on reset will be used.

5.1.2. Harvard Architecture

It can be seen in the block diagrams that the memory and file register address lines are separate from the data paths within the processor. This is referred to as Harvard architecture; it improves the speed of processor operation because data and addresses do not have to share the same bus lines. The reduced size of the instruction set also speeds up decoding and the short data path length in a single chip design reduces data transmission time. The program execution hardware also uses a 'pipeline' arrangement; as one instruction is executed, the next is being fetched from program memory, overlapping instruction processing and thus doubling the overall execution rate. All these features contribute to a high speed of operation, compared with traditional microprocessors that use a conventional (Von Neumann) architecture, in which the program and data share the same data bus and memory space.

5.2. Program Execution

The program consists of a sequence of 14-bit codes, which contain both the operation code and operand in a fixed length instruction. This machine code program is derived from a source code program created as a text file on a host PC, assembled and downloaded, as detailed in Chapter 4. At the moment we are not too concerned about exactly how the downloading is carried out; all we need to know for now is that the program is received in serial form from the host PC via an input/output (I/O) port pin (RB7/PGD/ICSPDAT). The data transfer is controlled by a data clock signal on an adjacent pin (RB6/PGC/ICSPCLK), while a high programming voltage (14 V) is applied to !MCLR. A low-voltage programming option is also available, which eliminates the need for this high programming voltage. In-circuit serial programming (ICSP) can be carried out while the chip remains in circuit; the programmer module is then connected via a six-pin connector on the application board (see Figure 1.9). The PIC program memory is usually implemented as flash read memory (ROM), so an existing program can be replaced by simply overwriting with a new version.

5.2.1. Program Memory

The program counter register holds the current address, reset automatically to 0000 when the chip is powered up or reset. The user program must therefore start at address 0000, but the first instruction is often GOTO the start of the program at some other labeled address. This is necessary when using interrupts, as we shall see later, because the interrupt service routine

(GOTO ISR) must be placed at address 0004. For now, we can place the start of the program at address zero.

The program counter consists of a pair of 8-bit registers. The program counter low (PCL) register holds the low byte of the current address, and the program counter high (PCLATH) register holds the high byte. In the original PIC 16 specification, the current address is fed to program memory from the program counter via a 13-bit address bus, so the high bits (5, 6 and 7) of PCLATH are unused, and the maximum program size accessible is $2^{13} = 8\,\text{k}$ (see Appendix C).

Program memory capacity has been extended in more recent chips, using additional bits of PCLATH and a wider address bus. A maximum 64 k program instruction is possible with a 16-bit address. In normal operation, the PCL is incremented during each instruction cycle, and PCLATH incremented when PCL overflows (i.e. when PCL rolls over from 255 to 0). The memory space is therefore divided into pages of 256 instructions, the range addressed by the 8 bits of PCL. See Appendix C for details of memory sizing.

5.2.2. Instruction Execution

The program execution section of the MCU contains the instruction register, instruction decoder, and timing and control logic. The 14-bit instructions stored in program memory are copied to the instruction register for decoding; each instruction contains both the operation code and operand. The instruction decoder logic converts the op-code bits into settings for all the internal control lines. The operand provides a literal, file register address or program address, which will be used by the instruction.

If, for example, the instruction is MOVLW (Move a Literal into W), the control lines will be set up to feed the literal operand to W via literal data bus to the multiplexer and ALU. If the instruction is MOVWF, the control lines will be set up to copy the contents of W to the specified file register via the internal data bus. The operand will be the address of the file register (00 to 4F) required. If we look at the 'move' instruction codes quoted in the instruction set, we can see the difference in the code structure for the three move instructions:

```
MOVLW  k        =     11  00xx  kkkk  kkkk

MOVWF  f        =     00  0000  1fff  ffff

MOVF   f,d      =     00  1000  dfff  ffff
```

In the MOVLW instruction, the operation code is the high 4 bits (1100), 'x' are 'don't care' bits, and 'k' represents the literal bits, the low byte of the instruction. In the MOVWF instruction, the operation code is 0000001 (7 bits) and 'f' bits specify the file register address. Only 7 bits are used for the register address, allowing a maximum of $2^7 = 128$ registers to be addressed. In the MOVF instruction the operation code is 001000, and the file register address

is needed as before to identify the data source register. Bit 7 (d) controls the data destination. This bit must be 0 to direct the data into W, the usual operation. For example, to move an 8-bit data word from file register 0C to W requires the syntax MOVF 0C,W.

5.2.3. Data Processing

The arithmetic and logic unit (ALU) can add, subtract or carry out logical operations on single data bytes or pairs of numbers (see Chapter 2). These operations are carried out in conjunction with the data multiplexer and working register. The multiplexer allows new data to be fed from the instruction (a literal) or a register. This may be combined with data from W, or register data manipulated in a single register operation. W is used in register pair operations as a temporary data source or store, but the final result must usually be copied back into a file register, since W may be needed for the next operation.

5.2.4. Jump Instructions

If a GOTO instruction is executed, the program counter will be loaded with the program memory address of the jump destination given as the instruction operand. The program label used in the source code will have been replaced by the destination address by the assembler. For conditional branching (making decisions), any file register bit can be tested by a 'Bit Test & Skip' instruction, which is then followed by a GOTO or CALL instruction.

When a CALL instruction is executed, the destination address is loaded into the PC in the same way as for the GOTO, but in addition, the address following the CALL is stored on the stack, the return address. The subroutine is then executed until a RETURN instruction is encountered. At this point, the return address is automatically pulled from the stack and replaced in the PC, allowing program execution to pass back to the original point. The stack works on a last in, first out (LIFO) basis, with the last address stored being the first to be recovered. In conventional processors, the stack can be modified directly as it is located in the main memory, but in the PIC16 this not possible.

5.3. File Register Set

All the file registers are 8 bits wide. They are divided into two main blocks: the special function registers (SFRs), which are reserved for specific purposes, and the general purpose registers (GPRs), which can be used for temporary storage of any data byte. The basic file register set (16F84A) is shown in Figure 5.2.

The registers in Bank 0 (file addresses 00−4F) can be directly addressed, and it is suggested that the register labels given in Figure 5.2, which match the data sheet, are used

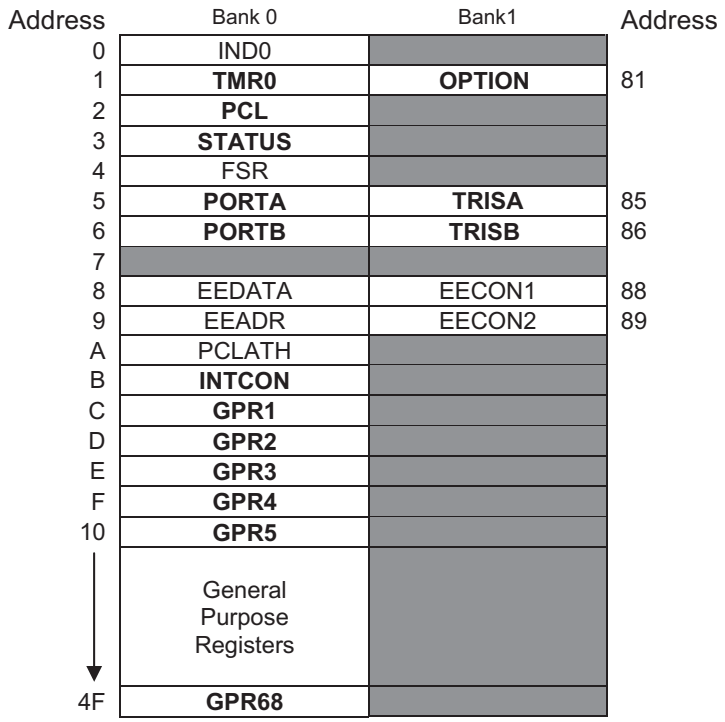


Figure 5.2
PIC basic file register set (16F84A)

as the register labels. These labels are also used by default in MPLAB, and standard header files can be included in your programs, which define all the register names using these labels.

Special instructions are available to access the Bank 1 registers. For simplicity, we have already used the instruction TRIS to access the data direction registers TRISA and TRISB. Similarly, we will use the instruction OPTION to access the option register, which will be used later to set up the hardware timer. Alternatively, a register bank select bit in the status register can be used to access Bank 1 file registers. This operation is most easily implemented by using a special instruction BANKSEL (see Program 1.1); this is the preferred method once the basics of programming have been established.

5.3.1. Special Function Registers

The operation of the 16F84A SFRs is summarized below, describing the operation of those that are used most frequently. The functions of all the registers are detailed in each chip data sheet. The shaded registers in Figure 5.2 either do not exist, or are repeated at addresses 80–CF (page 1).

```
PCL             Program Counter Low Byte
                File Register Number = 02
```

The program counter contains the address of (points to) the instruction currently being executed, and counts from 000 to 3FF, unless there is a jump (GOTO or CALL). The PCL register contains only the low 8 bits (00−FF) of the whole program counter, with the high stored in the PCLATH register (address 0A). We only need to worry about the high bits if the program is longer than 255 instructions in total, which is not the case for any of the demonstration programs, and then only if the program counter is being modified directly. The PC is automatically incremented during the instruction execution cycle, or the contents replaced for a jump.

```
PORTA           Port A Data Register
                File Register Number = 05
```

Port A has five I/O bits, RA0−RA4. Before use, the data direction for each pin must be set up by loading the TRISA register with a data direction code (see below). If a bit is set to output, data moved to this register appears at the output pins of the chip. If set as input, data presented to the pins can be acted on immediately, or stored for later use by moving the data to a spare register. Examples of this have already been seen in earlier chapters. In the 16F84A, RA4 can alternatively be used as an input to the counter timer register (TMR0) for counting applications. The use of the hardware timer will be covered in Chapter 6. The PORTA register bit allocation is shown in Table 5.1. In other PIC chips, most port pins will have at least two different functions, selected by setting up the relevant SFRs.

All registers are read and written as an 8-bit word, so we sometimes need to know what will happen with unused bits. When the port A data register is read within a program (using MOVF), the 3 unused bits will be seen as ‘0’. When writing to the port, the high 3 bits are simply ignored. An equivalent circuit for each port pin is given in the 16F84A data sheet, Section 5. The components of this block diagram are explained in Appendix B.

```
TRISA           Port A Data Direction Register
                File Register Number = 85
```

The data direction of the port pins can be set bit by bit by loading this register with a suitable binary code, or the hex equivalent. A ‘1’ sets the corresponding port bit to input, while a ‘0’ sets it to output. Thus, to select all bits as inputs, the data direction code is 1111 1111 (FFh), and for all outputs is 0000 0000 (00h). Any combination of inputs and outputs can be set by loading the TRIS register with the appropriate binary code.

When the chip first is powered up, these bits default to ‘1’, so it is not necessary to initialize for input, only for output. This makes sense, because if the pin is incorrectly wired up, it is more

Table 5.1: Port bit functions (16F84A)

Register Bit	Pin Label	Function
	Port A	
0	RA0	Input or Output
1	RA1	Input or Output
2	RA2	Input or Output
3	RA3	Input or Output
4	RA4/T0CKI	Input or Output or Input to TMR0
5	—	None (read as zero)
6	—	None (read as zero)
7	—	None (read as zero)
	Port B	
0	RB0/INT	Output or Input or Interrupt Input
1	RB1	Output or Input
2	RB2	Output or Input
3	RB3	Output or Input
4	RB4	Output or Input + Interrupt on change
5	RB5	Output or Input + Interrupt on change
6	RB6	Output or Input + Interrupt on change
7	RB7	Output or Input + Interrupt on change

easily damaged if set to output. For instance, if the pin is accidentally grounded, and then driven to a high state by the program, the short-circuit current is likely to damage the output circuit. If the pin is set as an input, no damage will be done.

The data direction register TRISA is loaded by placing the required code in W and then using the instruction TRIS 05 or TRIS 06 for port A and port B, respectively. Alternatively, all file registers with addresses 80—CF can be addressed directly, using the BANKSEL command, and this option will be used in later programs.

```
PORTB           Port  B  Data  Register

                File  Register  Number  =  06
```

Port B has the full set of eight I/O bits, RB0—RB7. If a bit is set to output, data moved to this register appears at the output pins of the chip. If set as input, data presented to the pins can be read at this address. The data direction is set in TRISB, using the TRIS or BANKSEL command, and all bits default to input on power up. The PORTB register bit allocation is shown in Table 5.1.

Bit 0 of port B has an alternative function; it can be initialized, using the interrupt control register (INTCON), to allow the processor to respond to a change at this input with an interrupt sequence. In this case, the processor is forced to jump to a predefined interrupt service routine

Table 5.2: STATUS register bit functions

Bit	Label	Name	Function
0	C	Carry Flag	Set if register operation causes a carry-out of bit 8 of the result (8-bit operations)
1	DC	Digit Carry Flag	Set if register operation causes a carry-out of bit 3 of the result (4-bit operations)
2	Z	Zero Flag	Set if the result of a register operation is zero
3	PD	Power Down	Cleared when the processor is in sleep mode
4	TO	Time Out	Cleared when watchdog timer times out
5	RP0	Register Bank	RP0 selects file registers 00—7F or 80—FF
6	RP1	Select Bits	RP1 not used in '84 chip
7	IRP		IRP not used in '84 chip

(ISR) upon completion of the current instruction (see Section 6.3). The processor can also be initialized to provide the same response to a change on any of the bits RB4—RB7.

```
TRISB           Port  B  Data  Direction  Register

                File  Register  Number  =  86
```

As in port A, the data direction can be set bit by bit by loading this register with a suitable binary code, or the hex equivalent, where '1' (default) sets an input and '0' sets an output (initialization required). The command TRIS 06 moves the data direction code from W to TRISB register; the command BANKSEL allows direct access to bank 1 and the data direction registers.

```
STATUS          Status  (or  Flag)  register

                File  Register  Number  =  03
```

Individual bits in the status register record information about the result of the previous instruction. Probably the most commonly used is the zero flag, bit 2; when the result of any operation is zero, this zero flag bit is set to '1'. It is used by the Decrement/Increment and Skip if Zero instructions, and can be used by the Bit Test & Skip instructions, to implement conditional branching of the program flow. The status register bit functions are shown in Table 5.2, where the function of the other bits is indicated.

```
TMR0            Timer  Zero  Register

                File  Register  Number  =  01
```

A timer/counter register counts the number of pulses applied to a clock input; the binary count can be read from the register when the count is finished. TMR0, an 8-bit register, can count up to 255 pulses. For external inputs, the pulses are applied at pin RA4. When used as a timer, the internal instruction clock is used to supply the pulses. If the processor clock frequency is

Table 5.3: OPTION register bit functions

Bit	Label	Name	Function
0	PS0	Prescaler Rate Select Bit 0	3-bit code to select one of eight prescale values
1	PS1	Prescaler Rate Select Bit 1	
2	PS2	Prescaler Rate Select Bit 2	
3	PSA	Prescaler Assignment	Assigns prescaler to WTD or TMR0
4	T0SE	Timer Zero Source Edge Select	Select rising or falling edge trigger for T0CKI input at RA4
5	T0CS	Timer Zero Clock Source Select	Selects timer/counter input as RA4 or internal clock
6	INTEDG	Interrupt Edge Select	Selects rising or falling edge trigger for RB0 interrupt input
7	RBPU	Port B Pull-up Enable	Enables pull-ups on port B pins so input data defaults to '1'

known, the time taken to reach a given count can be calculated. When the counter rolls over from FF to 00, an interrupt flag (see INTCON below) is set, if enabled. This allows the processor to check if the count is complete, or to be alerted via an interrupt when a set time interval has elapsed. The timer register can be read and written directly, so a count can be started at a preset value to generate a known interval. The 'Timer Zero' label refers to the fact that other PICs have more than one timer/counter register, but the 16F84A has only the one. More details on using the TMR0 are given in Chapter 6.

```
OPTION          Option  Register

                File  Register  Number  =  81
```

Table 5.3 details the option register bit functions. The TMRO counter/timer operation is controlled by bits 0–5. When used as a timer (T0CS = 0), the processor instruction clock signal increments the counter register. Prescaling can be selected (PSA = 0) to increase the maximum time interval. Bits PS2, PS1 and PS0 control the prescale factor, which can be set to divide the clock frequency by 2 (000), 4 (001), 8 (010), 16 (011), 32 (100), 64 (101), 128 (110) or 256 (111). If the ÷ 256 option is selected, the maximum count will be (256 × 256) − 1 = 65 535 cycles. As is the case with the TRISA and TRISB registers, the option register is accessed using a special instruction, OPTION, or by bank selection. There is more on using the timer in the next chapter. The option register is labeled OPTION_REG in more recent processors, to avoid confusion.

```
INTCON          Interrupt  Control  Register

                File  Register  Number  =  0B
```

The INTCON bit functions are given in Table 5.4. An interrupt is a signal that causes the current program execution to be suspended, and an ISR carried out. An interrupt can be generated by an external device, via port B, or from the timer. In all cases, the ISR must start at address 004 in the program memory. If interrupts are in use, an unconditional jump from address zero, the program start address, to a higher start address, is needed. The INTCON register contains three interrupt flags and five interrupt enable bits, and these must be set up as

Table 5.4: Interrupt control register (INTCON) bit functions

Bit	Label	Name	Function
0	RBIF	Port B Change Interrupt Flag	Set when any one of RB4–RB7 changes state
1	INTF	RB0 Interrupt Flag	Set when RB0 detects interrupt input
2	T0IF	Timer Overflow Interrupt Flag	Set when timer TMR0 rolls over from FF to 00
3	RBIE	Port B Change Interrupt Enable	Set to enable port B change interrupt
4	INTE	RB0 Interrupt Enable	Set to enable RB0 interrupt
5	T0IE	Timer Overflow Interrupt Enable	Set to enable timer overflow interrupt
6	EEIE	EEPROM Write Interrupt Enable	Set to enable interrupt on completion of write
7	GIE	Global Interrupt Enable	Enable all interrupts which have been selected

required during the program initialization by writing a suitable code to the INTCON register. Program 6.2 in Chapter 6 demonstrates the use of interrupts. The precise function of the INTCON bits varies in different PIC chips.

Other SFRs

More SFRs are listed in Table 5.5. EEDATA, EEADR, EECON1 and EECON2 are used to access the non-volatile ROM data area. PCLATH acts as a holding register for the high bits (12:8) of the program counter. The file select register (FSR) acts as a pointer to the file registers. It is used with IND0, which provides indirect access to the file register selected by FSR. This is useful for a block read or write to the GPRs, for example, for saving a set of data read in at a port at intervals. More information on this is given in Chapter 6, Section 6.4.3. Larger chips have more SFRs, which are required to control the additional peripheral blocks: timers, analogue converters, serial ports and so on. These will be described later.

5.3.2. General Purpose Registers

The GPRs are numbered 0C–4F in the 16F84A, 68 in all, and larger chips have more. They are also referred to as random access memory (RAM) registers, because they can be used as a small block of static RAM for storing blocks of data. We have already seen an example of using the

Table 5.5: Selected special function registers

SFR	Name	Function
00	INDF	File Register Memory indirect addressing
04	FSR	for block access
0A	PCLATH	Program Counter High Byte
08	EEDATA	Data EEPROM indirect addressing
09	EEADR	for block access
88	EECON1	Data EEPROM Read & Write Control
89	EECON2	

GPR1 (address 0C) as a counter register in a delay loop. The register was labeled 'timer', preloaded with a value and decremented until it reached zero. This is a common type of operation, and not only used for timing loops. For example, a counting loop can be used for repeating an output operation a certain number of times, such as when performing multiplication by successive addition. We could have used any of the GPRs for this function because they are all operationally identical. When using more than one, a different label is needed for each, declared using the EQU directive.

Questions 5

With reference to the PIC16 family of mid-range microcontrollers:

1. Describe the function of the following blocks within the MCU: program memory, program counter, instruction decoder, multiplexer, W. (10)
2. Why is it not necessary to initialize a PIC port for input? (2)
3. State the main functions of the ALU, and the three sources of its data input. (5)
4. Why is the stack needed for subroutine execution? (2)
5. State the function of the following PIC file registers: PORTA, TRISA, TMR0, PCLATH, GPR1. (5)
6. State the function of the register bits: STATUS,2; INTCON,1; OPTION,5. (6)

Answers on page 420. (Total 30 marks)

Activities 5

1. A trace table is shown below. By referring to the PIC 16F84A instruction set given in the data sheet, complete the trace table to show the binary code present on the internal data connections and in the registers during or after each instruction cycle while the program BIN1 (refer to Chapter 3) is executed. Copy this table and complete the additional columns to the right for each of the remaining four instructions. The first is given as a guide. You need to separate each instruction into op-code and operand, and work out the destination for each.

Activity 1: Trace table

Instruction no.	**1**	**2**	**3**	**4**	**5**
Address	0000				
Instruction	MOVLW 00				
Machine code	3000				
Program address bus (13 bits)	0 0000 0000 0000				
File register address (5 bits)	X XXXX				
Instruction code register (8 bits)	0011 0000				
Literal bus (8 bits)	0000 0000				

(*Continued*)

Activity 1: Continued

Instruction no.	1	2	3	4	5
Data bus (8 bits)	XXXX XXXX				
Working register (8 bits)	0000 0000				
Port B data register (8 bits)	XXXX XXXX				
Port B data direction register (8 bits)	XXXX XXXX				

X: don't know/don't care.

2. In the PIC 16F84A data sheet, Section 4.0, a block diagram of the internal circuit connected to pin RA0 is shown. Refer to Appendices B and C if necessary, and complete the tasks below. The FETs at the output form a complementary pair of switches, a P-type and an N-type. The PFET is on when its gate is low. The NFET is on when its gate is high. For the port pin to operate as an output, the TRIS latch is loaded with the data direction bit 0, and the data is loaded into the data latch from the internal data bus.
   (a) Construct a logic table to represent the operation of the output logic when the TRIS latch is clear ($Q = 0$), that is, the pin is set as an output. Prove that the output pin follows the latched output data.
   (b) Extend the logic table and prove that P and N are both off when the pin is initialized for input.
   (c) Describe how a data bit is read onto the data bus when the pin is set for input.
   (d) What is the function of the output FETs in the operation of the I/O pin?

CHAPTER 6

Programming Techniques

Chapter Outline

PIC Microcontrollers. DOI: 10.1016/B978-0-08-096911-4.10006-0

Chapter Points

- Instruction cycle time is four clock periods; jumps take two instruction cycles.
- Hardware timers are clocked by the instruction clock, or used to count input pulses.
- Programmable prescalers extend the range of the hardware timers.
- Interrupts force the execution of an interrupt service routine.
- EEPROM provides non-volatile memory operation.
- RC, crystal or internal clock oscillator options.
- Power-on timer, watchdog timer, sleep mode, in-circuit programming and code protection.
- Assembler directives, macros, special instructions.
- Numerical types include hex, decimal, binary, octal and ASCII.

Now that the basic programming methods have been introduced, we can look at some further techniques. Sample programs demonstrating use of the timer, interrupts and data tables are included in this chapter. A modified version of the 16F84A demo hardware schematic used for these examples is shown in Figure 6.1. It has jumpers that allow RB0 to be used as an output to a light-emitting diode (LED) or input to RB0 for demonstrating interrupts. The programs in this section can all be downloaded for simulation in Proteus VSM, or tested in MPLAB.

6.1. Program Timing

Microcontroller (MCU) program execution is driven by a clock signal generated by an internal oscillator, which may be controlled by an external RC circuit or crystal. This signal is divided into four internal clock phases (Q1−Q4) that run at a quarter of the oscillator frequency ($F_{osc}/4$). These

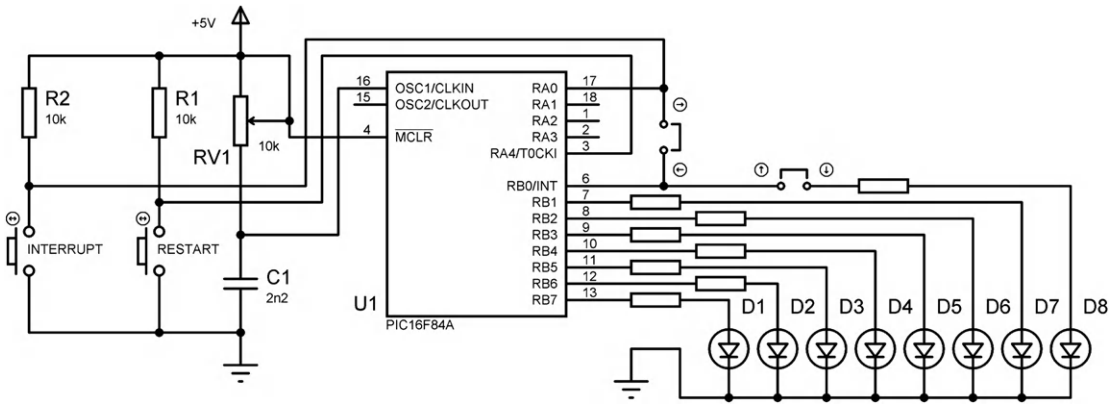
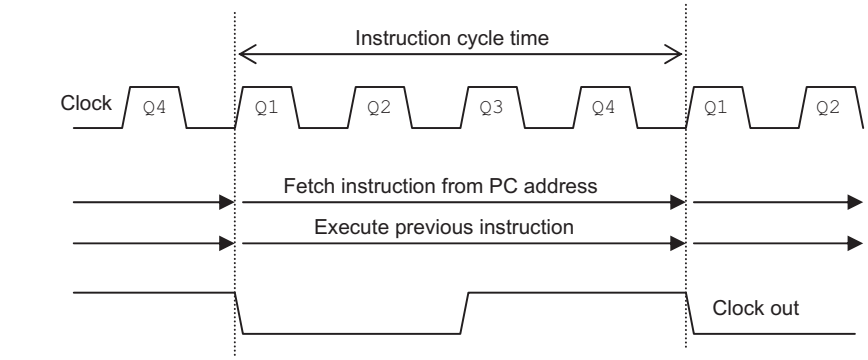


Figure 6.1
Modified demo board with jumpers

(a)



(b)

Figure 6.2

PIC program timing: (a) instruction timing cycle; (b) BIN5 MPLAB simulation, showing output timing

provide four separate pulses during each instruction cycle, which trigger the processor operations. These include fetching the instruction code from the program memory and copying it to the instruction register. The instruction code is then used by the decoder to set up the control lines to carry out the required process. The four clock phases are used to operate the data gates and latches within the MCU in sequence to complete the data movement and processing (see Microchip's 'PIC Mid-Range MCU Family Reference Manual' and Appendix C).

The instruction timing is illustrated in Figure 6.2. Most instructions are executed within these four clock cycles, unless a jump (GOTO or CALL) occurs. These will take eight clock cycles,

because the program counter contents have to be replaced, taking an extra instruction cycle. The PIC® chip operates a simple pipelining scheme which overlaps the fetch cycle of one instruction with the execution cycle of the previous one, doubling the speed in linear sequences at the expense of a delay of one instruction cycle when branching. An output instruction clock signal at $F_{osc}/4$ is available at the CLKOUT pin to operate external circuits synchronously; it can also be used in hardware testing to check that the clock is running, and to monitor its frequency.

If the clock rate is known, the execution time for a section of code can be predicted. A frequency of 4 MHz, using a crystal oscillator, is a convenient value as it gives an instruction cycle time of 1 μs. This is also the default frequency of the 8 MHz internal oscillator. The NOP (No OPeration) is useful here to adjust the sequence execution time; it can be used to insert a delay of one instruction cycle, that is, four clock cycles.

This point is illustrated in the simulation of program BIN5, Figure 6.2, which is a modification of BIN4 designed to give a binary count output with an output period of exactly 2 ms (500 Hz) at the LSB. A delay of 1 ms overall can be created using a counting loop set to 247 and a NOP in the loop to make the loop execution time 4 μs. The total loop time is then $247 \times 4 = 988$ μs plus 12 cycles for the loop control, giving a total of exactly 1000 μs. This is displayed on the stopwatch by setting a breakpoint at the start of the loop, and zeroing the stopwatch before running.

6.2. Hardware Counter/Timer

Accurate event timing and counting are often needed in microcontroller programs. For example, if we have a sensor on a motor shaft that produces one pulse per revolution of the shaft, the number of pulses per second will give the shaft speed. Alternatively, the interval between pulses can be measured, using a timer, to obtain the speed by calculation. A process for doing this would be:

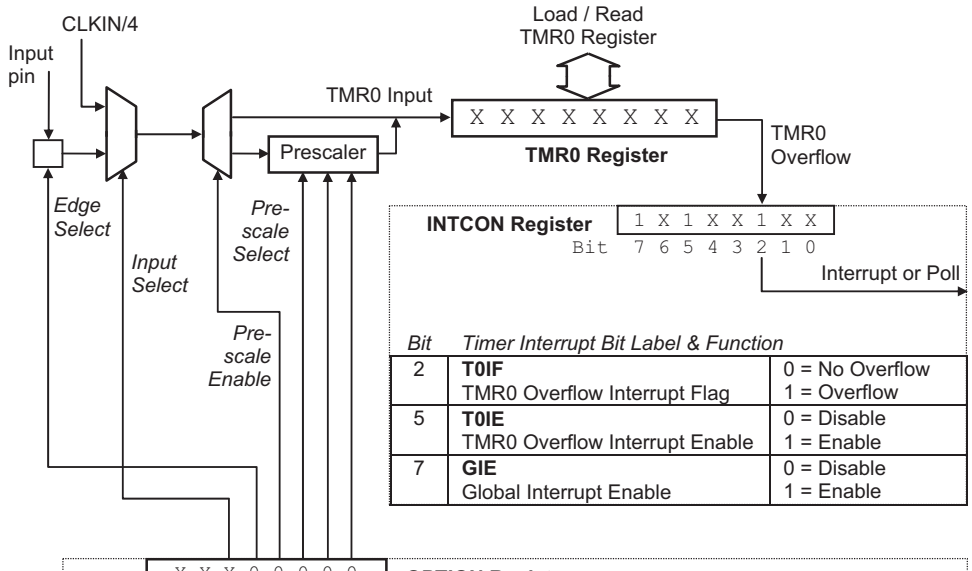
1. *Wait for pulse.*
2. *Read and reset the timer.*
3. *Restart the timer.*
4. *Process previous timer reading.*
5. *Go To 1.*

If an independent hardware timer is used to make the measurement, and the timer interrupt used, the controller program can carry on with other operations, such as processing the timing information, controlling the outputs and checking the sensor input, while the timer simultaneously records the time elapsed.

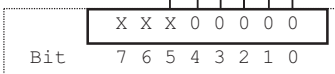
6.2.1. Using TMR0

The special file register timer zero (TMR0) is found in all PIC 16 devices. It is an 8-bit counter/timer register, which, once started, runs independently. This means it can count inputs or clock pulses concurrently with (at the same time as) the main program execution. A block diagram of TMR0 and its associated hardware and control registers is shown in Figure 6.3.

As an 8-bit register, TMR0 can count from 00 to FF (255). The operation of the timer is set up by moving a suitable control code into the OPTION register. The counter is then clocked by an external pulse train or, more often, from the instruction clock. When it reaches its maximum value, FF, and is incremented again, it rolls over to 00. This register overflow is



Bit	*Timer Interrupt Bit Label & Function*	
2	**T0IF** TMR0 Overflow Interrupt Flag	0 = No Overflow 1 = Overflow
5	**T0IE** TMR0 Overflow Interrupt Enable	0 = Disable 1 = Enable
7	**GIE** Global Interrupt Enable	0 = Disable 1 = Enable



Bit	*Timer Control Bit Label & Function*	*Prescaler Division Ratio* 2 4 8 16 32 64 128 256
0	**PS0** Prescaler Rate Select Bit 0	0 1 0 1 0 1 0 1
1	**PS1** Prescaler Rate Select Bit 1	0 0 1 1 0 0 1 1
2	**PS2** Prescaler Rate Select Bit 2	0 0 0 0 1 1 1 1
3	**PSA** Prescaler Assignment Bit	0 = Select Prescaler for TMR0 1 = Deselect Prescaler for TMR0
4	**T0SE** TMR0 Source Edge Select Bit	0 = Increment on rising edge of input pulse 1 = Increment on falling edge of input pulse
5	**T0CS** TMR0 Clock Source Select Bit	0 = Instruction Clock = Ext Clock/4 1 = Select Input pin

Figure 6.3
Timer0 setup and operation

recorded by the INTCON (interrupt control) register, Bit 2 (T0IF); assuming that it has been previously enabled and cleared, it is set to 1 when TMR0 overflows. This condition can be detected by bit testing in the program, or it can be used to trigger an interrupt (see Section 6.3 below).

In many PIC chips the 8-bit TMR0 is supplemented with additional 8-bit and 16-bit counters. The latter provide an extended count (up to 65 535) that gives greater accuracy or range. These timers are typically used to measure input intervals or generate output signals with a controlled period, using a similar setup process to that described below for TMR0.

6.2.2. Counter Mode

The simplest mode of operation of TMR0 is counting pulses applied to the relevant input (RA4 in the 16F84A, RA2 in the 16F690), which has the alternative name T0CKI, timer zero clock input. These pulses could be input manually from a push button, or produced by some other signal source, such as the sensor on the motor shaft mentioned above. If the sensor produces one pulse per revolution of the shaft, and one of the PIC outputs controls the motor, the microcontroller could be programmed to rotate the shaft by a set number of revolutions. If the motor were geared down, a positioning system can be designed to move the output through a set angle, for example, in a robot.

In order to increase the range of this kind of measurement, the prescaler allows the number of pulses received by the TMR0 register to be divided by a factor of 2, 4, 8, 16, 32, 64, 128 or 256. The ratio is selected by loading the least significant three bits in the OPTION register as follows: 000 selects divide by 2, 001 divide by 4 and so on up to 111 for divide by 256. With the maximum rate of division set, the register will count only 1 in 256 of the input pulses. TMR0 can also be preloaded with a value, say 156, using a standard MOVWF instruction. When it has been topped up by a suitable number of pulses (100) the overflow can be detected and acted upon by the program.

6.2.3. Timer Mode

The internal clock is selected as the TMR0 input source by setting the OPTION register, Bit 5, to 0. For accurate timing, a crystal oscillator must be used, mode XT. With a 4 MHz clock, the instruction clock will be 1 MHz, and the counter will then be clocked every 1 μs, taking 256 μs to count from zero round to zero again. By preloading with the value 156(9Ch) as above, the overflow would occur after 100 μs.

If the time period measured is extended using the prescaler, the maximum timer period will be 512 μs, 1024 μs and so on to 65.536 ms. Crystals are available that operate at frequencies more conveniently divisible by 2; for example, a 32.768 kHz crystal frequency

will produce an instruction clock at 8192 Hz. If this is now passed through the prescaler set to divide by 32, the counter will be clocked at 256 Hz and will time out once per second. Some PIC chips have an additional internal oscillator set to approximately this frequency (31 kHz).

In Figure 6.3, TMR0 is set up with $xxx00000_2$ in the Option register, selecting the internal clock source, with a prescale value of 2. The INTCON register has been set up with the timer interrupt enabled and the timer overflow interrupt flag has been set (overflow has occurred). Interrupts are explained in more detail in Section 6.3 below.

6.2.4. TIM1 Timer Program

A timer demonstration program TIM1 is listed as Program 6.1. It is designed to increment the binary output at port B once per second. The program uses the same demonstration BIN hardware as the previous programs, with eight LEDs displaying the contents of the output port. An adjustable CR clock is used, set to give a frequency of approximately 65 536 Hz. This frequency is divided by four, and is then divided by 64 in the prescaler, giving an overall frequency division of $4 \times 64 = 256$. The timer register is therefore clocked at $65\,536/256 = 256$ Hz. The timer register counts from zero to 256, overflows every second, and the output is then incremented. It will take 256 s to complete the 8-bit binary output count.

6.2.5. Problems with TIM1

The program TIM1 works by 'polling' the timer interrupt bit. This means that the program has to keep checking to see if this timeout flag has been set by the timer overflowing. This is not an efficient use of processor time. In real applications, it is usually preferable to allow the processor to carry on with some other process while the timer runs, and allow the timeout condition to be processed using an interrupt. The overall performance of the MCU is thus improved.

If the program is tested in MPSIM, and the stopwatch is used to measure the output step time, using a breakpoint at the increment instruction, it measures just over one second. The extra time is taken in completing the program loop before the timer is restarted. This will cause a small error, which may not be significant in this case, but in other applications it may be important. If the clock is adjustable (CR mode with variable resistor), overall timing can be tweaked in the hardware, or the program timing adjusted using NOPs.

6.3. Interrupts

Interrupts are generated by an internal event such as timer overflow, or an external asynchronous (i.e. not linked to the program timing) event, such as a switch closing. The interrupt signal can be

```
; ************************************************************
;        TIM1.ASM       M. Bates        6/1/99       Ver 1.2
; ************************************************************
;
;  Minimal program to demonstrate the hardware timer operation.
;
;  The counter/timer register (TMR0) is initialised to
;  zero and driven from the instruction clock with a
;  prescale value of 64.
;
;  T0IF is polled while the program waits for time out.
;  When the timer overflows, the Timer Interrupt Flag (T0IF) is
;  set. The output LED binary display is then incremented.
;  With the clock adjusted to 65536 Hz, the LSB LED flashes at
;  1 Hz.
;
;       Processor:      PIC 16F84A
;
;       Hardware:       PIC BIN Demo Hardware
;       Clock:          CR = 65536 Hz (approx)
;       Outputs:        RB0 - RB7: LEDs (active high)
;       WDTimer:        Disabled
;       PUTimer:        Enabled
;       Interrupts:     Disabled
;       Timer:          Internal clock source
;                       Prescale = 1:64
;       Code Protect:   Disabled
;
;       Subroutines:    None
;       Parameters:     None
;
; ************************************************************

; Register Label Equates.....................................

TMR0    EQU     01              ; Counter/Timer Register
PORTB   EQU     06              ; Port B Data Register (LEDs)
INTCON  EQU     0B              ; Interrupt Control Register

T0IF    EQU     2               ; Timer Interrupt Flag

; ************************************************************

; Initialize Port B (Port A defaults to inputs)..............

        MOVLW   b'00000000'     ; Set Port B Data Direction
        TRIS    PORTB

        MOVLW   b'00000101'     ; Set up Option register
        OPTION                  ; for internal timer/64

        CLRF    PORTB           ; Clear Port B (LEDs Off)


;  Main output loop .......................................

next    CLRF    TMR0            ; clear timer register
        BCF     INTCON,T0IF     ; clear timeout flag

check   BTFSS   INTCON,T0IF     ; wait for next timeout
        GOTO    check           ; by polling timeout flag

        INCF    PORTB           ; Increment LED Count
        GOTO    next            ; repeat forever...

        END                     ; Terminate source code
```

Program 6.1
TIM1 hardware timer program

received at any time during the execution of another process. In the PC, when you hit the keyboard or move the mouse, an interrupt signal is sent to the processor from the keyboard interface to request that the key be read in, or the mouse movement transferred to the screen. The code that is executed as a result of the interrupt is called the 'interrupt service routine' (ISR).

When the ISR has finished its task, the process that was interrupted must be resumed as though nothing has happened. This means that any information being processed at the time of the interrupt needs be stored temporarily, so that it can be recalled later. This is known as context saving. As part of the ISR execution, the program counter is saved automatically on the stack, as when a subroutine is called, so that the program can return to the original execution point after the ISR has been completed. This system allows the CPU to get on with other tasks without having to keep checking all the possible input sources.

6.3.1. Interrupt Setup

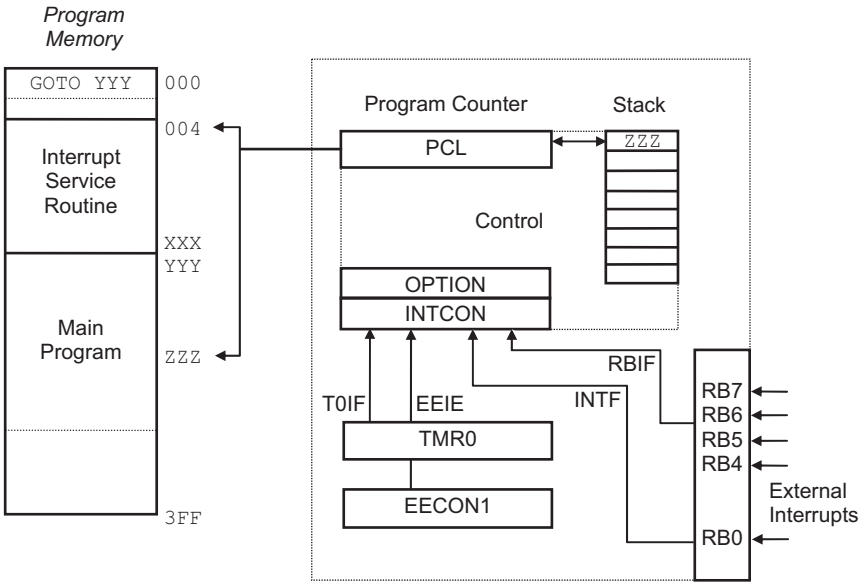
A block diagram of the interrupt system in the 16F84A is shown in Figure 6.4. There are four possible interrupt sources:

- RB0 can be set up as an edge triggered interrupt input by setting INTCON,4 (INTE).
- RB7 to RB4 can be set up to trigger an interrupt if any changes state, by setting INTCON,3 (RBIE).
- TMR0 overflow interrupt can be selected by setting INTCON,5 (T0IE).
- EEPROM write operation completion can be used to trigger the interrupt.

The interrupt source must be selected in the INTCON (Interrupt Control) register. Then, the global interrupt enable (GIE) bit, which enables all interrupts, must be set (INTCON,7). The MCU will then be ready to respond to the enabled interrupt (RBIF, INTF or T0IF). When the interrupt condition is detected (e.g. TMR0 overflow), the program counter will be automatically loaded with the address 004. That means that an ISR, or a jump to it, must be located at this address. This is the same for any of the interrupts, so if more than one has been enabled, a mechanism for identifying which is active must be included in the ISR. This means checking the interrupt flags to see which is set and jumping to the appropriate ISR. There is generally an interrupt associated with each peripheral interface, so most PICs have numerous interrupt sources. For example, the 16F690 has 12.

6.3.2. Interrupt Execution

Interrupt execution is also illustrated in Figure 6.4. Each interrupt source has a corresponding flag, which is set if the interrupt event has occurred. For example, if the timer overflows, T0IF (INTCON,2) is set. When this happens, and the interrupt is enabled, the current instruction is completed and the next program address is saved on the stack. The program counter is then loaded with 004, and the routine found at this address is executed. Alternatively, location 004



Interrupt Control Bit Functions

	Bit	*Label*	*Function*	*Settings*
INTCON	0	**RBIF**	Port B (4:7) Interrupt Flag	0 = No change 1 = Bit change detected
	1	**INTF**	RB0 Interrupt Flag	0 = No Interrupt 1 = Interrupt detected
	2	**T0IF**	TMR0 Overflow Interrupt Flag	0 = No Overflow 1 = Overflow detected
	3	**RBIE**	Port B (4:7) Interrupt Enable	0 = Disabled 1 = Enabled
	4	**INTE**	RB0 Interrupt Enable	0 = Disabled 1 = Enabled
	5	**T0IE**	TMR0 Overflow Interrupt Enable	0 = Disabled 1 = Enabled
	6	**EEIE**	EEPROM Write Interrupt Enable	0 = Disabled 1 = Enabled
	7	**GIE**	Global Interrupt Enable	0 = Disabled 1 = Enabled
OPTION	6	**INTEDG**	RB0 Interrupt Active Edge Select	0 = Falling Edge 1 = Rising Edge
EECON1	4	**EEIF**	EEPROM Write Interrupt Flag	0 = No Interrupt 1 = Write completed

Figure 6.4
16F84A interrupt setup and operation

can contain a 'GOTO addlab' (address label) if the ISR is to be placed elsewhere in the program memory. This is known as an interrupt vector.

If interrupts are to be used, a GOTO should be used at the reset address, 000, to redirect the program execution to the start of the main program at a higher memory address, because the

ISR (or GOTO addlab) will occupy address 004. The ISR must be created and placed at address 004 (ORG 004) as part of the program source code, or, alternatively, the interrupt vector placed at this address.

Context saving may be included in the ISR; this is illustrated in the interrupt demonstration program INT1 (Program 6.2) by saving and restoring the contents of port B data register. The ISR must be terminated with the instruction RETFIE, Return From Interrupt. This causes the address following that of the instruction that was interrupted to be pulled from the stack, with program execution resuming from that point.

6.3.3. INT1 Interrupt Program

A demonstration program, Program 6.2, illustrates the use of interrupts. The BIN hardware must be modified to run this program, with the push buttons connected to RB0 and RA4. This is necessary because only port B pins can be used for external interrupts (Figure 6.1).

The program outputs the same binary count to port B (except RB0), as seen in the previous BINx programs, to represent its normal activity. This process is interrupted by RB0 being pulsed manually. The interrupt service routine causes all the outputs to be switched on, and then waits for the restart button to be pressed. The routine then terminates, restores the value in port B data register and returns to the main program at the original point. The program structure and sequence are illustrated by the flowcharts in Figure 6.5.

The program is in three parts: the main sequence which runs the output count, the delay subroutine which controls the speed of the output count and the interrupt service routine. The delay process in the main program is implemented as a subroutine, and expanded in a separate flowchart. The ISR must be shown as a separate chart because it can run at any time within the program sequence. In this particular program, most of the time is spent executing the software delay, so this is the process that is most likely to be interrupted. If the program included additional tasks that could be carried out concurrently with the delay, a hardware timer interrupt could be added.

The interrupt routine is placed at address 004. The instruction ‘GOTO setup’ jumps over it at run time to the initialization process at the start of the main program. The interrupt and delay routines are assembled before the main program, because they contain the subroutine start address labels referred to in the main program. The last instruction in the ISR must be RETFIE. This instruction pulls the interrupt return address from the stack, and places it back in the program counter, where it was stored at the time of the interrupt call.

To illustrate context saving, the state of the LEDs is saved in register ‘tempb’ at the beginning of the interrupt, because port B is going to be overwritten with ‘FF’ to switch on all the LEDs. Port B is then restored after the program has been restarted. Note that writing a ‘1’ to the input bit has no effect. During the ISR execution, the stack will hold both the ISR return address and the subroutine return address.

```
; *********************************************************
;       INT1.ASM       M. Bates        12/6/99      Ver 2.1
; *********************************************************
;
;       Minimal program to demonstrate interrupts.
;
;       An output binary count to LEDs on PortB, bits 1 - 7
;       is interrupted by an active low input at RB0/INT.
;       The Interrupt Service Routine sets all outputs high,
;       and waits for RA4 to go low before returning to
;       the main program.
;       Connect push button inputs to RB0 and RA4
;
;       Processor:      PIC 16F84A
;       Hardware:       PIC Modular Demo System
;                           (reset switch connected to RB0)
;       Clock:          CR ~100kHz
;       Inputs:         Push Buttons
;                       RB0 = 1 = Interrupt
;                       RA4 = 0 = Return from Interrupt
;       Outputs:        RB1 - RB7: LEDs (active high)
;
;       WDTimer:        Disabled
;       PUTimer:        Enabled
;       Interrupts:     RB0 interrupt enabled
;       Code Protect:   Disabled
;
;       Subroutines:    DELAY
;       Parameters:     None
;
; *********************************************************

;   Register Label Equates.................................

PORTA   EQU     05              ; Port A Data Register
PORTB   EQU     06              ; Port B Data Register
INTCON  EQU     0B              ; Interrupt Control Register
timer   EQU     0C              ; GPR1 = delay counter
tempb   EQU     0D              ; GPR2 = Output temp. store

;   Input Bit Label Equates ...............................

intin   EQU     0               ; Interrupt input = RB0
resin   EQU     4               ; Restart input = RA4
INTF    EQU     1               ; RB0 Interrupt Flag

; *********************************************************

;   Set program origin for Power On Reset..................

        org     000             ; Program start address
        GOTO    setup           ; Jump to main program start

;   Interrupt Service Routine at address 004...............

        org     004             ; ISR start address

        MOVF    PORTB,W         ; Save current output value
        MOVWF   tempb           ; in temporary register
        MOVLW   b'11111111'     ; Switch LEDs 1-7 on
        MOVWF   PORTB

wait    BTFSC   PORTA,resin     ; Wait for restart input
        GOTO    wait            ; to go low
        MOVF    tempb,w         ; Restore previous output
```

Program 6.2
INT1 interrupt program

```
        MOVWF   PORTB           ; at the LEDs
        BCF     INTCON,INTF     ; Clear RB0 interrupt flag
        RETFIE                  ; Return from interrupt

;   DELAY subroutine.....................................

delay   MOVLW   0xFF            ; Delay count literal is
        MOVWF   timer           ; loaded into spare register
down    DECFSZ  timer           ; Decrement timer register
        GOTO    down            ; and repeat until zero then
        RETURN                  ; return to main program

;   Main Program  ***************************************

;   Initialize Port B (Port A defaults to inputs)...........

setup   MOVLW   b'00000001'     ; Set data direction bits
        TRIS    PORTB           ; and load TRISB
        MOVLW   b'10010000'     ; Enable RB0 interrupt in
        MOVWF   INTCON          ; Interrupt Control Register

;   Main output loop ....................................

count   INCF    PORTB           ; Increment LED display
        CALL    delay           ; Execute delay subroutine
        GOTO    count           ; Repeat main loop always

        END                     ; Terminate source code
```

Program 6.2: (*continued*)

The active edge of the RB0 interrupt is, by default, the rising edge (OPTION,6 = 1). When this input is operated, the interrupt only takes effect when the button is released. This eliminates the need for debouncing the switch in hardware or software. Mechanical switches often momentarily bounce open again when closed, before finally closing, and this can cause program malfunction in the real hardware which would not necessarily be apparent in simulation mode.

It is easier to test this particular application by interactive simulation using Proteus VSM (ISIS), because the interrupt is a real-time process, and the effect can be seen immediately on the output LEDs. By contrast, when tested in MPSIM, the effect of the program on the MCU registers can only be seen when the program execution is stopped. However, MPSIM provides more powerful simulation control and a complete record of the simulation. Both can provide a virtual logic analyzer to display the output changes in time. The simulation methods are shown together in Figure 6.6 for comparison. The files for both forms of simulation may be downloaded from the support website.

6.3.4. Multiple Interrupts

In larger PIC 16 chips, many additional interrupt sources are present, such as analogue inputs, serial ports and additional timers. These all have to be set up and controlled via additional special function registers, but there is still only one interrupt vector address, 004, to handle

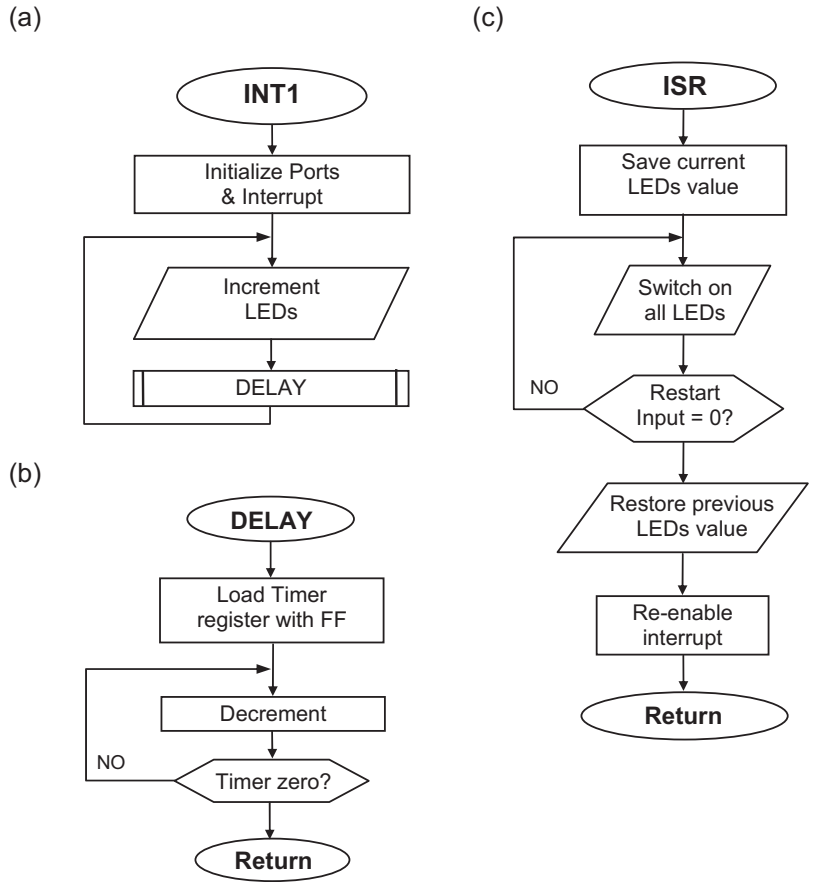


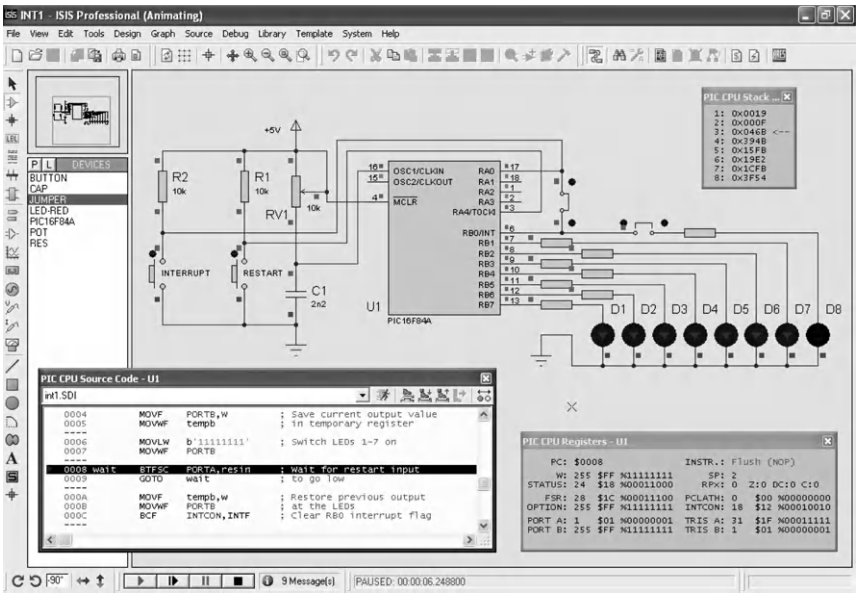
Figure 6.5
INT1 interrupt program flowcharts: (a) main sequence; (b) delay subroutine; (c) interrupt service routine

them. Therefore, when an interrupt is requested, these individual interrupt bits must be checked in the software to see which is active before calling the appropriate ISR. The stack will still only hold eight return addresses, meaning that only eight levels of interrupt or subroutine are allowed. The limit of eight levels of subroutine or interrupt can easily be exceeded if the program is too highly structured (i.e. multiple subroutine levels), so this must be borne in mind when planning the program design. Higher power PIC chips have deeper stacks.

6.4. Register Operations

We will now briefly review some of the options available when using the file registers, which provide more flexibility in programming.

(a)



(b)

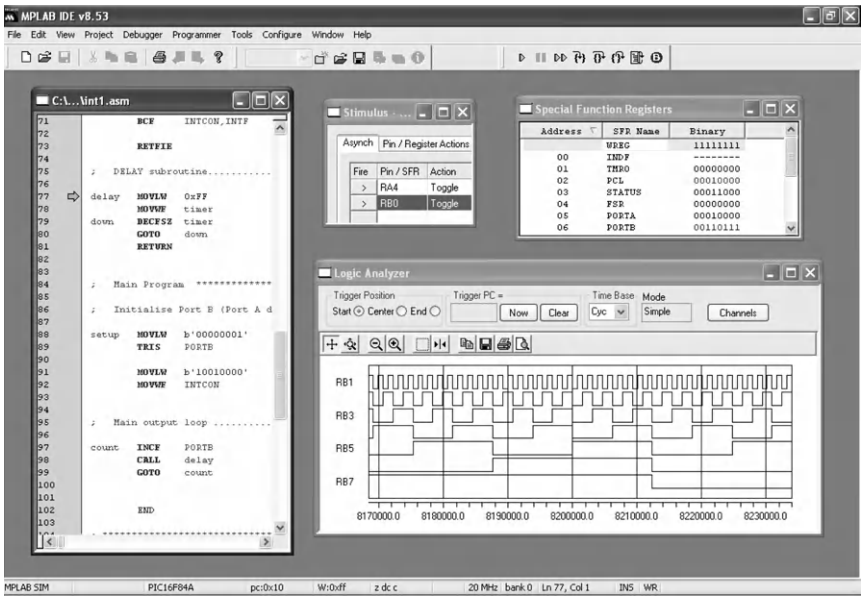


Figure 6.6

INT1 simulation test: (a) interrupt program INT1 ISIS interactive test; (b) interrupt program INT1 MPSIM test

6.4.1. Result Destination

The default destination for single register operations can simply be specified by label or number. For example,

```
INCF spare
```

increments the register labeled ‘spare’, with the result being left in the register. The above syntax may generate a message when the program is assembled to remind the user that the ‘default’ destination is being used, unless messages are suppressed by specifying a list file option. The full syntax for the instruction is:

```
INCF spare,1
```

or

```
INCF spare,f
```

where ‘1’ indicates the file register itself as the destination.

If the result of the operation were required in the working register (W), it could be moved using a second instruction:

```
MOVF spare,W
```

However, the whole operation can be done in one instruction by specifying the destination as W as follows

```
INCF spare,0
```

or

```
INCF spare,W
```

The label W is automatically given the value 0 by the assembler. The result of the operation is stored in W, while the original value is left unchanged in the file register. All the register arithmetic and logical byte operations have this option, except CLRF (Clear File Register) and CLRW (Clear Working Register), which are by definition register specific, and MOVWF and NOP (No OPeration). This option offers significant savings in execution time and program memory requirements, and also compensates to some extent for the absence of instructions that allow data to be transferred direct between file registers. These are available in higher power PICs that have a more extensive instruction set.

6.4.2. Register Bank Select

The smallest PICs, including the 16F84A, have a file register set (Figure 5.2) organized in two banks, with the most commonly used registers in the default bank 0. Some of the control registers, such as the port data direction registers, TRISA and TRISB, and the OPTION

register, are mapped into bank 1. Many of the special function registers (SFRs) can be accessed in either bank. Others have used special access instructions, namely TRIS to write the port A and B data direction registers, and OPTION which is used to set up the hardware timer counter. The assembler warns that the instructions TRIS and OPTION may not be supported in future. However, at the time writing, they still work, and provide a simplified method of bank 1 access for the beginner.

Newer, more powerful PIC 16 chips can have up to 32 banks of RAM, so a more general method of bank selection is needed. Bank selection bits are provided in one of the SFRs, and these can be modified directly using BSF and BCF instructions. In the 16F84A, only one bit is needed, bit 5 in the status register, named RP0. Bank 0 is enabled by default (RP0 = 0), thus bank 1 registers OPTION, TRISA, TRISB, EECON1 and EECON2 are accessed by setting RP0 = 1, prior to operating on the required register. This explicit bank selection method is illustrated in the code fragment in the sequence below, which sets port B as output:

```
STATUS   EQU     03        ; label for status register
TRISB    EQU     86        ; label for data direction register
         BSF     STATUS,5  ; select bank 1
         CLRW              ; load W with data direction code
         MOVWF   TRISB     ; set Port B as outputs
         BCF     STATUS,5  ; reselect bank 0
```

It is a good idea to reselect bank 0 immediately, as this is the most commonly used. If further bank 1 access is required, leave this step until later. Once a bank has been selected, it remains accessible until deselected. The larger PIC chips will need additional bank select bits.

An easier option is to use the pseudo-operation 'BANKSEL', which carries out the above process automatically:

```
BANKSEL   TRISB   ; select bank containing TRISB, bank 1
CLRW              ; load code for all outputs
MOVWF     TRISB   ; set Port B as outputs
BANKSEL   PORTB   ; reselect bank containing PORTB, bank 0
```

BANKSEL selects the bank that the specified register is in, so any register in the required bank will do. BANKSEL is effectively a predefined 'macro', a sequence of instructions that are bundled together by the assembler and invoked using a user-defined label. Macros are explained more fully in Section 6.6, below.

6.4.3. File Register Indirect Addressing

Register 04 in the PIC 16 chip is the file select register (FSR). It is used for indirect or indexed addressing of the other file registers. A target file register address is loaded into FSR, and the

contents of that file register can then be read or written at file register 00, the indirect file register (INDF). It is copied automatically to or from the target register. This method can be used for accessing a block of general purpose registers (GPRs), by reading or writing the data via INDF, and then selecting the next register in the data block by incrementing FSR. This indexed, indirect file register addressing is useful, for example, for storing a set of data that is read in at a port over a period of time. The technique is illustrated in Figure 6.7.

The demonstration program INX1 loads a set of file registers, 20 to 2F, with dummy data (AA), using FSR as the index register. FSR operates as a pointer to a block of locations, and is incremented between each read or write operation. Notice that the data actually has to be explicitly moved into INDF each time to trigger the file register write. The source code is seen in Program 6.3.

6.4.4. EEPROM Memory

PIC chips have a block of electrically erasable programmable read-only memory (EEPROM), which operates as non-volatile, read and write memory, where the data is retained when the power is off. This is useful, for example, in applications such as electronic lock, where the correct combination can be stored for comparison with an input code, but occasionally changed. Read from and write to EEPROM is illustrated in MPLAB in Figure 6.8. The code sequence can be seen in the source code window. Notice that the simulated input (09h) at port A is generated in a stimulus workbook window. The source code is listed as Program 6.4.

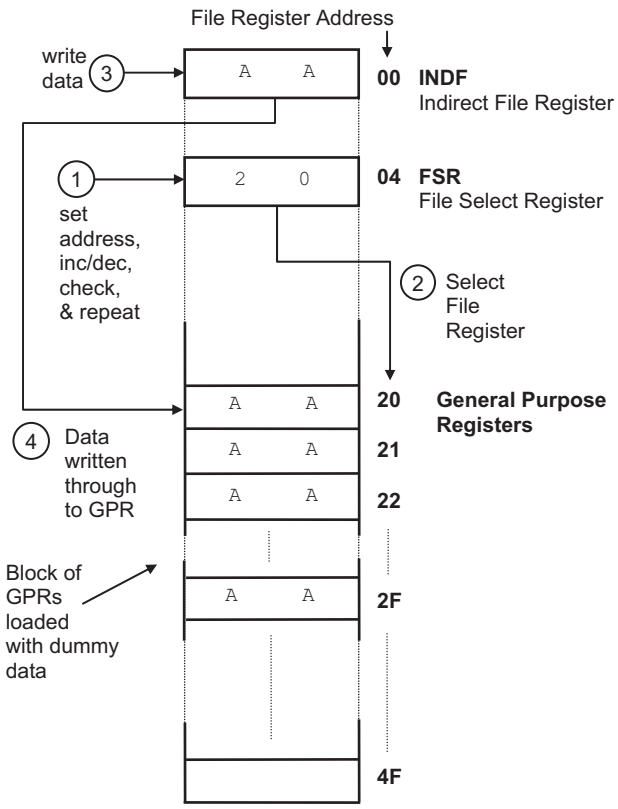
The set of registers used to access the EEPROM is EEDATA, EEADR, EECON1 and EECON2. The data to be stored is placed in EEDATA, and the address at which it is to be written in EEADR. Bank 1 must then be selected, and a read or write sequence included in the program as specified in the data sheet EEPROM section. The write sequence is designed to reduce the possibility of accidentally overwriting EEPROM, whereby essential data is lost. Reading the EEPROM is more straightforward, as seen in the second sequence in the source code.

Other devices use a different technique to access the EEPROM. For example, the 8-pin PIC 12CE518/9 devices use serial access via the unused bits of the port register. More recently introduced chips have extended the EEPROM write mechanism to include program memory read and write. The individual device data sheet must be studied carefully before using this feature.

6.4.5. Program Counter High Register, PCLATH

The basic 16 series PIC program memory can hold up to 8192 14-bit instructions (8k addresses). This requires a 13-bit address ($2^{13} = 8192$), so most of the chips in this group have a 13-bit program counter, even if the actual memory available is less than the maximum. Larger chips have a full 16-bit program counter, addressing up to 64k memory.

(a)



(b)

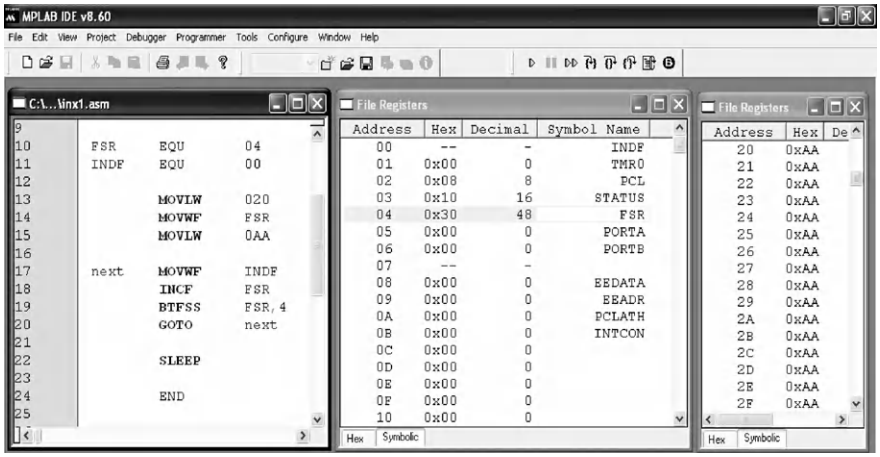


Figure 6.7
Indirect and indexed file register addressing: (a) file register operations; (b) file registers after INX1 run in MPSIM

```
; INX1.ASM             M Bates        29-10-03
; ..................................................
; Demonstrates indexed indirect addressing by
; writing a dummy data table to GPRs 20 - 2F
; ..................................................

   PROCESSOR 16F84A           ; select processor

FSR     EQU       04          ; File Select Register
INDF    EQU       00          ; Indirect File Register

        MOVLW      020        ; First GPR = 20h
        MOVWF     FSR         ; to FSR
        MOVLW     0AA         ; Dummy data

next    MOVWF     INDF        ; to INDF and GPRxx
        INCF      FSR         ; Increment GPR Pointer
        BTFSS     FSR,4       ; Test for GPR = 30h
        GOTO      next        ; Write next GPR

        SLEEP                 ; Stop when GPR = 30h

        END                   ; of source code
```

Program 6.3
Indexed addressing

The 8-bit PCL (program counter low byte) can only select one of 256 addresses, so the program memory is effectively divided into pages of 256 instructions, in the same way that random access memory (RAM) is divided into banks of 256 locations. PCL provides the address within each page of memory and is fully readable and writable. The PCH (program counter high) register, which provides the high bytes of the program address, is not directly accessible, but can be manipulated via the PCLATH (program counter latch high byte) register. The way this works is different for programmed jumps and direct writes to PCL, as illustrated in Figure 6.9, with a 13-bit address. In either case, the data sheet must be studied carefully to avoid problems with jumps over page boundaries.

GOTO and CALL

When a programmed jump is requested, the low three bits of PCH are written with the high three bits from the 11-bit operand of the GOTO or CALL instruction. The PCLATH register provides the remaining two bits of the address. If the chip has 2k program memory or less, these bits have no effect. However, if the chip has more than 2k program memory (up to 8k, or 4×2k blocks), a GOTO or CALL across a 2k memory block boundary will need the PCLATH bits 3 and 4 to be modified explicitly. The CALL instruction must store all 13 bits of the return address on the stack before the high bits are replaced.

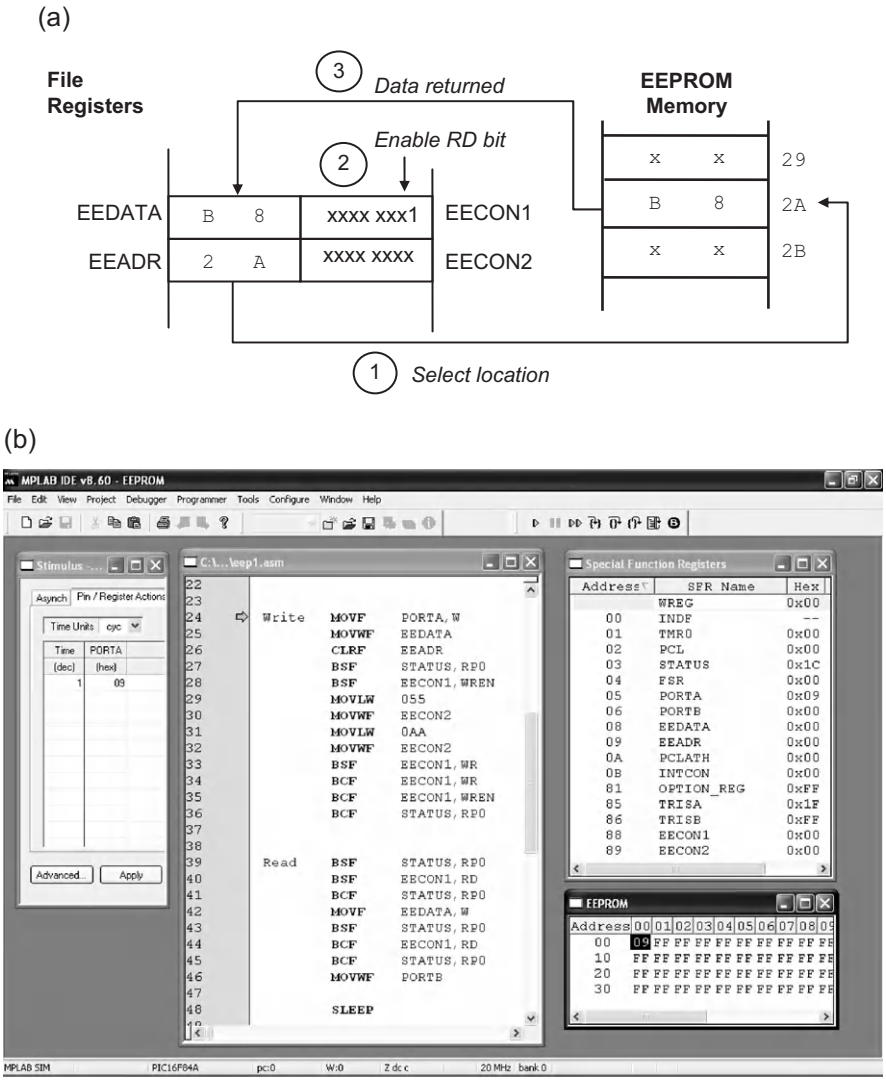


Figure 6.8
EEPROM operation: (a) register read process; (b) simulation of test program EEP1

PCL Write

If PCL is modified by a direct write under program control, the upper five bits of the program counter will be loaded from PCLATH. If the jump crosses a page boundary, these bits must be corrected accordingly. This may be relevant if, for example, a data table crosses a page boundary (see Section 6.9 on data tables below). See Microchip's 'PIC Mid-Range MCU Family Reference Manual' for further details. In other PIC chips, there may be other limitations to program branching operations. For example, CALL instructions in the 12C5XX

```
;       EEP1.ASM       MPB     02-02-11

;       Reads in data at Port A, stores it in EEPROM
;       and displays it at Port B

        PROCESSOR 16F84A

PCL     EQU     02      ; Program Counter Low
PORTA   EQU     05      ; Port A Data
PORTB   EQU     06      ; Port B Data
STATUS  EQU     03      ; Flags
EEDATA  EQU     08      ; EEPROM Memory Data
EEADR   EQU     09      ; EEPROM Memory Address
EECON1  EQU     08      ; EEPROM Control Register 1
EECON2  EQU     09      ; EEPROM Control Register 2

RP0     EQU     5       ; STATUS - Register Page Select
RD      EQU     0       ; EECON1 - EEPROM Read Initiate
WR      EQU     1       ; EECON1 - EEPROM Write Initiate
WREN    EQU     2       ; EECON1 - EEPROM Write Enable



Write   MOVF    PORTA,W         ; Read in data from port
        MOVWF   EEDATA          ; Load EEPROM data
        CLRF    EEADR           ; Select first EEPROM location
        BSF     STATUS,RP0      ; Select Register Bank 1
        BSF     EECON1,WREN     ; Enable EEPROM write
        MOVLW   055             ; Write initialisation sequence
        MOVWF   EECON2          ;
        MOVLW   0AA             ;
        MOVWF   EECON2          ;
        BSF     EECON1,WR       ; Write into current address
        BCF     EECON1,WR       ;
        BCF     EECON1,WREN     ; Disable EEPROM write
        BCF     STATUS,RP0      ; Re-select Register Bank 0


Read    BSF     STATUS,RP0      ; Select Register Bank 1
        BSF     EECON1,RD       ; Enable EEPROM read
        BCF     STATUS,RP0      ; Re-select Register Bank 0
        MOVF    EEDATA,W        ; Copy EEPROM data to W
        BSF     STATUS,RP0      ; Select Register Bank 1
        BCF     EECON1,RD       ; Disable EEPROM read
        CLRF    PORTB           ; Set PortB as outputs
        BCF     STATUS,RP0      ; Re-select Register Bank 0
        MOVWF   PORTB           ; Display data

        SLEEP

        END
```

Program 6.4
EEPROM operation

Figure 6.9
Program counter operation: (a) GOTO or CALL; (b) PCL direct load

group are limited to the first 256 locations of the program, even though the overall memory may be up to 1k.

6.5. Special Features

These include options such as oscillator type, internal timers to make the chip operation more reliable, code protection and internal hardware to support in-circuit programming and debugging. Most of these options are selected via the chip configuration word.

6.5.1. Clock Oscillator Type

The PIC MCU can be driven by an external RC network, a crystal oscillator or an internally generated clock signal. An external system clock can also be applied to synchronize its operation with other system components. The clock type is selected in the chip configuration word, which is programmed into a special location at the same time as downloading the user code. The configuration options can be set via a dialogue in MPLAB or at the top of the program using the __CONFIG directive (see Section 6.6 below).

The default clock option is normally the internal oscillator, if available. It reduces the number of external components required, and provides a default clock rate of 4 MHz in standard chips and a maximum clock rate of 32 MHz in the more recent 16F1xxx series chips. The internal oscillator is factory calibrated, but can be tuned to a more precise figure using a set of bits in the control register OSCTUNE.

For applications where the precise timing of the program is not important, and an internal oscillator is not available, an inexpensive RC low-frequency clock circuit can be used. This

requires only a resistor and capacitor connected to the CLKIN pin of the chip. If a variable resistor is used, as in the BIN hardware, the clock rate can be adjusted, within limits, and therefore all output signal frequencies can be changed simultaneously. This can be very useful, but the clock will not be very precise or stable.

The external quartz crystal oscillator option is slightly more expensive, but far more precise. The crystal is connected across the OSC1 and OSC2 pins, with a capacitor (15−22 pF) to ground from each pin, and an internal amplifier completing the circuit. The crystal resonates at a precise frequency, with an accuracy of around 50 ppm (parts per million), or 0.005%. This allows the hardware timer to measure exact intervals, and to generate accurate output signals. The overall execution time of the program blocks can also be predicted more precisely.

Three types of external crystal can be used: low power (LS), standard (XT) or high speed (HS). LS mode should be selected for low-speed crystals up to 32.768 kHz, which provides a frequency that is conveniently divisible by two. XT mode should be selected for clock speeds up to 4 MHz (1 μs instruction period), and HS used up to 20 MHz; these select a higher gain in the clock oscillator. Note that the overall power consumption is broadly proportional to the clock speed. The full supply voltage (5.0 V) is generally needed to run at high frequency, so a battery supply may not be suitable in this case.

Refer to the data sheet for specific devices for more information, and various application circuits for the external component connections.

6.5.2. Power-up Timer

When a power supply is switched on, the voltage and current initially rise in an unpredictable way, depending on the design of the supply and the load connected to it. If the processor program tries to start immediately, before the supply has settled down, it may malfunction. The PIC has a power-up timer (PWRT) built into the chip to overcome this problem. This timer is also invoked if the chip is reset at some later stage or the power supply dips temporarily (brownout).

When the PIC is powered up, it waits until the minimum operating voltage has been reached (typically 2.0 V), then generates an internal reset, which starts the PWRT. This times out after approximately 64 ms and the program starts executing. As a precaution, PWRT should normally be enabled when programming the chip, as the resulting delay on start-up will usually be insignificant.

The !MCLR (master clear) input (active low) can be used to restart the program at any time. This is useful as the power does not need to be switched off to restart, but particularly so when debugging, since the processor may hang for no obvious reason. If no reset input is required, this pin must be tied high to enable the processor to run; it is recommended that the !MCLR input is decoupled with

a 1k0 resistor (minimum value) and a 100 nF capacitor to protect against power supply transients causing a random reset, and electrostatic discharge, which can damage the input.

6.5.3. Watchdog Timer

The watchdog timer (WDT) is an internal independent timer that automatically forces the PIC to restart after a selectable period. The purpose is to allow the processor to escape from an endless loop or other error condition, without having to be reset manually. This option will be used by more advanced programs, so our main concern here is to prevent watchdog timeout occurring when not required, because it will disrupt the normal operation of our demo programs. WDT will therefore normally be disabled by selecting the appropriate configuration setting during program downloading, or specifying it in the configuration word in the source code.

If the watchdog is to be enabled, the WDT must be regularly reset within the program loop using the instruction CLRWDT. If this happens at least every, say, 1 ms (1000 instructions at 4 MHz), the WDT auto-reset can be prevented. If a program misbehaves in the simulator, check that WDT is disabled. If the WDT option is enabled, an interrupt is generated, so a suitable service routine to restart the processor must be set up at address 004, the ISR vector address.

6.5.4. Sleep Mode

The instruction SLEEP causes normal operation to be suspended and the clock oscillator is switched off. Power consumption is minimized in this state, which is useful for battery-powered applications. The PIC is woken up by a reset or interrupt; for example, when a key connected to port B is pressed. The SLEEP instruction is also used to terminate the program if it is not required to loop continuously (see Program 6.3). This prevents program execution running on into unused locations, where the program memory bits default high. This code (all 1s) is in fact a valid instruction in the PIC 16 instruction set, ADDLW FF, which will be repeated throughout the unused locations. If the program is not terminated, these meaningless instructions will be executed up to the end of program memory. The program counter will then roll over and the program will be restarted at address zero, so the program will restart by default.

6.5.5. Code Protection

In commercial applications, the PIC firmware may need to be protected from piracy. The code protect fuses, selected during programming, will prevent unauthorized copying of the code. The chip can also be given a unique identification code during programming, if required. In the demo programs, the code protection is not enabled, as the program could not then be read back for verification.

6.5.6. Configuration Word

The oscillator selection bits, watchdog timer, power-up timer, code protection and other options are selected by setting the bits of a configuration word, located at a special address that is only accessible when the chip is being programmed. These bits can be set via the programming dialogue in MPLAB. Alternatively, the configuration options can be set by including an assembler directive in the source code (see __CONFIG directive, Section 6.6, below). The default settings suggested here are:

- Clock source as required
- Watchdog timer disabled
- Power-up timer enabled
- Master clear enabled
- Code protection disabled
- All other features disabled.

6.6. Assembler Directives

Assembler directives are commands inserted in PIC source code that control the operation of the assembler. They are not part of the program itself and are not converted into machine code. Many assembler directives will only be used when a good knowledge of the programming language has been achieved, so we will refer to a small number of the more useful ones at this stage. Some of these are demonstrated in Program 6.5, ASD1. In order that the effect of the directives can be seen, the list file is reproduced here rather that just the source code, which can be seen in the right-hand columns.

The assembler directives are placed in the second column of the source code. They are not case sensitive, but are conventionally written in upper case to distinguish them. We have already met some of the most commonly used directives, but END is the only one that is essential. All the others are simply available to make the programming process more efficient. For definitive information refer to the documentation and help files supplied with your current assembler version. Some of the more useful directives are explained below.

PROCESSOR

This directive specifies the PIC processor for which the program has been designed, and allows the assembler to check that the syntax is correct for that processor. In MPLAB, the MCU type can also be specified via the menu items Configure, Select Device. It is therefore not essential to use this directive in MPLAB, but it is required if using the bare assembler in the Proteus VSM simulator. The include file also specifies the processor, if used (see INCLUDE directive below).

(a)

```
                     00001 ; *******************************************************
                     00002 ;   ASD1.ASM            M. Bates   13/11/10     Ver 1.2
                     00003 ; *******************************************************
                     00004 ;   Assembler directives, a macro and a pseudo-
                     00005 ;   operation are illustrated in this counting
                     00006 ;   program ...
                     00007 ; *******************************************************
                     00008
                     00009 ;   Directive sets processor type:
                     00010     PROCESSOR 16F84A
                     00011
                     00012 ;   Set configuration fuses:
2007   3FF3          00013     __CONFIG B'11111111110011'
                     00014 ;   Code protection off, power up timer on,
                     00015 ;   watchdog timer off, RC clock
                     00016
                     00017 ;   SFR equates are inserted from disk file:
                     00018     INCLUDE P16F84A.INC
                     00001         LIST
                     00002 ; P16F84A.INC  Standard Header File, Version 2.00
                     00134         LIST
                     00019
                     00020 ;   Constant values can be predefined by directive:
  00FF               00021     CONSTANT maxdel=0xFF, dircb=b'00000000'
                     00022
  0000000C           00023 timer   EQU     0C        ; delay counter register
                     00024
                     00025 ; Define DELAY macro ********************************
                     00026
                     00027 DELAY   MACRO
                     00028
                     00029         MOVLW   maxdel    ; Delay count literal
                     00030         MOVWF   timer     ; loaded into spare register
                     00031
                     00032 down    DECF    timer     ; Decrement spare register
                     00033         BNZ     down      ; Pseudo-Operation:
                     00034                           ; Branch If Not Zero
                     00035         ENDM
                     00036
                     00037 ;*******************************************************
                     00038
                     00039 ;       Initialize Port B (Port A defaults to inputs)
                     00040
0000   1683          00041         BANKSEL TRISB     ; Select Bank 1
0001   3000          00042         MOVLW   dircb     ; Port B Data Direction Code
0002   0086          00043         MOVWF   TRISB     ; Load the DDR code into F86
0003   1283          00044         BANKSEL PORTB     ; Reselect Bank 0
                     00045
                     00046 ;       Start main loop ..............................
                     00047
0004   0186          00048         CLRF    PORTB     ; Clear Port B Data & restart
0005   0A86          00049 again   INCF    PORTB     ; Increment count at Port B
                     00050         DELAY             ; Insert DELAY macro
                         M
0006   30FF              M         MOVLW   maxdel    ; Delay count literal
0007   008C              M         MOVWF   timer     ; loaded into spare register
0008   038C              M down    DECF    timer     ; Decrement spare register
0009   1D03 2808         M         BNZ     down      ; Pseudo-Operation:
                          M                           ; Branch If Not Zero
000B   2805          00051         GOTO    again     ; Repeat main loop always
                     00052
                     00053         END               ; Terminate source code
```

Program 6.5

ASD1 list file components: (a) main program; (b) include file labels; (c) memory map

(b)

```
SYMBOL TABLE
  LABEL                             VALUE

C                                 00000000
DC                                00000001
DELAY
EEADR                             00000009
EECON1                            00000088
EECON2                            00000089
EEDATA                            00000008
EEIE                              00000006
EEIF                              00000004
F                                 00000001
FSR                               00000004
GIE                               00000007
INDF                              00000000
INTCON                            0000000B
INTE                              00000004
INTEDG                            00000006
INTF                              00000001
IRP                               00000007
NOT_PD                            00000003
NOT_RBPU                          00000007
NOT_TO                            00000004
OPTION_REG                        00000081
PCL                               00000002
PCLATH                            0000000A
PORTA                             00000005
PORTB                             00000006
PS0                               00000000
PS1                               00000001
PS2                               00000002
PSA                               00000003
RBIE                              00000003
RBIF                              00000000
RD                                00000000
RP0                               00000005
RP1                               00000006
STATUS                            00000003
T0CS                              00000005
T0IE                              00000005
T0IF                              00000002
T0SE                              00000004
TMR0                              00000001
TRISA                             00000085
TRISB                             00000086
W                                 00000000
WR                                00000001
WREN                              00000002
WRERR                             00000003
Z                                 00000002
_CP_OFF                           00003FFF
_CP_ON                            0000000F
_HS_OSC                           00003FFE
_LP_OSC                           00003FFC
_PWRTE_OFF                        00003FFF
_PWRTE_ON                         00003FF7
_RC_OSC                           00003FFF
_WDT_OFF                          00003FFB
_WDT_ON                           00003FFF
_XT_OSC                           00003FFD
__16F84A                          00000001
again                             00000005
dircb                             00000000
down                              00000008
maxdel                            000000FF
timer                             0000000C
```

Program 6.5: (*continued*)

(c)

```
MEMORY USAGE MAP ('X' = Used,  '-' = Unused)

0000 : XXXXXXXXXXXX---- ----------------
2000 : -------X-------- ----------------

All other memory blocks unused.

Program Memory Words Used:    12
Program Memory Words Free:  1012


Errors   :     0
Warnings :     0 reported,     0 suppressed
Messages :     3 reported,     0 suppressed
```

Program 6.5 (*continued*)

__CONFIG

In MPLAB, the menu selection Configure, Configuration Bits opens a window where the configuration bits can be set up prior to downloading, but this will be overridden by the source code __CONFIG directive if the 'Configuration Bits set in code' box is checked. The double underscore that starts the directive indicates an operation on the MCU registers. The significance of each bit is shown in the MCU data sheet. The configuration bits for two sample chips are shown in Table 6.1.

In the 16F84A, only clock type, power-up timer, watchdog and code protection need configuring. There are more options in the 16F690, reflecting the more extensive range of peripheral features. Bits 0, 1 and 2 set the clock type (111 = RC, 001 = XT, 010 = HS, 101 = INTOSC), bit 2 disables the watchdog timer if cleared and bit 3 enables the power-up timer if cleared. All the other bits are set to 1 to disable code protection, brownout protection and other clock options. In the list file ASD1.LST (Program 6.5), the bits are specified in binary, and the hex equivalent code and destination register (2007) are listed in the left column.

ORG

This sets the code 'origin', referring to the address to which the first instruction following this directive will be assigned. We have already seen (Program 6.2) how it is necessary to set the origin of the interrupt service routine as 004. The default origin is 000, the first program memory location, so if ORG is not specified, the program will be placed at the bottom of the memory. This is the reset address where the processor always starts on power-up or reset. If using interrupts, an unconditional jump 'GOTO label' must be used at the reset address 000, as the first instruction to take the execution point to the main program starts higher up the memory, above the ISR vector location. For in-circuit debugging, a NOP may be necessary in the second location (001).

Table 6.1: Configuration bits

Bit	16F84A		16F690	
	Name	Function	Name	Function
0	FOSC0	Oscillator type: 00 = LP, 01 = XT,	FOSC0	000 = LP, 001 = XT, 010 = HS,
1	FOSC1	10 = HS, 11 = RC	FOSC1	011 = EC, 100 = INTOSCIO,
2	WDTE	Watchdog timer enable = 1	FOSC2	101 = INTOSC, 110 = RCIO, 111 = RC (affect I/O on OSC1 and OSC2)
3	PWRTE	Power-up timer enable = 0	WDTE	Watchdog timer enabled = 1
4	CP	Code protected = 0	PWRTE	Power-up timer enabled = 0
5	CP	Code protected = 0	MCLRE	Master clear enabled = 1
6	CP	Code protected = 0	CP	Program memory code protected = 0
7	CP	Code protected = 0	CPD	Data memory code protected = 0
8	CP	Code protected = 0	BOREN0	Brownout protection modes
9	CP	Code protected = 0	BOREN1	disabled = 00
10	CP	Code protected = 0	IESO	Switchover enabled = 1
11	CP	Code protected = 0	FCMEN	Fail-safe clock monitor enabled = 1
12	CP	Code protected = 0	–	Reserved – do not use
13	CP	Code protected = 0	–	Reserved – do not use
Typical value		XXX1 1111 1111 0011 (FFF3h) Code protect and WDT off PWRTE on, CR clock		XX00 0000 1110 0100 (00E4h) All disabled except MCLR, PWRT With internal oscillator

LIST

A text file PROGNAME.LST is produced by the assembler, which contains the source code (with line numbers), machine code, memory allocation and symbol (label) table. This can be studied for error checking or reference using any text editor, or printed out. The LIST directive has a number of options which allow the format and content of the List File to be modified, e.g. number of lines and columns per page, error levels reported, processor type and so on. These can be selected in the MPLAB build output options. The three main elements of the list file are seen in Program 6.5, ASD1.LST: the main program, label definitions and memory map. In the main program section, the machine code and corresponding memory locations are listed in the left-hand columns. The memory map (Program 6.5c) summarizes the program memory usage and the assembler messages.

EQU

This is a commonly used directive for representing numerical values with a more memorable label (symbol in the list file). It is used in the include file (see below) to define standard symbols for the special function registers for a specific processor (e.g. PORTA), and by the user for

additional file register labels. The numerical value can be specified as hexadecimal, binary, decimal or ASCII (see Section 6.8, below). In the list file, all the user-defined labels' values (constants, equates, addresses) are listed after the include file label values.

INCLUDE

This directs the assembler to include a block of source code from a named file on disk. If necessary, the full file path must be given, but if the file is copied into the application folder with the source code and the files generated by the assembler, only the file name is needed. In the example ASD1.LST (Program 6.5), the file P16F84A.INC provided by Microchip is included at line 18, but the listing has been suppressed, as it is 134 lines long. It defines labels for all the special function registers and individual control bits in this device, which can be seen in the label value listing (Program 6.5b). The file also includes directive codes for setting the configuration bits individually, e.g. directive _PWRTE_ON will switch on the power-up timer if used in the program header.

These standard header files, which use labeling that is consistent with the data sheet register names, are supplied with the development system files for all processors. They are currently found in the folder 'MPASM Suite' in the 'Microchip' system folder alongside the MPASMWIN.EXE file. The text file is included as though it had been typed into the source code editor, so it must conform to the usual assembler syntax; any program block, subroutine or macro can be included in this way. This allows separate source code files to be combined together, and opens the way for the user to create libraries of reusable program modules.

MACRO .... ENDM

A macro is a block of source code that is inserted into the program by using its label as an instruction. In ASD1 (Program 6.5), for example, DELAY is the name of the macro, and its insertion in the main program can be seen in the list file. Using a macro is equivalent to creating a new instruction from standard instructions, or an automatic copy and paste operation. The directive MACRO defines the start of the block with a label, and ENDM terminates it. The advantage of a macro over a subroutine to perform the same function is that it is reduces overall execution time by eliminating the extra instruction cycle required by CALL and RETURN. It is therefore most suitable for short sequences or where speed is important. Subroutines, on the other hand, will use less memory, as they are only assembled once.

BANKSEL

This directive allows access to register banks other than the default, zero, which contains the main SFRs. In ASD1 (Program 6.5), it is used to access and initialize the port B data direction register. The operand is the register required (TRISB) and the effect is to set the

register select bit(s) in the status register. Remember that bank 0 must be reselected before using the main SFRs. See Section 6.4.2 above for more details.

END

The END directive informs the assembler that the end of the source code has been reached. This is the one directive that must be present; an error message will be generated if it is missing.

6.7. Pseudo-Instructions

These additional instructions are essentially macros that are predefined in the assembler. An example is shown in the program ASD1 (Program 6.5), 'BNZ down', which stands for 'Branch if Not Zero to label'. It is replaced by the assembler with the instruction sequence Bit Test and Skip if Set and GOTO:

```
BNZ   down   =   BTFSS   3,2
                 GOTO    down
```

The zero flag (bit 2) in the status register (register 3) is tested, and the GOTO skipped if it is set (as a result of the previous operation being zero). If the result was not zero, the GOTO is executed, and the program jumps to the address label specified (down).

Other examples are BZ (Branch if Zero), BC (Branch if Carry), BNC (Branch if no Carry). This type of instruction is included in the main instruction set of the more powerful PICs. Other pseudo-instructions are simply alternative forms of standard instructions, such as SETC (=BSF 3,0). LGOTO and LCALL are long jumps that automatically adjust PCLATH for branch over program memory page boundaries (see PCLATH, Section 6.4.5 above).

6.8. Numerical Types

Literal values given in PIC source code can be written using different number systems. The default is hexadecimal, that is, if the type is not specified, the assembler will assume it is hex. However, it is very important to note that the assembler will still get confused between numbers and labels if the hex number starts with a letter (i.e. A, B, C, D, E or F). The literal must start with a number, so use a leading zero at all times. Therefore, 8-bit literals should be written as three hex digits, including the leading zero (000–0FF).

The numerical types supported by the MPASM assembler are:

- Hexadecimal
- Decimal

- Binary
- Octal
- ASCII.

If necessary, refer to Appendix A for more details on hex and binary number systems. Octal is a base 8 number system, which has limited use as far as we are concerned here. ASCII is described below. To specify a type, the initial letter of the type can be used with quotes, such as:

```
Hex:       H'3F'   (or 0x3F or default 03F)
Decimal:   D'47'
Binary:    B'10010011'
ASCII:     A'K' (or 'K')
```

The numerical type prefix is not case sensitive. Hex has an alternative form that is used in C programming (e.g. 0xFA). Binary is useful for specifying register values that are bit oriented, especially when setting up control registers; the state of each bit is then explicit.

ASCII code represents text characters. The codes are listed in Table 6.2; the high bits and low bits for each character code must be combined together to form a 7-bit code. Notice that most of the characters on a standard keyboard are available, including upper and lower case, numbers, punctuation and other symbols.

Table 6.2: ASCII character set

Low Bits	High Bits					
	0010	**0011**	**0100**	**0101**	**0110**	**0111**
0000	Space	0	@	P	‘	p
0001	!	1	A	Q	a	q
0010	’	2	B	R	b	r
0011	#	3	C	S	c	s
0100	$	4	D	T	d	t
0101	%	5	E	U	e	u
0110	&	6	F	V	f	v
0111	’	7	G	W	g	w
1000	(	8	H	X	h	x
1001	)	9	I	Y	i	y
1010	*	:	J	Z	j	z
1011	+	;	K	[	k	{
1100	,	<	L	\	l	|
1101	-	=	M	]	m	}
1110	.	>	N	^	n	~
1111	/	?	O	_	o	Del

If an ASCII character is specified in the program source code, the corresponding code in the range 00−7F will be generated. This option is used in sending data to alphanumeric displays or serial ports, for example:

```
MOVLW   'Y'     ; Converted to binary 01011001
MOVWF   PortB   ; send to display
```

Note that the A for ASCII can be left out in the operand, and the character will still be correctly recognized by the assembler.

6.9. Data Table

A program may be required to output a set of predefined data bytes, for example, the codes to light up a seven-segment display with the correct pattern for each display digit. The data set can be written into the program as a table within a subroutine, and the data list accessed using CALL and RETLW. To fetch the table value required, the position in the table is placed in W. '0' will access the first item, '1' the second and so on. At the top of the subroutine, ADDWF PCL is used to add this table pointer value to the program counter register so that the execution point jumps to the required item in the list. RETLW is then used to return the table value in W, and it can then be moved to the required file register.

Program 6.6, TAB1, shows how such a table may be used to generate an arbitrary sequence at the LEDs in our BIN demonstration hardware. In this case, it is a bar graph display, which lights the LEDs from one end, using the binary sequence 0, 1, 3, 7, 15, 31, 63, 127, 255.

GPRs labeled 'timer' and 'point' are used. Port B is set as outputs, and subroutines are defined for a delay and to provide a table of output codes. In the main loop, the table pointer register 'point' is initially cleared, and will then be incremented from 0 to 9 as each code is output. The value of the pointer is checked each time round the loop to see if it is 9 yet. When 9 is reached, the program jumps back to 'newbar', and the pointer is reset to zero.

For each output, the pointer value (0−8) is placed in W and the 'table' subroutine called. The first instruction, 'ADDWF PCL', adds the pointer value to the program counter. At the first call, this value is zero, so the next instruction, 'RETLW 000', is executed. The program returns to the main loop with the value 00 in W. This is output to the LEDs, the delay is run, and the pointer value incremented. The new value is tested to see if it is 9 yet, and if not, the next call is made to the table, until finally the ninth code (0FF) is returned to the main output loop for display. After this, the test of the pointer being equal to 9 succeeds, the jump back to 'newbar' is taken, and the process repeats. Note the use of 'W' as the destination for the result of the subtract (SUBWF) instruction. This is necessary to avoid the pointer value being overwritten with the result of the subtraction.

```
;*******************************************************
;       TAB1.ASM              MPB  4-2-11
;*******************************************************
;
;   Output binary sequence gives a demonstration of a
;   bar graph display, using a program data table..
;
; *****************************************************

        PROCESSOR 16F84A

; Register Label Equates.............................

PCL     EQU     02          ; Program Counter Low
PORTB   EQU     06          ; Port B Data Register
timer   EQU     0C          ; GPR1 used as delay counter
point   EQU     0D          ; GPR2 used as table pointer

; *****************************************************

        ORG     000
        GOTO    start       ; Jump to start of main prog

; Define DELAY subroutine............................

delay   MOVLW   0xFF        ; Delay count literal
        MOVWF   timer       ; loaded into spare register
down    DECFSZ  timer       ; Decrement timer register
        GOTO    down        ; and repeat until zero
        RETURN              ; then return to main program

; Define Table of Output Codes ......................

table   ADDWF   PCL         ; Add pointer to PCL
        RETLW   000         ; 0 LEDS on
        RETLW   001         ; 1 LEDS on
        RETLW   003         ; 2 LEDS on
        RETLW   007         ; 3 LEDS on
        RETLW   00F         ; 4 LEDS on
        RETLW   01F         ; 5 LEDS on
        RETLW   03F         ; 6 LEDS on
        RETLW   07F         ; 7 LEDS on
        RETLW   0FF         ; 8 LEDS on

; Initialise Port B (Port A defaults to inputs)........

start   MOVLW   b'00000000' ; Port B Data Direction
        TRIS    PORTB       ; and load into TRISB

; Main loop ..........................................

newbar  CLRF    point       ; Reset pointer
nexton  MOVLW   009         ; Check if all done yet
        SUBWF   point,W     ; (note: destination W)
        BTFSC   3,2         ; and start a new bar
        GOTO    newbar      ; if true...
        MOVF    point,W     ; Set pointer to
        CALL    table       ; access table...
        MOVWF   PORTB       ; and output to LEDs
        CALL    delay       ; wait a while...
        INCF    point       ; Point to next value
        GOTO    nexton      ; and repeat...

        END                 ; Terminate source code
```

Program 6.6
TAB1 table program

For full details on topics in this chapter related to the assembler, refer to the 'MPASM User's Guide' at www.microchip.com.

Questions 6

1. State (a) the number of clock cycles in a PIC instruction cycle, and the number of instruction cycles taken to execute the instructions (b) CLRW and (c) RETURN. (3)
2. If the PIC clock input is 100 kHz, what is the instruction cycle time? (2)
3. Calculate the preload value required in TMR0 to obtain a delay of 1 ms between the load operation and the T0IF going high, if the clock rate is 4 MHz and the prescale ratio selected is 4:1. (3)
4. List the bits in the SFRs that have to be initialized to enable an RB7:RB3 interrupt. (2)
5. State one advantage each of the RC, XT, HS and INTOSC clock options. (4)
6. State the assembler directive that must be used in all PIC programs. (2)
7. Explain the difference between a subroutine and a macro, and one advantage of each. (4)

Answers on pages 420–1. (Total 20 marks)

Activities 6

1. Calculate the time taken to execute one complete cycle of the output obtained from TAB1 with a clock rate of 100 kHz. Check this result by simulation.
2. Modify the program TIM1 to use a timer interrupt rather than polling to control the delay.
3. Devise a program to measure the period of an input pulse waveform at RB0, which has a frequency range of 10–100 kHz. When measured, the input period should be stored in a GPR called 'period' as a value where $0A_{16} = 10$ μs and $64_{16} = 100$ μs (resolution of 1 bit per microsecond). The MCU clock frequency is 4 MHz.

CHAPTER 7

PIC Development Systems

Chapter Outline

Chapter Points

- In-circuit programming is the usual method of PIC program downloading.
- PICkit2 and 3 are low-cost USB programmer/debugger modules.
- LPC is a standard Microchip demonstration board based on the PIC 16F690 chip.
- The test program demonstrates analogue input and other programming techniques.
- The program can be tested in MPSIM or Proteus VSM before downloading.
- Microchip supplies a range of demo kits for training and application development.
- A range of in-circuit debugging modules offers three levels of cost and performance.

PIC Microcontrollers. DOI: 10.1016/B978-0-08-096911-4.10007-2

A development system is the software and hardware package that supports a particular range of microcontrollers (MCUs). The software provides the program development environment, including utilities for testing and downloading the program to the MCU. Many of the more advanced features available in MPLAB are not covered in this book, in particular, use of the linker and librarian features of the assembler tool suite. These support the creation of new applications from modular, reusable code, and are used by advanced programmers to improve the efficiency of the development process for more complex programs. Third-party suppliers also provide tools that are designed to support a range of different types of MCU, of which, for us, Proteus VSM is the most useful.

The development system also requires hardware peripherals to complete the toolset. A programming module is the main requirement, and there are various options depending on the MCU type and the application development context (hobby, education, commercial). There are also various interfacing options (e.g. headers required for some chips), demo boards and so on. Some basic options are described below, focusing in particular on the 16F690 LPC demo board and similar kits.

7.1. In-Circuit Programming

MPLAB IDE and a hardware programmer are the essential components of the Microchip toolset. Originally, PIC® chips had to be removed from the circuit for programming in a separate module, and then replaced in the target application board. Now, in-circuit programming allows the chip to be programmed without being removed, which avoids possible mechanical (broken/bent legs) and electrical (static) damage. The MCU must incorporate the necessary hardware features to support this option. If in-circuit debugging (ICD) is supported by the design of the target hardware (six-pin connector and suitable connections to the MCU), the programming module can act as both a programming and a debugging interface.

The connections shown in Figure 7.1 are common to the low-price PICkit2/3 and the ICD2/3 programmer/debugger modules. Each has a six-pin connector, with the PICkit using a single in-line (SIL) connector on the target board. The program is downloaded via ICSPDAT/PGD (in-circuit serial programming data) synchronized by ICSPCLK/PGC (clock). If the target board does not need too much current, it can be powered from the host computer via the universal serial bus (USB) programming module (V_{DD} and V_{SS}). For example, the PIC 16F690-based low-pin count (LPC) demo board can be programmed without an external supply. The target board reset can be controlled from MPLAB (!MCLR) and V_{PP} provides the programming voltage.

If the chip and development system support ICD, the same MPLAB simulation tools can be used to test the program as it runs in the actual chip, with the real hardware providing the inputs and outputs. This allows the interaction with the target hardware to be examined more closely and a final debugging stage implemented to ensure correct operation of the MCU in the actual

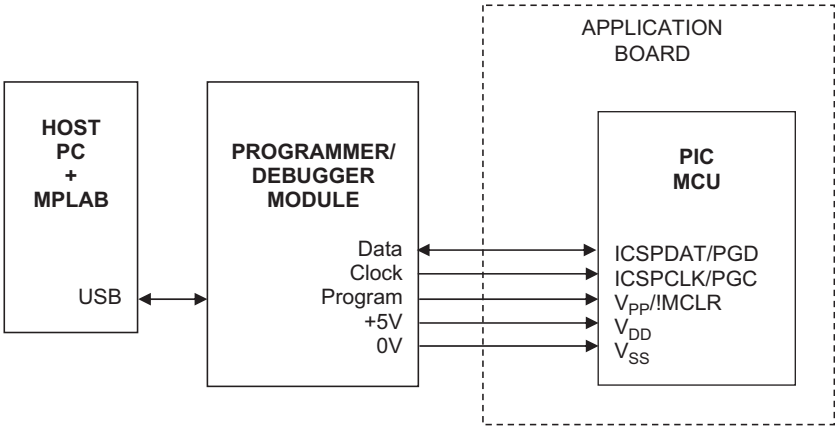


Figure 7.1
Programmer/debugger connections

circuit. All the usual techniques are available: single stepping, breakpoints, register monitoring and so on. The finished program can then be run at full speed in the target hardware, and any final bugs removed that become apparent. Previously, an expensive in-circuit emulator would have been needed for this type of testing.

Unfortunately, the 16F690 chip does not support ICD without a header. If this feature is required, the 44-pin demo board with the 16F887 chip on board is available, since this chip contains the ICD interface while the 16F690 does not.

7.2. PICkit2 Demo System

The PICkit2 is an in-circuit programming module that supports a full range of PIC microcontrollers. PICkit3 is now available. The PICkit2 Starter Kit also includes the LPC board incorporating the 16F690 MCU and some minimal test circuitry (Figure 7.2). The programmer is connected to the USB port of the host PC running MPLAB, with the six-pin in-line output plugged into a six-pin male connector on the target board. The connections are shown in Table 7.1.

The board has four light-emitting diodes (LEDs) to display programmed output sequences, a push button connected to !MCLR and a small pot providing an analogue test input. The LPC board can be powered from the USB port via pins 2 and 3. For programming, +12 V is applied to pin 1, but after programming is complete, it reverts to the reset (!MCLR) input function. When under control of the host PC, the on-board reset button is overridden by a command/button in the MPLAB toolbar. When detached from the programmer, the push button can be configured as a reset input or as a digital input. Pins 4 and 5 carry the program data and clock, which write the code into program memory in the target chip. Pin 6 is available for additional functions of the programmer. These features can be seen in the schematic for the LPC (Figure 7.3).

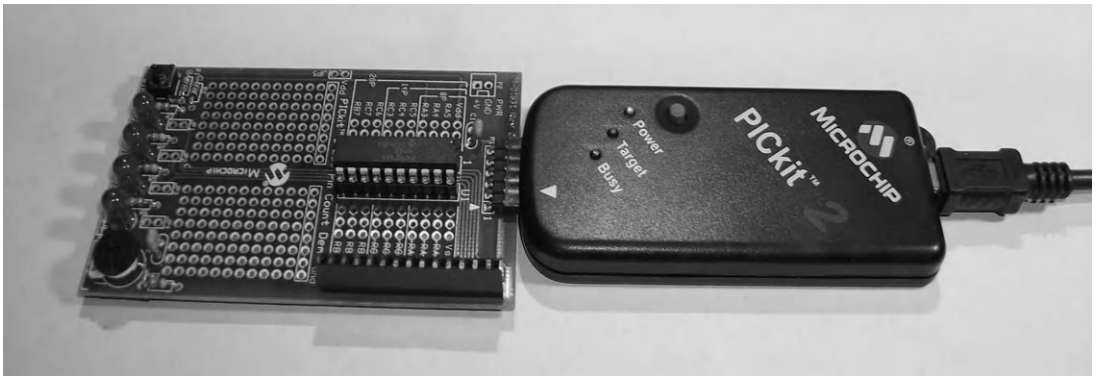


Figure 7.2
PICkit2 Starter Kit including LPC board

Table 7.1: PICkit2 pin functions

Pin	Label	Function
1	V_{PP}/!MCLR	Programming voltage or reset input when application running
2	V_{DD} Target	Power supply positive voltage (can supply LPC board) +5 V
3	V_{SS} (ground)	Power supply reference voltage (can supply LPC board) +0 V
4	ICSPDAT/PGD	Programming data: bidirectional serial signal (program download and verify)
5	ICSPCLK/PGC	Programming clock: unidirectional clock signal supplied by programmer
6	Auxiliary	Connected to T1G/CLKOUT on LPC

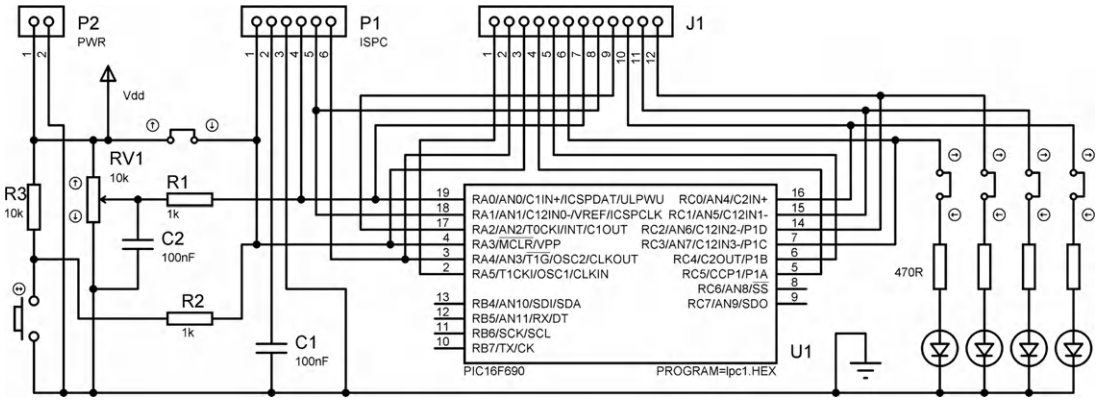


Figure 7.3
LPC board schematic

The board has additional connections to all the chip pins and a small prototyping area, which can be used to add peripheral components. Links are provided in the LED circuits so that they can be disconnected if these outputs are needed for another load. When the board is removed from the programmer, it needs to be connected to an external supply via connector P2.

7.3. PIC 16F690 Chip

The data sheet for this chip should be downloaded from www.microchip.com and studied in conjunction with this section. The pin-out can be seen in the schematic Figure 7.3; it has only 20 pins in total, hence the 'low pin count' description. The 16F690 is representative of the 16 series chips as it has a typical range of interfaces, including:

- Digital input/output
- Analogue inputs (12)
- Multi-mode timers (3)
- Serial ports (USART, SPI, I^2C)
- An internal clock oscillator (4 MHz).

The chip has 4k of program memory, with 256 bytes each of random access memory (RAM) and electrically erasable programmable read-only memory (EEPROM). It can be initialized to provide simple digital input/output (I/O) on 18 of the 20 pins, which are grouped as ports A (6), B (4) and C (8). Notice that port bits RB0 to RB3 are missing, and the chip is programmed via RA0 and RA1. In common with most current chips, the 16F690 has analogue inputs, which allow voltage measurement interfaces to be connected. The basic setup will be explained here, and there is further information on the principles of analogue to digital conversion in Section 12.3.3 (Chapter 12). For digital I/O the ports must be initialized as shown in the test program below because the analogue pins default to analogue inputs if not explicitly set up.

The analogue inputs use a single analogue to digital (A/D) converter, which can be connected to any one of 12 input pins (AN0 to AN11) via a multiplexer. The A/D converts an input voltage to a corresponding 10-bit binary code, which is placed automatically in special function registers (SFRs) ADRESH and ADRESL when the conversion is finished. The conversion is triggered by setting bit ADCON0,1 (A/D control register 0) and is complete when the same bit is set low by the hardware. This bit can be polled (checked repeatedly in a loop) or an interrupt set to indicate completion.

An alternative method of checking an analogue input is to use a comparator, which simply indicates which of two inputs is at a higher voltage (the input polarity). The analogue comparator has two inputs, labeled plus (+) and minus (−). Signals are applied to both, and the output goes logic high if the voltage at the (+) input is higher than at the (−) input, otherwise it is low. In the 16F690, several inputs are multiplexed with reference voltages so different combinations of inputs can be detected (see data sheet).

The hardware timers can be used in the usual counter or timer mode, but in addition can be used in capture, compare or pulse width modulation (PWM) mode. Capture means the timer value is stored when a selected input changes, allowing, for example, the period of an input to be measured. Compare mode is the inverse operation: the timer value is compared after each increment to a reference register and an output changed or interrupt generated when they match. This can be used to generate an output pulse waveform of a set period. PWM is similar, designed to provide a pulse waveform with a set mark/space (high/low) ratio.

Serial ports allow communication with other devices (microprocessors or computers) via single- or two-wire connection. There are various methods (protocols) available; the 16F690 supports RS232, RS485, LIN, SPI and I^2C. The function of each pin must be selected during initialization, since each has multiple operating modes. The internal clock frequency is selected in the OSCCON register. The internal clock defaults to a frequency of 4 MHz, with 8 MHz the maximum. The default is accepted in the test program.

All the peripheral interfaces mentioned here are explained further in Chapter 12, and typical applications are described in detail in *Interfacing PIC Microcontrollers: Embedded Design by Interactive Simulation* (Newnes 2006) by this author.

7.4. Test Program

The test program LPC1 (Program 7.1) exercises the analogue input and LEDs, and is used to explain the testing and downloading processes. Its function is to rotate an LED through the bits of port B, with the speed controlled by the pot. The test program has the following structure:

```
Main
     Processor  configuration
     Control  register  setup
     Main  Loop
          Rotate  LED
          Read  pot  via  ADC
          Call  delay  using  ADC  result
     Repeat  always
     Subroutine
          Delay  using  ADC  result
```

In the processor configuration word !MCLR (reset input) is enabled as this allows the program execution in the LPC board to be controlled from the MPLAB programming toolbar. Bank selection is used to access the control registers, in descending order (bank 2, 1, 0) so the port data register bank does not need to be reselected at the end of the register initialization sequence.

```
;
;       LPC1.ASM       MPB  Ver 1.0
;       Test program for LPC demo board
;       Rotates LED, pot controls the speed
;
;*****************************************************************

        PROCESSOR 16F690       ; Specify MCU for assembler
        __CONFIG 00E5          ; MCU configuration bits
                               ; PWRT on, MCLR enabled
                               ; Internal Clock (default 4MHz)
        INCLUDE "P16F690.INC"  ; Standard register labels

        LOCO    EQU     20     ; GPR labels
        HICO    EQU     21

; Initialize registers.....................................................

        BANKSEL ANSEL          ; Select Bank 2
        CLRF    ANSEL          ; Port C digital I/O
        BSF     ANSEL,0        ; except AN0 Analogue input
        CLRF    ANSELH         ; Port C digital I/O

        BANKSEL TRISC          ; Select Bank 1
        CLRF    TRISC          ; Initialise Port C for output
        MOVLW   B'00010000'    ; A/D clock setup code
        MOVWF   ADCON1         ; A/D clock = fosc/8

        BANKSEL PORTC          ; Select bank 0
        CLRF    PORTC          ; Clear display outputs
        MOVLW   B'00000001'    ; Analogue input setup code
        MOVWF   ADCON0         ; Left justify, Vref=5V,
                               ; Select RA0, done, enable A/D

; Start main loop.........................................

        CLRF    PORTC          ; LEDs off
        BSF     PORTC,0        ; Switch on LED0
loop    RLF     PORTC          ; Rotate output LED
        BSF     ADCON0,1       ; start ADC..
wait    BTFSC   ADCON0,1       ; ..and wait for finish
        GOTO    wait
        MOVF    ADRESH,W       ; store result high byte
        MOVWF   HICO
        INCF    HICO           ; avoid zero count
        CALL    slow
        GOTO    loop           ; Repeat main loop

; Subroutine..............................................

slow    CLRF    LOCO           ; delay block
fast    DECFSZ  LOCO
        GOTO    fast
        DECFSZ  HICO
        GOTO    slow
        RETURN

        END                    ; Terminate assembler..........
```

Program 7.1
LPC1 test program

The main sequence lights up an LED, rotates it through port B bits, reads the pot voltage and uses that value in the delay counter, with the result that the pot controls the speed of the LED sequence. At the mid-position of the pot, with a clock rate of 4 MHz, the whole cycle takes about 1 second. Note the high bit is rotated through all 8 bits but only 4 are displayed, so there is a delay between the last LED going out and the first coming on.

7.5. Analogue Input

The registers used in setting up and operating the analogue input are listed in Table 7.2. The ANSEL and ANSELH (analogue input select) register bits are configured with 0 for digital and 1 for analogue inputs. Only AN0 is required in this case (ANSEL,0), the others will be set for digital input. The register bits default to 1, or analogue inputs, so must be initialized for digital I/O with 0. Usually it is convenient to clear all the control bits, and then set those that are required as analogue inputs.

The analogue-to-digital converter (ADC) works by a successive approximation method, and uses the system clock to drive the converter that generates the binary equivalent of the input voltage. Since the conversion takes a minimum time per bit, the clock must not be too fast, so a frequency divider is provided which can be set to a suitable value. The recommended division ratio is given in the data sheet (Table 9-1) for each oscillator frequency. In this example, a 4 MHz system clock requires division by 8, to provide a 500 kHz A/D clock. ADCON1, bits 4, 5 and 6 are used to select the recommended ADC clock rate.

ADCON0 has several functions. ADCON0,0 enables the ADC and bit 1 starts the conversion process when set to 1 in the program, and also indicates the conversion is finished when it is cleared in hardware. In the test program, this bit is polled in a loop that repeats until it is cleared. Bits 2–5 select the current input as the corresponding binary number (0000 = AN0,

Table 7.2: A/D registers setup for analogue input at AN0

Register name	Setup	Bits	Comment
ANSEL	00000001	Bit 0 = 1	Input AN0 = analogue
		Bits 1–7 = 0	AN1 to AN7 = digital
ANSELH	0000 0000	Bits 0–3 = 0	AN8 to AN11 = digital
ADCON1	0 001 0000	Bits 6–4 = 001	A/D clock = f/8
ADCON0	00 0000 01	Bit 7 = 0	Left justify result
		Bit 6 = 0	V_{ref} = +5 V internal
		Bits 5–2 = 0	Select RA0 as input
		Bit 1 = 0	Done bit cleared
		Bit 0 = 1	A/D enabled
ADRESH	XXXX XXXX	Result	High bits only

0001 = AN1, etc., up to AN11 = 1011). Since only one ADC is available, only one input can be selected and converted at a time.

The ADC needs a reference voltage to set the range of the input that will be converted. ANCON0,6 selects between an internal reference of 5 V and an external reference which must be supplied from a constant voltage circuit, usually based on a zener diode. A 10-bit conversion gives results from 0000000000 (0_{10}) to 1111111111 (1023_{10}) or 1024 steps, giving a resolution of better than 0.1%. If an accurate reference voltage of say 4.096 V is supplied, the resolution will be 4096/1024 = 4.00 mV per bit. With a 5 V reference, the resolution would not be such a convenient value (5000/1024 = 4.88 mV), but no external circuit is needed. In the test program, an accurate measurement is not needed, so the internal reference is used.

Since the result is more than 8 bits, two registers are needed to receive the result: ADRESH and ADRESL. The justification of the result controls how it is placed in these. Left justification places the high 8 bits in ADRESH and the low 2 bits in ADRESL (bits 6 and 7). In the test program, therefore, the whole range is covered (0−5 V) by reading ADRESH, but at reduced 8-bit resolution (19.5 mV per bit). Right justification places the low 8 bits in ADRESL, providing 25% of the range (0−1.25 V) but at full resolution.

7.6. Simulation Test

The program can be tested in simulation mode before downloading to the LPC board. Assuming it has been edited and assembled in MPLAB, MPSIM can be invoked and the program run with the SFRs, stopwatch, etc., displayed. However, an analogue input stimulus is not available in MPSIM, so the ADRESH must be loaded via a register stimulus or a modified simulation version of the program used where a literal is loaded in place of the input. Otherwise, the program can only then be tested with the delay count loaded with 00 from ADRESH, giving the maximum delay.

Interactive simulation using Proteus VSM (Figure 7.4) is, therefore, in this case, simpler and more convenient. The program is written, assembled and attached to the MCU in the schematic, and the simulation run with source code and SFRs displayed (see Appendix E for tutorial notes). The pot can be adjusted on screen and the LEDs will animate in real time, so correct program function can quickly be demonstrated. The program timing can be checked on the display of simulated time elapsed in the status bar.

The advantages of both systems can be realized by running the interactive simulation from within MPLAB. The debug tools provided by MPLAB are used to control the VSM simulation in a viewing window, while the interactive features are still available.

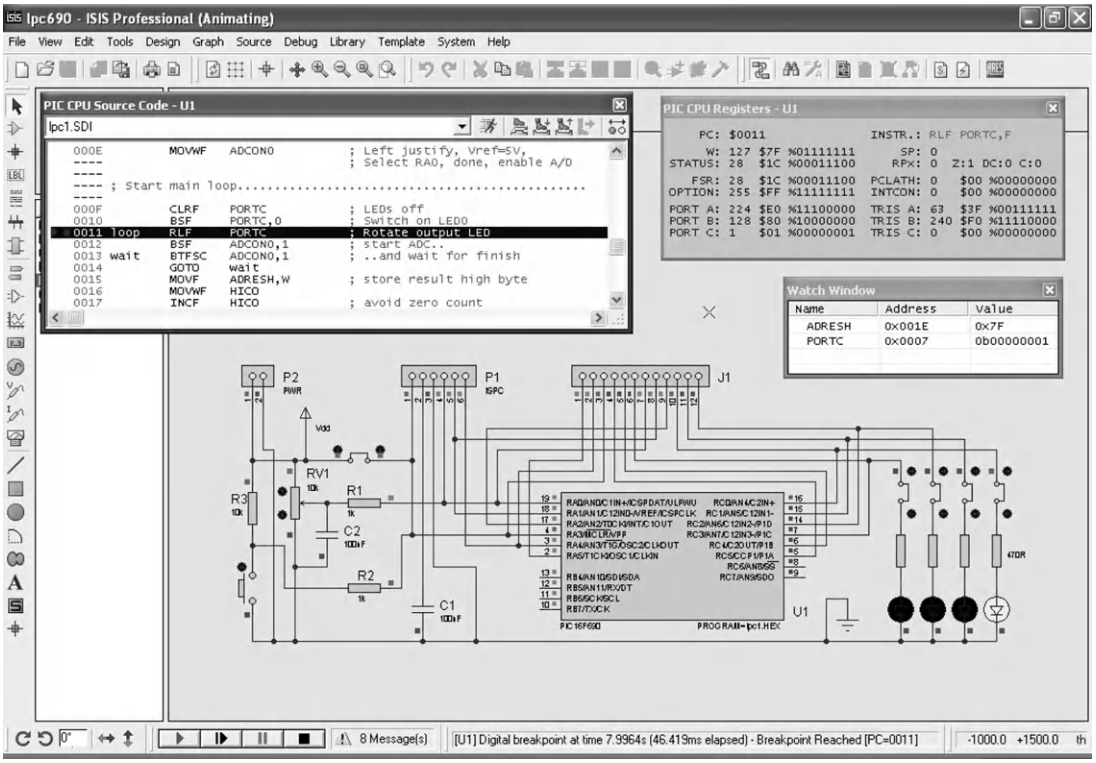


Figure 7.4
LPC board VSM simulation

The simulation test allows the basic program syntax and logic to be checked before downloading. Any syntax errors will be detected by the assembler, with a line number indicated and the error type indicated in the output window. If the scanning output is not obtained, check the main sequence by single stepping through the main loop, stepping over the delay subroutine. If the sequence appears correct, view the SFRs and check that the changes are correct. If the main sequence and intialization are correct, step into the delay loop and make sure the program is not getting stuck in an endless loop and failing to return.

7.7. Hardware Test

When simulating correctly, the program can be downloaded. The PICkit programmer is plugged into a USB port and Programmer, Select Programmer, PICkit2 from the menus. Successful connection to the programmer should be confirmed in the output window. Sometimes an updated version of the programmer operating system needs to be downloaded. A programming toolbar also appears.

Assuming the source has been successfully assembled into program memory (View, Program Memory), download it by hitting 'Program the Target Device' button, and run the program using the 'Bring Target Device MCLR to Vdd' button. The LEDs on the LPC board should start scanning, and the speed should be controlled by the on-board pot. Note that the SW1 push button on the board has no effect as it is overridden by MCLR from the programmer.

Thus, the LPC board provides a convenient demonstration of all the main features of PIC program development, except for ICD.

7.8. Other PIC Demo Kits

There are several other Microchip demonstration kits that allow the user to investigate a range of devices and techniques and provide convenient hardware platforms for further application development. The features of some of the currently available range are summarized in Table 7.3, and described below.

44/28-Pin Demo Boards

The 44-pin demo board incorporates the 16F887 MCU, which has a full range of features in a surface-mounted TQFP package. The chip has 33 I/O pins (ports A−E), so is useful if more peripherals are needed. Additional features are a full set of eight LEDs, a 32 kHz crystal clock input for timer 1 and a 10 MHz system crystal clock. A major advantage of the board is that the '887 supports direct ICD (without a header). Otherwise, the board facilities are similar to the LPC demo board with a small prototyping area and extra connections to the chip. The 28-pin board has similar features, but with the smaller sibling of the '887 chip, the 16F886, fitted.

PICDEM2 Plus Demo Board

This board has an alphanumeric liquid crystal display (LCD) commonly used to display simple messages in microcontroller applications. As well as push-button switches and a buzzer, it has a serial EEPROM that allows the I^2C serial protocol to be examined, and a temperature sensor, which can provide real-time data for storage. It has 18-, 28- and 40-pin dual in-line (DIL) sockets, allowing a range of different chips to be fitted.

PICDEM Lab Development Kit

This kit allows users to build peripheral circuits on a plug board, and is therefore a good choice for training purposes. It includes a set of different processors and a brushed direct current (dc) motor. Flowcode programming software is included, which is a user-friendly option that avoids the need to learn the details of assembler programming. Programs are entered as a flowchart (see Section 4.2 on program design in Chapter 4), which is then compiled directly to downloadable code.

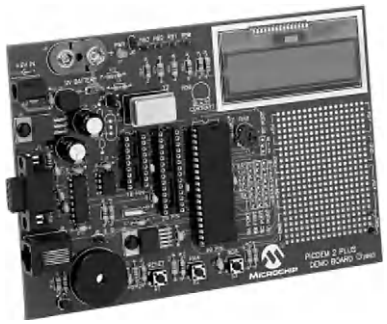
Table 7.3: PIC demo systems

PICkit2 44-Pin Demo Board

PIC 16F887 MCU fitted
Prototyping area
PICkit2 programmer/debugger
LEDs, pot and button
20 MHz and 32 kHz crystals

PICDEM2 Plus Demo Board

Supports 16 and 18 series MCUs
18-, 28- and 40-pin devices
Prototyping area
2 × 16 LCD alphanumeric display
Serial EEPROM
Temp sensor and RS232 port
Requires ICD2/3 prog/debug

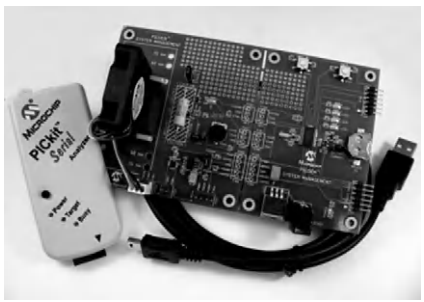


PICDEM Lab Development Kit

5 different PIC MCUs supplied
PICkit2 programmer/debugger
Prototyping area
DC motor and other components
Demo Flowcode software

PICDEM System Management Kit

PIC 16F886 MCU
Prototyping area
Fan and tacho. sensor
Heater and temperature sensor
Serial comms and analyzer
Free C compilers



Courtesy of Microchip® Technology Inc.

PICDEM System Management Kit

This kit, again based on the 16F886 MCU, has a small computer fan on board, incorporating a brushless dc motor and a sensor to monitor the fan speed, allowing experimentation with closed loop motor control, as well as a heater and temperature sensor. C compilers are included, so that more complex programs can be developed, especially applications requiring real-time calculations. A serial analyzer pod also allows the communications signals produced in the board to be examined at the outputs.

7.9. In-Circuit Debugging

In-circuit debugging (ICD) is the most powerful fault-finding technique available for microcontrollers. It allows the chip to be programmed and tested in circuit using the standard MPLAB debugging tools to control program execution in the actual target board. This is obviously a major advantage, as it allows the interaction of the PIC chip with the real hardware to be more fully examined than in a purely software simulation. Microchip currently offers three main debugging interfaces, of increasing cost and power, which all support the whole range of PIC chips. These are:

- PICkit3
- ICD3
- Real ICE.

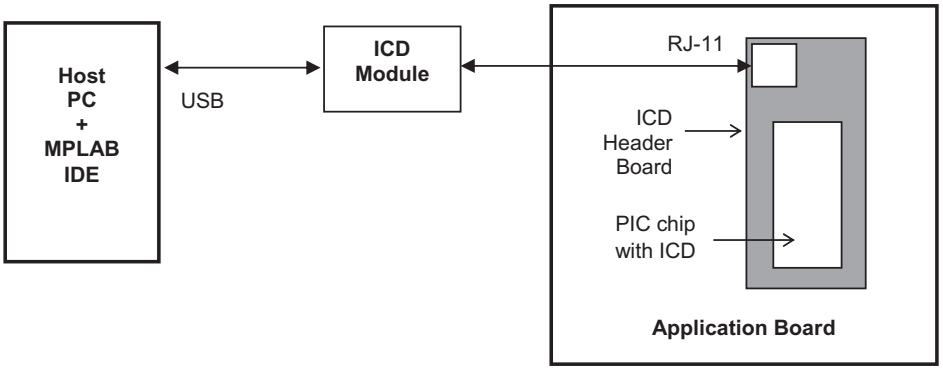
They all possess the following features:

- USB connection
- Program download, read and verify
- In-circuit debugging, including
- Unconditional and conditional breakpoints
- Register display and stopwatch timing.

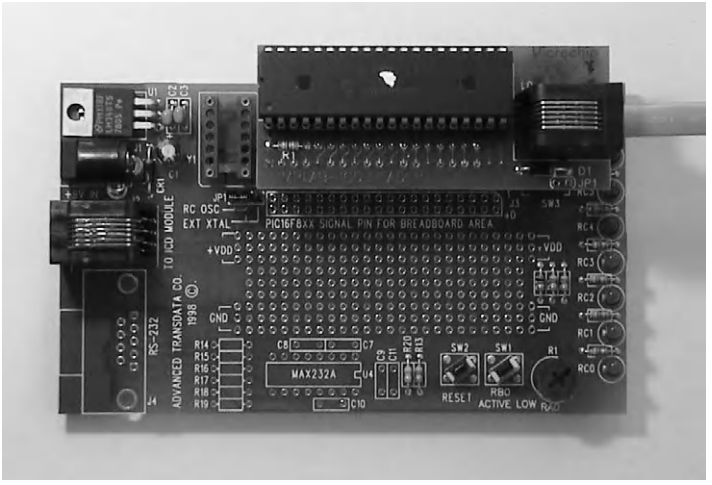
PICkit3 is the most cost-effective solution for non-professional developers, providing all the necessary features for learning and hobby applications in a compact and easy-to-use package. It is an enhanced version of PICkit2, operating at a USB full speed data rate of 12 Mb/s. It uses the six-pin in-line board connector, which will normally connect direct to the chip in circuit.

ICD3 is more powerful, operating with high-speed USB (up to 480 Mb/s) to provide real-time ICD with maximum MCU clock rates and more complex breakpoint triggering options. It uses the six-pin RJ-11 connector, designed to connect directly to chips that support ICD, or to a header board (see below) for those that do not.

(a)



(b)



(c)

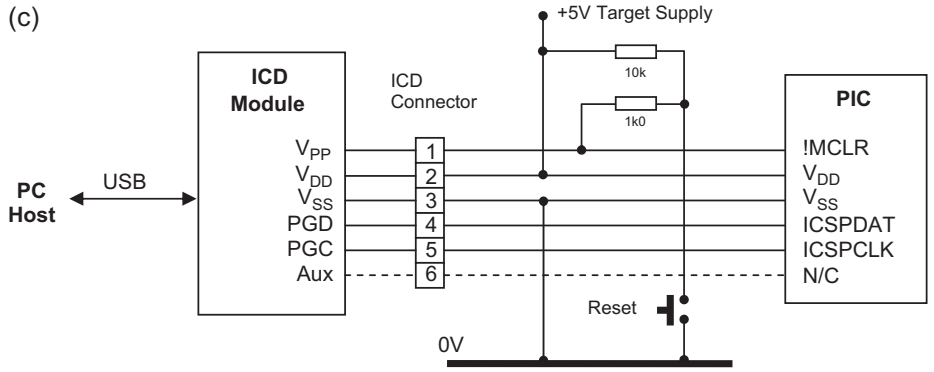


Figure 7.5
In-circuit debugging system with header board: (a) block diagram; (b) ICD header with 20-pin DIL MCU; (c) ICD connections

PICkitX and ICDX programmers are both capable of supporting ICD. Unfortunately, the smaller mid-range (16FXXX) chips, including the 16F690 chip fitted in the LPC board, do not support ICD internally, owing to pin-out limitations and cost constraints. For these chips, ICD can be implemented instead by using a header board connected between the ICD module and the chip socket on the application board. The header board carries a version of the target chip that incorporates the on-chip ICD circuitry, which substitutes for the target device while the system is under development (these chips are not available separately).

The ICD header system configuration is shown in Figure 7.5(a). The ICD module sits in between the host PC running MPLAB IDE and the application board MCU socket (Figure 7.5b). When debugging is complete, the chip can be programmed to run independently and plugged directly into the board. The ICD signals are shown in Figure 7.5(c), with definitions provided in Table 7.1. The on-board reset circuit has been included to show how it is isolated from the V_{PP} by a 1k0 resistor.

7.10. In-Circuit Emulation

An in-circuit emulator (ICE) traditionally allows processor systems to be tested without the microcontroller or microprocessor present. A host computer with a hardware dedicated emulator pod replaces the target processor, with a header connector with the same pin out as the processor connected to its socket on the application board. The emulator then substitutes for the processor operating at full speed with the real hardware, giving complete control over the target system. In the microcontroller, however, only the ports are accessible on the pins, so internal debug circuitry is needed to feed register status information out to the debugger in real time, or a header is needed to substitute for the MCU and generate the same data.

The Microchip REAL ICE debugger offers the most comprehensive facilities of the range of in-circuit programmer/debuggers, with multiple modes of operation for interactive hardware testing. As well as the standard connections to the target, or substitute header board, high-speed options are available which can also employ the serial and parallel ports to supply additional debug information. PIC 32 devices have special trace outputs to enhance the debugging operation.

For professional development, the PIC REAL ICE provides superior performance and additional debug facilities, while using the same programming and debugging connections in the target device. For programming chips on a commercial scale, the Microchip PM3 and a number of third-party programmers are available. For current product information, visit www.microchip.com.

Questions 7

1. List the functions of pins 1—5 on the six-pin programming connector of the PICkit2 module. (5)
2. A PIC 16F690 is to be used for digital input on all pins of port C, and therefore no port initialization is performed. Why will this not work correctly? (2)
3. Calculate the maximum value and resolution of an A/D conversion if the reference voltage is 1.024 V, the result is right justified and only the contents of ADRESL are used. (4)
4. Explain why testing a program in simulation mode speeds up the development process. (3)
5. Why does the push button on the LPC board not work when the board is attached to the PICkit2 programmer? (3)
6. Compare the Microchip LPC board with the 44-pin demo board and summarize the additional features of the latter. (3)

Answers on page 421. (Total 20 marks)

Activities 7

1. Download the program LPC1.ASM from the support website www.picmicros.org.uk, load it into MPLAB, assemble and test it in MPLAB. Ensure that the output port bit rotates as required. Modify the program so that the delay count is fixed at 0x80, and check the cycle time is about 1 s. Make sure the clock frequency is set to 4 MHz.
2. If you have access to Proteus VSM, download LPC690.DSN and test the program as above with the pot in mid-position. Setup the display as per Figure 7.4. Check that the output speed is controlled by the pot, and ADRESH and PORTB are displayed correctly.
3. Obtain the PICkit2 demo kit with LPC and test the system using the program LPC1. Connect the hardware, load the program into MPLAB, select PICkit2 programmer, download and run. The MCLR buttons should switch the sequence on and off, and the pot should control the speed.
4. Log onto www.microchip.com and research the current range of starter kits (home/development tools/starter kits).

CHAPTER 8

Application Design

Chapter Outline

Chapter Points

- The application requirements and target performance specification are stated.
- A block diagram is used to outline the hardware to be converted into a schematic.
- The application program consists of statements which allow sequence, selection and iteration.
- The software algorithm is represented with a suitable software design technique.
- The program outline is elaborated until sufficiently detailed to translate into source code.
- Flowcharts should be structured, using separate charts to expand the lower level processes.
- Source code should be fully commented for future reference, maintenance and modification.

PIC Microcontrollers. DOI: 10.1016/B978-0-08-096911-4.10008-4

In this chapter, we will go through the complete process of application design and development, based on a simple motor drive system, to illustrate the principles outlined in the preceding sections. At each step, basic design techniques will be explained and a suitable implementation developed.

Before designing hardware or writing a program, we have to describe as clearly as possible what an application is required to do; this means a specification is needed which defines the user's requirement. Once the specification has been written, a prototype hardware design can be attempted; a useful starting point for hardware design is a block diagram. We have already seen some examples in previous chapters. It should represent the main parts of a system and the signal/data flow between them, in a simplified form.

This can later be converted to circuit diagrams and the hardware connections laid out and constructed on a printed circuit board (PCB). In a similar way, software can be designed using techniques that allow the application program to be outlined, and then the details progressively filled in. Flowcharts have been used already, and this chapter will explain in more detail the basic principles of using flowcharts to help with program design.

Pseudocode is another useful method for designing software. This is a program outline in text form that can be entered directly into the source code editor as a set of general statements that describe each major block, which would be defined as functions and procedures in a high-level language, and subroutines and macros in a low-level language. Detail is then added under each heading until the pseudocode is suitable for conversion into source code statements for the assembler or compiler for the target processor or programming language.

At this stage, we will concentrate on flowcharts, as their pictorial nature makes them a useful learning tool. The first step in the software design process is to establish a suitable algorithm for the program; that is, a processing method that will achieve the specification using the features of an available programming language. This obviously requires some knowledge of the range of languages that might be suitable, and experience in the selected language. Formal software design techniques cannot be properly applied until the software developer is reasonably familiar with the relevant language syntax. However, when learning programming we have to develop both skills together, so some trial and error is unavoidable. When learning, it is useful to apply these design techniques retrospectively, that is, as an analytical tool or as part of the application documentation. For instance, a final version of a flowchart might be drawn after the program has been written and tested, when the suitability of the design algorithm has been proven.

Real software products will generally be far more complex than the simple examples considered here, but the same basic design principles may be applied. If the design brief is not specific about the hardware, considerable experience and detailed knowledge of the options available is required to select the most appropriate hardware and software combination. The relative costs in the planning, development, implementation, testing, commissioning and

support of the product should also be estimated to obtain the most cost-effective solution. Naturally, the example used here to illustrate the software development process has been chosen as suitable for PIC® implementation.

8.1. Design Specification

The application program will be required to generate a pulse width modulated (PWM) output to drive a small, brushed direct current (dc) motor. This can be generated by a specially designed hardware interface in many microcontrollers (MCUs), including PICs, as it is a common requirement. The software implementation will help us to understand the operation of the hardware-based PWM interface, which will be described later.

Under PWM control, the motor runs at a speed that is determined by the average level of a pulsed signal, which in turn is dependent on the ratio of the on (mark) to off (space) time, or 'duty cycle'. This method provides an efficient method of using a single digital output to control output power from a motor, heater, lamp or similar power output transducer. PWM is also used to control small digital position (hobby) servo units, as used in radio-controlled models. The basic drive waveform is shown in Figure 8.1.

A variable mark/space ratio (MSR) of 0–100%, with a resolution of 1%, is required. The frequency is not critical, but should be high enough to allow the motor to run without any significant speed variation over each cycle (> 10 Hz). It is desirable to operate at a frequency above the audible range (> 15 kHz) because some of the signal energy can radiate as sound from the windings of the motor, which can be quite irritating! A higher frequency of operation also ensures full averaging of the current. However, it is more practical to implement this using the dedicated (hardware) PWM interface, so we will aim for lower frequency of operation just to demonstrate the principles involved. The interfacing hardware is also simplified; a single field effect transistor (FET) drive transistor is used to drive the motor in one direction only, such as

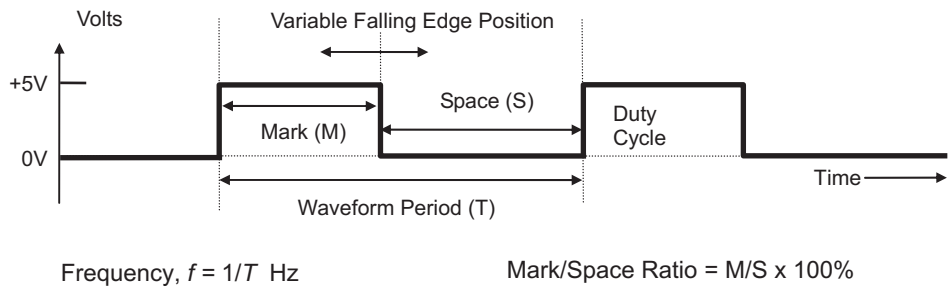


Figure 8.1
Pulse width modulated signal

would be required in a ventilation fan. A full bridge driver with four FETs is normally used to provide bidirectional motor control, which would be needed in a position controller, for example.

The motor speed will be controlled by two active low inputs, which will increment or decrement the PWM output. An active low enable signal is also required to switch the drive on and off, while preserving the existing setting of the MSR. The system should start on reset or power up at 50% MSR, that is, with equal mark and space, and a reset input should be provided to return the output to the default 50% MSR at any time. The increment and decrement operations must stop at the maximum and minimum values; in particular, 0% must not roll over to 100%, causing a zero to maximum motor speed transition in a single step. The inputs and outputs must be TTL (transistor–transistor logic) compatible (+5 V nominal signals) for interfacing purposes, allowing PWM control from a separate master controller. A programmed device (i.e. PIC) allows the control parameters to be modified to suit different motors and to enable future enhancement of the controller options and performance. A performance specification and a control logic table (Table 8.1) define the operational characteristics required.

Table 8.1: MOT1 application specification

(a) Performance Specification

```
Project: MOT1
Variable Speed Controller for Small DC Motor


1.  Maximum load: 500mA @ 5V (2.5W @ 100% MSR)
2.  Manual or remotely controlled variable MSR:
    2.1   Start: at 50% MSR
    2.2   Reset: to 50% MSR
    2.3   Range: Min < 2%, Max > 98%
    2.4   Step Resolution: < 1%
    2.5   Manual Control:
          2.5.1  Push Button Increment, Decrement
          2.5.2  Hold MSR when inputs inactive
```

(b) Control Logic Table

Inputs				Operation description	Output dc motor
!MCLR	**!RUN**	**!UP**	**!DOWN**		
0	x	x	x	Initialize – set speed to 50%	Off
1	1	x	x	Disabled	Off
1	0	1	1	Run with MSR = 50% or run at current speed	Default speed or Speed Constant
1	0	0	1	Increment MSR (until max.)	Speed increasing
1	0	1	0	Decrement MSR (until min.)	Speed decreasing

8.2. Hardware Design

The hardware is typically designed before the software, although it may need to be revisited subsequently. It is possible that the initial choice of MCU may need to be changed when the overall design requirements have been finalized.

8.2.1. Block Diagram

In the block diagram, the system inputs and outputs can be identified, and a provisional arrangement of subsystems worked out. The blocks and their connections should be labeled, indicating their function. The direction and type of information flow between the blocks should be described using arrows. Inset diagrams can be used to illustrate the waveform of analogue signals. Parallel data paths will be shown as block arrows, or with suitable signal labeling. The block diagram for MOT1 is seen in Figure 8.2.

The block diagram can be readily created using the drawing tools in Microsoft® Word or a general purpose drawing package, since it needs only basic shapes, arrows and text boxes. In Word, the drawing toolbar may need to be selected via the main menu, 'View, Toolbars, Drawing'. The drawing can be embedded in a text file, but beware of interaction of drawing objects with the text cursor, which can disrupt the drawing. It is usually a good idea to move the text cursor below the drawing area. A drawing grid can be switched on to help line up the main drawing objects; from the 'Draw' menu, select 'Grid' and check the 'Snap to grid' option. To make fine adjustments to drawing objects, the grid can later be switched off. It may be found that drawing directly onto the text page by switching off the default option of inserting a drawing canvas is more convenient (Tools, Options, General).

The main elements can be drawn using text boxes, and the same object used for labeling with the 'No line' and 'No fill' options selected. Various line and arrow styles are available, and the 'Freeform' line style in the 'Autoshapes' menu is useful for multi-segment lines. This menu also provides various standard shapes for block diagrams and flowcharts. When the drawing is

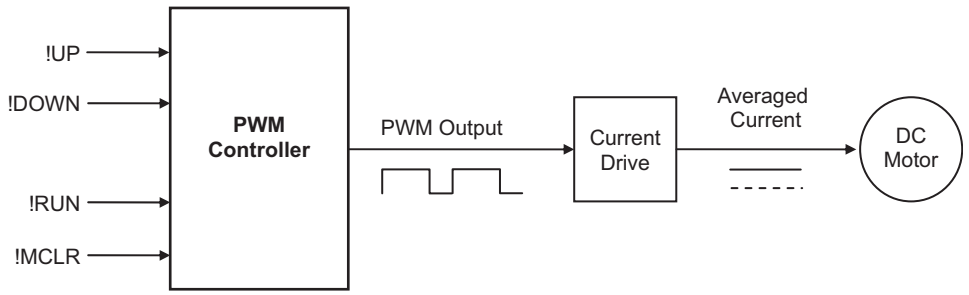


Figure 8.2
MOT1 system block diagram

finished, select all the drawing elements by looping with the 'Select objects' tool and select 'Draw, group'. This will create a single drawing object, which will no longer be affected by the text cursor, and allows the whole drawing to be repositioned on the page if necessary.

8.2.2. Hardware Implementation

Unless the program is being written for an existing hardware system, the general hardware configuration must be worked out as part of the design exercise. The nature and complexity of the software are important considerations in the selection of a microprocessor or microcontroller, as are the number and type of inputs and outputs, data storage and interfacing.

Various types of control system could be applied here, some of which are described in more detail in Chapter 14. The requirement is for minimal complexity with no special interfacing. A purely hardware solution could be based around the 555 timer, a standard pulse generator chip whose output is controlled by external CR (analogue) networks. However, this would not provide the push-button (digital) continuously variable output required. The microcontroller also provides a more flexible solution, in that the software is easily reconfigured.

A circuit derived from the block diagram is shown in Figure 8.3. The input control uses simple active low push buttons, with the additional feature of connections for remote system control. The motor is controlled by an FET, which acts as a current switch operated by the PIC digital output. An FET is selected whose input gate operates at TTL levels (0 V or +5 V with respect to the source terminal), so that it can be connected directly to the PIC output, and can handle the motor current anticipated (about 500 mA for a small motor). The motor forms an inductive

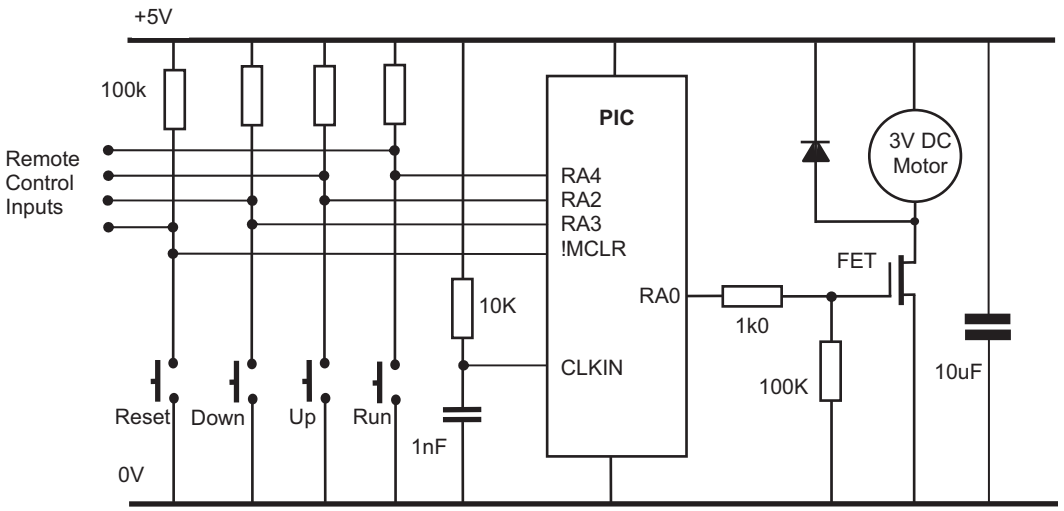


Figure 8.3
MOT1 circuit diagram

Table 8.2: MOT1 I/O allocation using PIC 16F84A

Signal	Type	Pin	Description	Comment
Clock	System	CLKIN	RC Clock	~100 kHz
Reset	System	!MCLR	Active low	Restart at default speed
PWM	Output	RA0	Pulse	FET drive
!Run	Input	RA4	Active low	Enable motor
!Up	Input	RA2	Active low	Increase speed
!Down	Input	RA3	Active low	Decrease speed

load, so a diode is connected to protect the FET from the back electromotive force (emf) normally generated by the motor. Additional decoupling is applied across the supplies to prevent the motor switching transients from disrupting the PIC supply.

The microcontroller only needs four input/output (I/O) pins, so a 12XXX series device with six I/O could be considered. However, an external reset is required, so our basic device 16F84A will be used, at least for simulation purposes. The controller I/O allocation can then be specified as shown in Table 8.2.

The PIC thus provides motor speed control with a PWM output at RA0. The !RUN ('Not Run', active low) input has been allocated to RA4. This will be programmed to enable the PWM output to run the motor when low. When RA2 (!UP) is low, the MSR at RB0 should increase, and the motor speed up. When RA3 (!DOWN) is low, the MSR should be reduced, slowing the motor down. !MCLR (Master Clear) is the reset input to the PIC, which will restart the program when pulsed low, and hence reset the speed to the default value of 50% MSR.

8.3. Software Design

We can now start work on the software using a flowchart to outline the program. A few simple rules will be used to help devise a working assembly code program; these have been discussed in more detail in Chapter 2.

A program consists of a sequence of instructions that are executed in the order that they appear in the source code, unless there is an instruction or statement that causes a jump, or branch. Usually jumps are 'conditional', which means that some input or variable condition is tested and the jump is made, or not, depending on the result. In PIC assembler, 'Bit Test and Skip if Set/Clear' and 'Decrement/Increment File Register and Skip if Zero' provide conditional branching when used with a 'GOTO label' or a 'CALL label' immediately following.

A loop can be created by jumping back at least once to a previous instruction. In our standard delay loop, for instance, the program keeps jumping back until a register that is decremented within the loop reaches zero. In high-level languages, conditional operations are created using

IF (a condition is true) THEN (do a sequence), and loops created using statements such as DO (a sequence) WHILE (a condition is true/not true). This terminology can be used in a program outline to clarify the logical sequences required.

8.3.1. MOT1 Outline Flowchart

Flowcharts illustrate the program sequence, selections and iterations in a pictorial way, using a simple set of symbols. Some basic rules for constructing flowcharts are all that are needed to ensure consistency in their use, leading to well-structured programs. An outline flowchart for the motor speed control program MOT1 is shown in Figure 8.4.

The outline flowchart shows a sequence where the inputs (Run, Speed Up and Speed Down) are checked and the delay count is modified if either of the speed control inputs is active. The output is then set high and low for that cycle, using the calculated delays to give the MSR. The loop repeats endlessly, unless the reset is operated. The reset operation is not represented in the flowchart, because it is an interrupt, and therefore may occur at any time within the loop. The program name, MOT1, is placed in the start terminal symbol. Most programs need some form of initialization process, such as setting up the ports at the beginning of the main program loop. This will normally only need to be executed once. Any assembler directives, such as label equates, should not be represented, as they are not part of the executable program itself.

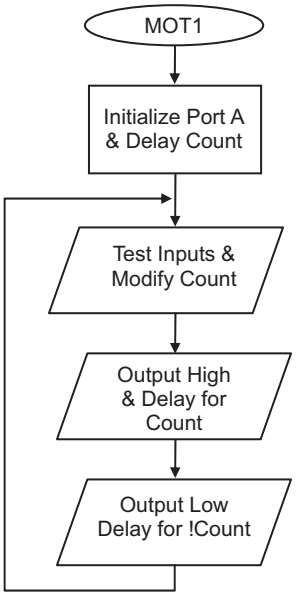


Figure 8.4
MOT1 outline flowchart

In common with most 'real-time' applications, the program loops continuously until reset or switched off. Therefore, there is an unconditional jump at the end of the program back to start, but missing out the initialization sequence. Since no decision is made here, the jump back is simply represented by the arrow, and no process symbol is needed. It is suggested here that the loop back should be drawn on the left side of the chart, and any loop forward on the right, unless it spoils the symmetry of the chart or causes line segment cross-overs. Note that when branching, the flow junctions must be BETWEEN process boxes, to preserve a single input, single output rule for each process. Each process then always starts and ends at the same point.

8.3.2. MOT1 Detail Flowchart

The outline flowchart given in Figure 8.4 may show enough information for an experienced programmer. If more detail is needed, boxes in the main program can be elaborated until there is enough detail for the less experienced programmer to translate the sequence into assembly code. A detail flowchart is shown in Figure 8.5.

After the initialization sequence, a set of conditional jumps is required to enable the motor, check the 'up' and 'down' inputs, and test for the maximum and minimum values of the value of 'Count' (FF and 01). Two different forms of the decision box have been used in this example, both of which may be seen. The diamond-shaped decision symbol is used here to represent a 'Bit Test and Skip If Zero/Not Zero' operation, while the elongated symbol represents an 'Increment/Decrement and Test for Zero' operation, which essentially combines two instructions in one. In either case, the decision box should contain a question, with its outputs representing a 'Yes' or 'No' result of the test. Note that only the result producing the jump needs to be specified.

'Decrement and Skip if Zero' is used to create the software delay loop. Two different delays are required, one for the mark time, one for the space. Since the delay needed is relatively short, and only a single loop is needed, a delay subroutine is not necessary.

8.3.3. Flowchart Symbols

A minimal set of flowchart symbols is shown in Figure 8.6. The data flowchart symbols provided in Word (Autoshapes) may be used, but they do not necessarily have the same meaning here. The text can be inserted via the drawing object properties menu (right click, add text).

Terminals

These symbols are used to start or end the main program or a subroutine. The program name or routine start label used in the source code should be specified in the start box. If the program loops endlessly the END symbol is not needed, but RETURN must always be used to terminate a subroutine. In PIC programming, use the project name (e.g. MOT1) in the start symbol of the

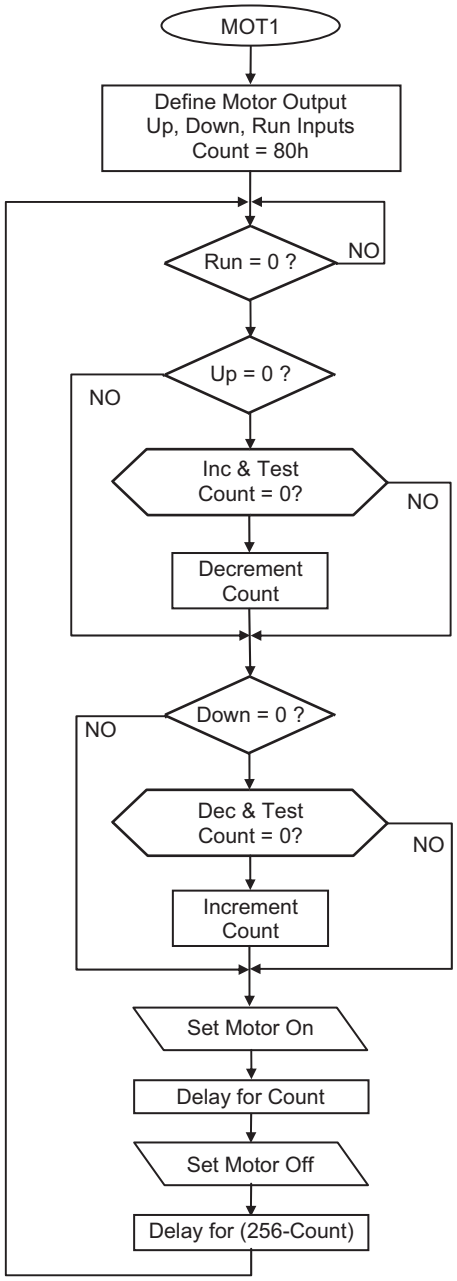


Figure 8.5
MOT1 detail flowchart

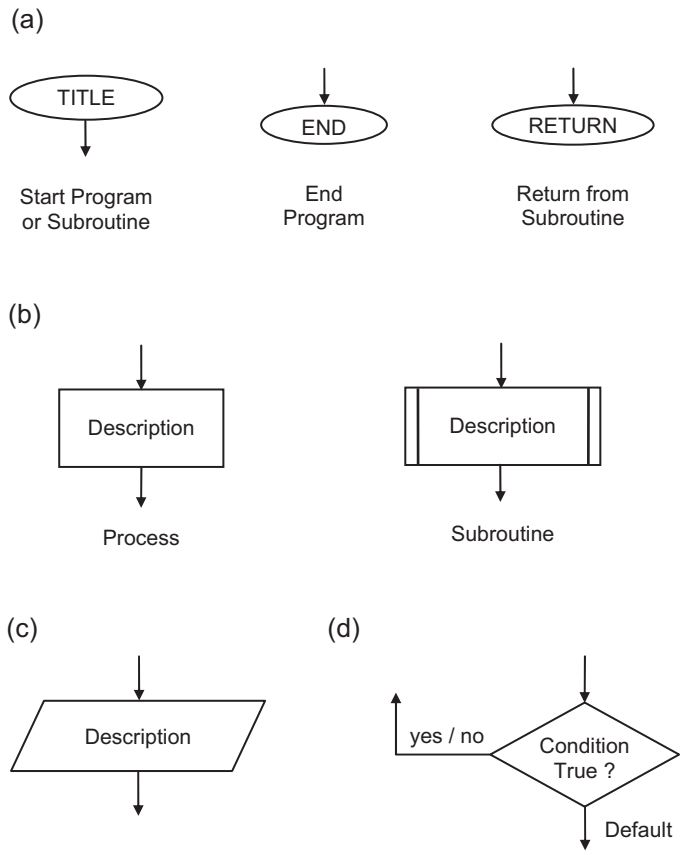


Figure 8.6
Flowchart symbols: (a) terminals; (b) processes; (c) input or output; (d) decision (conditional branch)

main program, and the subroutine start address label in subroutine start symbols. Terminals have only one input or output.

Processes

The process box is a general purpose symbol that represents a sequence of instructions, possibly including loops inside it. The top-level flowchart of a complex program can be simplified, with a lot of detail concealed in each box. A subroutine is a process that will be implemented in the source code as a separate block, and may be used more than once within a program. It may be expanded into a separate subroutine flowchart, using the same name in the start symbol as that shown in the calling process. Subroutines can be created at several levels in a complex program. Processes should have only one input and one output.

Input/Output

This represents processes whose main function is input or output using a port data register in the microcontroller. Use a statement in the box that describes the general effect of the I/O operation, for example, 'Switch Motor On' rather than 'Set RA0'. This will make the flowchart easier to understand. This symbol should also have only one input and one output.

Decisions

The decision symbol contains a description of the conditional branch as a question. There will be two alternative exit paths, for the answers 'yes' and 'no'. Only the arrow looping back or forward needs to be labeled 'yes' or 'no'; the default option, which continues the program flow down the center of the chart, need not be labeled. In PIC assembly language, this symbol would refer to the 'Test and Skip' instructions. In the MOT1 detailed flowchart, an enlarged decision box is used to represent the 'Decrement/Increment and Skip if Zero' operation. This symbol allows more text inside, so is a useful alternative to the standard diamond shape. The decision symbol thus contains a logical question, and has one input and two outputs, 'yes' or 'no'.

8.3.4. Flowchart Structure

In order to preserve good program structure, there should be single entry and exit points to and from all process blocks, as illustrated in the complete flowcharts. Loops should rejoin the main flow between symbols, and *not* connect into the side of a process symbol, as is sometimes seen. Terminal symbols have a single entry or exit point. Decisions in assembler programs only have two outcomes, branch or not, giving two exits. Loops back should be drawn on the left of the main flow, and loops forward on the right of the main flow, if possible. For the main flow down the page, the arrowheads may be omitted as forward flow is clearly implied.

Connections between pages are sometimes used in flowcharts, shown by a circular labeled symbol. It is recommended here that such connections be avoided; it should be possible to represent a well-structured program with a set of separate flowcharts, each of which should fit on one page. An outline flowchart should be devised for the main sequence, and then each process detailed with a separate flowchart, so that each process can be implemented as a subroutine or macro. In this case, the main program sequence should be as small as possible, consisting of subroutine calls and the main branching operations.

Therefore, the program should initially be represented as an outline flowchart on a single page, and each process expanded using subroutines or functions on separate pages. Keep expanding the detail until each block can be readily converted to source code statements. A well-structured program like this will be easier to debug and modify. Subroutines can be 'nested', to a depth that depends on the MCU hardware stack size (including interrupts). Mid-range PICs have space for eight return addresses in a hardware stack, which means that only eight levels of subroutine or interrupt are allowed.

8.3.5. Structure Chart

The structure chart is another method for representing complex programs. Each program block is placed in a hierarchical diagram to show how it relates to the rest of the program. This technique is most commonly used in data processing and business applications running on larger computer systems, but may be useful for more ambitious microcontroller applications.

The program shown in the structure diagram (Figure 8.7) has four levels. The main program calls subroutines to initialize, process inputs and process outputs. The input processing routine in turn calls Sub1 and Sub2 subroutines. Output processing only requires Sub3, but Sub2 calls Sub4 and Sub5 at the lowest level. At this level, 3 stack locations will be used up.

8.3.6. Pseudocode

Pseudocode is a text outline of the program design. The main operations are written as descriptive statements that are arranged as functional blocks. The structure and sequence are represented by suitable indentation of the blocks, as is the convention for high-level languages. An outline of MOT1 is shown in Figure 8.8.

The pseudocode version of the program uses high-level style syntax, such as IF ... THEN, to describe the selections in the program. It has the advantage that no drawing is required, and the

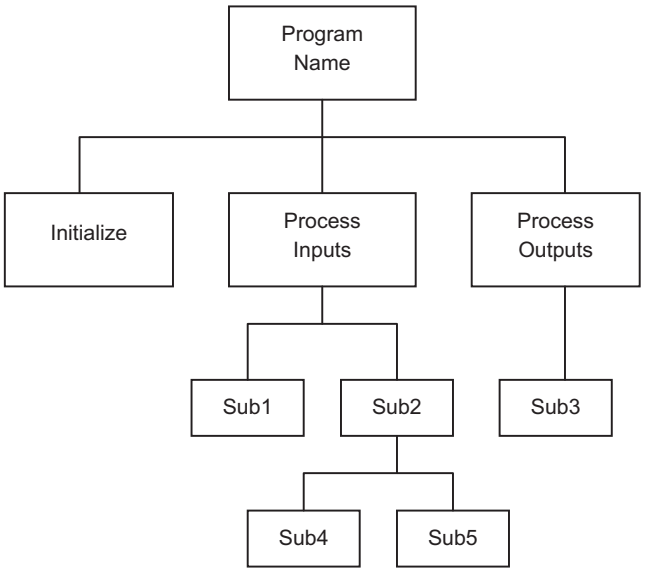


Figure 8.7
Structure chart

```
MOT1
Program to generate PWM output to Motor

       Initialize
              Outputs
                     Motor
              Inputs
                     Speed up
                     Speed down
                     Run enable
              Registers
                     Count = 128

       Start loop

              IF Run enable = off THEN wait
              IF Speed up = on THEN inc Count
              IF Count = 0 THEN dec Count
              IF Speed down = on THEN dec Count
              IF Count = 0 THEN inc Count

              Switch on Motor
              Delay for Count

              Switch off Motor
              Delay for 256-Count

       End loop
```

Figure 8.8
MOT1 pseudocode

pseudocode can be entered directly into the text editor used for writing the source code. It can be started as a brief outline and developed in stages, until it is ready to be translated into assembler syntax. The pseudocode can be left in the source code as the basis of program comments, or replaced, whichever suits the programmer. Although used here to represent an assembler program, pseudocode is probably most useful for developing 'C' programs for applications for the more powerful PIC microcontrollers.

8.4. Program Implementation

When the program logic has been worked out using flowcharts, or otherwise, the source code can be entered using a text editor. Normally, the program editor is part of an integrated development package such as MPLAB. Most programming languages are now supplied as part of an integrated edit and debug package. Assembler source code can also be entered directly into the Proteus VSM system, which includes an editor and the PIC assembler, and this is

preferable where the schematic has already been created in ISIS and the program is not too complex.

8.4.1. Flowchart Conversion

The program design method should be applied so as make the program as easy as possible to translate into source code. The PIC has a 'reduced' instruction set, meaning that the number of available instructions has deliberately been kept to a minimum to increase the speed of execution and reduce the complexity of the chip. While this also means that there are fewer instructions to learn, the assembler syntax (the way the instructions are put together) can be a little trickier to work out. For example, the program branch is achieved using the 'Bit Test and Skip' instruction. In CISC assembly code languages, branching and subroutine calls are implemented using single instructions. The PIC assembler requires two instructions. However, recall that 'Special Instructions' (essentially predefined macros) are available which combine 'test', 'skip' and 'goto' instructions to provide equivalents to conventional conditional branching instructions (see Chapter 6).

The representation of the program with different levels of detail is illustrated in Figure 8.9. Figure 8.9(a) shows the process in enough detail that each process box converts into only one or two lines of code. This may be necessary when learning the programming syntax. Later, when the programmer is more familiar with the language and the standard processes that tend to recur, such as simple loops, then a more condensed flowchart may be used, such as Figure 8.9(b), where the loop is concealed within the 'delay' process. As we have seen above,

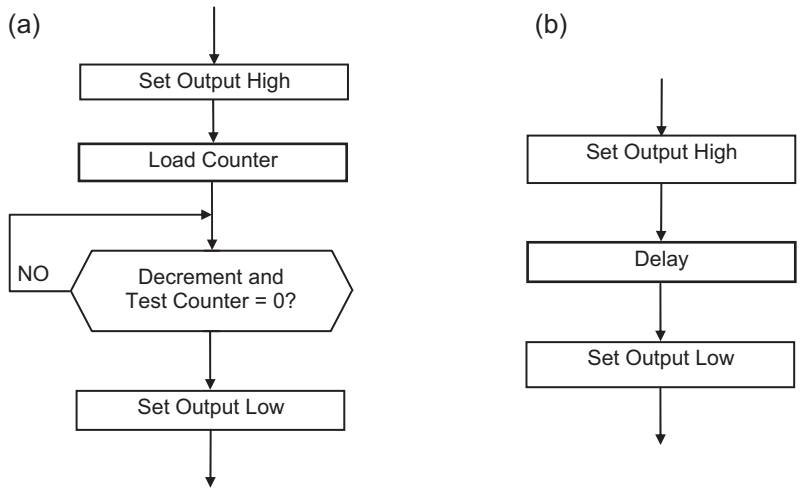


Figure 8.9
PIC program branch flowchart fragments: (a) detail flowchart; (b) outline flowchart

Table 8.3: PIC program branch code fragment

```
;         Branch Program Fragment

          .
          .
          BSF     PortA,0       ; Set Output

          MOVLW   0FF           ; Set Count Value
          MOVWF   count1        ; Load Count
back1     DECFSZ  count1        ; Dec. Count & Skip if 0
          GOTO    back1         ; Jump Back

          BCF     PortA,0       ; Reset Output
          .
          .
```

this process can also be written as a separate, reusable, software component: a subroutine. The corresponding source code fragment is shown in Table 8.3.

Another limitation in PIC 16 assembler is found when moving data between registers. It is not possible to copy data directly between file registers; it has to be moved into the working register (W) first, and then into the file register. This requires two instructions instead of the single instruction available in other processors. This problem is overcome to some extent by the availability of the destination register option with the byte processing operations, which allows the result to be placed in W or F, as required. Nevertheless, the advantage of simplicity in PIC assembly language outweighs these limitations, especially in the early stages of learning.

8.4.2. MOT1 Source Code

The program source code for the MOT1 program is listed in Program 8.1. The program produces the PWM output by toggling RA0 with a delay. A register labeled ‘timer’ holds the current value for the ‘on’ delay. The program does not use a subroutine for the delay, because the ‘timer’ value has to be modified for the ‘off’ delay. Note the use of the COMF instruction, which complements the contents of the timer register, which effectively subtracts the value from 256. The total PWM cycle time stays constant as a result. When incremented, the ‘timer’ value has to be checked to prevent it rolling over from FF to 00, by decrementing it again if the new value is zero. The roll-under at the low end of the scale is prevented in a similar way.

The program source code has instruction mnemonics in upper case to match the instruction set in the data sheet. However, by default, they are not case sensitive, so you will often see them in lower case. On the other hand, labels ARE case sensitive, so they must match exactly when

```
; ********************************************************
;       MOT1.ASM            M. Bates        14/6/99
; ********************************************************
;
;       DC Motor Control using Pulse Width Modulation
;       Motor (RA0) starts with 50% MSR when enabled with
;       RA4. Speed controlled with RA2, RA3.
;
;       Hardware:       Simple Motor Circuit
;       Clock:          CR ~100kHz
;       Inputs:         Push Buttons (active low):
;                       RA2 = Speed Up
;                       RA3 = Slow Down
;                       RA4 = Run
;       Outputs:        RA0 (active high) = Motor
;
;       Chip Fuse Settings:
;       WDTimer:        Disable
;       PUTimer:        Enable
;       Interrupts:     Disable
;       Code Protect:   Disable
;
; ********************************************************

  PROCESSOR 16F84A

; Register Label Equates.................................

PORTA   EQU     05              ; Port A
Timer   EQU     0C              ; Delay Counter
Count   EQU     0D              ; Delay Count Pre-load

; Input Bit Label Equates ...............................

motor   EQU     0               ; Motor Output = RA0
up      EQU     2               ; Speed Up Input = RA2
down    EQU     3               ; Slow Down Input = RA3
run     EQU     4               ; Motor Enable Input = RA4

; ********************************************************

; Initialize ............................................

        MOVLW   b'11111110'     ; Port A bit direction code
        TRIS    PORTA           ; Set the bit direction
        MOVWF   PORTA           ; Initialise input bits
        MOVLW   080             ; Initial value
        MOVWF   Count           ; ...for delay
; Next Page .............................................
```

Program 8.1
MOT1 source code

```
; Input Test ..........................................

start   BTFSC   PORTA,run       ; Test Run input
        GOTO    start           ; & wait if HIGH

        BTFSS   PORTA,up        ; Test Up input, if hi
        INCFSZ  Count           ; ...inc Count, test
        GOTO    test            ; and check down button
        DECF    Count           ; or dec Count again if 00

test    BTFSS   PORTA,down      ; Test Down input, if hi
        DECFSZ  Count           ; ...dec Count, test
        GOTO    cycle           ; and do an output cycle
        INCF    Count           ; or inc Count again if 00


; Output High and Delay ..................................

cycle   BSF     PORTA,motor     ; Switch on motor

        MOVF    Count,W         ; Get delay count
        MOVWF   Timer           ; Load timer register
again1  DECFSZ  Timer           ; Decrement timer register
        GOTO    again1          ; & repeat until zero then


; Output Low and Delay ...................................

        BCF     PORTA,motor     ; Switch off motor

        MOVF    Count,W         ; Get delay count again
        MOVWF   Timer           ; Reload timer register
        COMF    Timer           ; Complement timer value
        INCF    Timer           ; Inc to avoid 00 value
again2  DECFSZ  Timer           ; Decrement timer register
        GOTO    again2          ; & repeat until zero then

; Repeat Endlessly .......................................

        GOTO    start           ; Restart main loop
        END                     ; Terminate source code
```

Program 8.1: (*continued*)

declared and used. The label case sensitivity can be switched off as an assembler option if you wish. Upper case characters for the special function register names (PORTA) have been used to match the register names used in the data sheet, and lower case characters with the first letter capitalized used for general purpose registers (Timer, Count). The bit labels are lower case (motor, up, down, run), as are the address labels. Using source code editing conventions like this is not obligatory, but consistent layout and presentation improves program readability and makes it easier to understand.

Most programming languages allow comments to be included in the source code as a debugging aid for the programmer, and information for other software engineers who may need to fix the code at a later date. Comments in PIC source code are preceded by a semicolon; the assembler ignores any source code text following, until a line return is detected.

A header should always be created for the main program and the associated routines. It should contain relevant information for program downloading, debugging and maintenance. Examples have already been given. The layout should be standardized, especially in commercial products. The author's name, organization, date, and a program version number and description are essential. Hardware information on the processor or system type is important; for example, when a PIC program is assembled, the processor type must be specified, because there is variation in the SFRs available. The processor type may be specified in the header block as an assembler directive (essential when using the bare assembler in Proteus VSM), or by selecting the processor in the MPLAB development system options. Alternatively, the standard header file can be included, which defines the MCU and registers. Target hardware details such as input and output pin allocation are useful, and the design clock speed needs to be specified in programs where code execution time is significant. Programmer settings that enable or disable hardware features such as the watchdog timer, power-up timer and code protection should also be listed, unless specified explicitly using the available assembler directives.

The general layout of the source code should be designed to make the structure clear, with subroutines headed with their own brief functional description. The asterisk symbol (*) is often used to separate and decorate comments; rows of dots are also useful, but there is some scope here for individual touches! The main object is to make the source code and program structure as easy as possible to understand. Blank lines should separate the functional program blocks; that is, instructions that together carry out an identifiable operation. In this way, the source code can be presented in a way that makes it as easy to interpret as possible.

When the program has been finalized and the memory and I/O requirements of the PIC have been established, the final choice of MCU can be made. The range of 16F devices will be reviewed in Chapter 12. When finalizing the circuit design, a more detailed consideration of a range of interfacing techniques can be found in *Interfacing PIC Microcontrollers: Embedded Design by Interactive Simulation* by this author (Newnes 2006).

Questions 8

1. Sketch a PWM waveform and explain how PWM controls the power delivered to dc loads from a single digital output. (4)
2. Explain briefly the role of the block diagram and flowchart in creating the final hardware and software design for an application. (4)
3. Describe two alternative techniques to flowcharts for program design and one advantage of each. (4)
4. Explain the function of the following statements in the source code for MOT1, expanding on the comment given in the program source code:
   (a) motor EQU 0
   (b) BTFSC PORTA, run
   (c) INCF Count
   (d) COMF timer. (8)

Answers on pages 421–2. (Total 20 marks)

Activities 8

1. Compare the source code for MOT1 with the flowchart in Figure 8.5, and the pseudocode in Figure 8.8, and note how they correspond. Draw a structure chart for the application using the drawing tools in Word, or equivalent word-processing application.
2. Test MOT1 in MPLAB and confirm correct operation, or in Proteus VSM if available.
3. (a) Devise a set of structured flowcharts for making a nice cup of tea, manually!
   (b) Draw a block diagram of a coffee machine, and devise a set of flowcharts for a control program. You may assume a PIC microcontroller will be used with suitable interfacing, sensors and actuators.
4. (a) Devise a block diagram for a motor control system which has a bidirectional drive, and inputs which select the motor on/off and direction of rotation. Separate active low outputs should be provided to enable the motor in each direction. Investigate full bridge driver circuits and modify the MOT1 circuit to incorporate this output stage.
   (b) Modify the outline flowchart for MOT1 to operate the full bridge bidirectional output, allowing the direction to be changed only while the motor is disabled. Produce a logic table and detail flowchart, and amend the source code observing the recommendations for source code documentation.

   Compare your design with that provided in Chapter 11.

CHAPTER 9

Program Debugging

Chapter Outline

Chapter Points

- MPLAB IDE includes editor, assembler, simulator, programming and debugging utilities.
- Two main types of error can occur in source code: syntax and logical error.
- Syntax errors are invalid statements which are detected by the assembler, generating error messages.
- Logical errors are mistakes in the program design or implementation detected by simulation.
- Simulation identifies logical errors using single stepping and breakpoints with register monitoring.
- In-circuit debugging allows the software to be tested at full speed on the target hardware.

PIC Microcontrollers. DOI: 10.1016/B978-0-08-096911-4.10009-6

The design of a simple PIC® motor control application, MOT1, has been described in Chapter 8, and an assembler code program developed (Program 8.1). In practice, it is likely that such a program will contain errors, especially when learning the language, so we need now to look further at the techniques and tools that are available for debugging (removing the errors from) PIC programs. We are going to continue with MOT1 as our example application, and will see how to resolve two main types of error.

The syntax of a language refers to the way that the words are put together. Any language, programming or spoken, must follow certain rules so that the meaning is clear and the usage consistent (English is a very poor example of this!). The rules in programming languages are very strict, because the source code must be converted into machine code without any ambiguity. Syntax errors are mistakes in the source code, such as the misspelling of an instruction mnemonic, or failure to declare a label before using it in the program. These errors are detected by the MPLAB assembler (MPASM), resulting in error messages being generated and displayed in a separate window. The source code is color coded in recent versions of MPLAB to highlight correct syntax and errors. If a syntax error is detected, the correct use of the instruction set and assembler directives must be checked against the programming rules.

Once a program has been assembled without any syntax errors, it does not mean that it will function correctly when run. Logical errors may well be present which prevent correct operation. The software simulator (MPSIM) can be used to detect and correct these errors prior to downloading to the chip. It allows the program to be run in a virtual processor. Logical errors are detected by inspecting the program output and comparing it with the performance specification. This usually requires inputs to be simulated in a sequence that represents the normal usage of the application, using either asynchronous (user-generated) inputs or a stimulus file (workbook) that generates the same test sequence each time. Alternatively, interactive simulation can offer a schematic with animated input and output devices, and instant results. Proteus VSM is the most user-friendly package for microcontroller (MCU)-based circuits, offering a highly intuitive user interface and comprehensive co-simulation of analogue, digital and programmed devices, including a full range of PIC MCUs.

9.1. Syntax Errors

When the program source code for a PIC program has been created in the editor, it must be converted into machine code for downloading to the chip. This is carried out by the assembler program, MPASMWIN.EXE, which analyzes the source code text line by line, converts the instruction mnemonics into a list of corresponding binary codes and saves it as PROGNAME.HEX for downloading into the chip. Only valid statements as defined in the

PIC instruction set will be recognized and successfully converted. Assembler directives provide additional programming instructions, but these are not converted to machine code (see Chapter 6).

Before starting a project, create a folder to keep the project files in, named, for example, MOT1. In MPLAB, the source code is created in an edit window opened by hitting the New File button. Type in the file name and save it immediately in the application folder as MOT1.ASM, or any file name with an ASM extension. It is a good idea to keep a backup version of the file set on a different drive (USB stick, portable drive or network).

When the program has been entered and saved, the project menu item 'Quickbuild' will assemble a single file. If required, a project can be created, but this is only really necessary for more complex applications using multiple source files or a higher level language (usually C). In the Project menu, select 'New …' and call the project MOT1, or the same name as the source code. Now select 'Add Files to Project …' and select the source code created above. The program can now be assembled by selecting 'Build All', and the project saved. Alternatively, the workspace can be saved, that is, the screen configuration and working files.

In the source code file, numerical formatting, assembler directives and so on must all be used correctly. If they are not, error messages will be generated when the program is assembled. These describe the syntax errors that have been found. The error messages are saved in a text file PROGNAME.ERR, and displayed when the assembler is finished. A typical set of messages is shown below:

```
Executing:  "C:\Program Files\MPLAB IDE\MCHIP_Tools\mpasmwin.exe" /q /p16F84
"MOT1.asm" /l "MOT1.lst" /e"MOT1.err"
Warning[205]  C:\MOT1.ASM 24 : Found directive in column 1. (PROCESSOR)
Warning[224]  C:\MOT1.ASM 44 : Use of this instruction is not recommended.
Message[305]  C:\MOT1.ASM 56 : Using default destination of 1 (file).
Warning[207]  C:\MOT1.ASM 58 : Found label after column 1. (DEC)
Error[122]  C:\MOT1.ASM 58 : Illegal opcode (Count)
Error[113]  C:\MOT1.ASM 71 : Symbol not previously defined (Timer)
Error[113]  C:\MOT1.ASM 73 : Symbol not previously defined (again1)
Error[129]  C:\MOT1.ASM 92 : Expected (END)
Halting build on first failure as requested.
BUILD FAILED
```

To generate this list, deliberate errors were introduced into the demo program MOT1.ASM, and the messages selected from the error file MOT1.ERR. There are three levels of error shown: 'Message', 'Warning' and 'Error'. The source code line number where the problem was found is indicated, and the type of problem that the assembler thinks is present. However, a word of warning: owing to the presence of the error itself, the assembler may be misled as to

the actual error. Consequently, the message generated is not always entirely accurate. For example, the incorrect instruction mnemonic at line 58 caused the assembler to misinterpret 'Count' as an illegal op-code.

The PROCESSOR directive was misplaced, causing a non-fatal warning, which would not itself prevent successful assembly of the program. The TRIS instruction also caused a warning in the MPLAB assembler, because its use is not recommended, but will still be successfully assembled. It is used in our examples because the alternative method of port initialization, using register bank selection, is more complicated.

The instruction mnemonic DECF was misspelt as DEC, causing the errors at line 58. This contributed to the register label count being misinterpreted. The register label 'Timer' was missed out of the EQU statements at the top of the program, causing the error at line 71. The jump destination 'again1' has been incorrectly labeled 'again', causing the error at line 73. Finally, the END directive had been omitted at the end of the program, causing the message 'Expected (END)'.

The message 'Using default destination of 1 (file)' refers to the fact that the full syntax for MOVWF instruction has not been used. Using the full syntax, the destination for the result of the operation is specified as the file register or the working register, by placing a W (0) or F (1) after the destination register number or label. In the examples throughout this text, we take advantage of the assumption by the assembler that the destination is the file register if not specified in the instruction; this simplifies the source code. When the error messages have been studied carefully, and printed out if necessary, the source code must be re-edited and reassembled until it is correct. The different levels of error message (message, warning and error) can be selectively suppressed in the list file output using the list file options with the directive LIST. These options can also be set in the Project, Build Options dialogue in MPLAB by selecting the Output Category under the Assembler tab, then the preferred Diagnostics level.

9.2. Logical Errors

When all syntax errors have been eliminated the program will assemble successfully, and the hex file will be created. However, this does not necessarily mean that it will function correctly when downloaded to the chip; in fact, it probably won't! Usually there will be logical errors, particularly when learning the programming method. Mistakes in the program functional sequence or syntax will prevent it operating as required. For instance, the wrong register may be operated on or a loop may execute correctly, but the wrong number of times. There may also be 'run-time' errors, that is, mistakes in the program logic that only show up when the program is actually executed. A typical run-time error is 'Stack Overflow', which is caused by CALLing

a subroutine, but failing to use RETURN at the end of the process. This is not detected by the assembler.

9.2.1. Simulation

Once the program has been successfully assembled, it could be tested by downloading to the hardware and running it. If there are logical errors, and the output is incorrect, the source code must be modified, rebuilt, downloaded and tested again. In the early days of microprocessors this was the only option, but it is time consuming and inefficient. The only way round this was the use of an emulator system, which plugged into the processor socket in place of the MPU and allowed register monitoring, single stepping and breakpoints to be used. Such emulator systems were expensive, and the program read-only memory (ROM) still had to be reprogrammed out of circuit each time the source code was corrected, unless this was also emulated.

Some form of virtual testing of the program sequence was therefore desirable, to eliminate logical errors before downloading. The rise in the use of microcontrollers meant that, in small systems at least, the program memory, processor and peripheral interfaces were all on one chip, which could be simulated in software as a complete package. This allowed the program logic to be tested and modified more quickly, with all the internal registers displayed while tracing through the program for the correct sequence of operations. Source code debugging allows the execution point reached in the source code to be identified while single stepping through the program. Step-over and breakpoint setting allow execution of selected portions of the program at full speed to reach the problem areas or the code quickly.

Thus, a major advantage of the microcontroller over an equivalent discrete microprocessor system is that the design of the chip is fixed, so a full simulation model can be supplied for each device. The Microchip simulator tool MPSIM allows windows to be opened to show the source code, machine code, registers, simulated input, timing checks and so on. The MPLAB development system, including the assembler and simulator, has always been provided free by Microchip to encourage the development of the market for its chips.

Proteus VSM provides an alternative debugging environment, which is integral with ISIS schematic capture. It can be run independently or as a debugging tool from within MPLAB. The latter arrangement is preferred for more complex assembler and C applications, when the MPLAB project management tools are required. For simple assembler programs, MPLAB is not required. The schematic is entered using the ISIS graphics editor, and the program written using the integrated source code editor. It is assembled using the same Microchip MPASM assembler as provided in MPLAB integrated development environment (IDE), which is included in the Proteus package.

The resulting HEX machine code program file is then attached to the MCU. The program can then be run and debugged in a source code window, which includes three single stepping options and simple breakpoints. The procedure for interactive testing in Proteus VSM is detailed in Appendix E.

9.2.2. Program Testing in MPLAB

The simulator must model the operation of the selected microcontroller as completely as possible. The user must be able to provide the inputs that would occur in the actual system, and to monitor the effect on relevant registers, especially the outputs. The program will need to be started and stopped at critical points, single stepped to check the sequence of operations, and timing measured. All possible input events and sequences must be anticipated and tested, to ensure that no unforeseen problems arise when the application is in use.

In MPLAB, with the source code loaded and assembled, select 'Debugger', 'Select Tool' and 'MPLAB SIM' from the main menu. The debugging toolbar should appear with the buttons:

- RUN: Execute the program.
- HALT: Stop the program with the current execution point indicated.
- ANIMATE: Run the program in auto-step mode.
- STEP INTO: Execute program one instruction at a time, including subroutines.
- STEP OVER: Single step current routine, but execute subroutines at full speed.
- STEP OUT: Exit from the current subroutine at full speed and wait.
- RESET: Start again at the top of the program.

These controls allow the source code to be debugged using the single stepping options with breakpoints. Various windows can be opened via the View and Debugger menus, which assist with debugging in MPSIM. The most commonly used are described below (see Figure 9.1).

Edit Window (Open/New File Buttons)

The Edit window is used to create and subsequently modify the source code, PROG1.ASM, as a text file. Ensure it is saved in a project folder with the rest of the assembler fileset, as further simulation files will be added. When the program has been assembled and is then run from the debug controls and halted, the current instruction is indicated in the source code window by a green arrow. Breakpoints can be inserted by double-clicking on the line required, causing a red marker to appear in the margin.

Special Function Register Window (View Menu)

All the special function registers (SFRs) are displayed in this window in hex, binary and decimal (right-click on the column headings bar to select the format). We may want to read

Table 9.1: PIC 16F84A SFRs

Address	Name	Function
—	WREG	Working Register (not an SFR)
00	INDF	Indirect file addressing access register
01	TMR0	Timer0 8-bit hardware counter
02	PCL	Program Counter keeps track of execution point
03	STATUS	Flag register, Zero = bit 2
04	FSR	File Select Register for indirect addressing
05	PORTA	I/O port bits connected to external pins RAx
06	PORTB	I/O port bits connected to external pins RBx
08	EEDATA	EEPROM data access register
09	EEADR	EEPROM address access register
0A	PCLATH	Program counter latch high byte
0B	INTCON	Interrupt control register
81	OPTION_REG	Option register for bank selection
85	TRISA	Data direction register for PORTA
86	TRISB	Data direction register for PORTB
88	EECON1	EEPROM control register
89	EECON2	EEPROM control register

a counter in decimal, but a data direction (TRIS) register in binary. A basic set of SFRs found in the 16F84A is shown in Table 9.1.

The functions of these registers have been described in Chapters 5 and 6. More complex chips have additional SFRs, and the address of a register given here may be different. For example, the 16F887 has ports A−E, using registers 05−09, so the electrically erasable programmable read-only memory (EEPROM) access registers are moved to bank 2, address 10Ch and 10Dh. The SFRs displayed will therefore change according to the chip selected in Configure, Select Device.

Watch Window (View Menu)

A watch window allows selected registers to be displayed, that is, only those of interest in a particular application. SFRs and user-labeled registers are added separately, and the numerical format for each can be selected individually. This allows, for example, counters to be displayed in decimal, and port and status bits in binary.

Simulator Stimulus (Debugger Menu)

In most applications, a sequence of inputs needs to be generated to test the response to all possible combinations. In MPLAB, a workbook is created to specify and store the simulated inputs. The simplest method is to use the Asynch (asynchronous input) table; each input pin is assigned a row in a stimulus table, and is operated manually during the course of the simulation. Alternatively, a schedule of input changes may be created which allow the same test sequence to be generated each time the program is run using the Pin/Register Actions tab.

Stopwatch (Debugger Menu)

The stopwatch window records the simulated time elapsed and number of instructions executed. It can be zeroed to measure intervals between events, for example, the delay created by a software loop, or the period of an output pulse. The MCU clock rate must be entered via the Debugger, Settings dialogue to match the oscillator frequency to be used in hardware.

Trace Window (View Menu)

As the program is executed, the trace window displays a disassembled version of each line with the corresponding machine code and addresses, so the changes can be checked and recorded. At the same time, the original source code is displayed in a lower window.

Logic Analyzer (View Menu)

This displays the input and output bits individually or in groups on a timebase display, as would be seen on an oscilloscope. This gives a much more immediate view of the system performance and allows event timing to be more easily checked.

9.2.3. Setting up MPSIM

The setup for a typical application, MOT1, will now be described step by step. If necessary, assemble the program, using the Project, Quickbuild option, or rebuild the project using the Build All button. Ensure that the correct processor is selected, via the Configure, Select Device dialogue (16F84A for the MOT1 project). At the same time, it is advisable to set the chip fuses via Configure, Configuration Bits. For MOT1, uncheck 'Configuration Bits set in code', selecting oscillator = RC, watchdog timer off, power-up timer on and code protection off. Set the processor clock frequency via Debugger, Settings, Osc/Trace (100 kHz for MOT1).

The program can now be run and stopped to make sure the simulator is working. When halted, the current execution point is indicated in the margin of the source code window. We now need to set up the simulator so that the relevant information is displayed and the correct program function can be confirmed, or logical errors corrected. Simply running the program in the simulator does not generally provide enough information to confirm its correct operation, let alone to debug it. Single stepping allows the program to be executed one instruction at time; the registers can then be checked for the right contents as the execution progresses.

In the Watch window, use Add SFR to display the PCL, WREG, PORTA and TRISA registers, and Add Symbol to display the Count and Timer registers using the buttons. Select and right-click on PORTA, select Properties from the drop-down menu and change the display format to binary so the state of the individual bits can be seen. Repeat for TRISA. In the same dialogue, change the format of the Count and Timer registers to decimal.

9.2.4. Testing with Asynchronous Inputs

We are aiming for a setup as shown in Figure 9.1. The simplest method of simulating inputs is the asynchronous stimulus. Single-bit inputs are changed by the user via on-screen buttons while the program is executed in single-step mode. The source code, Stopwatch, Stimulus, Watch and Trace windows are displayed.

The trace facility (View, Simulator Trace) provides a comprehensive record of program execution by displaying the program line from the list file alongside the source and destination register address and contents (SA, SD, DA, DD) after each instruction, as well as the number of cycles completed. This can be saved, printed and studied. The original source code is also displayed in the lower pane when single stepping.

To set up the simulated inputs, select Debugger, Stimulus, New Workbook and the Asynch tab. In the Pin/SFR column, select RA2, RA3 and RA4, and Toggle mode for each in the Action column. Add a comment stating its function. Set all the inputs high by clicking on each fire button and stepping once. Check that the port A input bits are set high in the SFR window.

Reset the program and single step through the initialization sequence. Step through the initialization sequence to the label ‘start’ and check that the register ‘Count’ is initialized correctly. With the ‘run’ input high, the program should wait at the ‘start’ label. Now clear

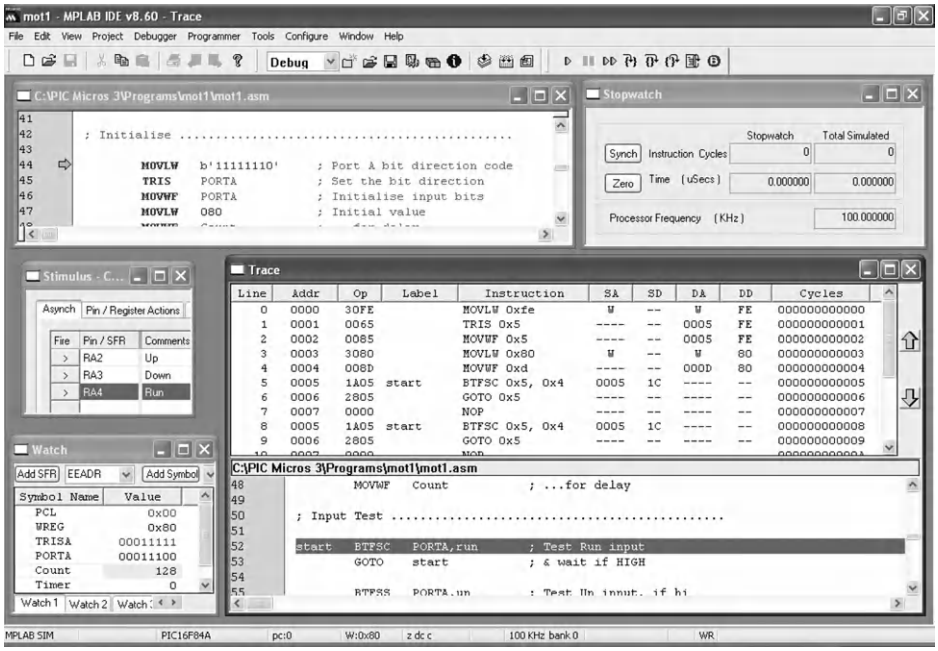


Figure 9.1
MPSIM test with asynchronous inputs and trace

input RA4 by hitting the 'Fire' button, and step to the start of the delay sequence. Once in the delay sequence, single stepping is not helpful, so we will now use breakpoints to test the main loop. Set a breakpoint at the 'cycle' label by double-clicking in the source code line number margin; a red marker appears. Set a breakpoint at the BCF instruction as well. Reset the program and run to the first breakpoint.

Zero the stopwatch, run to the next breakpoint and note the time (15 ms). Zero the stopwatch again and run to the first breakpoint. The time should be similar (16 ms), indicating a mark/space ratio (MSR) of approximately 50%. At the same time, check that RA0 is toggling each time, which indicates that the output PWM signal is present.

The up and down control can now be tested. Hit the 'Up' button (RA2) and check that the Count value increments for each output cycle. Disable 'Up' (high) and enable 'Down' (low). The Count value should decrement. Now disable the breakpoints (right-click) and run the program. The Count value should go down to 1 and stop there. Now enable the 'Up' input and check that Count goes to its maximum value (255) and stops there. With the breakpoints reinstated, the stopwatch can be used to check the minimum and maximum MSR values.

The single-step facility has two options, 'step into' and 'step over'. Step into means execute all the instructions including those in subroutines. As we have seen above, the delay sequence is not suitable for single stepping as it is repetitive. If the delay is in a subroutine, as in BIN4, step over can be used; this will step through the current routine, but will run any subroutines called at full speed. This allows each program block to be tested separately. However, as the delays in MOT1 are not subroutines, breakpoints are used to allow them to run at full speed.

9.2.5. Testing with Scheduled Inputs

Testing a program can be automated using a stimulus workbook. The state of an input or file register is changed at a particular step in the program, and the same test sequence can then be applied each time the simulation is run, making the testing quicker and easier, particularly in more complex applications.

The workbook is opened via Debugger, Stimulus, New Workbook. Select the 'Pin/Register Actions' in the workbook window, and 'Click here to Add Signals'. The output (RA0) and input (RA2,3,4) bits are added to the active window, and the bit state is changed at specific times, measured in instruction cycles. The stimulus sequence used for MOT1 can be seen in the stimulus window in Figure 9.2.

The simulator logic analyzer in the view menu will provide a time-based trace of changes in individual input/output (I/O) register bits. The Channels button opens a dialogue where the required bits are selected for display. In order to record the outputs over a suitable time period, a break must be set after a suitable number of cycles. This can be setup in the Debugger,

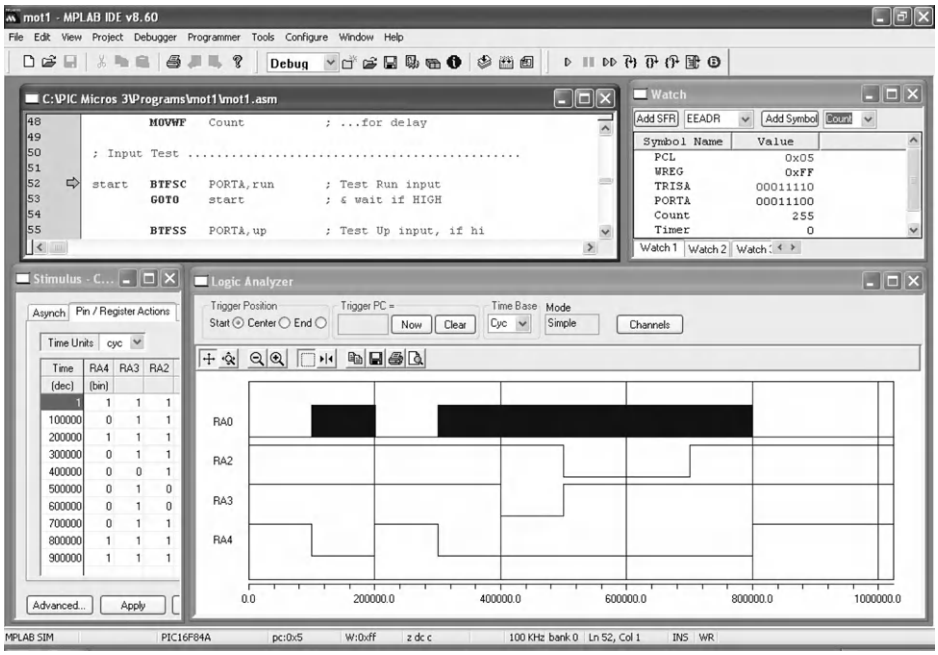


Figure 9.2
MPSIM test with scheduled inputs and logic analyzer

Settings dialogue, by checking the Break on Trace Buffer Full box and setting the buffer size to, say, 1000k for the stimulus inputs scheduled in the workbook as above. This provides enough time for the inputs to take effect, over a period of about 10 s.

A trace of the output at RA0 is obtained as shown with these settings (it appears solid at full scale). It can be seen that it is switched on when RA4 (run input) is low, and off when RA4 is high. RA3 (down) low decreases the MSR and RA2 (up) increases it. To see the effect of these inputs on the output waveform, the zoom controls must be used to expand the waveform in a selected area. The overall timescale has to be adjusted so that the duty cycle has enough time to reach the minimum and maximum values. Some experimentation is needed to get the right combination of overall timescale and input timing. The scheduled test allows exactly the same test sequence to be used each time, and a record kept if required, at the cost of increased setup time.

9.3. Test Schedule

Using the simulator, the program function can be tested against the design requirements. The specification for MOT1 has been converted into a test procedure designed to exercise all its functions; we also need to anticipate incorrect input sequences that may cause a problem. A test procedure for MOT1 is suggested in Table 9.2.

Table 9.2: Simulation test schedule for MOT1

Project: MOT1					**Simulator: MPLAB 8.60**
Setup: Source code: MOT1.ASM Watch registers: PORTA, Timer, Count *Simulator Stimulus: RA2 = up, RA3 = down, RA4 = run (toggle mode)* Stopwatch: clock frequency 100.00 Hz *Optional: Program memory*					
	Test	**Action**	**Required performance**	**OK?**	**Fault/comment**
1	Initialize	RA2, RA3, eRA4 = 1	Check Watch Window, PORTA		All inputs inactive
2	Start	Step Over	Count = 80 Waits in Start Loop		Waiting for Run Enable
3	Enable Run	Input: RA4 = 0 Step Over	RA0 = 1 Runs into high delay Timer decrements to 0 RA0 = 0 Runs into low delay Timer decrements to 0 Repeats		One cycle of output at default MSR 50% Stopwatch: Output Period ~ 33 ms
4	Disable Run	Input: RA4 = 1 Step Over	Returns to start loop		Waiting for Run Enable
5	Select Increment	Input: RA2 = 0 RA4 = 0 Step Over	Count increments to 81, 82 after next cycle etc.		Count increments MSR increasing
6	Test for no roll-over	Run at full speed & stop	Maximum Count = FF Count NOT to 00		Roll-over prevented
7	Select decrement	Input: RA3 = 0 RA4 = 0 Step Over	Count decrements to FE FD after next cycle, etc.		Count decrements MSR decreasing
8	Test for no roll-under	Run at full speed & stop	Minimum Count = 01 Count NOT to 00		Roll-under prevented
9	Restart	All inputs = 1 Run & Stop	Returns to start loop		Restart correct
10	Program reset	Reset	Execution reset to first instruction		Reset correct
11	Recheck default output	RA4 = 0	Count = 80		Output Toggles MSR ~ 50%
Tested by: Comments:			Date:		

9.3.1. Typical Logical Errors

It is difficult to anticipate exactly what kinds of logical errors will arise, as they are generally the result of inexperience, but the following are typical:

- *Port initialization*: If a port does not appear to respond to output operations, check that the initialization is correct.
- *Register operations*: If a register is not responding as it should, ensure that the correct register is being modified, and the address label is correct. Check for correct bank selection during setup.
- *Bit test & skip*: Obtaining the correct sequence of operations in the program depends on these instructions. Make sure the skip condition logic is correct, as it is easy to get this wrong.
- *Jump destinations*: Make sure that the destination specified is correct, and the loop sequence includes all the necessary steps.
- *Program structure*: If the program gets lost during subroutine execution, check that call address labels are correct, and all subroutines are terminated with 'return' instructions.

9.3.2. Limitations of Software Simulation

The simulator allows the program logic to be tested before running the program in the actual hardware, to ensure that it is functionally correct. However, the simulation is cannot be 100% realistic, and its limitations need to be taken into account in testing the real system. The following is given as an example of the kind of problem that might easily be missed, but seriously affects the operation of the application, and would compromise safe operation of the real system if a more powerful motor were used.

The data sheet for the 16F84A shows that the state of the port A bits is unknown after a power-on reset. Therefore, the motor output may come on during the power-on timer phase, before the program starts executing. This is obviously a potential problem, which is only partially addressed by following the port initialization instructions with one to clear the motor output bit. There could still be an on pulse to the motor before the start of the program. If this caused a problem in practice, a suitable fix would be needed; for example, a separate fail-safe contactor in the power circuit that ensured that the motor was not powered up until the controller had been successfully started, which is probably desirable in any case.

9.3.3. Hardware Testing

The test procedure looks very detailed when written down, but in practice it does no more than test all the features of MOT1. It can be converted to workbook stimulus sequence as indicated in the previous section. The software product needs to meet the specification only once, in

prototype hardware. Once the software is proved, the hardware in the production units can be tested in conjunction with firmware (final ROM program) that is known to be correct. A similar test schedule could be used to test each unit, using an oscilloscope to monitor the output waveform at RA0. It would then be useful to measure the actual resulting motor speed for the range of output duty cycles generated (the motor will typically not run below a minimum MSR). Alternatively, a special test program could be written to exercise the hardware, before downloading the working version.

9.4. Interactive Debugging

Interactive simulation of microcontroller circuits provides a powerful extra dimension in application design and debugging. Proteus VSM offers a complete design package, with schematic editing and capture, using a library of components which provides a graphical symbol for the schematic, a mathematical model for circuit analysis and component pin-out for use in the circuit layout. These are provided in two software packages, ISIS schematic capture and ARES PCB layout. Detailed guidance on using Proteus VSM is given in Appendix E.

9.4.1. ISIS Schematic

The application design concept may be outlined using a block diagram and converted to a provisional circuit, which can be entered into ISIS. The components are selected from the library of parts, placed on the screen and connected using virtual wiring. The result is a schematic such as that for MOT1 seen in Figure 9.3, which includes a virtual oscilloscope for monitoring the output.

Mixed mode simulation uses standard circuit modeling techniques for the analogue parts of the circuit. For example, the mathematical model for a resistor is $V = IR$. The behavior of reactive components (e.g. capacitors and motors) is modeled using complex mathematics, which is used to represent the phase relationships between voltage and current, and hence transient and frequency response in switching circuits. Overall, the analogue parts of a circuit are represented as a set of nodes connected together by these components, forming a mesh. This is then represented by a set of simultaneous equations in the form of a matrix, which can be solved for any given input signals. This process is repeated at intervals to predict the output and provide a dynamic, interactive model of the circuit.

Modeling the digital circuit elements, on the other hand, is simply a question of representing the logical processes performed by the components using a suitable logical model. For example, the output of an AND gate is $A \cdot B$ (see Appendix B). The microcontroller is represented by a combination of its internal logic and the application program attached. The effects of all types of component can then be combined to give a reasonably complete model

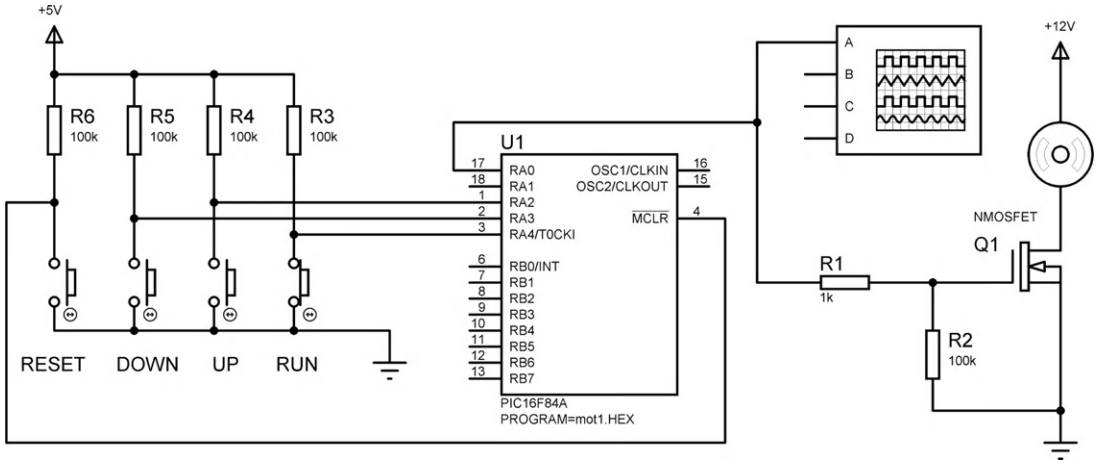


Figure 9.3
MOT1 circuit schematic

of the whole circuit. The main limitation of the simulation is that it sometimes fails to converge on a solution, especially in complex or high-speed circuits. The model then needs to be simplified, resulting in a less accurate simulation. Of course, the model is only an approximation to the real circuit in any case, especially when modeling complex devices such as motors.

The components used in MOT1 are mostly generic types, that is, they do not represent a specific component. For example, the resistor has a model described as an 'analogue primitive' or generic model. The capacitor is the same. The push button is an active device, that is, it can be operated on screen, in either transient (click on button) or latched (click on control spot) mode. The generic NMOS FET (see Appendix B) could be replaced by a specific component that has a model tailored to the operating parameters of that particular device. The dc motor parameters can be changed in the properties dialogue (i.e. operating voltage, nominal speed and coil resistance).

The microcontroller, in contrast, is a specific device from a particular manufacturer. It needs a machine code program (hex file) to be attached to determine its operation. This can be created in a built-in editor and, if a PIC, assembled using a copy of MPASM included with the simulation package. The hex file is attached via the component properties dialogue (double-left-click on the component). The clock speed (100 kHz) and configuration word can also be defined in this dialogue. The components are selected from the component libraries as illustrated in Figure 9.4.

The oscilloscope is one of a selection of virtual instruments. The power supplies are represented by a ground (0 V) connection and a power terminal whose supply voltage is

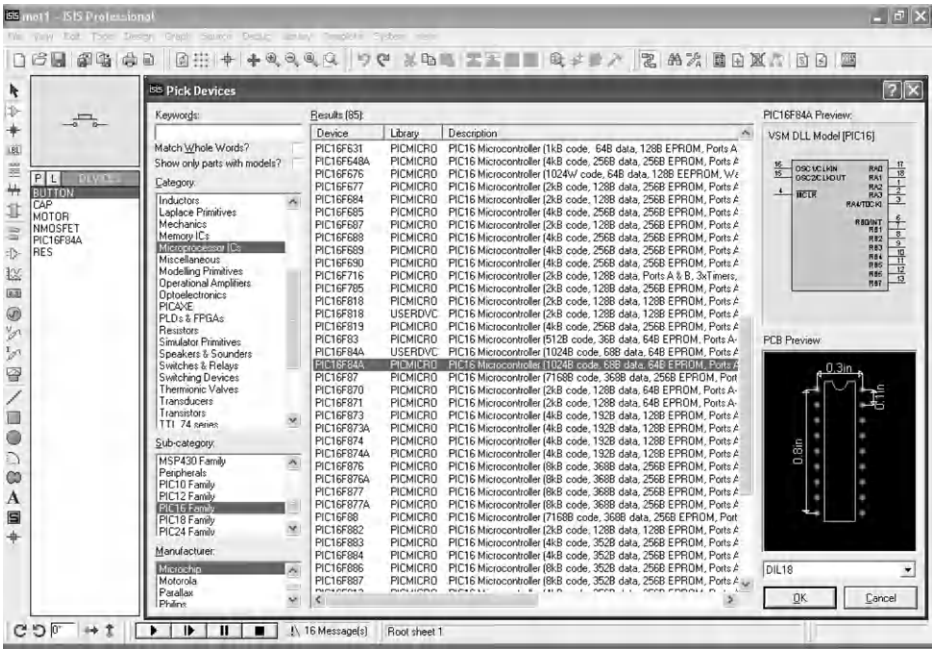


Figure 9.4
VSM component selection

specified by the label (+5 V and +12 V). These additional items are selected via the mode toolbar buttons at the left of the edit window (see Appendix E for details).

9.4.2. VSM Debugging

When the circuit is complete, and the MCU program attached, the simulation can be run. The logic levels on each connection are displayed as blue (low) and red (high). Clicking on the Run button in the schematic should make the motor appear to turn. The Up button should make it speed up and the Down button slow down. The Reset will restart it at the default speed. This can be seen more easily by displaying the PWM signal on the virtual oscilloscope. If the program needs debugging, the source code can be displayed and single stepping and breakpoints applied. A screenshot of the VSM debugging setup is shown in Figure 9.5.

The benefits of both the more extensive debugging and project management tools in MPLAB and the user-friendly interface of Proteus VSM can be realized by running VSM as the debugging tool from within MPLAB (Figure 9.6). When selected as the debugging tool in the MPLAB menu, a VSM viewing window opens, allowing the current program in memory to be run in interactive mode. Debugging and program modification are still carried out using the

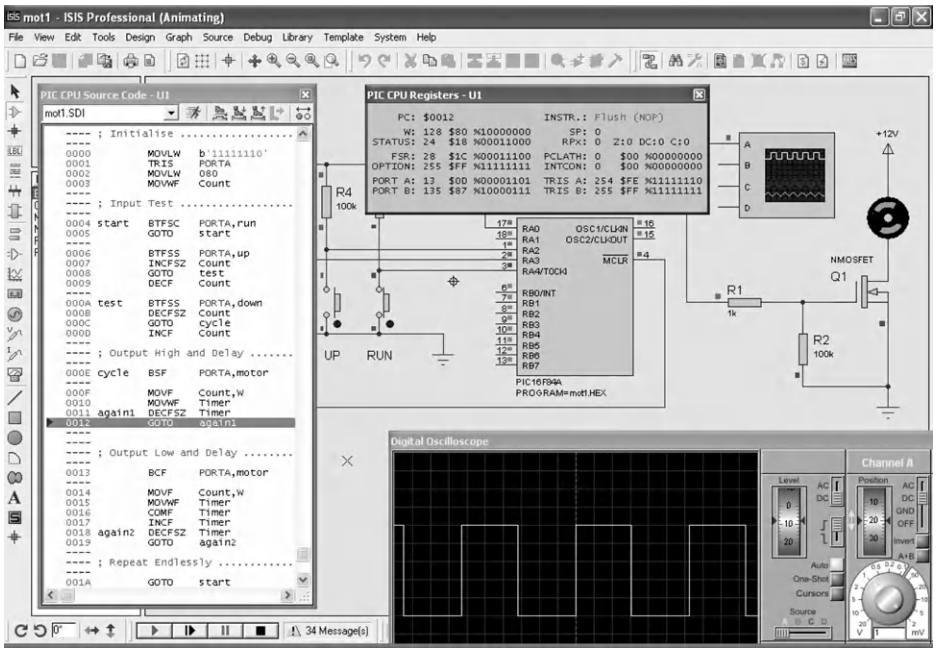


Figure 9.5
VSM debugging screenshot

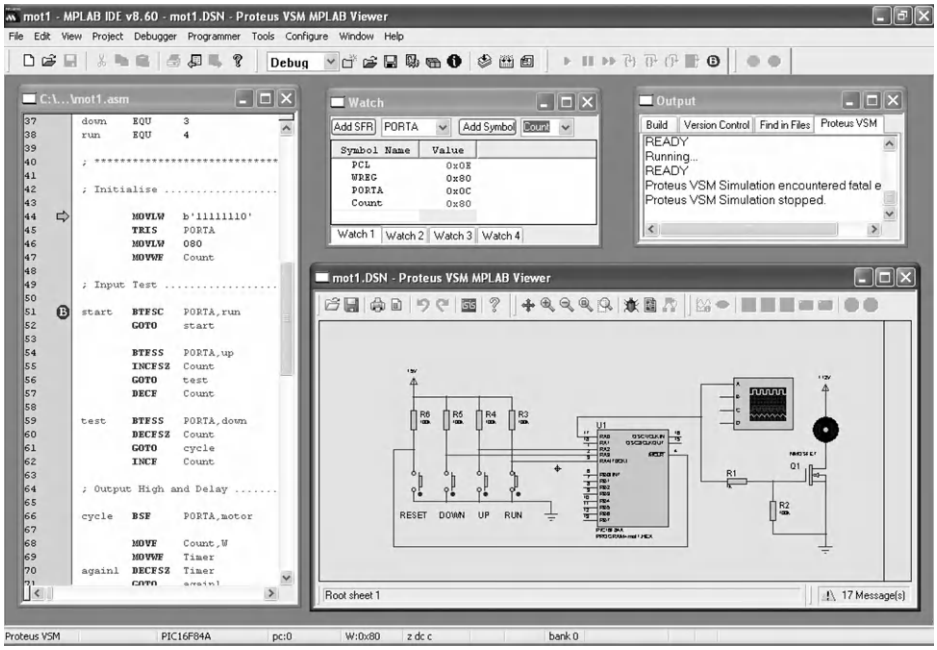


Figure 9.6
VSM running from MPLAB

same MPLAB controls and facilities. However, if the schematic needs modifying, it must be re-edited in ISIS itself.

9.5. Hardware Testing

Hardware construction will be covered in more detail in Chapter 10. For the moment, we will assume that suitable hardware has been constructed for MOT1 and we are ready to test it. When the program has been fully debugged in a simulator, it can be downloaded to the chip. In some cases, the chip must be programmed before fitting, specifically 16F84A-based circuits. For most current PIC chips, however, it is preferable to program it in-circuit via the six-pin in-circuit serial programming (ICSP) connection (see Chapter 7). In-circuit debugging, as described in Chapter 7, can also be used in the final stages of testing. This allows the program to be exercised in conjunction with the final hardware, using the debugging tools in MPLAB. The state of the registers in response to real inputs can then be monitored for fault-finding purposes.

The target hardware layout and connections should be carefully checked and tested before inserting the chip. The populated circuit board should be inspected for correct assembly; wrong component orientation, solder bridges between tracks and dry joints are common faults. The connections can be buzzed out with a continuity tester and checked against the circuit diagram.

Before fitting the microcontroller chip, it is a good idea to apply power and check the rest of the circuit. Make sure the components connected to the chip outputs can be safely powered up with an open circuit input. For example, in the MOT1 circuit, the field effect transistor (FET) gate input should not be allowed to float, so there is a pull-down resistor fitted. The supply current should not be excessive, and components should be checked for overheating. The voltages at the chip power supply pins (V_{DD} and V_{SS}) of the chip should also be checked, as incorrect connection of the supply is likely to damage most integrated circuits.

If all is well, switch off the power and fit the chip using a suitable tool. Antistatic precautions should be observed, but PIC chips have not been found to be particularly sensitive in practice. Make sure it is the right way round! Pin 1 should be marked on the board. Switch on and check that the chip is not overheating or drawing excessive current. If left to overheat for more than a few seconds, the chip will probably be destroyed.

Connect an oscilloscope to the output. On power-up, there should be no output from MOT1. When the ‘Run’ button is pressed, the default output waveform with a 50% MSR should be observed, running at a frequency of about 30 Hz. The speed ‘Up’ and ‘Down’ buttons should be operated to ensure that the speed control stops at the minimum and maximum value, and does not roll over from zero to full speed in one step. Note that the program algorithm does not

give an MSR of 100% or 0%, but stops one step short of the maximum and minimum. Since there are 255 steps altogether, the step size is less than 1%.

The circuit should also be tested for fail-safe operation, that is, no unplanned or potentially dangerous output is caused by an incorrect input operating sequence. In this case, operating both the 'Up' and 'Down' buttons together would be an erroneous input combination, which should result in no change in speed, because the increment and decrement operations cancel out.

Other examples of potential problems that would need to be considered are input switch bounce, variation in component performance (check specifications), dynamic operation of motor, minimum MSR required to make the motor run, and so on. Complex applications are likely to have more potentially incorrect input conditions and component-related problems, but the test schedule should ideally anticipate all possible fault modes (not easy!). If the circuit is being produced on a commercial basis, a formal test schedule would be needed, and the performance certificated as being correct to the product specification.

A basic test schedule for the MOT1 program running in a PIC 16F84A has already been outlined for the simulator test in Table 9.2, and this can be adapted for hardware testing. Additional documentation should be prepared according to circumstances (education, commercial, research) to provide the application user or product customer with the relevant information on using the system.

Questions 9

1. Explain briefly the difference between syntax and logical program errors, and how they are detected. (4)
2. How are the following used in program debugging: single stepping, breakpoint, pin stimulus, watch window? (4)
3. An instruction in the program memory listing appears as follows:

```
0005  1A05 start    btfsc     0x5,0x4
```

   Explain the meaning of each of the six elements in the line, such that its correct effect can be predicted when debugging. (6)
4. State two advantages of interactive debugging using Proteus VSM. (2)
5. State two checks to be carried out before powering up a new prototype microcontroller board. (4)

Answers on pages 422–3. (Total 20 marks)

Activities 9

1. (a) In MPLAB, open a source file edit window, enter or download the source code for MOT1 and assemble it using the Quickbuild option. Note any error messages generated. If the program assembles correctly first time, put some deliberate errors in the source code and inspect the error messages.
   (b) In MPLAB, create a project MOT1, assign MOT1.ASM to the project, reassemble and test the program using scheduled inputs as described in Section 9.2.5.
2. (a) Design an application for the LPC demo board with 16F690 to control the brightness of an LED using PWM. When the input button is pressed, the LED should increase in brightness up to a maximum and then reduce back to the minimum, so that it can be stopped at any point by releasing the button. Demonstrate correct operation in MPLAB using an asynchronous stimulus.
   (b) If Proteus VSM for 16 series PIC is available, download the simulation of the LPC board to test the dimmer program interactively. Modify it so that the LED brightness is controlled by the pot.

CHAPTER 10

Hardware Prototyping

Chapter Outline

Chapter Points

- Breadboard allows circuits to be prototyped quickly and easily.
- Stripboard is more reliable, but not reusable.
- ISIS software is used to create a schematic, and to test the design by simulation.
- ARES software is used to create a circuit board design.
- The DIZI board demonstrates a range of simple applications.

PIC Microcontrollers. DOI: 10.1016/B978-0-08-096911-4.10010-2

Circuit design, simulation and layout software has developed to the point where comprehensive packages are now available at reasonable cost, which can be used to create microcontroller-based circuits. Schematic capture software allows a circuit to be created on screen, printed and saved as a netlist of components and connections. This can be imported into a printed circuit board (PCB) design package, where the circuit is laid out on screen. The layout can be printed onto a masking sheet to transfer the layout to copper-coated board manually, or a file can be generated which is used to manufacture the PCB on a production system. Proteus VSM™ software from Labcenter Electronics is used here as it provides the most complete support for PIC® designs currently available. This consists of two main parts: ISIS™ for circuit design and schematic capture, and ARES™ for PCB layout.

Traditional prototyping methods, breadboard and stripboard, will also be discussed, as these require minimum expenditure and need only simple tools to create a prototype microcontroller test circuit. Breadboard is purely temporary, easy to construct, but unreliable, particularly if moved about. Stripboard is more permanent, and the standard technique for hobbyists to build semi-permanent boards, but is not suitable for production.

10.1. Hardware Design

Before computer-based electronic computer-aided design (ECAD) was widely available, circuits would be designed as sketches on paper and a layout produced manually. In the absence of fast and powerful computer simulation methods such as SPICE, this process relied more heavily on the experience of the design engineer to be able to predict the circuit performance from theoretical knowledge and practical experience. Numerous prototypes might be needed to arrive at a working solution.

Since the development of increasingly powerful desktop computers, the design process has been radically improved. The designer still has to come up with the original ideas, but circuits can now be rapidly drawn and tested on screen, and a working design quickly produced. The hardware prototype is much more likely to work first time, or at least require less development time. The time taken from design concept to market is regarded as a major competitive factor, so ECAD is now a vital tool for the electronics engineer, just as computer-aided design (CAD) has become in mechanical engineering.

A circuit schematic can now be created, tested and converted into a PCB layout within a single software package, such as Proteus VSM. Complete libraries of all the most commonly used components and microcontrollers are available, which consist of a mathematical model, on-screen circuit symbol and, in selected cases, a physical component pin-out for each. A set of animated components also allows interactive simulation. A circuit can be drawn on screen, the application program attached to the microcontroller and the program tested by operating the on-screen inputs, such as switches and or a keypad, with the mouse. The outputs are then seen

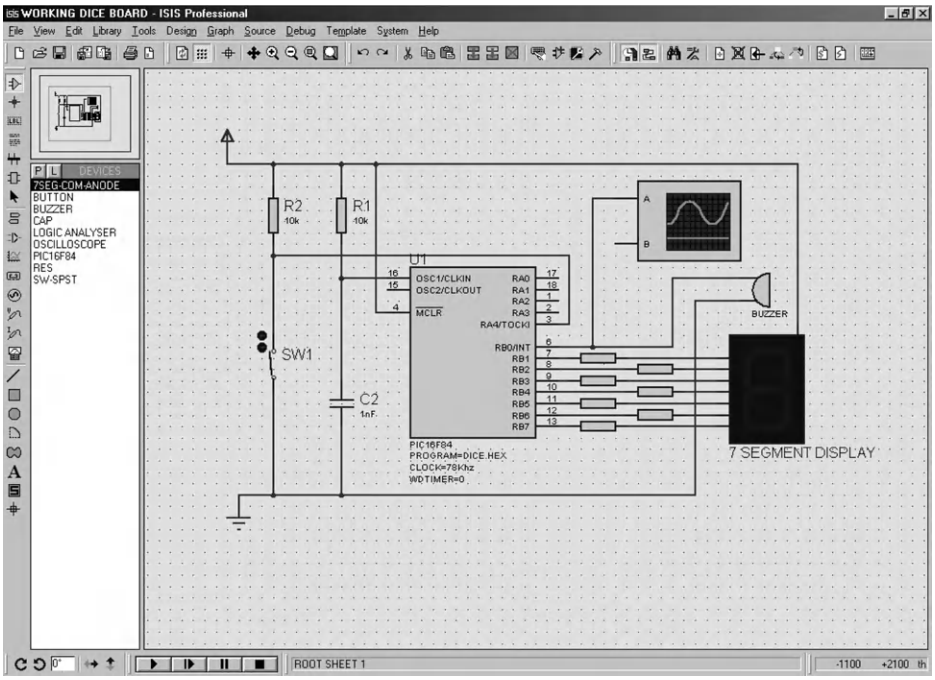


Figure 10.1
ISIS circuit design and test for dice board

on simulated displays (LED and LCD), or operate animated output devices, such as relays and motors. Examples can be seen in previous chapters.

A further example is shown in Figure 10.1, which will be converted into a PCB layout. It is an electronic dice board with a push button, seven-segment display and buzzer controlled by a PIC 16F84A. It can be programmed to display a random number between 1 and 6 when the button is pressed.

When a suitable program is assigned to the PIC in the simulation (see below), the circuit becomes interactive on screen. When the switch is operated, the display will operate in the same way as the real device. If the chip is programmed to generate a sound output via the buzzer, the waveform can be displayed on a virtual oscilloscope, and reproduced at the PC audio output. The techniques for developing the firmware (Flash ROM software) have been explained in detail in previous chapters.

10.2. Hardware Construction

First, we will look briefly at some traditional techniques suitable for building one-off boards and prototypes, using the dice circuit in Figure 10.1. Then a slightly more elaborate general

purpose demonstration board will be designed and laid out in prototype form, and some programs provided to demonstrate its features and the related programming principles.

10.2.1. Printed Circuit Board

The PCB is the standard method for making electronic circuits. In its basic form, it starts life as a sheet of insulating glass fiber-reinforced epoxy resin, with a layer of copper on one side. The circuit connections are made by printing or photographically transferring a pattern of conducting tracks and pads onto the copper.

The layout for a simple PIC circuit is shown in Figure 10.2. It has a PIC 16F84A, push button, seven-segment display, buzzer and associated components. It can be programmed to operate as electronic dice, generating a random number between 1 and 6. The pattern of the copper tracks is shown, as well as the 'silk screen' printing, which is applied to the component side of the board to show where to place the components.

The layout is reversed, as it will be printed onto a translucent mask, which is then used to create the pattern of connections on the copper side of the board. The copper layer is coated with a light-sensitive material, which is exposed to ultraviolet light through the mask. In the exposed areas of the board, the photosensitive material becomes soluble and is removed by a caustic solvent, exposing the copper below. This is then dissolved (etched) in an acid bath, leaving behind the copper layout where it was protected by the etch-resistant layer. The components are then fitted to the top side of the board, and the leads and pins soldered to the pads.

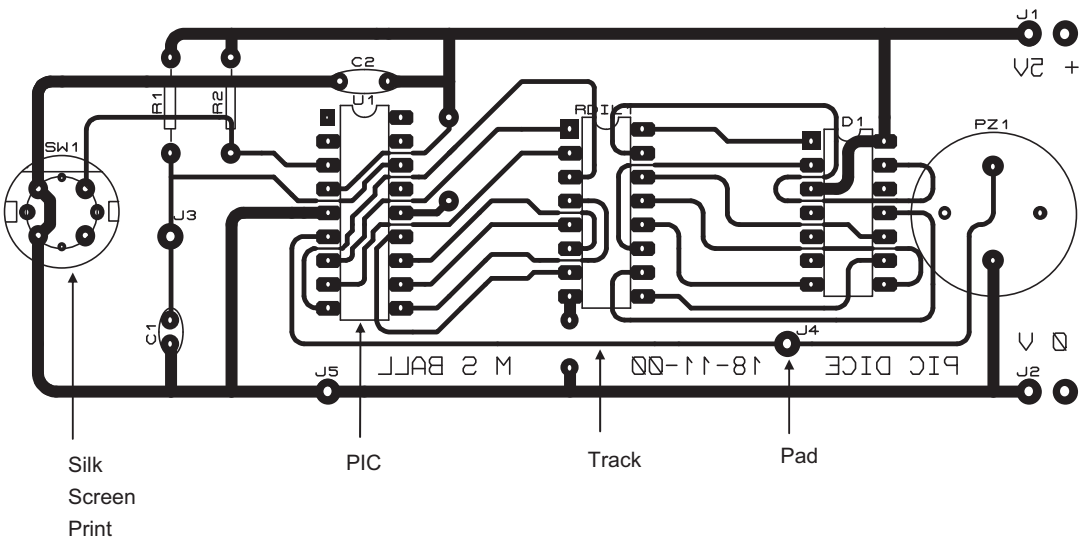


Figure 10.2
PIC dice board layout (courtesy Melvyn Ball, SCCH)

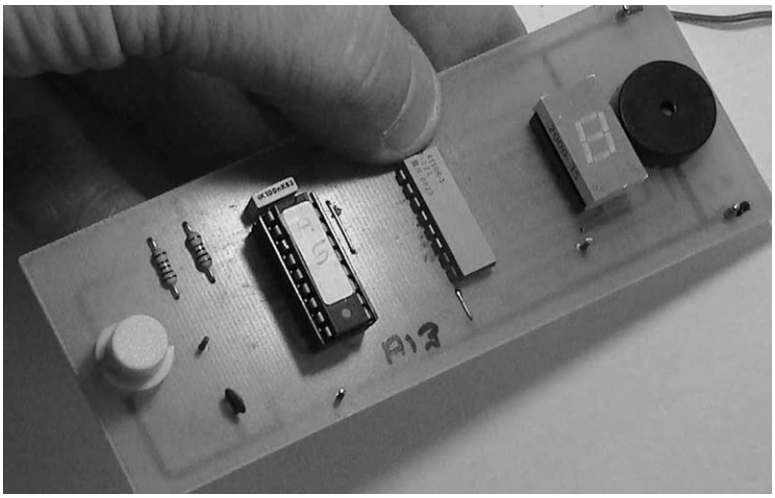


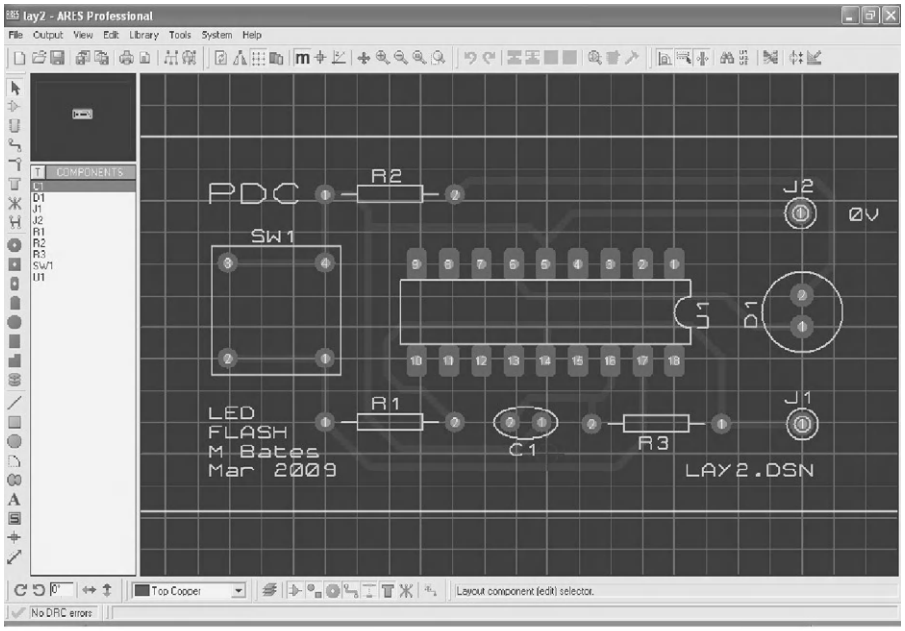
Figure 10.3
PIC 16F84 dice board

Once the layout has been designed, it can be used for batch production of the application hardware. Specialist companies are now often used to manufacture the boards direct from the file output of the PCB design software, as the cost of short production runs is now lower, owing to the application of advanced manufacturing techniques. The final product is shown in Figure 10.3.

Another simple single-sided layout is shown in Figure 10.4(a) on the edit screen of ARES PCB software. This package allows the ISIS schematic to be imported and converted to a layout for printing or export to a manufacturing system in a standard format (usually a Gerber file). Before transfer from ISIS, each component needs a suitable pin arrangement attached, depending on the actual component to be used. For example, the physical size of a resistor depends on its power rating, which consequently affects the pin spacing. Switches have a great variety of pin outs, or may be mounted off the board so suitable terminals must be provided. The ARES library provides standard pin-outs, or they can be created if necessary for non-standard components.

The netlist is then exported to ARES, and the list of components appears in a window at the left of the edit screen (Figure 10.4a). The components may be selected and placed individually on the layout edit screen and their positions adjusted for the most compact arrangement (auto-placement is also available). Initially, the connections are shown as direct lines between pins ('rat's nest'). These are converted into tracks when the auto-router utility is invoked. Final adjustments to the track layout are completed manually. If necessary, wire links may be used on a single-sided board to achieve a viable layout. A double-sided board simplifies the track arrangement a great deal, but plated through-holes or connecting pins are then necessary. When the layout is complete, a three-dimensional preview of the populated board can be generated (Figure 10.4b). The use of ARES is described in more detail in Appendix E.

(a)



(b)

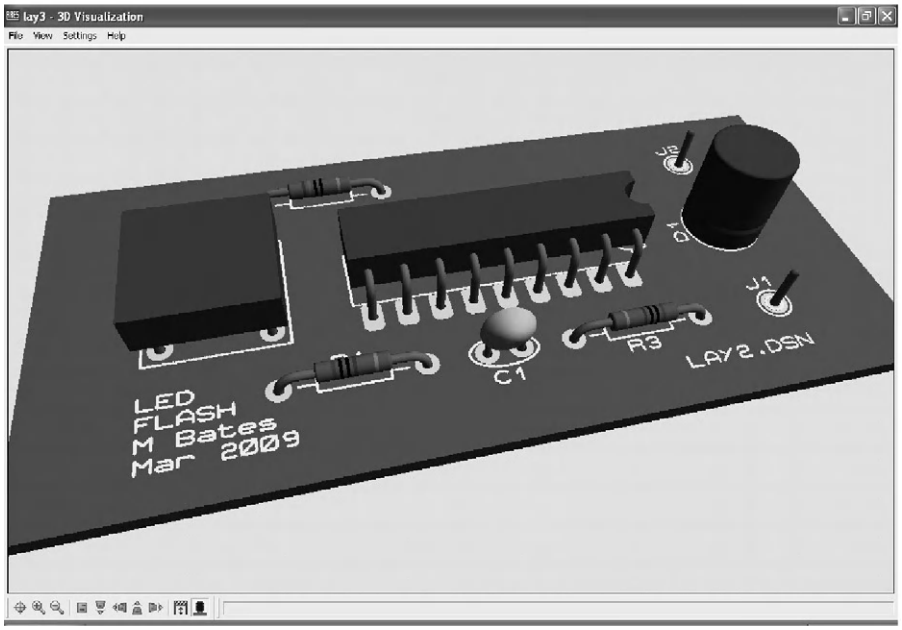


Figure 10.4
ARES PCB layout screenshots: (a) editing the layout; (b) 3D view of final layout

The Microchip LPC demo board is an example of a double-sided board commercially produced in large quantities. Using both sides of a board allows it to be more compact overall, and simplifies the track layout. In general, tracks are oriented in a common direction on each side, perpendicular to the other side. More complex boards, such as the PC motherboard, have multiple layers sandwiched together to provide the large number of connections required by the system busses. Holes are plated through to make the connections between different layers, and a printed silk-screen layer carries the component labeling (or legend) on the top side. High-resolution printing techniques in the commercial production process allow a finer track width (where the current is small) and, overall, a more precise, compact layout.

Most production designs now use surface-mounted components, which are smaller and do not require through-holes for mounting, but are all soldered to the surface layer simultaneously by flow soldering. The surface-mount version of the PIC 16F887 can be seen on the Microchip 44-pin board. Full details of these boards can be downloaded from www.microchip.com, with the layouts provided in the user manuals. Both boards have prototyping areas so that simple peripheral circuits can be added without having to design a test board from scratch.

An alternative method for making simple prototype boards, which has recently become viable for the hobbyist, training organizations and small businesses, is the PCB mill. This is essentially a small 2.5D (dimensions) milling machine. An engraving tool is mounted on X, Y and Z (limited travel) axes and programmed to outline the copper tracks and pads to isolate them from the rest of the copper layer (Figure 10.5). This avoids the use of corrosive chemicals, and is viable for small-scale production.

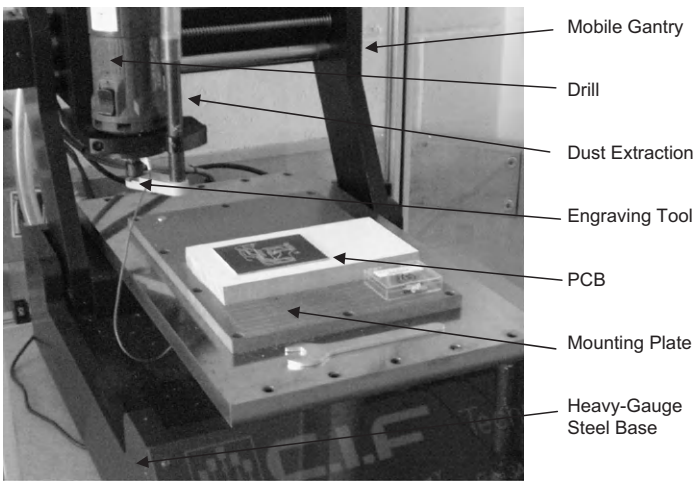


Figure 10.5
PCB Mill

Even with the current user-friendly ECAD packages, the PCB layout can take some time to create, and a considerable amount of skill is needed to use the software. Therefore, we will also look at how to prototype our hardware using traditional methods, which do not require specialist software or PCB fabrication equipment.

10.2.2. Breadboard

Breadboard (plugboard) has sets of miniature sockets laid out on a 0.1 inch grid which will accept the manual insertion of component leads and tinned copper wire (TCW) links (Figure 10.6a). It has rows of contacts interconnected in groups placed either side of the center line of the board, where the integrated circuits (ICs) are inserted, giving multiple contacts on each IC pin. At each side of the board, there are long rows of common contacts, which are used for the power supplies. Some types of breadboard are supplied in blocks that link together to accommodate larger circuits, or are mounted on a base with built-in power supplies.

The layout for a simple circuit is shown in Figure 10.6(b), with a PIC 16F84A driving a light-emitting diode (LED) at RB0 via a current-limiting resistor. The only other components required are a capacitor and resistor to form the clock circuit, but we must not forget to connect the !MCLR (Master Clear) pin to the positive supply, or the chip will not run. The chip could now be programmed to flash the output at a specified rate.

Breadboard circuits can be built quickly, with no special tools required, other than a supply of insulated wire (recycled telephone cables are ideal) and wire cutters. However, the connections

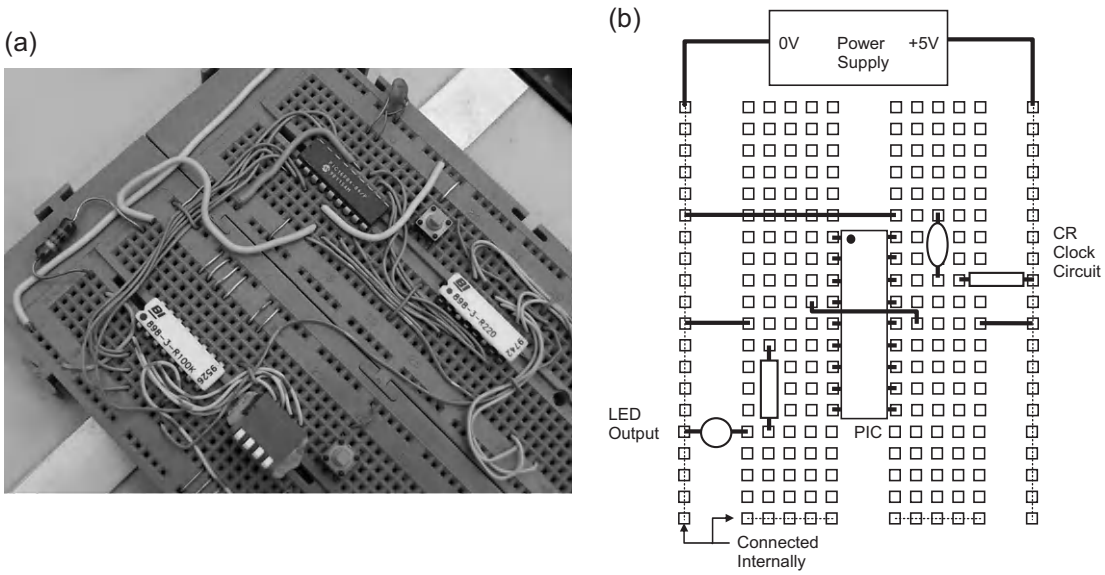


Figure 10.6
Breadboard: (a) PIC breadboard prototype circuit; (b) simple PIC circuit layout

are relatively unreliable, so bad connections are likely in more complicated circuits. Therefore, a method of producing prototype circuits with more reliable soldered connections might be preferred.

10.2.3. Stripboard

Stripboard (veroboard) requires no special tools or chemical processing. The components are connected via copper tracks laid down in strips on a 0.1 inch grid of pinholes in an insulating board (Figure 10.7a).

The components are soldered in place and the circuit is completed using wire links on the component side soldered to the tracks on the copper side. The tracks must be cut where necessary to isolate the connection nodes in the circuit using a hand drill. The components are generally placed across the tracks, so that each pin connects with a separate track. The tracks must be cut between the rows of pins in each dual in-line (DIL) chip. Care is required to avoid dry joints (too little solder) or short-circuits between tracks due to solder splashes and whiskers (too much solder). A manual drawing may used to draft the layout, if necessary, but experienced constructors will often build the circuit ad hoc, with maybe some additional wastage of board area.

Figure 10.7(b) shows how the simple PIC circuit can be laid out for construction on stripboard using general purpose drawing tools, such as those provided with Word®. In the word processor, the drawing toolbar needs to be switched on, and page layout view selected. In the Draw menu, the grid should be switched on and set to 0.1 inch; this allows layouts to be drawn actual size, since this is the spacing between standard in-line pins. The circuit can then be drawn using suitable line styles, text boxes and so on. When finished, use the Select Objects tool to select the whole drawing and Group it in the Draw menu. This prevents text cursor movement from disrupting the drawing, and the whole diagram can be repositioned on the page if required.

Naturally, the circuit can also be hand-drawn in the traditional manner.

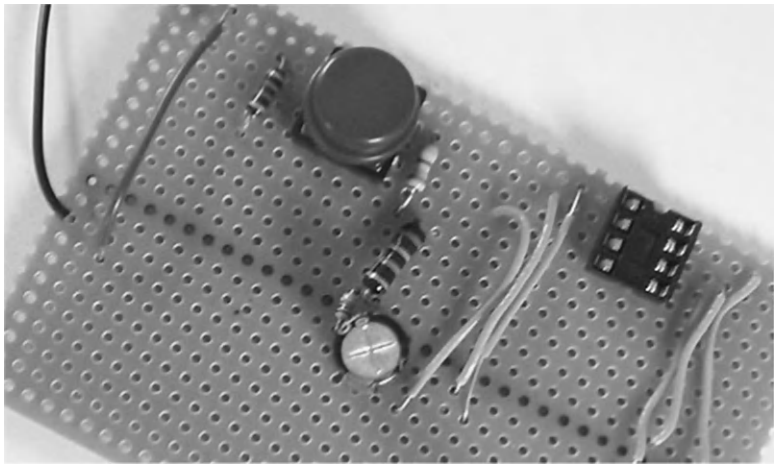
10.3. Dizi84 Board Design

A circuit will now be designed, and a set of programs provided, to illustrate the hardware design process and programming principles discussed in previous sections. The DIZI board will allow the user to experiment with the various features of the PIC hardware and programming techniques.

10.3.1. Hardware Specification

The microcontroller demonstration board will be suitable for demonstrating a range of processes incorporating display, audio, counting, timing and interrupt operations. The

(a)



(b)

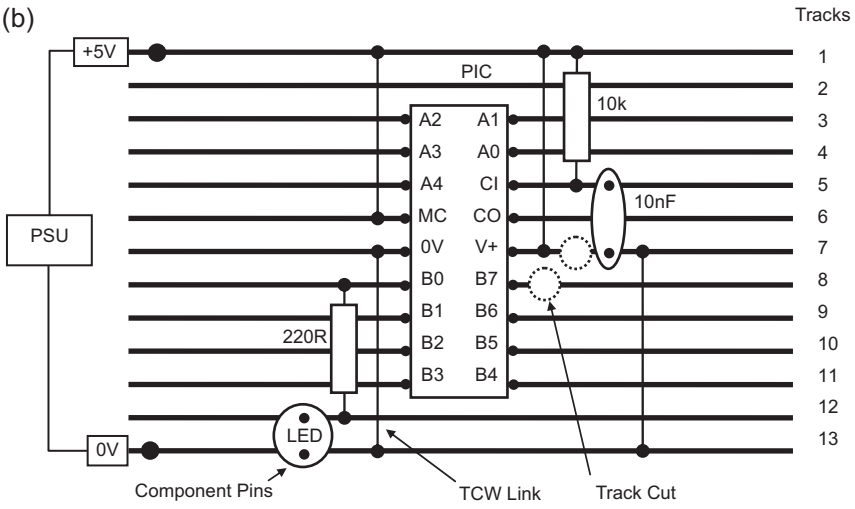


Figure 10.7
Stripboard: (a) a simple stripboard circuit; (b) PIC stripboard layout

board will have a single-digit seven-segment display for showing output data in hexadecimal or decimal form, and a low-power audio transducer. Manually operated toggle switches will provide a 4-bit parallel input. Two input push buttons will be used for general control (e.g. run, clear), to simulate input events to be counted, or to generate an external interrupt. Timed events should be measured or generated with an accuracy of better than 1%. The circuit will be battery powered, with a push button power switch to ensure that the power cannot be left on, and a power 'on' indicator. The board will be as

small as possible, and the microcontroller must be easily reprogrammable, with flash memory.

10.3.2. Hardware Implementation

The seven-segment display will require seven outputs from the microcontroller. Active high operation can be provided by a common cathode LED display, and the display decimal point can be used as the power indicator. The audio transducer requires one output. A piezo buzzer has sufficient bandwidth and output power, and its power consumption is low. A miniature DIP switch bank will be used for 4-bit input, and miniature push buttons used, to conserve space.

Fourteen input/output (I/O) pins are required; the PIC 16F84A has only 13, so a chip with more I/O, such as the 16F690, could be considered. However, the audio output and interrupt input can share the same I/O pin, because the high impedance of the buzzer will not interfere with input signals on the same pin. RB0 will be used as the dual function pin, since it is defined as the principal interrupt input, but can also be used as an output. The outputs can source up to 25 mA, but current-limiting resistors will restrict the current per display segment to 10−15 mA to control the maximum load on the port when all the segments are on. The I/O allocation for the project is shown in Table 10.1.

A crystal clock of 4 MHz will be used to obtain the required timing precision, and the convenience of a 1 μs instruction cycle. The 16LF84A-04 (LF = low voltage) can operate from a supply of between 2.0 V and 5.5 V, so the circuit will be powered from 2 × 1.5 V dry cells, giving a 3.0 V supply. The '04' suffix indicates that a maximum 4 MHz clock frequency can be used. A block diagram of the proposed system is shown in Figure 10.8. The inputs and outputs are given the labels that will be assigned in the application programs.

10.3.3. Implementation

A circuit for the DIZI board is shown in Figure 10.9. The PIC 16LF84A drives an active high- (common cathode)−low-current seven-segment LED display at port B, RB1−RB7, via a block of 220R current-limiting resistors. RB0 drives an audio sounder when set as

Table 10.1: DIZI board I/O allocation

Device	Type	Pin(s)
7-Segment display	Outputs	RB1−RB7
4-Bit switch bank	Inputs	RA0−RA3
Push button	Input	RA4
Push button interrupt	Input	RB0 (dual function)
Audio transducer	Output	RB0 (dual function)

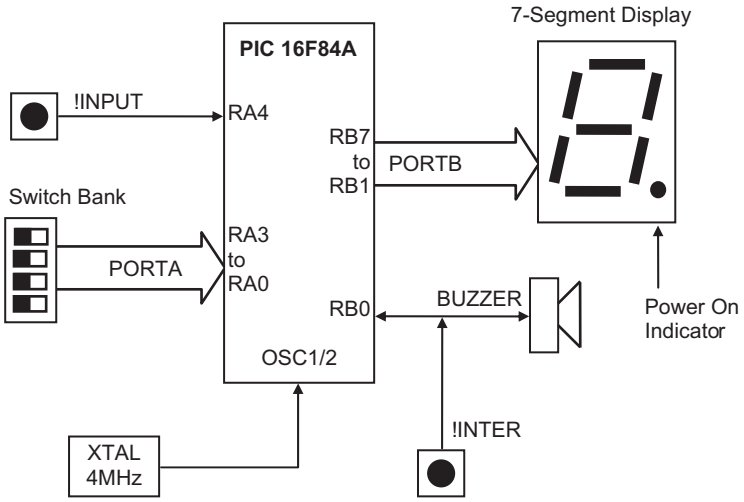


Figure 10.8
Block diagram of DIZI demonstration board

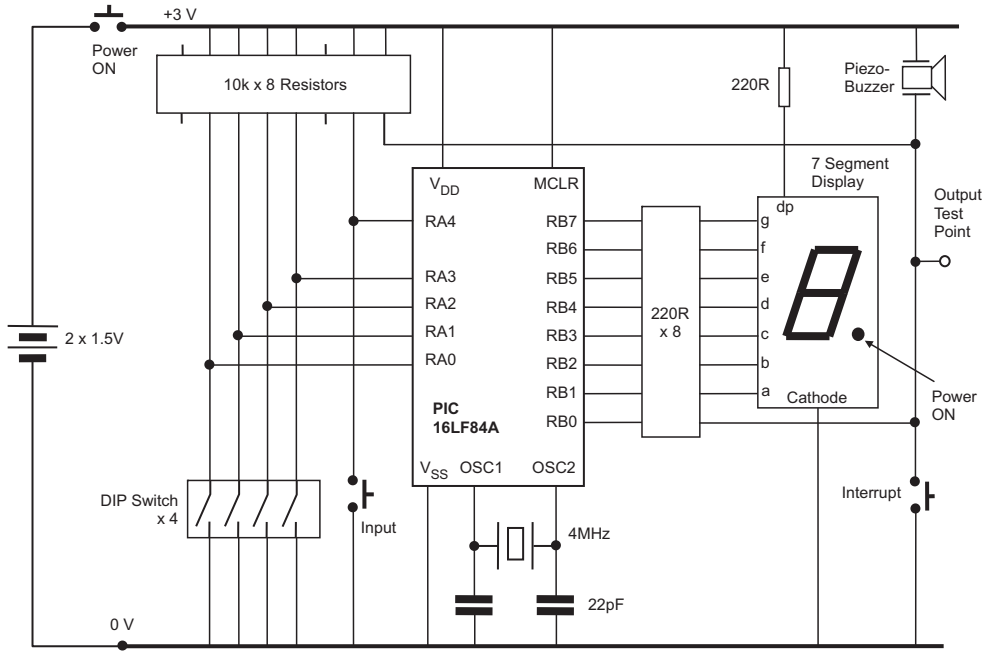


Figure 10.9
DIZI board circuit diagram

an output, but can also be used to detect the 'Interrupt' push button when set as an input and the chip is initialized for this option. To prevent RB0 being shorted to ground if set as an output, the spare 220R resistor is connected between the push button and RB0. This does not affect the operation of the sounder, which has a relatively high resistance. A 4-bit DIP switch input is connected to port A, RA0–RA3, with a push button connected to RA4, which can be used as an external pulse input to the Counter/Timer Register TMR0. These operate as active low inputs with 100k pull-up resistors, as does the interrupt push button.

A stripboard layout for the DIZI board is shown in Figure 10.10. The detail of the component pin connections has been omitted owing to the reduced scale of the illustration, but this information can be obtained from the component pin out data, when selecting particular components. The finished stripboard circuit is shown in Figure 10.11. The

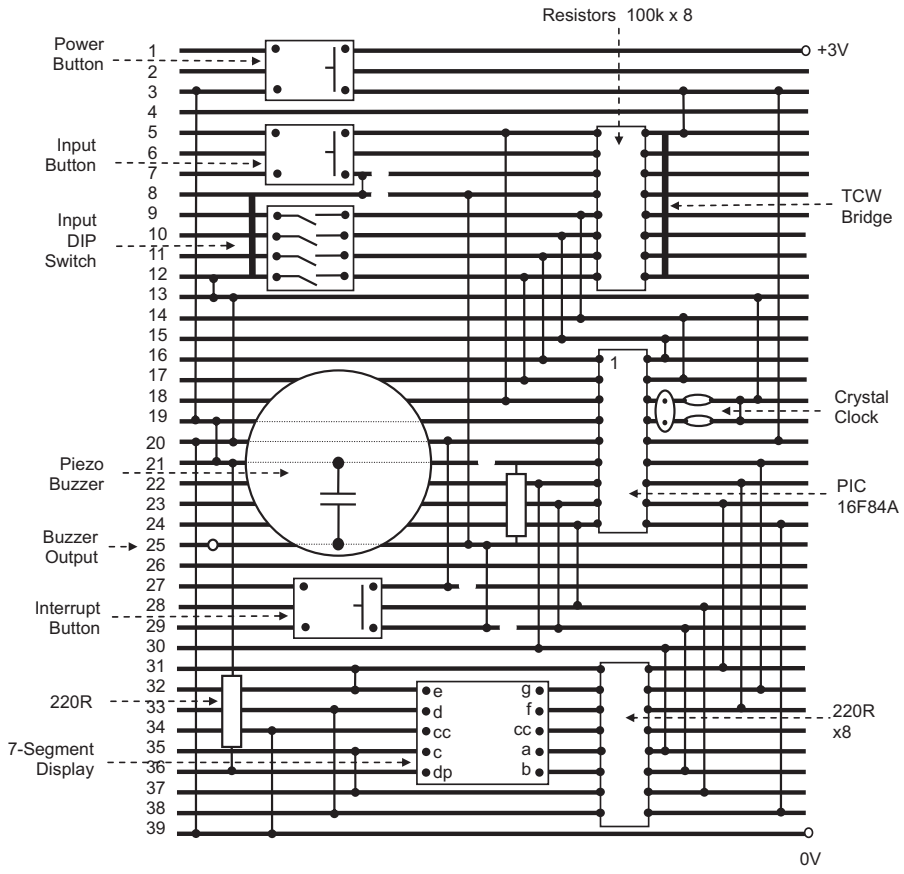


Figure 10.10
Stripboard layout for DIZI board

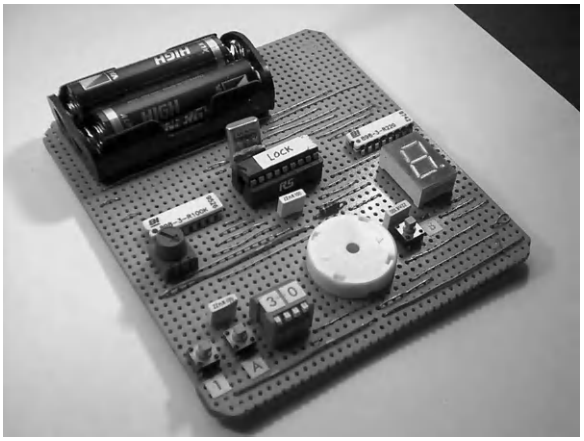


Figure 10.11
DIZI stripboard circuit

construction process for a slightly modified board is described in a little more detail in Appendix D.

10.4. Dizi84 Applications

Titles for a set of programs to run on this hardware are suggested below. Three applications (*) will be described in some detail, while a further eight will be specified and the source code listed. These form a set of applications suitable for group assignment work in a training program.

Display Apps

FLASH1	Flash all Segments
STEP1	Step through Segments
HEX1	Binary to Hex Converter
MESS1	Message Display
SEC1	One Second Clock
REACT1	Reaction Timer
DICE1*	Electronic Dice

Sound Apps

BUZZ1*	Output Single Tone
SWEEP1	Sweep Tone Frequency
TONE1	Switch Tone On/Off
SEL1	Select Tone on Switches
GEN1	Audio Frequency Generator
MET1	Metronome

GIT1	Guitar Tuner
SCALE1*	Musical Scale
BELL1	Doorbell Tune

Interrupt Apps

STEP1	Step Through Scale
STEP2	Step Scale and Display Note
BUZZ2	Output Tone using TMR0
REACT2	Reaction Timer using TMR0
SEC2	One Second Clock using TMR0
MET2	Metronome using TMR0

EEPROM Apps

STORE1	Store a display sequence in EEPROM
STORE2	Store a tone sequence in EEPROM
LOCK1	Store a code and buzz if matched

An ISIS schematic for the DIZI board is shown in Figure 10.12, which allows the above applications to be tested by simulation. All can be downloaded from the support website (www.picmicros.org.uk), and tested in MPLAB if Proteus VSM is not available. If VSM is available, virtual instruments can be used for checking the outputs. The frequency counter and oscilloscope are shown in Figure 10.13.

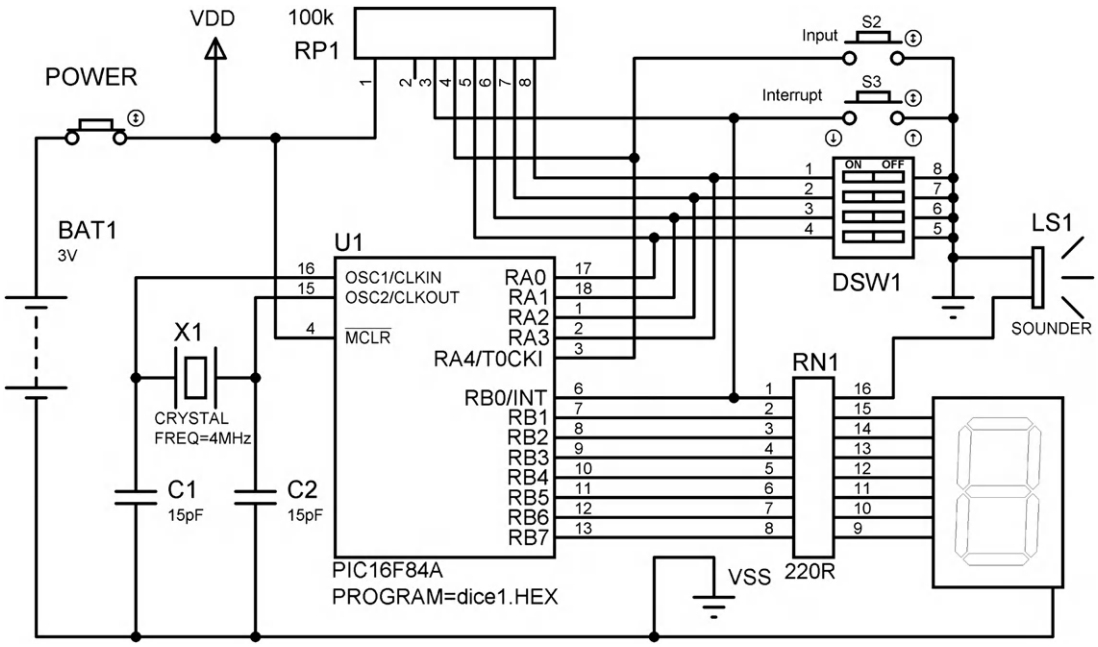


Figure 10.12
DIZI board schematic

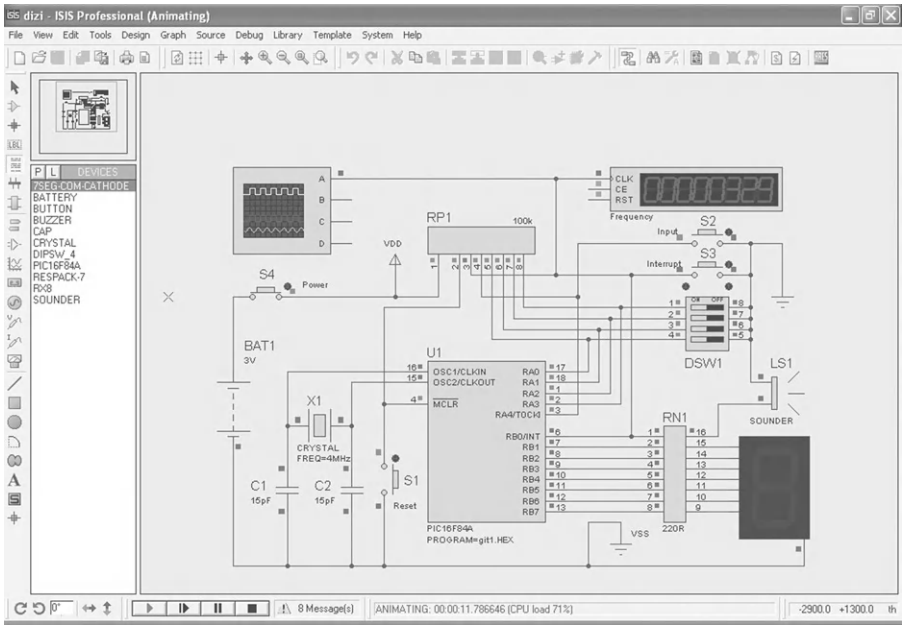


Figure 10.13
DIZI simulation screenshot

10.4.1. Program BUZZ1

A flowchart for the program BUZZ1 is shown in Figure 10.14. It will generate a single tone at the buzzer when the input button is operated, by toggling the output to the buzzer, with a delay between each change of output state. If a count of 255 is used with a 1 μs

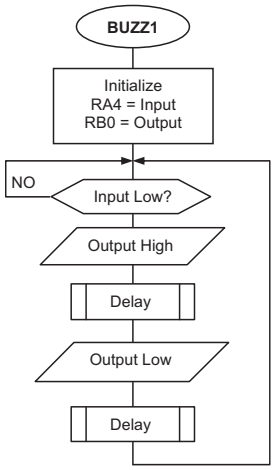


Figure 10.14
BUZZ1 flowchart

instruction cycle time, we have seen that the loop itself will take $255 \times 3 \times 1 = 765\ \mu s$, which will give an output frequency of $10^6/(765 \times 2) = 654$ Hz, which is well within the audible range.

This frequency can be adjusted by simply reducing the count value in the delay loop; 654 Hz is the minimum frequency available. A more precise calculation of the delay loop can be used to obtain a more exact frequency, or the hardware timer can be used. In either case, the period can be checked using the stopwatch in the simulator before downloading. The source code is listed in Program 10.1.

10.4.2. Program DICE1

This program will generate a random number at the display between 1 and 6 when the input button in pressed. A continuous loop will increment a register from 1 to 6, and back to 1. The loop is stopped when the button is pressed and the number displayed. The display is retained when the button is released.

First, the allocation of the segments to the pins on the display chip must be established. The segments of the display are labeled from a to g, as shown in Figure 10.15. They must be lit in the appropriate combinations to give each display number; for instance, segments 'b' and 'c' must be lit for the number '1' to be displayed. A table is useful here to work out the codes required for output to the display (Table 10.2).

The display is active high in operation. This means a 1 at the pin will light that segment. This arrangement is also described as common cathode, as all the LED cathodes are connected together at the common terminal. A common anode display will therefore operate active low. The binary or hexadecimal code for each digit will be included in the program in the form of a program data table.

The program represented in the flowchart (Figure 10.16) uses a spare register as a counter, which is continuously decremented from 6 to 0. When the button is pressed, the current number is used to select from the table of codes using the method described in Program 10.2. This results in the pseudo-random number code being displayed, and remaining visible until the button is pressed again. Because the number is selected by manually stopping a fast loop, the number cannot be predicted. In the flowchart, the jump destinations have been labeled, and these labels will be used in the program source code. The table subroutine is also named 'table' to match the source code subroutine start label.

10.4.3. Program SCALE1

This program will output a musical scale of eight tones. The frequencies for a musical scale from middle C upwards are 262, 294, 330, 349, 392, 440, 494 and 523 Hz. These can be

```
; *********************************************************
;       BUZZ1.ASM           MPB           30-11-10 Ver 1.1
; *********************************************************
;
;       Generates an audio tone at Buzzer when the
;       Input button is operated..
;
;       Hardware:       PIC 16F84 DIZI Demo Board
;       Clock:          XTAL 4MHz
;       Inputs:         RA4: Input (Active Low)
;       Outputs:        RB0: Buzzer
;       MCLR:           Enabled
;
;       PIC Configuration Settings:
;
;       WDTimer:        Disable
;       PUTimer:        Enable
;       Interrupts:     Disable
;       Code Protect:   Disable
;
        PROCESSOR 16F84A     ; Declare PIC device

; Register Label Equates.................................

PORTA   EQU     05           ; Port A
PORTB   EQU     06           ; Port B
Count   EQU     0C           ; Delay Counter

; Register Bit Label Equates ............................

Input   EQU     4            ; Push Button Input RA4
Buzzer  EQU     0            ; Buzzer Output RB0

; Start Program *****************************************

; Initialize (Default = Input) ..........................

        MOVLW   b'00000000'  ; Define Port B outputs
        TRIS    PORTB        ; and set bit direction
        CLRF    PORTB        ; Switch off display
        GOTO    check        ; Start main loop

; Delay Subroutine ......................................

delay   MOVLW   0FF          ; Standard Routine
        MOVWF   Count
down    DECFSZ  Count
        GOTO    down
        RETURN

; Main Loop .............................................

check   BTFSC   PORTA,Input  ; Check Input Button
        GOTO    check        ; and wait if not 'on'

        BSF     PORTB,Buzzer ; Output High
        CALL    delay        ; run delay subroutine
        BCF     PORTB,Buzzer ; Output Low
        CALL    delay        ; run delay subroutine
        GOTO    check        ; repeat always

        END                  ; Terminate source code
```

Program 10.1
BUZZ1 source code

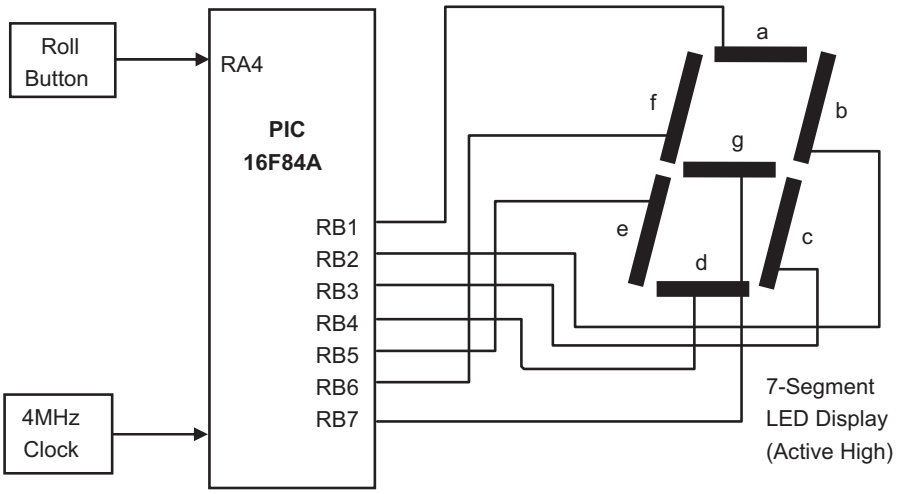


Figure 10.15
Block diagram for DICE1 system

translated into a table of delay counts which give the required tone period, since period $T = 1/f$ (s), where f = frequency (Hz). The buzzer on the DIZI board is driven from RB0, so this needs to be toggled at a rate determined by the frequency of each tone. We therefore need to use a counter register or the hardware timer to provide a delay corresponding to half the period of each tone. We have previously seen how to calculate the delay time for a loop. Using a formula for the count value derived from this analysis, figures were calculated for a half cycle of each tone, which were then placed in the data table in SCALE1.ASM. To keep the program simple, each tone will be output for 255 cycles, so we will use another register to count the number of cycles competed during each tone. The scale will then be played over a period of about 5 s. The table of values can later be modified to play a tune in the doorbell program.

Instead of a flowchart, the SCALE1 program source code listing (Program 10.3) has been annotated with arrows to show the execution sequence. This informal method of analysis

Table 10.2: DICE1 display encoding table

Displayed Digit	Segment Code (1 = Segment On)							
	g	f	e	d	c	b	a	hex
	RB7	RB6	RB5	RB4	RB3	RB2	RB1	(RB0 = 0)
1	0	0	0	0	1	1	0	0C
2	1	0	1	1	0	1	1	B6
3	1	0	0	1	1	1	1	9E
4	1	1	0	0	1	1	0	CC
5	1	1	0	1	1	0	1	DA
6	1	1	1	1	1	0	1	FA

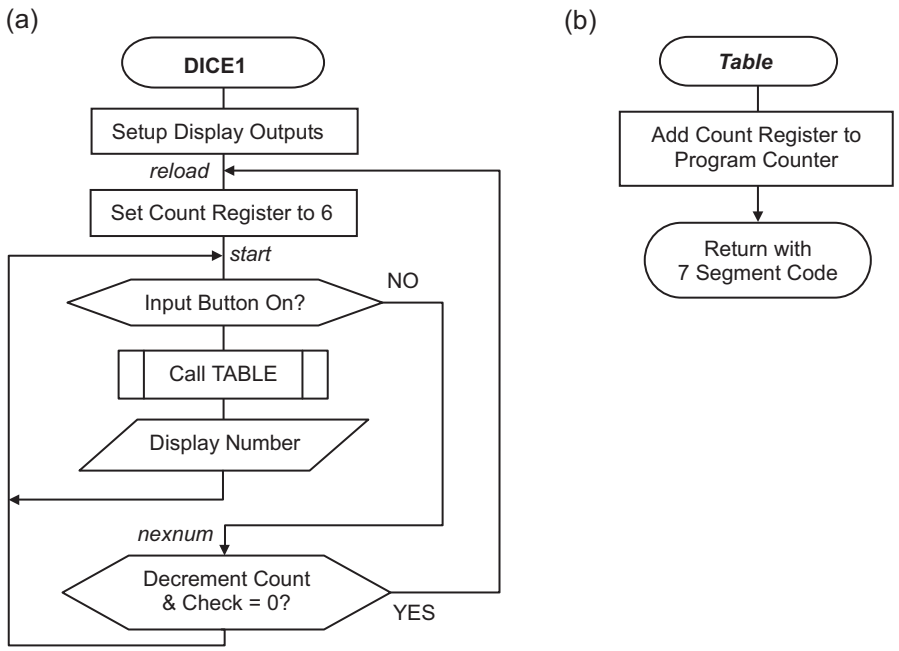


Figure 10.16
DICE1 program flowcharts: (a) main routine; (b) table routine

can be used to check the program logic before simulation. The eight tone frequencies are controlled by the value of 'HalfT', obtained from the program data table at 'getdel'. 'HalfT' is a counter value, which will give a delay corresponding to half a cycle of the frequency required when the chip is clocked at 4 MHz. The eight tones are selected in turn by the value of 'TonNum', which is initialized to 8. This is used as the program counter offset in the data table fetch operation. It is decremented in the main loop after each tone has finished to select the next. The 'HalfT' values are thus selected from the bottom of the table upwards.

The tone is generated in the routine 'note', where RB0 is set high, the delay using 'HalfT' runs, RB0 is cleared, and the second half cycle delay executed. No OPeration instructions (NOP) have been inserted to equalize the duration of each half cycle. RB0 is toggled 255 times using the 'Count' register, which gives a duration of around half a second, depending on which tone is being generated (the lower frequencies are output for longer). The main loop thus selects each of the eight values of 'HalfT' in turn, and outputs 255 cycles of each tone.

10.4.4. DIZI Application Outlines

A further eight applications are specified below, and the source code for each is listed in Programs 10.4. They can be downloaded from www.picmicros.org.uk, and tested in simulation

```
; ********************************************************
;       DICE1.ASM               MPB       30-11-10 Ver 2.0
; ********************************************************
;
;       Displays pseudo-random numbers between 1 and 6
;       when a push button is operated.
;
;       Hardware:               PIC 16F84A DIZI Demo Board
;       Clock:                  XTAL 4MHz
;       Inputs:                 RA4: Roll (Active Low)
;       Outputs:                RB1-RB7: 7seg LEDs (AH)
;       MCLR:                   Enabled
;
;       PIC Configuration Settings:
;       WDTimer:                Disable
;       PUTimer:                Enable
;       Interrupts:             Disable
;       Code Protect:           Disable
;
; Set Processor Options.....................................

          PROCESSOR 16F84A     ; Declare PIC device

; Register Label Equates....................................

PCL       EQU       02         ; Program Counter
PORTA     EQU       05         ; Port A
PORTB     EQU       06         ; Port B
Count     EQU       0C         ; Counter (1-6)

; Register Bit Label Equates ...............................

Roll      EQU       4          ; Push Button Input

; Start Program ********************************************

; Initialize (Default = Input)

          MOVLW     b'00000001'          ; Define RB1-7 outputs
          TRIS      PORTB                ; and set bit direction
          MOVLW     0FF                  ; Switch on..
          MOVWF     PORTB                ; ..all segments
          GOTO      reload               ; Jump to main program

; Table subroutine .........................................

table     MOVF      Count,W              ; Put Count in W
          ADDWF     PCL                  ; Add to Program Counter
          NOP                            ; Skip this location
          RETLW     00C                  ; Display Code for '1'
          RETLW     0B6                  ; Display Code for '2'
          RETLW     09E                  ; Display Code for '3'
          RETLW     0CC                  ; Display Code for '4'
          RETLW     0DA                  ; Display Code for '5'
          RETLW     0FA                  ; Display Code for '6'

; Main Loop ................................................

reload    MOVLW     06                   ; Reset Counter
          MOVWF     Count                ; to 6
```

Program 10.2
DICE1 source code

```
start     BTFSC     PORTA,Roll          ; Test Button
          GOTO      nexnum              ; Jump if not pressed
          CALL      table               ; Get Display Code
          MOVWF     PORTB               ; Output Display Code
          GOTO      start               ; start again

nexnum    DECFSZ    Count               ; Dec & Test Count=0?
          GOTO      start               ; Start again
          GOTO      reload              ; Restart count if zero

          END                           ; Terminate source code
```

Program 10.2: (*continued*)

mode in MPSIM or ISIS (if available). If the DIZI hardware is constructed, they can be programmed into a 16F84A chip using an out-of-circuit programmer.

HEX1 Hex Converter

The hexadecimal number corresponding to the binary setting of the DIP switch inputs is displayed. The input switches select from a table of 16 seven-segment codes which drive the display in the required pattern for each hex digit: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, b, C, d and E. Note that numbers B and D are displayed in lower case so that they can be distinguished from 8 and 0, respectively.

MESS1 Message Display

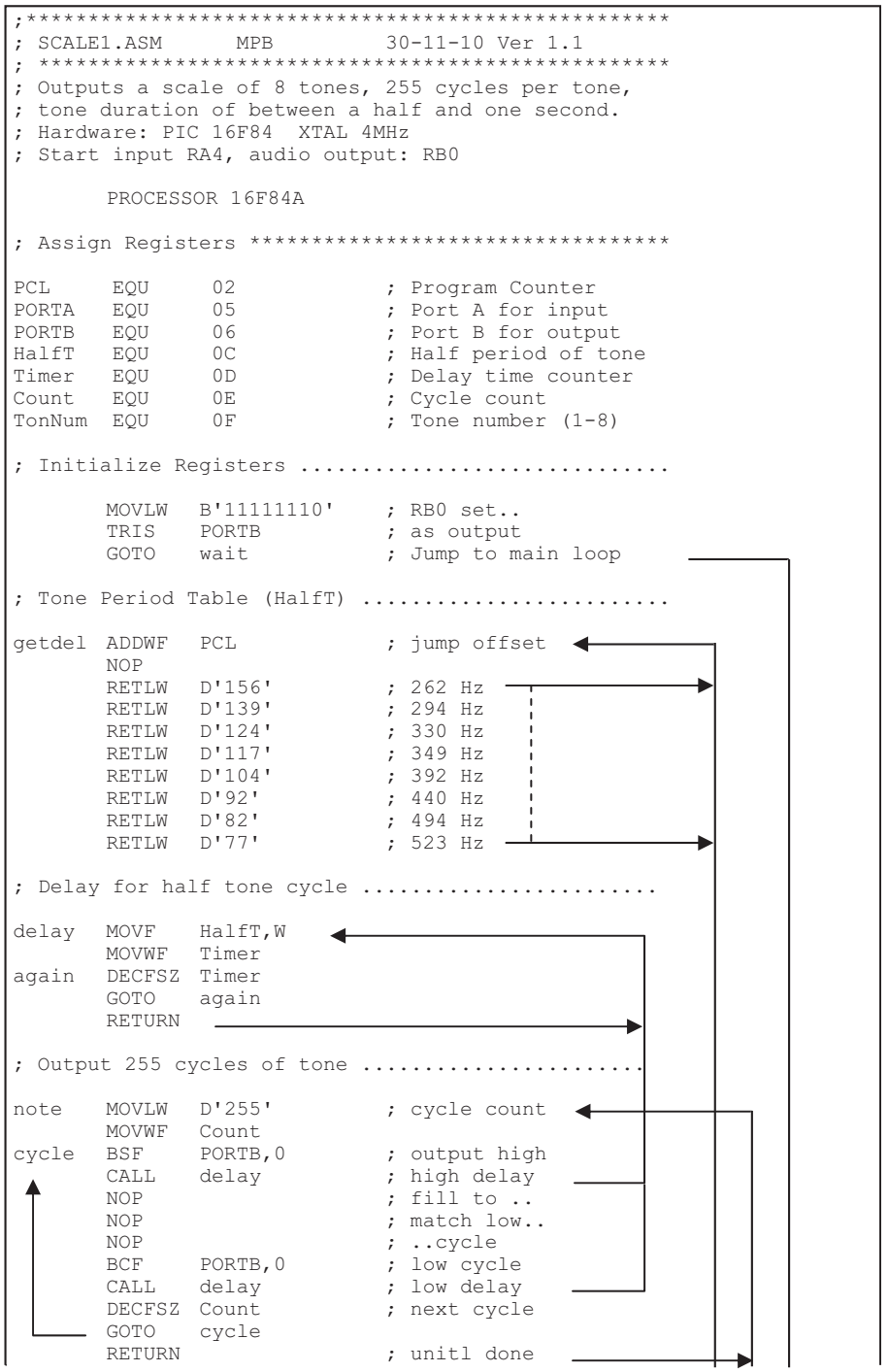
A sequence of characters is displayed for about 0.5 s each. Most letters of the alphabet can be obtained on the seven-segment display in either upper or lower case, for instance 'HELLO'. The number of characters must be set in a counter, or a termination character used.

SEC1 Second Counter

An output is displayed which counts down exactly once per second, from 0 to 9, and then repeats. A table of display codes is required as in the Hex Converter application. A 1 s time delay can be achieved using the hardware timer (Chapter 6) and spare register. A tick could be produced at the audio output by pulsing the speaker at each step.

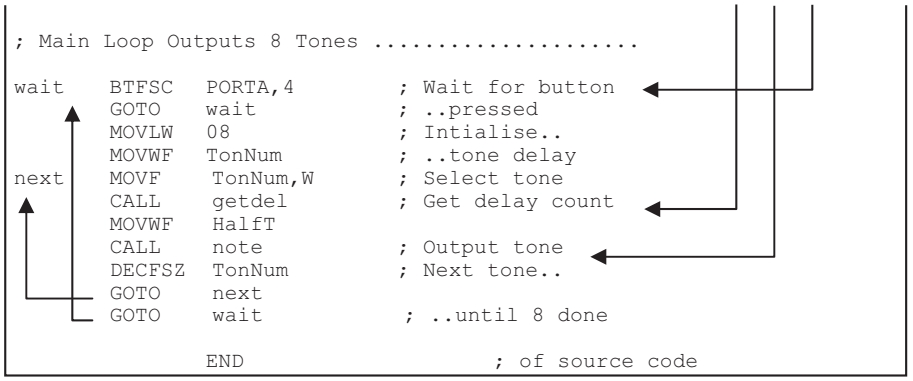
REACT1 Reaction Timer

The user's reaction time is tested by generating a random delay of between 1 and 10 s, outputting a beep, and timing the delay before the input button is pressed. A number representing the time between the sound and the input, in multiples of 100 ms, should be displayed as a number 0—9, giving a maximum reaction time of 900 ms.



```
;*************************************************
; SCALE1.ASM      MPB         30-11-10 Ver 1.1
; *************************************************
; Outputs a scale of 8 tones, 255 cycles per tone,
; tone duration of between a half and one second.
; Hardware: PIC 16F84  XTAL 4MHz
; Start input RA4, audio output: RB0

        PROCESSOR 16F84A

; Assign Registers ********************************

PCL     EQU     02              ; Program Counter
PORTA   EQU     05              ; Port A for input
PORTB   EQU     06              ; Port B for output
HalfT   EQU     0C              ; Half period of tone
Timer   EQU     0D              ; Delay time counter
Count   EQU     0E              ; Cycle count
TonNum  EQU     0F              ; Tone number (1-8)

; Initialize Registers ..............................

        MOVLW   B'11111110'     ; RB0 set..
        TRIS    PORTB           ; as output
        GOTO    wait            ; Jump to main loop

; Tone Period Table (HalfT) .........................

getdel  ADDWF   PCL             ; jump offset
        NOP
        RETLW   D'156'          ; 262 Hz
        RETLW   D'139'          ; 294 Hz
        RETLW   D'124'          ; 330 Hz
        RETLW   D'117'          ; 349 Hz
        RETLW   D'104'          ; 392 Hz
        RETLW   D'92'           ; 440 Hz
        RETLW   D'82'           ; 494 Hz
        RETLW   D'77'           ; 523 Hz

; Delay for half tone cycle .........................

delay   MOVF    HalfT,W
        MOVWF   Timer
again   DECFSZ  Timer
        GOTO    again
        RETURN

; Output 255 cycles of tone .........................

note    MOVLW   D'255'          ; cycle count
        MOVWF   Count
cycle   BSF     PORTB,0         ; output high
        CALL    delay           ; high delay
        NOP                     ; fill to ..
        NOP                     ; match low..
        NOP                     ; ..cycle
        BCF     PORTB,0         ; low cycle
        CALL    delay           ; low delay
        DECFSZ  Count           ; next cycle
        GOTO    cycle
        RETURN                  ; unitl done
```

Program 10.3
SCALE1 source code



```
; Main Loop Outputs 8 Tones ......................

wait    BTFSC   PORTA,4         ; Wait for button
        GOTO    wait            ; ..pressed
        MOVLW   08              ; Intialise..
        MOVWF   TonNum          ; ..tone delay
next    MOVF    TonNum,W        ; Select tone
        CALL    getdel          ; Get delay count
        MOVWF   HalfT
        CALL    note            ; Output tone
        DECFSZ  TonNum          ; Next tone..
        GOTO    next
        GOTO    wait            ; ..until 8 done

                END                     ; of source code
```

Program 10.3: (*continued*)

GEN1 AF Generator

An audio frequency generator outputs frequencies in the range 20 Hz to 20 kHz. The sounder output is toggled with a delay between each operation determined by the frequency required, as in the BUZZ1 program. For example, for a frequency of 1 kHz, a delay of 1 ms is required, which is 1000 instruction cycles at a cycle time of 1 μs. The information on program timing must be studied in Chapter 6. The delay time, and hence the frequency, can then be incremented using the input button, and range selection with the input switches might be incorporated, as there are only 255 steps available when using an 8-bit register as the period counter.

MET1 Metronome

An audible pulse is output at a rate set by the DIP switches or input buttons. The output tick can be adjustable from, say, 1 up to 4 beats per second, using the interrupt button to step the speed up and down, and the input button to select up or down. A software loop or the TMR0 register can be used to provide the necessary time delays.

BELL1 Doorbell

A tune is played when the input button is pressed, using a program look-up table for the tone frequency and duration. Each tone must be played for a suitable time, or number of cycles, as required by the tune. The program can be elaborated by selecting a tune using the DIP switches, and displaying the number of the tune selected.

GIT1 Guitar Tuner

The program will allow the user to step through the frequencies for tuning the strings of a guitar, or another musical instrument using the input button, or selecting the tone at the DIP switches. The program could be enhanced by displaying the string number to be tuned. The tone frequencies will be generated as for the doorbell application. The digit display codes would also be required in a table.

```
;**************************
;  BEL1.ASM   MPB 2-12-10
;.........................
;  Program to output a tone
;  Sequence (random) of 8
;  RBO = Output Buzzer
;  RAO = Input Button
;  ......................

        PROCESSOR 16F84

PCL     EQU     02
PortB   EQU     06
PortA   EQU     05
Notnum  EQU     0C
Tabnum  EQU     0D
Cycnum  EQU     0E
Count   EQU     0F

;Initialise ..............

        MOVLW   B'11111110'
        TRIS    PortB
Wait    BTFSC   PortA,4
        GOTO    Wait

; Get note ................

Start   MOVLW   08
        MOVWF   Notnum
Nexnot  MOVF    Notnum,W
        CALL    Table
        MOVWF   Tabnum

; 256 Cycles of note ......

        CLRW
        MOVWF   Cycnum
Cycle   BSF     PortB,0
        CALL    Half
        BCF     PortB,0
        CALL    Half
        DECFSZ  Cycnum
        GOTO    Cycle

; Next note of 8 .........

        DECFSZ  Notnum
        GOTO    Nexnot
        GOTO    Wait

;Half cycle delay ........

Half    MOVF    Tabnum,W
        MOVWF   Count
Down    DECFSZ  Count
        GOTO    Down
        RETURN

;Table of delay values....

Table   ADDWF   PCL
        NOP
        RETLW   D'124'
        RETLW   D'82'
        RETLW   D'117'
        RETLW   D'156'
        RETLW   D'77'
        RETLW   D'156'
        RETLW   D'92'
        RETLW   D'104'

        END
```

```
;******************************
;  GEN1.ASM  MPB 2-12-10
;  Audio generator 200Hz-20kHz
;  RB0 = Output to buzzer
;  RA0 = Decrease frequency
;******************************

        PROCESSOR 16F84A

PORTA   EQU     05
PORTB   EQU     06
Multi   EQU     0C
Count1  EQU     0D
Count2  EQU     0E

; Initialise ..............

        MOVLW   B'11111110'
        TRIS    PORTB
        MOVLW   02
        MOVWF   Multi

; Output one cycle .........

Cycle   BSF     PORTB,0
        CALL    Half
        BCF     PORTB,0
        CALL    Half
        BTFSC   PORTA,4
        GOTO    Cycle

; Reduce frequency.........

        INCF    Multi
        CLRF    Count2
Down2   DECFSZ  Count2
        GOTO    Down2
Wait    BTFSS   PORTA,4
        GOTO    Wait
        GOTO    Cycle

; Delay one half cycle...

Half    MOVF    Multi,W
        MOVWF   Count1
Down1   NOP
        NOP
        NOP
        NOP
        NOP
        DECFSZ  Count1
        GOTO    Down1
        RETURN
```

Programs 10.4
8 DIZI applications.

```
;***********************************
;  GIT1.ASM   MPB  2-12-10
;  Guitar Tuner
;  Outputs standard frequencies
;  330,245,196,147,110,82Hz
;  3030,4081,5102,6802,9090,12195us
;  Count = 30,41,51,68,91,122 x50us
;  Measured accurate to about 1%
;  RB0 = buzzer(string tone)
;  RA4 = button(next string)
;
;***********************************

PortA     EQU       05
PortB     EQU       06
String    EQU       0C
Count1    EQU       0D
Count2    EQU       0E
PCL       EQU       02

          PROCESSOR 16F84A

; Initialise ...................

          MOVLW     B'11111110'
          TRIS      PortB
          MOVLW     06
          MOVWF     String

; Output one cycle .............

Next      BSF       PortB,0
          CALL      Cycle
          BCF       PortB,0
          CALL      Cycle
          GOTO      Next

; Delay and check inputs ......

Cycle     MOVF      String,W
          CALL      Table
          CALL      Tone
          BTFSS     PortA,4
          CALL      Wait1
          RETURN

; Select next tone ...........

Wait1     BTFSS     PortA,4
          GOTO      Wait1
          DECFSZ    String
          RETURN
          MOVLW     06
          MOVWF     String
          RETURN

;Table of tone values..........

Table     ADDWF     02
          NOP
          RETLW     D'122'
          RETLW     D'91'
          RETLW     D'68'
          RETLW     D'51'
          RETLW     D'41'
          RETLW     D'30'

; Subroutine to generate Tone..

Tone       MOVWF     Count1
Loop1     CALL      Fifty
          DECFSZ    Count1
          GOTO      Loop1
          RETURN
```

```
; Subroutine 50us   delay .....

Fifty     NOP
          NOP
          MOVLW     08
          MOVWF     Count2
Loop2     NOP
          NOP
          DECFSZ    Count2
          GOTO      Loop2
          RETURN

          END
```

```
;***********************************
;       HEX1.ASM  MPB  2-12-10
;       Program to convert binary
;       input to 7 segment output
; ***********************************

        PROCESSOR 16F84A

PortA   EQU     05
PortB   EQU     06
PCL     EQU     02

        MOVLW   B'0000000'
        TRIS    PortB

Start   MOVF    PortA,W
        ANDLW   B'00001111'
        CALL    Table
        MOVWF   PortB
        GOTO    Start

Table   ADDWF   PCL
        RETLW   07E
        RETLW   00C
        RETLW   0B6
        RETLW   09E
        RETLW   0CC
        RETLW   0DA
        RETLW   0FA
        RETLW   00E
        RETLW   0FE
        RETLW   0CE
        RETLW   0EE
        RETLW   0F8
        RETLW   072
        RETLW   0BC
        RETLW   0F2
        RETLW   0E2

        END
```

Programs 10.4: (*continued*)

```
;*************************
;       MESS1.ASM
;       MPB 2-12-10
;       Message display
; *************************

        PROCESSOR 16F84A

PCL     EQU     02
PortA   EQU     05
PortB   EQU     06
Timer1  EQU     0C
Timer2  EQU     0D
Timer3  EQU     0E
count   EQU     0F

; Initialise..............

        CLRW
        TRIS    PortB

; Output loop.............

repeat  MOVLW   D'12'
        MOVWF   count

next    MOVF    count,w
        CALL    table
        MOVWF   PortB
        CALL    delay
        DECFSZ  count
        GOTO    next
        GOTO    repeat

; Meassage delays.........

delay   MOVLW   05
        MOVWF   Timer3

loop3   MOVLW   0FF
        MOVWF   Timer2
loop2   MOVLW   0FF
        MOVWF   Timer1
loop1   DECFSZ  Timer1
        GOTO    loop1

        DECFSZ  Timer2
        GOTO    loop2
        DECFSZ  Timer3
        GOTO    loop3
        RETURN

; Message characters.....

table   ADDWF   PCL
        NOP
        RETLW   B'00000000'
        RETLW   B'00000000'
        RETLW   B'01111110'
        RETLW   B'00000000'
        RETLW   B'01110000'
        RETLW   B'00000000'
        RETLW   B'01110000'
        RETLW   B'00000000'
        RETLW   B'11110010'
        RETLW   B'00000000'
        RETLW   B'11101100'
        RETLW   B'00000000'

        END
```

```
;******************************
;       MET1.ASM  MPB  2-12-10
;       Program to output beeps
;       between 0.1-10Hz
;       RB0 = Output Buzzer
;       RA0 = Input Button Up
;       RA1 = Input Button Down
; ...............................

        PROCESSOR 16F84A

PortB   EQU     06
PortA   EQU     05
Count1  EQU     0C
Count2  EQU     0D
Count3  EQU     0E
Wait1   EQU     0F
Count0  EQU     10

; Initialise...................

        MOVLW   B'11111110'
        TRIS    PortB
        MOVLW   D'10'
        MOVWF   Wait1

; Main loop...................

start   MOVLW   020
        MOVWF   Count0
beep    BSF     PortB,0
        CALL    delay1
        BCF     PortB,0
        CALL    delay1
        DECFSZ  Count0
        GOTO    beep

; Read buttons................

fup     BTFSS   PortA,0
        DECFSZ  Wait1
        GOTO    fdown
        INCF    Wait1
fdown   BTFSS   PortA,1
        INCFSZ  Wait1
        GOTO    Wait
        DECF    Wait1

; Wait 0.1 - 2.5s.........

Wait    MOVF    Wait1,w
        MOVWF   Count3
loop3   CALL    del100
        DECFSZ  Count3
        GOTO    loop3
        GOTO    start

; Wait 100ms..............

del100  MOVLW   D'100'
        MOVWF   Count2
loop2   CALL    delay1
        DECFSZ  Count2
        GOTO    loop2
        RETURN

; 1ms Delay..............

delay1  MOVLW   D'250'
        MOVWF   Count1
loop1   NOP
        DECFSZ  Count1
        GOTO    loop1
        RETURN
        END
```

Programs 10.4: (*continued*)

```
;*****************************
;  REACT1.ASM  MPB 30-11-10
;  Reaction time program
;  RB0 = Buzzer
;  RA4 = Test Input
;  RB1-RR7 = Display
;*****************************

        PROCESSOR 16F84A

PortA   EQU     05
PortB   EQU     06
Random  EQU     0C
Rtime   EQU     0D
Count3  EQU     0E
Count2  EQU     0F
Count1  EQU     10

; Initialise...................

        MOVLW   B'00000000'
        TRIS    PortB
        MOVLW   0FF
        MOVWF   PortB

; Generate random count 0-100..

wait    BTFSC   PortA,4
        GOTO    wait
        CALL    onehun
        CLRW
        MOVWF   PortB
reload  MOVLW   D'100'
        MOVWF   Random

down    BTFSC   PortA,4
        GOTO    randel
        DECFSZ  Random
        GOTO    down
        GOTO    reload

; Delay for random time(0-10s)..

randel  CALL    onehun
        DECFSZ  Random
        GOTO    randel

; Beep and start timer(512ms)..

        CLRF    Rtime
beep    BSF     PortB,0
        CALL    onems
        BCF     PortB,0
        CALL    onems
        BTFSS   PortA,4
        GOTO    stop
        INCFSZ  Rtime
        GOTO    beep

; Divide Reaction time by 32..

stop    MOVLW   4
        MOVWF   Count3
loop3   BCF     3,0
        RRF     Rtime
        DECFSZ  Count3
        GOTO    loop3
```

```
; Display reaction time..

        MOVF    Rtime,W
        CALL    table
        MOVWF   PortB
done    CALL    onehun
        BTFSS   PortA,4
        GOTO    done
        GOTO    wait

;100ms delay.............

onehun  MOVLW   D'100'
        MOVWF   Count2
loop2   CALL    onems
        DECFSZ  Count2
        GOTO    loop2
        RETURN

; 1ms delay..................

onems   MOVLW   D'249'
        MOVWF   Count1
loop1   NOP
        DECFSZ  Count1
        GOTO    loop1
        RETURN

; Display codes 0-9...........

table   ADDWF   002
        RETLW   0EC     ; H
        RETLW   00C     ; 1
        RETLW   0B7     ; 2
        RETLW   09E     ; 3
        RETLW   0CC     ; 4
        RETLW   0DA     ; 5
        RETLW   0EA     ; 6
        RETLW   00E     ; 7
        RETLW   0FE     ; 8
        RETLW   0CE     ; 9
        RETLW   0EC     ; H
        RETLW   0EC     ; H
        RETLW   0EC     ; H
        RETLW   0EC     ; H
        RETLW   0EC     ; H

        END
```

Programs 10.4: (*continued*)

```
;*****************************
;       SEC1.ASM
;       MPB 30-11-10
;       One second counter
;*****************************

        PROCESSOR 16F84A

PCL     EQU     02
PortA   EQU     05
PortB   EQU     06
count   EQU     0C
Timer0  EQU     0D
Timer1  EQU     0E
Timer2  EQU     0F

        CLRW
        TRIS    PortB

repeat  MOVLW   D'10'
        MOVWF   count

next    MOVF    count,w
        CALL    table
        MOVWF   PortB
        CALL    delay
        DECFSZ  count
        GOTO    next
        GOTO    repeat

delay   MOVLW   D'25'
        MOVWF   Timer0
loop0   MOVLW   D'100'
        MOVWF   Timer1
loop1   MOVLW   D'99'
        MOVWF    Timer2
loop2   NOP
        DECFSZ   Timer2
        GOTO     loop2
        DECFSZ  Timer1
        GOTO    loop1
        DECFSZ  Timer0
        GOTO    loop0
        RETURN

Table   ADDWF   PCL
        NOP
        RETLW   07E
        RETLW   00C
        RETLW   0B6
        RETLW   09E
        RETLW   0CC
        RETLW   0DA
        RETLW   0FA
        RETLW   00E
        RETLW   0FE
        RETLW   0CE

        END
```

Programs 10.4: (*continued*)

Questions 10

1. State one advantage and one disadvantage of: (a) breadboard; (b) stripboard; (c) simulation for testing prototype designs. (6)
2. State an output binary code for: (a) all segments off and (b) displaying a '2' in a common cathode seven-segment LED display, assuming the connections shown in Figure 10.15. (4)
3. Outline an algorithm for generating a fixed frequency output of approximately 1 kHz from the DIZI board using the hardware timer. (5)
4. Draw a flowchart representing the process of generating a 'random' delay between a button being pressed and an output LED being switched on. (5)

Answers on page 423. (Total 20 marks)

Activities 10

1. Build the DIZI circuit on breadboard, stripboard or PCB and test the programs BUZZ1, DICE1 and SCALE1.
2. Confirm by calculation or simulation that the values used in the program data table in SCALE1.ASM will give the required delays.
3. Devise a breadboard layout for the BIN circuit in Figure 3.3. Build the circuit and test the BINx programs.
4. Design and implement one of the programs outlined for the DIZI hardware, and compare your solution with the model programs provided for HEX1, MESS1, SEC1, REACT1, GEN1, MET1, BELL1 or GIT1.
5. (a) Investigate how input from a numeric keypad can be detected. Refer to Chapter 1, Section 1.4.1. The typical keypad, shown in Figure 10.17, has 12 keys in four rows of three: 1, 2, 3; 4, 5, 6; 7, 8, 9; *, 0, #. These are connected to seven terminals, and can

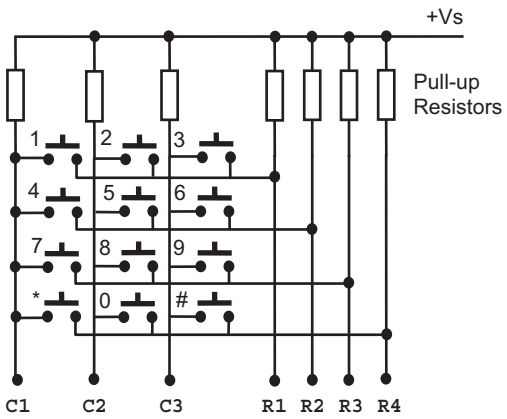


Figure 10.17
Keypad connections

be scanned in rows and columns. A key press is detected as a connection between a row and column. The pull-up resistors ensure that all lines default to logic '1'. If a '0' is applied to one of the column terminals (C1, C2, C3), and a key is pressed, this '0' can be detected at the row terminal (R1, R2, R3, R4). If the keypad terminals are connected to a PIC port, and a '0' output in rotation to the three columns, a key can be detected as a combination of the column selected and the row detected. Column terminals can be set as outputs, and rows as inputs. Draw a flowchart to represent the process for converting each decimal key into the corresponding BCD number.

(b) A lock function may be implemented by matching an input sequence with a stored sequence of, say, four digits, and switching on an output to a door solenoid if a match is detected. Specify the hardware and outline the program for the lock application.

(c) Design, build and test an electronic lock system using the keypad shown, a suitable PIC and an LED to indicate the state of the lock (ON = unlocked). Research the design for the interface to a solenoid operated door lock.

Note: Keypad scanning is used in Program 13.1, and a lock application outlined in Appendix D.

CHAPTER 11

PIC Motor Applications

Chapter Outline

Chapter Points

- Demonstration hardware MOT2 is based on the PIC16F690 (hardware and simulation).
- MOT2 board has a dc motor with index pulse sensor, full bridge driver, analogue and digital inputs.
- Bidirectional drive and position test programs are provided.
- Speed control uses pulse feedback to modify the PWM drive output.
- Position control module, mechatronics board, hobby servo and brushless motors are described.

PIC Microcontrollers. DOI: 10.1016/B978-0-08-096911-4.10011-4

Following on from Chapter 8, we will develop further the topic of driving small motors, since this is a significant application area and illustrates some important real-time control principles. Printers, DVD players, computer hard drives, robots, motor vehicles and many other consumer and industrial products contain microprocessor-controlled motors. The various types of motors have their own drive requirements and dynamic characteristics that must be taken into account in the program design, and can cause complications in practice. The simple permanent magnet brushed direct current (dc) motor is taken as a starting point.

11.1. Motor Control

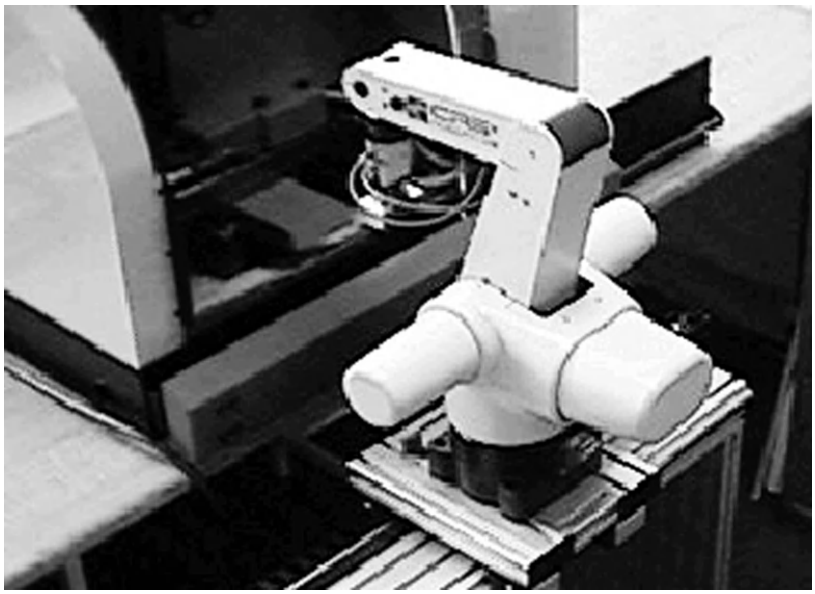
There are two main types of control system, open loop and closed loop. An open loop system is essentially manually controlled or involves operating a load under fixed conditions. For example, a cooling fan will usually not need precise speed control, and might simply be switched on and off from a fixed voltage supply. A closed loop system uses sensors to monitor the system outputs and control them automatically, so, in a motor, the output speed or position is more precisely controlled. The dynamic response (i.e. when there is a change in speed or position) should then be more predictable, particularly when starting or stopping. Position control in a robot arm (Figure 11.1) is a good example of a motor application using digital feedback in a closed loop system.

The block diagram in Figure 11.1(b) shows the operation of one axis. The motor is controlled via a PWM drive (see Chapter 8), and its position and speed are monitored via an incremental encoder, which produces a pulse train as the motor rotates. A sequence of positions is specified in the robot program, and the main controller sends the next position required to the axis controller as a certain number of steps from the current position. The axis is moved accordingly, with the axis controller accelerating and decelerating the motor to provide a smooth motion and accurate end position, using the feedback provided by the encoder.

A small, inexpensive, brushed (having a conventional commutator) dc motor will be used to demonstrate the use of the PIC® microcontroller (MCU) in a such a control application, allowing simple open and closed loop operation to be investigated. More sophisticated systems these days tend to use brushless motors, as they are more efficient and reliable, but are more complicated to drive, as they need electronic commutation. This entails the microcontroller switching the windings on and off in sequence, and monitoring the current to provide precise control.

Motor output is measured as the shaft speed or position. Open loop control of a motor would consist of simply switching it on and off for a fixed period to position it, or varying the speed, under manual control. There are obvious limitations to open loop control. A dc motor will not start from stationary until there is a significant current, owing to inertia, stiction and its electromagnetic characteristics. This makes its response non-linear, at least at low speeds,

(a)



(b)

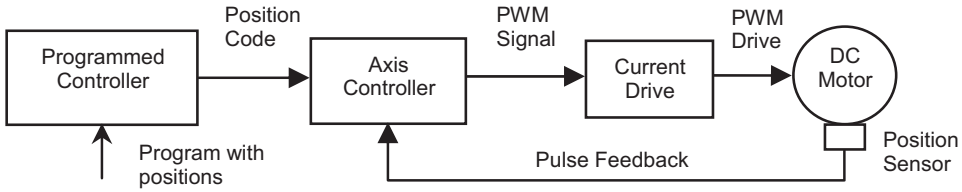


Figure 11.1
Robot arm and axis control: (a) robot loading a milling machine; (b) robot axis control block diagram

which means that the speed is not directly proportional to the current or voltage applied. In addition, the speed cannot be accurately predicted for any given current, because it will vary with the load on the shaft. The final position of the shaft when the motor stops cannot be precisely controlled either. Therefore, if the speed or position of a dc motor is to be controlled accurately, we need sensors to measure the output, and a control system for the motor drive.

A simple analogue potentiometer can measure position, by converting it to a voltage, or speed can be measured using a tachometer (essentially a small dc generator), which produces a voltage that is proportional to the motor speed. These transducers have traditionally been used in analogue motor control systems, where all the signals are continuously variable currents and voltages. With the development of digital control systems, feedback is usually

derived from switching sensors (optical or magnetic) and the microcontroller provides a programmable device in which the program can be designed to handle the motor characteristics and load requirements, and the dynamic response can be adjusted in software.

The speed of a dc motor is controlled by the current in the armature, which interacts with the magnetic field produced by the field windings (or permanent magnets in small motors) to produce torque. An analogue control system gives continuous control over the motor current, and a digital to analogue drive converter can be used at the output if the feedback and control are digital. However, the control interface can be simplified if pulse width modulation (PWM) is used. PWM is a simple and efficient method of converting a digital signal to a proportional drive current. Many microcontrollers now provide dedicated PWM outputs, but we are going to generate the control signal in software here for simplicity.

Digital feedback can be obtained from a sensor, which detects the shaft rotation, as in the robot axis above. One way of doing this is to use a perforated or sectored disk attached to the shaft and an optical sensor to detect the slots or holes in the disk. The shaft position can be detected by counting pulses, and the speed by measuring their frequency. This signal can be fed directly to a microcontroller, which monitors the pulse input, and varies the output to control the speed and/or position of the motor.

11.2. Motor Application Board MOT2

We will investigate these ideas via a general purpose motor test board design, MOT2, which can control a dc motor requiring up to 30 A drive current. This is provided by a full bridge driver that allows bidirectional speed and position control with pulse feedback (servo motor). MOT2 is based on the PIC 16F690 (as used in the LPC demo board). A block diagram is shown in Figure 11.2 and a circuit schematic in Figure 11.3.

A variety of motor control operations can be demonstrated using this system:

- Motor On/Off
- Motor Forward/Reverse
- Open/Closed Loop Position Control
- Open/Closed Loop Speed Control.

Command inputs can be received from an 8-bit switch bank, a remote 8-bit master controller, two push buttons, or from the analogue inputs or serial ports. The motor can be turned in either direction via a full bridge driver, providing position and speed control. Pulse width modulation (see Chapter 8) will be used to control the speed. The shaft speed and position are monitored by a shaft encoder, which has three outputs, but only the index output (one pulse per revolution) is connected in the initial design. In the simulation schematic, the motor and encoder are integrated into a single servo motor model, DCM.

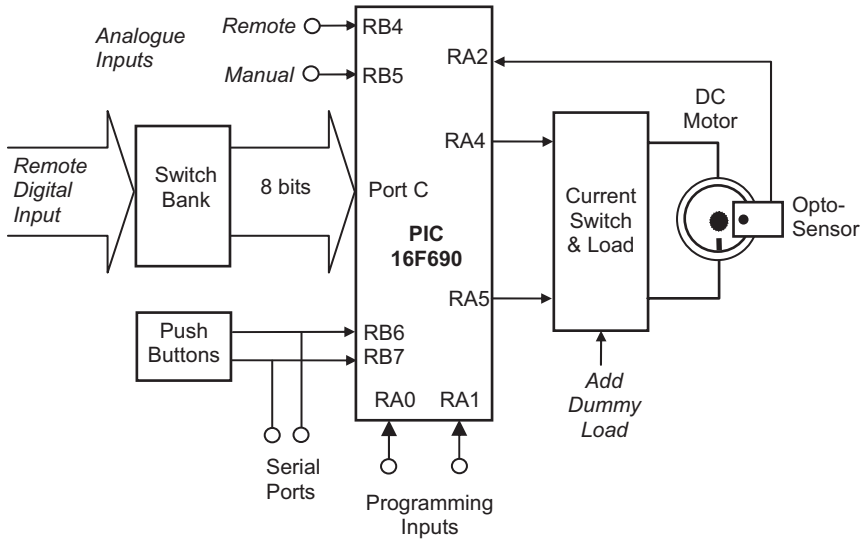


Figure 11.2
Block diagram of MOT2 board

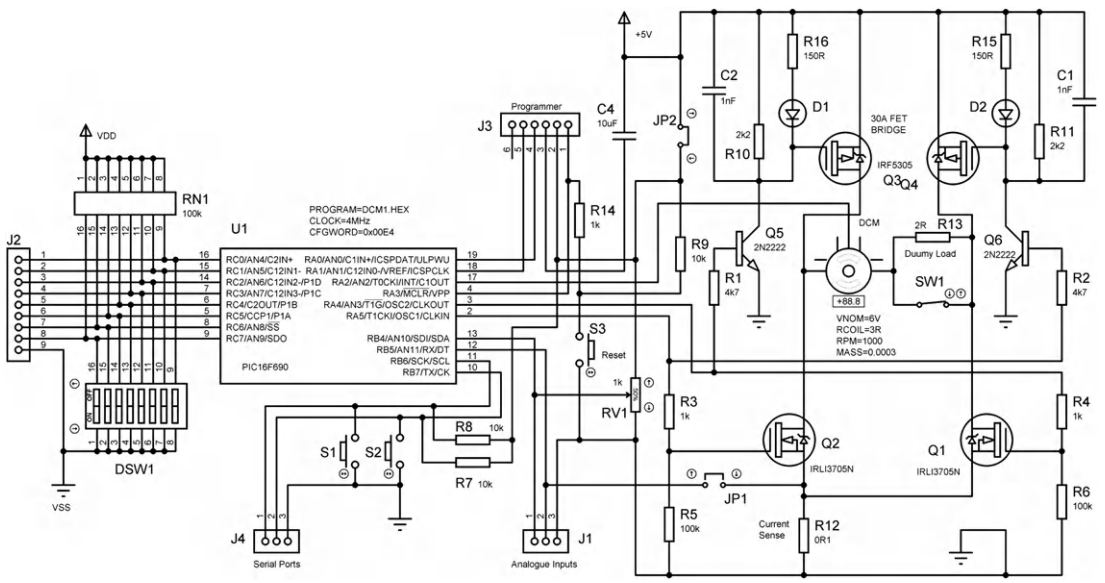


Figure 11.3
MOT2 dc motor board schematic

Motor Drive

A 6 V permanent magnet motor is used since it will run from the same 5 V supply as the PIC. However, a large decoupling capacitor (C4) should be fitted because the motor will generate a lot of noise on the supply, particularly when it switches off. The motor current direction, forward or reverse, is controlled by switching on two of the four MOSFETs from RA4 or RA5. Q1 and Q3 are switched on if RA4 output goes high (motor forward), and Q2 and Q4 if RA5 goes high (motor reverse). Q1 and Q2 switch on when the gate is high (N-FET), but Q3 and Q4 switch on when the gate is low (P-FET), so an inverting bipolar stage is needed on each gate. The current flows diagonally through the bridge and motor to drive it in either direction. The bridge is rated at 30 A, so a range of small to medium dc motors could be driven successfully. In the simulation, the motor characteristics can be adjusted to represent different motors: nominal voltage, coil resistance, coil inductance, zero load rpm, effective mass of the motor and load, and the number of encoder pulses per revolution.

Output Sensor

The rotary encoder represented in the simulation circuit schematic has three outputs. Two have the same number of slots per revolution (adjustable in simulation to represent a range of encoders, default 24), but they are offset by half a slot so that the direction of rotation can be detected from the phase difference in the output signals. The third output generates an index signal once per revolution, which can be used to generate an absolute initial position, or to measure the speed as time per revolution. In hardware, an opto-sensor containing a light-emitting diode (LED) and photodetector can be mounted either side of a perforated wheel attached to the motor shaft. This allows the light to pass through holes or slots causing digital pulses to be output from the sensor via a built-in amplifier, allowing the motor speed or position to be monitored by the controller. The simplest type can have a single slot producing one pulse per revolution. Alternatively, the sensor can work by reflection from a shaft surface, or magnetically. The index output on the simulated servo in Figure 11.3 is connected to the Timer0 (T0CKI) input of the PIC, so that the shaft revolutions can be counted. Alternatively, the pulse interval can be measured using timer mode, if that will produce a more accurate measurement. The pulse may also be used to trigger a Timer0 interrupt.

Switched Inputs

The control program can use the push buttons (S1, S2) connected to RB6 and RB7 to stop, start, or change speed or direction. The binary input switches could be used to select the speed or position. Alternatively, a remotely generated digital control code can be applied to the digital input connector pins (J2) from a master controller, which could be operating a number of motors in a robot system or machine tool. In this case, part of the digital input would be a motor select code, and part would be a position or speed command. Serial commands could also be used, with the port B pins reassigned for this purpose (see Section 12.4). If the parallel input is removed from the circuit, a smaller, cheaper PIC 12FXXX series device could be used instead.

These have six input/output (I/O) pins, so there would be three inputs available with which to control the motor speed, position and/or direction. Analogue inputs are also available, if the motor needs to be voltage controlled.

Analogue Input

The analogue input could be used to receive a voltage that sets the speed or position of the motor. For example, a position servo may use a potentiometer to provide position feedback to the controller, or a temperature sensor might control the speed of a fan. The analogue-to-digital converter (ADC) must be initialized to suit, in this case using the internal supply reference to set the range. For test purposes, a pot is connected to AN10, to provide a dummy analogue input. If the jumper is closed, AN11 can be used to monitor the motor drive current as a voltage across a 0R1 current sensing resistor connected in the common arm of the bridge (100 mV/A). This can be used as a feedback signal or to shut down the output if the motor current is too high. External analogue inputs can be connected at J1.

The default internal clock frequency of the PIC 16F690 is 4 MHz to give an instruction cycle time of 1 μs. The internal clock mode needs to be selected in the configuration word at the top of the program, along with the power-up timer and MCLR enable (00E4h). MCLR is controlled from the programmer during testing, but a reset button is also provided on board. The power-on timer should be enabled during programming to ensure a reliable start. The motor drive will need an external power supply providing sufficient current for the motor connected, so the supply from the programming connector may need to be disconnected at JP2 while programming.

11.3. Motor Control Methods

The programs described below have been tested on a previous version of the hardware, and in simulation mode with the current design.

11.3.1. Open Loop Control

Open loop control of a dc motor (MOT1) has been described in Chapter 8 and a program developed which allows the speed to be controlled manually. In the MOT2 circuit (Figure 11.3), the motor can be driven in either direction by setting RA4 or RA5 high, with both set low to turn the motor off. They must not be high together; this would switch on both transistors, resulting in no current through the motor, and possible damage to the power transistors. Open loop speed control can therefore be implemented by outputting a PWM signal at either RA4 or RA5.

The push-button inputs could be programmed to run the motor in either direction, or to increment and decrement the speed in one direction by modifying the delay in a PWM

program. Alternatively, the speed could be set at the binary inputs to port B, either manually at the DIP switches, or with an 8-bit digital input code supplied from a master controller. Analogue control is possible from a manual input (RV1) or from a remote voltage source. Any of these inputs could be used to set the duty cycle of the drive waveform. However, neither the speed nor position can be controlled accurately without feedback.

11.3.2. Closed Loop Control

The simulation design uses the index output of the servo module for feedback, because this is easier to implement in a prototype. The signal can be generated by making one opto-sensor mark on the shaft (or magnetic equivalent) or one slot or hole in a disk attached to the shaft. One pulse per revolution will also provide more time between pulses for the control program to complete its processing tasks. The sensor is connected to the Timer0 input, an 8-bit counter/timer register that can be clocked externally or from the system clock, and which we can use to measure the pulse total count, frequency or period, depending on the application.

Closed loop position control involves counting the revolutions as the shaft turns. This sounds straightforward, but the dynamic characteristics of the motor have to be taken into account. For example, the motor can be switched on from the controller, the pulses counted, and the motor turned off when a set number of pulses has occurred. However, the motor will probably overshoot the required position owing to inertia of the rotor and load. A simple solution is to keep counting the slots and turn the motor back by the requisite number of slots. This may have to be repeated several times, causing oscillation. Another improvement is to ramp the speed of the motor up and down at the start and end of the move.

With only one slot, the position can only be determined to the nearest whole revolution. This may be acceptable if a gearbox is fitted (often the case in position controllers), which reduces the angular rotation and speed. For instance, if the gearbox has a reduction ratio of 50:1, the output can be positioned within 1/50 of a revolution. If a shaft encoder is used, a known number of slots per revolution is generated, and a proportionate increase in accuracy obtained. With 100 slots, for example, and the gearbox, the accuracy will be $360/(50 \times 100) = 0.072$ degrees. This result can then be used to estimate the positional resolution of the load attached. For example, if a robot arm of length 300 mm is attached to this drive, the accuracy at the end of the rotating arm will be the length of arc:

$$\text{Circumference of working circle} = 2\pi r = 2 \times 0.3 \times \pi = 1.885\,\text{m}$$

$$\text{Arc of step} = 1.885 \times 0.072/360 = 0.38\,\text{mm} = \text{resolution}$$

Speed control will involve measuring the index pulse interval, and comparing it with a target value. The target value can be input from any of the analogue, digital or serial data sources available in MOT2. The PWM duty cycle is then adjusted continuously towards this target.

Since there is some delay in the response of the motor, owing to mechanical inertia, the speed may oscillate around the target value to some extent. This depends very much on the characteristics of the motor and load, which can be varied in the simulation circuit to investigate their effect.

11.4. Test Programs for MOT2

The following test programs for the MOT2 board will demonstrate aspects of direction, position and speed control.

11.4.1. Direction Test

A simple test program to drive the motor in each direction DCM1 is listed as Program 11.1.

When S1 is held on, the motor runs in the forward direction, and in the reverse direction when S2 is operated. In ISIS simulation mode, the motor properties should be set to supply = 6 V, armature resistance = 3 Ω and load mass = 0.0001, so that the motor responds quickly. A voltage probe can be attached to the bridge common node to measure the current in the sensing resistor (60 mA). The nominal speed should be set to 1000 because the maximum displayed rpm is 999. The simulated servo parameters can be adjusted to check the effect.

11.4.2. Position Control

A program POS2 that moves the motor to a position set on the pot is represented in the flowchart in Figure 11.4, and the source code listed as Program 11.2.

The principle of the program is to read a position from the pot in the range 0−255, as an 8-bit result from the ADC, and move the motor to a corresponding position. To allow the pot to be adjusted before the motor responds, push-button S1 is used to trigger the move. Initially, the motor position is set midway at 127. If the pot is moved to a forward or reverse position, the motor moves the same number of revolutions, i.e. ± 127. The index output of the servomotor is fed back to Timer0 (8-bit counter) in the MCU, which counts the number of revolutions. This is compared with the target value in the monitoring phase, and the motor stopped when the correct number of pulses has been received.

The main problem with motor control, which is illustrated in this example, is that the motor tends to overshoot the target position because of mechanical inertia. An attempt to correct this is incorporated in the program, where the overrun is counted and the motor moved back if necessary. This is achieved by waiting an arbitrary time after the motor has been switched off and checking the count again. However, if the program timing and motor characteristics are not closely matched, the motor may either oscillate about the target position (hunting) or not

```
;*************************************************************
;       DCM1.ASM       MPB  Ver 1.0
;       Test program for DC motor demo board MOT2
;       S1 = Forward, S2 = Reverse
;
;*************************************************************

        PROCESSOR 16F690       ; Specify MCU for assembler
                               ; MCU configuration bits
        __CONFIG 00E4          ; PWRT on, MCLR enabled
                               ; Internal Clock (4MHz)
        INCLUDE "P16F690.INC"  ; Standard register labels

; Initialize registers.....................................

        BANKSEL ANSEL          ; Select Bank 2
        CLRF    ANSEL          ; Port A digital I/O
        MOVLW   B'00001100'    ; Input setup code
        MOVWF   ANSELH         ; RB6, RB7 digital input

        BANKSEL TRISA          ; Select Bank 1
        MOVLW   B'11001111'    ; PortA setup code
        MOVWF   TRISA          ; RA4, RA5 output
        BANKSEL PORTA          ; Reselect Bank 0

; Start main loop.........................................

        CLRF    PORTA          ; Both FETs off
S1      BTFSS   PORTB,6        ; Test S1
        GOTO    For
        BCF     PORTA,4        ; If not,off
        GOTO    S2
For     BSF     PORTA,4        ; If on, forward

S2      BTFSS   PORTB,7        ; Test S2
        GOTO    Rev
        BCF     PORTA,5        ; If not,off
        GOTO    S1
Rev     BSF     PORTA,5        ; If on, reverse
        GOTO    S1             ; repeat always

        END     ; Terminate assembler.....................
```

Program 11.1
DCM1 source code

achieve an accurate position. This effect can be seen if the program is simulated in ISIS. The motor mass needs to be adjusted to about 0.0002 for the simulation to work properly; this can be varied to see the effect on the overshoot.

In this case, the position can only be controlled to a resolution of one revolution. Using the incremental encoder outputs of the servo would allow this performance to be improved, by counting more pulses per revolution. Another way of achieving better performance is for the current position to be continuously compared with the required position, and the motor driven

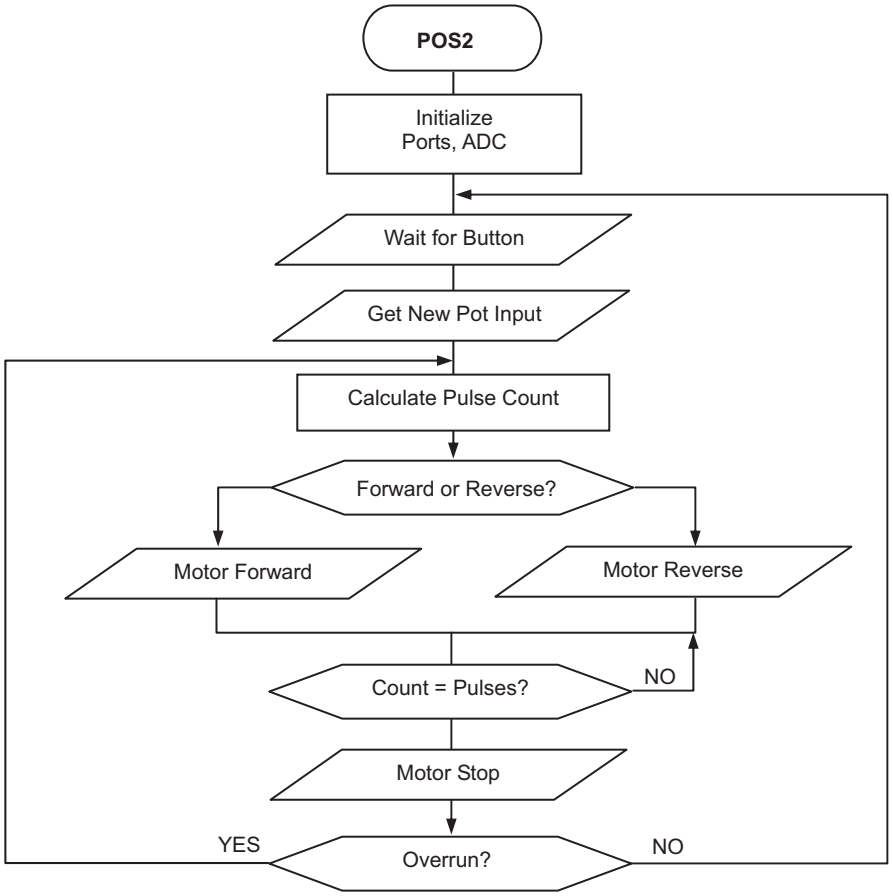


Figure 11.4
Flowchart for motor program POS2

at a speed proportional to the error. The motor will slow down as it approaches the target position. This type of process is referred to as PID (proportional, integral and differential) control, where the response of the system can be tuned to give the best compromise between speed of response, accuracy and overshoot. A simpler process called 'trapezoidal' control can also be used. This involves ramping the motor speed up and down at the ends of the move, with a constant speed period in the middle.

11.5. Closed Loop Speed Control

In this example, the motor board is to operate as a slave speed-controlled unit. A master controller supplies an 8-bit code to set the speed of the motor, with the local controller required to maintain it with a specified degree of precision. The MOT2 board allows for a test input at

```
;*************************************************************
;       POS2.ASM       MPB     Ver1.0
;       Test program for motor demo board MOT2
;       Position control from pot
;       Counts motor pulses in TMR0
;       Complete 15/12/10
;
;*************************************************************

        PROCESSOR 16F690        ; Specify MCU for assembler
                                ; MCU configuration bits
        __CONFIG 00E4           ; PWRT on, MCLR enabled
                                ; Internal Clock (4MHz)
        INCLUDE "P16F690.INC"   ; Standard register labels

PotLas  EQU     020             ; Pot start position
PotNow  EQU     021             ; Pot target position
Count1  EQU     022             ; Overrun counter
Count2  EQU     023             ; Overrun counter
PotDif  EQU     024             ; Pot change or overrun
Timer1  EQU     025             ; Delay timers
Timer2  EQU     026             ;
Timer3  EQU     027             ;
OverCo  EQU     028             ; Overcount holding

; Initialize registers.....................................

        CLRF    PORTA           ; Motor off

        BANKSEL ANSEL           ; Select Bank 2
        CLRF    ANSEL           ; All digital I/O
        CLRF    ANSELH          ; ..initially
        BSF     ANSELH,2        ; Analogue inputs
        BSF     ANSELH,3        ; ..AN10, AN11

        BANKSEL TRISA           ; Select Bank 1
        MOVLW   B'11001111'     ; RA4, RA5 output
        MOVWF   TRISA           ;
        MOVLW   B'01110000'     ; A/D clock setup code
        MOVWF   ADCON1          ; Internal clock

        BANKSEL ADCON0          ; Select Bank 0
        MOVLW   B'00101001'     ; Analogue setup code
        MOVWF   ADCON0          ; Left justify, Vref=5V,
                                ; RA10, done, enable A/D
        MOVLW   D'127'          ; Mid value
        MOVWF   PotLas          ; ..into position regs
        MOVWF   PotNow

        CLRF    PORTA           ; Switch off motor
        CLRF    PotDif          ; Zero pot movement

; Main loop ...............................................
```

Program 11.2
POS2 source code

```
start   BTFSC   PORTB,6        ; Wait for S1
        GOTO    start

getpot  MOVF    PotNow,W       ; Save previous pot input
        MOVWF   PotLas

        BSF     ADCON0,1       ; Start ADC..
finish  BTFSC   ADCON0,1       ; ..and wait for finish
        GOTO    finish
        MOVF    ADRESH,W       ; Store result high byte
        MOVWF   PotNow         ; Current pot value

        BCF     STATUS,Z       ; Clear zero flag
        BSF     STATUS,C       ; Set carry flag
        MOVF    PotLas,W       ;
        SUBWF   PotNow,W       ; W = PotNow - PotLas
        MOVWF   PotDif         ;
        BTFSC   STATUS,Z       ; If PotDif = 0
        GOTO    start

        BTFSC   STATUS,C       ; Pot moved negative?
        GOTO    forwrd
        COMF    PotDif         ; Convert to positive
        GOTO    revers         ; yes - reverse motor

forwrd  BCF     PORTA,5        ; Reverse off
        BSF     PORTA,4        ; Motor forward
        CALL    wait
        MOVF    PotDif,W       ; Motor at target?
        BTFSC   STATUS,Z       ;
        GOTO    start          ; Yes - start again

revers  BCF     PORTA,4        ; Forward off
        BSF     PORTA,5        ; Motor reverse
        CALL    wait
        MOVF    PotDif,W       ; Motor at target?
        BTFSC   STATUS,Z       ;
        GOTO    start          ; Yes - start again

        GOTO    forwrd

; Subroutine to stop motor and to correct overrun ........

wait    CLRF    TMR0           ; Count motor pulses
check   MOVF    TMR0,W
        SUBWF   PotDif,W
        BTFSS   STATUS,Z       ; until target reached
        GOTO    check
        CLRF    PORTA          ; Motor off

        CLRF    TMR0           ; Reset pulse count
stop    MOVF    TMR0,W
        MOVWF   Count1         ; Store pulse count
        CALL    long           ; Wait a while
```

Program: 11.2: (*continued*)

```
        MOVF    TMR0,W          ; Store count again
        MOVWF   Count2
        MOVWF   OverCo
        SUBWF   Count1          ; Check if changed
        BTFSS   STATUS,Z
        GOTO    stop            ; ..until unchanged
        CALL    long            ; Wait a while

        MOVF    OverCo,W        ; store overcount
        MOVWF   PotDif
        RETURN

; Long delay.....................................

long    MOVLW   D'10'
        MOVWF   Timer1          ; 1s
loop1   MOVLW   D'100'
        MOVWF   Timer2          ; 100ms
loop2   MOVLW   D'249'
        MOVWF   Timer3          ; 1ms
loop3   NOP
        DECFSZ  Timer3
        GOTO    loop3
        DECFSZ  Timer2
        GOTO    loop2
        DECFSZ  Timer1
        GOTO    loop1
        RETURN

        END     ; Terminate assembler..................
```

Program: 11.2: (*continued*)

the switch bank to simulate this external demand. Alternatively, the required speed could be input as a data byte at a serial port.

Suppose that the motor is to be controlled to a speed of exactly 600 rpm. This will produce 10 pulses per second (pps) with a single slot in the wheel. This relatively low speed is used for simulation in ISIS, because the DCM rev counter only reads up to 999 rpm. Real hardware needs to be controlled at speeds up to at least 3000 rpm, using a higher MCU clock speed. The speed can be measured in one of two main ways: by counting sensor pulses over a measured time period, or by measuring the period between sensor pulses.

11.5.1. Counting Pulses

The accuracy of the speed measurement using this method will depend on the number of slots counted, because the error is always ± 1 slot. If the rev count were made over a period of 1 s at 10 pps, the precision would be 10% and the speed could only be corrected once per second. This response time is too slow for most practical purposes, so this option will be rejected. It would be viable if the encoder had more slots per revolution or the motor was running at high speed.

11.5.2. Measuring Pulse Period

At 10 pps, the target speed, the pulse period will be 100 ms. This can be measured using a 100 ms timer, which can be set up using the 8-bit TMR0 hardware counter/timer (see Chapter 6). The counter is driven by the instruction clock (1/4 of the MCU oscillator frequency). The timer prescaler allows this to be divided by 2, 4, 8, 16, 32, 64, 128 or 256, by setting a 3-bit code in the option register. If the MCU clock is set to 1 MHz, the timer clock rate is 250 kHz, with period 4 μs, with the maximum prescaler setting of 256, the longest period measurable will be $256 \times 256 \times 4 = 26\,2144\ \mu s = 262$ ms. The count required, as a proportion of this maximum value, will be $100/262 \times 256 = 98$ (to the nearest whole number). Recall that the timer counts up to FF then 00, when the overflow flag is set, so the timer must be preloaded with the complement of this value, $256 - 98 = 158$.

11.5.3. PWM Motor Control

The speed of the motor is controlled using PWM, which switches the motor current on and off over a fixed period cycle. The ratio of the on/off periods controls the average current, and hence the speed. A software delay loop will be used to generate a PWM drive signal to the motor, while running a hardware timer to generate a timing reference to compare with the sensor feedback each time round the motor delay loop. The mark/space ratio (MSR) is adjusted to control the speed. The target speed is set using the switch inputs to generate the timer preload value.

The process is illustrated in the timing diagram (Figure 11.5). The timing cycle starts at the rising edge of the sensor pulse when the timer is started. The program waits for the falling edge of the sensor pulse, then starts checking if the next pulse has arrived, or if the timer has timed out, once per motor cycle. If the speed is too low, the timer times out first, before the pulse arrives, so the speed must be increased for the next timing cycle. If the slot arrives before the timer has timed out, it means that the motor is running too fast, so the speed must be decremented for the next cycle. User flags have been defined, one to record the fact that the falling edge has been detected and acted upon ('slot' flag) and another to record the fact that the timer has been restarted ('done' flag), to make the program wait for the next slot to restart the timer. When the speed is correct, the speed correction will alternate between incrementing and decrementing.

Figure 11.6 shows the top-level flow chart for the program. This initializes the program and switches the motor on and off. 'Speed' is a user register that holds the value for the PWM 'on' time. The 'off' time is derived by complementing this value. The total count for each motor drive cycle is then 256, which means the frequency will remain constant. At the start of the main loop, on the rising edge of the sensor pulse, Timer0 is loaded with the 8-bit value read

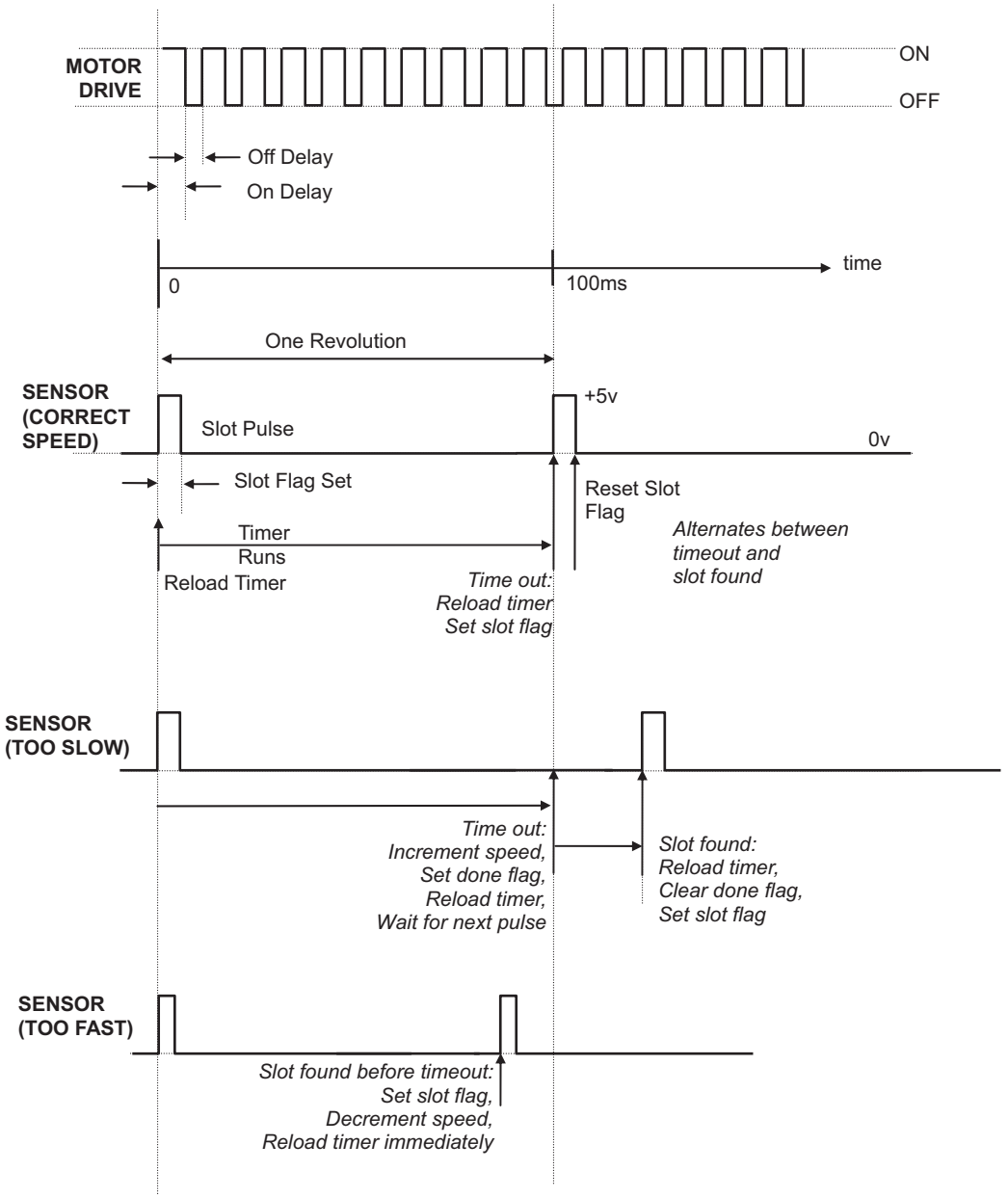


Figure 11.5
Timing diagram for PWM motor speed control (CLS2.ASM)

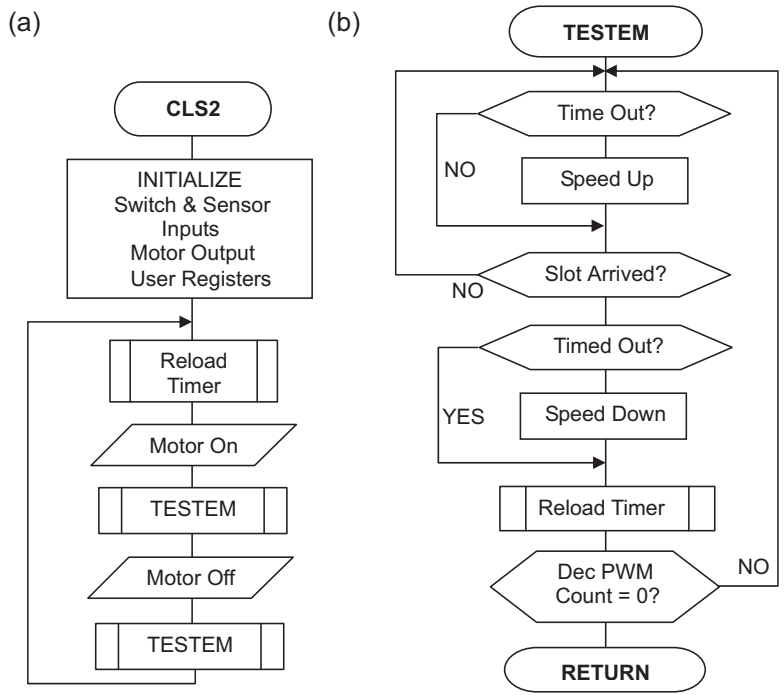


Figure 11.6
Flowcharts for closed loop motor speed control: (a) main loop; (b) main subroutine

from the switch bank. A low value will give a longer remaining count in Timer0, and a longer target cycle time, corresponding to a low speed. A high value will give a high target speed.

The main subroutine checks the input and timeout flag in a polling loop. If the timer times out before the next slot arrives, as is the case when starting up, the motor is going too slowly, so the PWM 'on' time is increased. When the motor is eventually going too quickly, the slot arrives before the timer has finished, so the 'on' time is reduced. The speed must be stopped from rolling over from FF (maximum) to 00 (minimum), so the speed value is tested for zero after incrementing or decrementing and set back to FF or 1, respectively.

When the motor is running at the correct speed, the sensor period should match the timer period. In practice, the motor will hunt around the target value owing to limited resolution in the measurements, program delays, motor imperfections and mechanical inertia. The source code is listed as CLS2.ASM in Program 11.3.

11.5.4. Program Simulation

The speed control application was tested in simulation mode (Figure 11.7). The forward drive, index sensor feedback and current monitoring signals can be seen on the

```
; *******************************************************
;       CLS2.ASM       MPB     Ver1.0
;       Status: Working OK in simulation 17-12-10
; *******************************************************
;       Closed Loop DC Motor Speed Control using Pulse
;       Width Modulation (software loop) to control speed
;       and hardware timer to set reference time interval
;
;       Hardware:      MOT2 Proteus VSM simulation
;       MCU:           16F690, 1MHz internal clock
;       Inputs:        RC0-RC7 DIP switches
;                      RA2 index sensor (high pulse)
;       Outputs:       RA4,5 Motor forward, reverse
;
; Set Processor Options.....................................

        PROCESSOR 16F690       ; Specify MCU for assembler
                               ; MCU configuration bits
        __CONFIG 00E4          ; PWRT on, MCLR enabled
                               ; Internal Clock (1MHz)
        INCLUDE "P16F690.INC"  ; Standard register labels

; Register Label Equates...................................

Speed   EQU     020            ; Counter Pre-load Value
Count   EQU     021            ; Delay Counter
Flags   EQU     022            ; User Flags

; Register Bit Label Equates ..............................

forwd   EQU     4              ; Motor Forward = RA4
revrs   EQU     5              ; Motor Reverse = RA5
sensor  EQU     2              ; Shaft Opto-Sensor = RA2
slot    EQU     0              ; Slot Found Flag
done    EQU     1              ; Time Out Done Flag
timout  EQU     2              ; Time Out Flag = TMR0,2

; Initialize, Port B defaults to input.....................

        BANKSEL ANSEL          ; Select Bank 2
        CLRF    ANSEL          ; All digital I/O
        CLRF    ANSELH         ; ..initially
        BSF     ANSELH,2       ; Analogue inputs
        BSF     ANSELH,3       ; ..AN10, AN11

        BANKSEL TRISA          ; Select Bank 1
        MOVLW   B'11001111'    ; RA4, RA5 output
        MOVWF   TRISA          ; and load dirc. reg.
        MOVLW   B'10000111'    ; Code for TMR0..
        MOVWF   OPTION_REG     ; sets prescale 1:256
        MOVLW   B'01000111'    ; Code to select..
        MOVWF   OSCCON         ; internal clock = 1MHz


        BANKSEL PORTA          ; Select Bank 0
        CLRF    PORTA          ; Motor off
        MOVLW   d'100'         ; Initial value for
        MOVWF   Speed          ; ..speed
```

Program 11.3
CLS2 source code for closed loop speed control

```
        CLRF    Flags           ; and clear user flags
        GOTO    start           ; Jump to main program

; Subroutine reloads Timer0.............................

reltim  MOVF    PORTC,W         ; Get input switches &..
        MOVWF   TMR0            ; Load Timer with input
        BCF     INTCON,timout   ; Reset 'TimeOut' Flag
        RETURN

; Subroutine checks for time out or slot.................

testem  BTFSS   INTCON,timout   ; Time Out?
        GOTO    tessen          ; NO: Skip Speed Increment
        BSF     Flags,done      ; YES: Set Time Out Flag
        INCFSZ  Speed           ; Test for maximum speed
        GOTO    reload          ; NO: jump to timer reload
        DECF    Speed           ; YES: Decrement to 255
        GOTO    reload          ; & jump to timer reload

tessen  BTFSC   PORTA,sensor    ; Slot Present?
        GOTO    teslot          ; YES: jump to test slot
        BCF     Flags,slot      ; NO: Reset 'Slot' Flag
        GOTO    datcon          ; & continue Count loop

teslot  BTFSC   Flags,slot      ; 'Slot' Flag Set?
        GOTO    datcon          ; YES: Skip speed decrement
        BTFSC   Flags,done      ; NO: 'Done' Flag Set?
        GOTO    clrdone         ; YES: Skip speed decrement
        DECFSZ  Speed           ; NO: Test for min. speed
        GOTO    clrdone         ; NO: continue loop
        INCF    Speed           ; YES: increment back to 1

clrdone BCF     Flags,done      ; Clear 'Done' Flag
setslot BSF     Flags,slot      ; Set 'Slot' Flag
reload  CALL    reltim          ; Reload timer
datcon  DECFSZ  Count           ; Decrement & Test Count
        GOTO    testem          ; Counter not zero yet
        RETURN                  ; End motor cycle if zero

; Drive loop outputs one cycle of PWM to motor ..............

start   CALL    reltim          ; reload timer to start

again   BSF     PORTA,forwd     ; Motor ON
        MOVF    Speed,W         ; Put ON delay value
        MOVWF   Count           ; into counter
        CALL    testem          ; Modify speed

        BCF     PORTA,forwd     ; Motor OFF
        MOVF    Speed,W         ; Put ON delay value
        MOVWF   Count           ; into counter
        COMF    Count           ; and convert to OFF value
        BTFSC   STATUS,Z        ; Test for zero count
        INCF    Count           ; ..and avoid
        CALL    testem          ; Modify speed
        GOTO    again           ; next drive cycle

        END     ; Terminate source code  ******************
```

Program: 11.3: (*continued*)

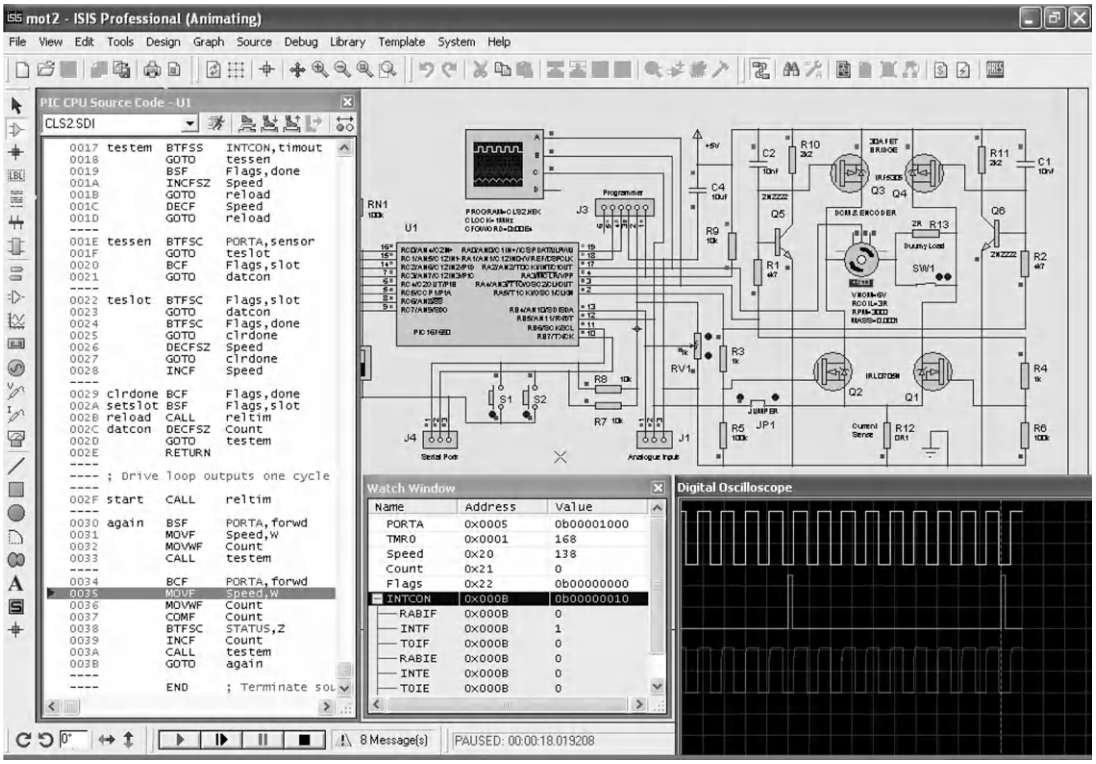


Figure 11.7
CLS2 simulation screenshot

virtual oscilloscope. The drive is operating at about 100 Hz and the feedback has a period of 100 ms, indicating a speed of 600 rpm (switch input = 0xA0). The maximum motor speed (switch input = FF) with this setup is just over 1000 rpm, the maximum displayed on the animated motor. The minimum speed (switch input = 00) is just below 240 rpm.

The closed loop response can be tested at 600 rpm by opening the switch across the dummy load of 2 Ω. This adds extra resistance in series with the motor causing a drop in the 'on' current, hence the speed. The drive program must compensate by increasing the PWM drive to restore the set speed.

The current in the bridge drivers, forward and reverse, can be monitored across the 0.1 Ω sensing resistor. The 'on' current generates a voltage of about 0.4 V, indicating a current of 4 A. This can be connected to the analogue input AN11 by closing the jumper JP1 so it can be measured for control purposes. This could also be used to prevent excessive current in the bridge if, for example, the motor stalls.

Depending on the processing power of your computer, the simulation speed may drop back from real-time operation to a lower speed when testing this circuit. Check the status information below the edit screen: when the processor load reaches 100%, the simulation clock will slow down.

11.5.5. Hardware Testing

A hardware implementation can be tested using a dual-beam scope as indicated above. The correct function of the closed loop control process can be tested by setting the binary input to B'10100000' and checking the actual speed of the motor by measuring the period of the sensor pulse, which should be 100 ms. The binary input can then be varied, and the sensor period should vary in proportion, within limits stated above. The transient and start-up response can be examined by stalling the motor (if not too powerful), and studying the motor response as it locks on to the target speed. When the dummy load is switched in, the drive should compensate and maintain the speed of the motor by increasing the drive PWM MSR.

11.5.6. Evaluation and Improvements

The output was found to be slightly unstable around the target value, owing to the unequal timing of the alternative routes through the program, but this may not be significant in a real motor where the load inertia will tend to maintain a constant speed. Ideally, the PWM speed control should operate at a frequency above about 15 kHz. The demonstration program CLS2 operates at a lower frequency, because of the time required to sample the timer status and sensor input, and to complete the software loop for one drive cycle, which only uses an 8-bit count counter, Timer0. The performance will be improved by operating at the maximum MCU clock speed of 20 MHz, using interrupts and the hardware PWM output associated with 16-bit Timer2. Timer1 is a 16-bit counter, which would provide greater range and/or precision in the speed measurement. The Timer1 interrupt could be incorporated into signal timeout. Alternatively, the sensor pulse could be detected as a change at port A interrupt.

11.6. Motor Control Modules

Some examples of more complete designs for position controllers are described below.

11.6.1. Serial Input Position Controller

A position controller with serial input is described in Microchip® application note AN532 (Figure 11.8a). Although based on a now obsolete MCU, it represents a commercially viable design which could be updated for applications such as printers and X,Y (two-dimensional)

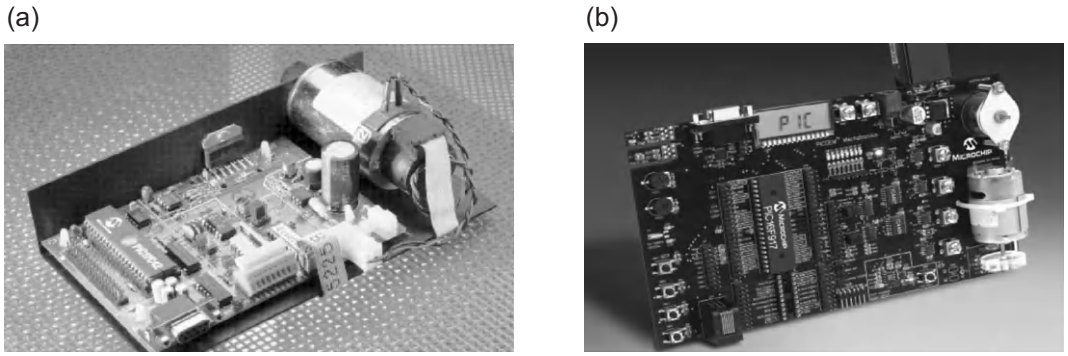


Figure 11.8
Direct current motor control boards: (a) AN532 Servo Module; (b) PIC Mechatronics Development Board

positioning systems. It incorporates a dc motor with quadrature (two-phase) encoder, integrated circuit full bridge driver and separate programmable logic device (PLD) encoder interface. The position demand input is received via an RS232 serial input from a master controller or PC, producing a trapezoidal output speed profile. The application note, which has some useful additional information about servo control design, can be found at www.microchip.com.

11.6.2. Microchip Mechatronics Kit

A very useful motor demo system is provided in the PIC Mechatronics Development Kit (Figure 11.8b). It has a dc brushed motor with a single slot wheel and a stepper motor, a reconfigurable full bridge driver and control logic. It is based on the PIC16F917, which has a dedicated interface to operate a 3.5-digit liquid crystal display (LCD). The board has a six-pin connector for PICkit2/3 programming and in-circuit debugging, and can be used as target hardware to test the programs (suitably adapted) in this chapter. For a fuller description, see the product information sheets or *Programming 8-bit PIC Microcontrollers in C with Interactive Hardware Simulation* by this author (Newnes 2008). A simulation schematic can be downloaded from www.picmicros.org.uk.

Recently, much attention has been directed towards brushless dc motors, which, as the name implies, eliminate the traditional commutator by using a permanent magnet rotor surrounded by a stationary set of windings. A rotating magnetic field drives the rotor, in a similar way to stepper motors. They are more efficient and reliable than brushed dc motors, but are more complex to control, requiring a three-phase full bridge driver with six transistors. Search for BLDC (Brushless DC Motor) among the Microchip Application Notes archive for background theory and demonstration circuits.

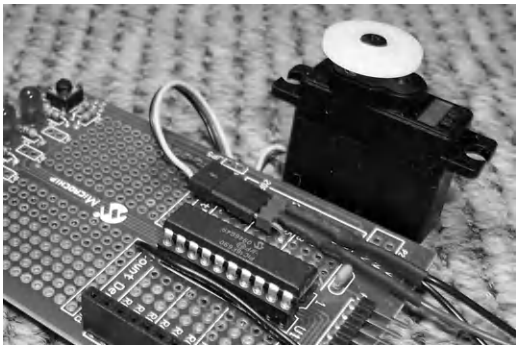


Figure 11.9
Hobby servo connected to LPC test board

11.6.3. Hobby Servo

A popular position controller is the hobby servo, as used in remote-controlled systems such as model boats and planes, mainly for steering control. A compact self-contained unit contains a small dc motor, feedback pot and control chip, which receives a standard PWM signal and moves the output shaft to a position determined by the pulse width. The signal is received via a radio link to a radio-frequency receiver module, into which the control servos are plugged. The transmitter is housed in a control console with two joysticks, which can operate up to four servos, each controlling a left/right and an up/down pot.

The same servo module can be connected directly to a PIC chip to provide a simple position control system. Figure 11.9 shows it connected to the LPC board for testing, as per the schematic in Figure 11.10, using program HOB1 (Program 11.4). The servo PWM input requires a TTL (transistor–transistor logic) positive pulse of between about 0.5 and 2.5 ms

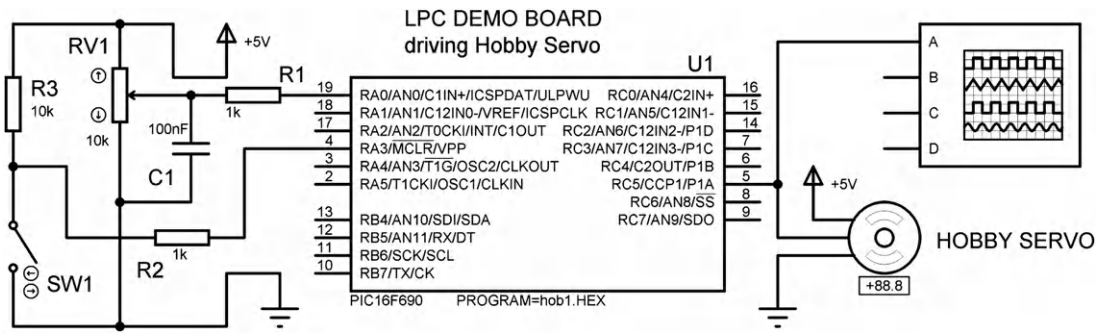


Figure 11.10
LPC driving a hobby servo

```
;*********************************************************
;       HOB1.ASM       MPB  Ver 1.0
;       Test program for LPC demo board
;       Pulse output to servo on RC5
;
;       Internal clock = 4MHz
;       Variable frequency pulse output to servo
;       0.5 - 1.5 - 2.5 ms pulse, overall 18ms period
;
;       Status: working 21-12-10
;
;*********************************************************

        PROCESSOR 16F690      ; Specify MCU for assembler

        INCLUDE "P16F690.INC" ; MCU register lables

        Count   EQU     20      ; Delay count
        Two     EQU     21      ; Delay multiplier
        Ten     EQU     22      ; Delay multiplier
        HiPer   EQU     23      ; High period count
        LoPer   EQU     24      ; Low period count
        ADres   EQU     25      ; AD result store

; Initialize registers.....................................

        BANKSEL ANSEL           ; Select Bank 2
        CLRF    ANSEL           ; Ports digital I/O
        CLRF    ANSELH          ; Ports digital I/O
        BSF     ANSEL,0         ; except AN0 Analogue input

        BANKSEL TRISC           ; Select Bank 1
        CLRF    TRISC           ; Initialise Port C for output
        MOVLW   B'00000000'     ; Analogue input setup code
        MOVWF   ADCON1          ; Left justify result, ref=Vdd

        BANKSEL PORTC           ; Select bank 0
        CLRF    PORTC           ; Clear display outputs
        MOVLW   B'00000001'     ; Analogue input setup code
        MOVWF   ADCON0          ; f/8, RA0, done, enable

; Start main loop.........................................

getpot  BSF     ADCON0,1        ; Start ADC..
wait    BTFSC   ADCON0,1        ; ..and wait for finish
        GOTO    wait
        MOVF    ADRESH,W        ; Test result
        BTFSC   STATUS,Z
        INCF    ADRESH          ; Avoid zero
        MOVF    ADRESH,W        ; store result
        MOVWF   HiPer           ; Store high time
        MOVWF   LoPer
        COMF    LoPer           ; Calculate low time
        BTFSC   STATUS,Z
        INCF    LoPer           ; Avoid zero
```

Program 11.4
Hobby servo test program (HOB1, ASM) for LPC board

```
        BSF     PORTC,5         ; output high pulse
        MOVWF   Count
        CALL    by12            ; and wait for HiPer

        BCF     PORTC,5         ; Output low
        MOVF    LoPer,W
        MOVWF   Count
        CALL    by12            ; and wait for LoPer

        MOVLW   D'10'           ; Extra delay for 15ms
        MOVWF   Ten             ; to make total cycle
by4     MOVLW   D'125'          ; time = 18ms
        MOVWF   Count
        CALL    by12
        DECFSZ  Ten
        GOTO    by4
        GOTO    getpot          ; Repeat main loop

; Wait for Count x 12us ..................................

by12    MOVLW   02              ; Loop delay adjusted
        MOVWF   Two             ; in simulation
by3     NOP
        DECFSZ  Two
        GOTO    by3
        DECFSZ  Count
        GOTO    by12
        RETURN

        END     ; Terminate assembler ***********************
```

Program 11.4: (*continued*)

(depending on the servo specification), corresponding to the limits of travel, with the mid-position set by a pulse of about 1.5 ms. Some live calibration is normally required. The overall signal period is about 18 ms, but this is not critical.

In the test circuit, the PIC reads the pot input and sets the output pulse width accordingly, so the position of the servo follows the pot. To keep it simple, software loops are used to generate the PWM, but hardware timers and interrupts will typically be used in a more complete application.

Questions 11

1. Outline a method of controlling the speed of a small dc motor by PWM. Identify the main hardware components required. (4)
2. Explain how an incremental encoder can be used to provide speed and position feedback from a shaft to a microcontroller. (4)

3. Based on your answers to Questions 1 and 2, explain how the speed of a dc motor can be accurately controlled by a microcontroller using digital feedback. (4)
4. (a) Calculate the positional resolution (smallest step size) in degrees at the output shaft of a robot arm drive with a 90:1 gearbox, if a shaft encoder with 200 steps per revolution is attached to the motor shaft. (4)
   (b) Calculate, to one significant figure, the positional resolution of a robot gripper positioned at 500 mm from the axis of rotation controlled by the above axis controller. (4)

Answers on page 424. (Total 20 marks)

Activities 11

Carry out the following investigations using test hardware or simulation, or both.

1. Construct the MOT2 board on stripboard and devise a test schedule. Confirm the correct operation of the hardware prior to fitting the PIC chip, OR download the simulation of the MOT2 board and confirm correct operation of DCM1.
2. Evaluate the performance of the position control program POS2 in terms of speed of response, accuracy and reliability. What are the characteristics of the motor that affect the performance?
3. (a) Investigate the performance of the program CLS2 in terms of reliability, response time, range of control (maximum and minimum speeds). Devise a method of loading the motor to test the performance of the controller with varying loads (the speed should be held constant within limits).
   (b) Modify CLS2 to read the input push buttons on the MOT2 board to increase or decrease the set speed.
4. (a) Modify the CLS2 program for the MOT2 board to use the timer interrupt to signal time out. Compare the performance of this alternative implementation with the original program.
   (b) Modify the program CLS2 to use the port A interrupt to monitor the feedback from the motor. Compare the performance of this alternative implementation with the original program.
   (c) Modify program CLS2 to control the set speed from the analogue input AN10.

CHAPTER 12

More PIC Microcontrollers

Chapter Outline

PIC Microcontrollers. DOI: 10.1016/B978-0-08-096911-4.10012-6

Chapter Points

- The PIC family of microcontrollers share a common core architecture and instruction set.
- PICs are designed with separate program and data buses, a reduced instruction set and two-stage pipelining.
- The 10, 12, 16 and 18 series MCUs offer increasing power and features.
- Selection criteria include I/O provision, program memory size, analogue input and serial ports.
- In-circuit programming is a common feature, with in-circuit debugging in selected devices.
- Serial communication interfaces include RS232, LIN, SPI, I^2C and CAN.

The PIC® 16F84A has been used as a reference device in this book as it the simplest of the PIC 16 range. However, it is now effectively obsolete, since chips with more features are available at lower cost. The 16F690 has been selected for particular scrutiny, since it is used in Microchip's own basic demo hardware, the low pin count (LPC) board (see Chapter 7). This chapter will summarize the additional features available in other 8-bit PIC devices, so that the most suitable may be selected for a given application. Essentially, this means selecting the PIC chip that has the required number and type of inputs and outputs, program memory capacity and other special features, all at the minimum price.

The main groups of 8-bit (internal data) PIC flash devices are shown in Table 12.1. They are divided into four groups: the 10F2xxx microcontrollers (MCUs) are four input/output (I/O) devices with minimal features, the 12Fxxxx series are eight-pin miniature PICs, the 16Fxxxx group is the mid-range series, and the 18Fxxxx devices are the high-performance group with much larger memory and extensive I/O facilities.

Comprehensive selection tables and individual data sheets can be downloaded at www.microchip.com.

12.1. Common Features

All PIC microcontrollers use the same basic architecture (Chapter 5) and instruction set (Chapter 4), to provide a design progression path from simple programs to the most complex applications. The architectural features may be compared by studying the block diagram for each device found in its data sheet. The shared features of the 8-bit devices considered here are:

- Harvard architecture with RISC instruction set
- Flash program memory with in-circuit programming
- RAM block including special function registers

Table 12.1: Eight-bit PIC flash microcontroller families

PIC Flash Device Group	Summary Details
10F2XXX	Low cost and minimal size 8 pin packages, 6 pins used, 4 I/O pins Set of 33 12-bit instructions Up to 512 program instructions Up to 23 RAM locations 4/8 MHz internal oscillator only 1 × 8-bit timer only Up to 2 analogue inputs* 6 devices total
12FXXXX	Low cost and small size 8 pin packages, 6 I/O pins Set of 33/35 14-bit instructions Up to 1024 program instructions Up to 256 RAM locations 20 MHz max. clock, 4 MHz internal oscillator 8-bit and 16-bit timer Up to 4 analogue inputs* 14 devices total
16FXXXX	Mid-range cost and performance 14—64 pin packages Up to 55 I/O pins 35/49 × 14-bit instructions Up to 16k program instructions Up to 1536 bytes RAM 20 MHz max. clock, 16 MHz internal oscillator* Up to 6 × 8-bit and 3 × 16-bit timers* Up to 30 analogue inputs Up to 4 serial ports* 92 devices total
18FXXXX	Higher cost and performance 18—100 pin packages Up to 70 I/O pins 77 × 16-bit instructions Up to 64k program instructions Up to 4096 bytes RAM 40 MHz max. clock, 16 MHz internal oscillator* Up to 6 × 8-bit and 5 × 16-bit timers Up to 28 analogue inputs Up to 4 serial ports, USB, Ethernet* 198 devices total

*Selected devices in the range.

- Single working register and dedicated, non-writable stack
- Power-up, brownout and watchdog timers
- Multiple interrupt sources, with single vector address
- 8- and 16-bit hardware timers with PWM, capture and compare mode
- Sleep mode and low-power operation in selected devices
- EEPROM non-volatile data memory in selected devices.

12.1.1. Harvard Architecture

In conventional processor systems, the instruction codes and associated operands have to be transferred from memory using the same address and data bus as the system data, that is, the data read in via inputs or generated by the processor. The PIC architecture has separate paths for the instructions and the system data. Therefore, the instruction fetch operation can be carried out at the same time as the results from the previous operation are stored. As a result, the program executes more quickly at the same clock speed, by carrying out two processes concurrently. The instruction fetch and execute cycles overlap to double the execution rate (pipelining; see Chapter 5).

12.1.2. RISC Instruction Set

The PIC has a relatively small number of instructions compared with a conventional complex instruction set computing (CISC) processor. This has two main benefits: the instruction set is easier to learn and the code executes more quickly, because the instruction decoding hardware is less complicated. The downside is that more complex operations may have to be constructed from simpler ones, ending up taking longer to execute. Overall, the reduced instruction set computing (RISC) performance is usually superior because, in a typical application, these complex instructions are not needed too often. The PIC 16 chip typically has 35 instructions, while the 18 series MCUs have up to 85, so the more powerful devices need more instructions for extra performance.

12.1.3. Flash Program Memory

The introduction of flash memory was a key stage in the development of microcontrollers. Writable, but non-volatile, memory is essential in embedded systems to store the control program. Previously, erasable programmable read-only memory (EPROM) was used, but this had to be removed from the system for erasing under ultraviolet light before reprogramming. Battery-backed random access memory (RAM) was an alternative, but battery life is limited. In-circuit serial programming allows flash ROM to be reprogrammed without the inconvenience, time-wasting and possible damage caused by having to remove it from the

circuit each time. Program memory capacity ranges from 256 instructions in the smallest 10F chip to 128k in the larger 18F MCUs. The program counter range varies accordingly.

12.1.4. RAM and Special Function Registers

The individual bits in the special function registers (SFRs) need to be read and written when initializing the chip or during program operation. Because they are located in the same RAM block as the general purpose registers (GPRs), they can be accessed using the same instructions. This means that special instructions for control register access are not needed, which helps to keep the instruction set small. RAM size varies from 16 bytes to over 2k bytes, with the number of SFRs also increasing with the chip complexity and the number of peripheral interfaces.

12.1.5. EEPROM Data Memory

This is useful in applications where data read in at the ports or produced by the processor needs to be stored in non-volatile memory. For example, in a keypad-operated electronic lock, the lock code is entered by the user and then retained to be checked against user keypad input to release the lock. Data logging applications, where sampled input data may need to be retained over a period of time, may also need to store the data while the power is off. Electrically erasable programmable read-only memory (EEPROM), unlike flash ROM, can be written as individual data bytes, but is not as physically compact, so is provided in smaller blocks. EEPROM capacity ranges between 256 and 1k bytes, and is not fitted in some chips.

12.1.6. ALU and Working Registers

CISC processors tend to have a block of registers for storing current data, rather than a single working register (W) as found in the PIC. They also tend to have much larger address spaces. This means that the instructions, including operands, are typically 4 bytes in total, often more, compared with 14 bits for the PIC 16. An architecture with only one working register, used in conjunction with the RAM register block, reduces the overall number and complexity of instructions required, as the options are reduced. This does mean, however, that all data transfers must go through W. In the more powerful MCUs, the arithmetic and logic unit (ALU) support hardware can be more elaborate. For example, the 18F4580 has an 8×8 multiplier, but still has only one working register.

12.1.7. Stack Size

The stack size determines the number of subroutine or interrupt levels that can be used in the application program. A CISC processor can have an unlimited stack, as general purpose RAM is used, unlike the PIC, which has a dedicated internal stack of limited depth. The 12 series

chips have only a two-level stack, the 16 series eight levels, and the 18 series 32 levels. This reflects the typical program complexity for each type. The application programmer needs to be aware of this limitation, and balance the advantages of a well-structured program using multiple subroutine levels, and the absolute limit imposed by the stack size. Unlike in CISC processors, the stack cannot be overwritten by a program instruction, making it more secure.

12.1.8. Protection Devices

Internal timers are used to ensure a reliable start-up on power-up, after a reset or a short-term dip in the supply voltage. In the 16F690, the power-up timer provides a nominal delay of 64 ms to allow the power supplies to stabilize before the program execution begins. The watchdog timer allows the chip to reset itself automatically if the program execution fails to follow the normal sequence, thereby improving overall reliability. Brownout protection allows the chip to reset in an orderly fashion if the power supply fails for a short period. Full details, including a block diagram, are provided in each chip data sheet. Over time, these protective features are becoming more elaborate, so that, if they are properly applied, the overall reliability of PIC applications is improved, at the expense of the firmware becoming more complex.

12.1.9. Interrupts

An interrupt is an internally or externally generated signal, which forces the processor to suspend the current operation and execute a interrupt service routine (ISR). The ISR thus has a higher priority than the background process. PIC chips provide a variety of interrupt sources, for example, a change on a selected input, or a hardware timer timeout. There is an interrupt priority system available in the more advanced 18 series PICs; this allows the chip to be set up to ignore an interrupt source if a more important one is already active. In CISC microprocessors, multiple interrupt vectors are usually available, and a different ISR can be invoked for different interrupt sources. In the PIC, all interrupts have to be serviced via the single interrupt vector, located at address 004 in program memory. To differentiate between them and to determine the action required, the ISR needs to check the relevant control register flags, to find out which interrupt source is active, before branching to the required routine. As the number of peripheral devices increases, such as additional timers, serial ports and so on, the number of potential interrupt sources increases, making interrupt servicing via a single vector more complicated.

12.1.10. Hardware Timers

The number of hardware timers available generally increases with the chip complexity. They are either 8-bit or 16-bit counters, with prescalers or postscalers, which divide down the input or output of the counter to extend its range. If we take the motor program in Chapter 11 as an

example, we can see how additional hardware timers could have been utilized; the 20 ms time interval and the motor output cycle delay could both have been implemented as hardware operations, while the sensor pulse monitoring could have used RB0 interrupt. As well as simple counting and timing, the hardware timers can be configured to measure input intervals, generate timed outputs and drive output loads using pulse width modulation (PWM; see Section 12.3.1 below). Sometimes, independent clocks are associated with the timers and protective devices, so that they can continue to function when the main clock fails, or a different timebase is needed.

12.1.11. Sleep Mode

This is useful for terminating programs that do not loop continuously, suspending operations pending an interrupt or saving power in battery applications. The processor switches off the clock and disables most of its normal functions when the instruction SLEEP is encountered, with current consumed dropping to around 1 μA, or less in low-power devices (LP designation). Using SLEEP to terminate a program prevents it continuing into unprogrammed memory locations. These default to all 1s, which generally corresponds to a valid instruction code, ADDLW FF in 14-bit code (W, C, DC and Z all affected). If a program is not terminated with a SLEEP or GOTO instruction, the program will carry on to the end of memory, the program counter will roll over to zero and the program will restart. A hardware reset input or suitable interrupt will be needed if the application code is terminated with SLEEP.

12.1.12. In-Circuit Programming

The PIC microcontrollers use a common program downloading system, which consists of transferring the machine code in serial form via one of the data pins when the chip is in programming mode. Previously, the chip would be placed in a programming unit for downloading the application code, and then physically transferred to the application board. Now, the chip is normally programmed in circuit, where the chip can be left in circuit, reducing the risk of damage. It can then be programmed after the circuit has been manufactured, and reprogrammed at any time, via a six-pin on-board connector, which can be seen in the 16F690-based LPC board.The connector can be in the form of a SIL (PICkit 2/3) or RJ-11 (ICD 2/3) connector (see Chapter 7).

12.1.13. In-Circuit Debugging

Mid-range PIC chips are now generally being designed to support in-circuit debugging (ICD), where, after programming, the firmware can be executed within the final target hardware under control of the MPLAB IDE via the programming interface. This allows the application to be more fully tested using the same debugging tools available in MPSIM and makes it easier to

eliminate firmware and hardware bugs in the final stages of testing. Smaller chips, which do not provide internal ICD support, must be debugged using a header connector that carries a version of the chip that contains the ICD circuitry. The PICkitX development programmer provides all programming and debugging features we need at minimal cost, while the ICDX module may be needed to debug smaller chips which require a header.

12.1.14. Power PICs

To complete the analysis of the whole PIC range, the features of the more powerful PICs must be considered. The PIC24 series of microcontrollers have a 16-bit internal data bus and ALU, and move from the single working register model to a set 16×16 bit working registers. Otherwise, the architecture is similar to the 8-bit MCUs. The dsPIC30 and dsPIC33 are also 16-bit processors, but include a digital signal processing (DSP) engine for fast floating-point calculations. Top of the range are 32-bit PIC32 chips with the kind of central processing unit (CPU) enhancements used to improve performance in CISC devices such as the Intel® PC processors: a split-bus architecture, instruction pre-fetch and cache, five-stage execution pipeline and high-performance interrupt controller. The same MPLAB IDE supports the whole range, but these processors will normally be programmed in C, to make optimum use of the additional features.

12.2. Device Selection

Each type of PIC microcontroller offers a different combination of features; the most suitable can be selected for any given application. The range is expanding all the time, with additional features and improved performance at lower cost. Tables of MCU families at www.microchip.com allow the current features and price of each to be readily compared. The key selection criteria are:

- Total number of I/O pins available
- Grouping of I/O in ports
- Program memory size
- Data RAM size
- EEPROM data memory availability
- Timers (8-bit or 16-bit), CCP, PWM
- Number of 10-bit analogue inputs
- Serial comms (USART, SPI, I^2C, CAN, LIN)
- Internal/external oscillator and maximum clock speed
- Package/footprint (DIP, SOIC, PLCC, QFP)
- Price

When developing an embedded application, the hardware will generally be specified and designed first. This will determine the number and type of inputs and outputs required. Simple switches will require single digital input, while a keypad will require several inputs. A temperature sensor will need an analogue input, while a motor will probably require a PWM output. Most systems use some kind of status or information display, and the type of display will determine the number of output pins needed to drive it. Serial communication will often be used if the PIC is part of a larger system or is connected to a master controller.

When the hardware requirements have been established, the program can be developed, and tested by simulation. The size of the program will then be known, so that chip memory size can be specified. In addition, the size of the stack in the selected device must be sufficient for the number of subroutine levels and interrupts; if not, the program can be restructured or a different chip used. If necessary, an overspecified chip can be used initially, and the chip that matches the application requirements more exactly substituted later.

When the design parameters, such as I/O requirements, program memory size and so on, have been finally established, the most suitable device can be selected using the search facilities on the manufacturer's website. Summary information for selected 8-bit PIC flash microcontrollers is provided in Table 12.2, as a guide to the range of features available. The current device selection tool on the website is illustrated in Figure 12.1.

The smallest 8-bit chip currently available has only four I/O, 256 instructions and one timer with an internal 4 MHz oscillator. The largest has 70 I/O, 128k program memory, a more extensive instruction set and multiple peripherals, and runs at 42 MHz.

12.2.1. Input/Output Pins

The number and type of inputs and outputs required needs to be considered at an early stage in circuit design. The convenient grouping of the pins for particular interfaces may also be relevant, with many chips having partial port implementations. For example, a 4-bit port can be conveniently used for a 4-bit input from a DIL switch, while a seven-segment display with a decimal point needs the full 8-bit port. The number of analogue inputs available must also be adequate for inputs requiring a voltage measurement. Most I/O pins have more than one function, one of which is selected during initialization by setting up the relevant control register. If no setup is performed for a particular pin, it will normally default to a digital input, or an analogue input if this is an option. Pins can be reconfigured within the program sequence to have a different function at different times. If this is the case, the designer must ensure that the two functions do not interfere with each other, in terms of both the hardware and the software.

Table 12.2: Selected 8-bit PIC flash microcontrollers

PIC Device no.	Total Pins	I/O Pins	Program ROM Words	File RAM Bytes	EEPROM Bytes	Analogue Inputs × 10 Bits	Timers 8+ 16 Bit	Max. Clock (MHz)	Internal Oscillator (MHz)	In-circuit Debug	CCP/ PWM modules	Serial Comms	No. of Instructions	Relative Cost
10F200	6	4	256	16	–	–	1	4	4	–	–	–	33	0.30
10F222	6	4	512	23	–	2	1	8	8	–	–	–	33	0.39
12LF1822*	8	6	3k5	128	256	4	2 + 1	32	32	✓	1	All	49	0.73
12F508	8	6	512	25	–	–	1	4	4	–	–	–	33	0.41
12F629	8	6	1k	64	128	–	1 + 1	20	4	✓	–	–	35	0.70
16F627A	18	16	1k	224	128	–	2 + 1	20	4	–	–	UART	35	1.30
16F648A	18	16	4k	256	256	–	2 + 1	20	4	–	–	UART	35	1.67
16F676	14	12	1k	64	128	8	1 + 1	20	4	✓	–	UART	35	0.98
16F690	20	18	4k	256	256	12	2 + 1	20	8	–	1	All	35	1.20
16F72	28	22	2k	128	–	4 × 8-bit	2 + 1	20	–	–	1	–	35	1.91
16F77	40	33	8k	368	–	8 × 8-bit	2 + 1	20	–	–	2	All	35	4.12
16F819	18	16	2k	256	256	5	2 + 1	20	8	✓	1	I^2C, SPI	35	1.78
16F84A	18	13	1k	64	64	–	1	20	–	–	–	–	35	3.11
16F877A	40	33	8k	368	256	8	2 + 1	20	–	✓	2	All	35	N/A
16F88	18	16	4k	368	256	7	2 + 1	20	8	✓	1	All	35	2.20
16F887	40	33	8k	368	256	14	2 + 1	20	8	✓	2	All	35	1.77
16LF1907*	40	33	8k	256	–	14	1 + 1	20	16	✓	–	UART	49	F
18F1220	18	16	2k	256	256	7	1 + 3	40	8	✓	1	UART	77	1.96
18F4320	40	36	4k	512	256	12	1 + 3	40	8	✓	2	All	77	4.81
18F4580	40	36	16k	1536	256	11	1 + 3	40	8	✓	2	All	83	4.38
18F6520	64	52	16k	2048	1024	12	1 + 3	40	–	✓	5	All	77	5.93
18F97J60*	100	70	128k	3808	–	16	2 + 3	42	–	✓	5	All + Ethernet	85	3.77

N/A: no longer available; F: future product at time of writing.
*Low-voltage operation (maximum 3.6 V).

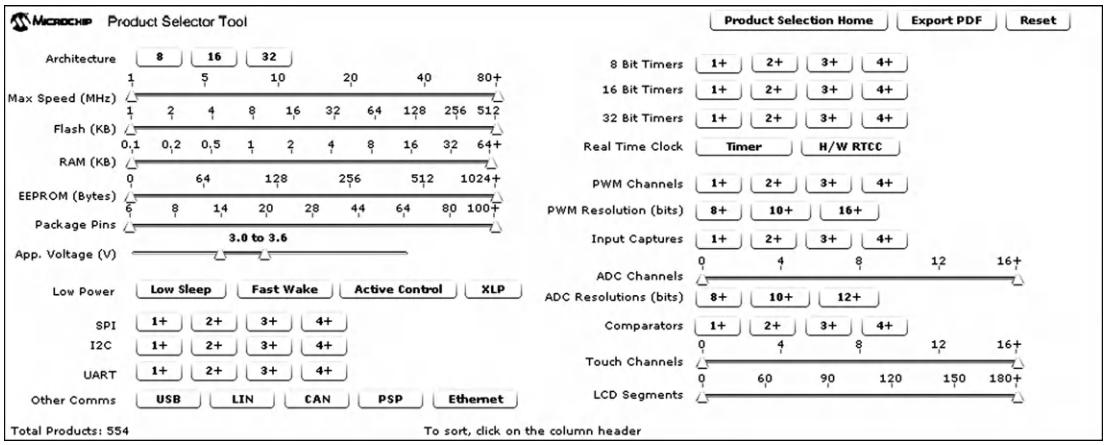


Figure 12.1
Microchip Product selector tool

12.2.2. Program Memory

The specification of memory size can only be finalized after the software has been developed, but an experienced application developer should be able to anticipate this requirement fairly early on. Otherwise, a chip with more than enough memory can be used at first, and one with the correct capacity selected later on. If the program is developed in 'C' language, the memory size required will be greater, because each program statement can expand into several machine code instructions. In this case, an 18 series device is likely to be the best choice. Microchip supply a free C compiler for the 18XXXX chips, and third party compilers are also available for both the 18 and 16 series chips. PIC microcontrollers are generally supplied with flash program memory, as this is the most flexible option for prototyping and production. If larger volumes of chips with a fixed program are required, masked ROM program memory (not reprogrammable) can be configured in the final manufacturing stage.

12.2.3. Data Memory

The file register RAM block, which includes the SRFs and GPRs, tends to increase in size with the program memory size and chip complexity, and ranges from 16 to 4096 bytes in the 8-bit PICs. Note that some blocks of RAM are unique, while others are common to all the RAM banks (that is, the same register is accessed at the corresponding address in different banks), while other address ranges may not be implemented at all. Therefore, the total amount of RAM cannot simply be calculated as the number of locations per bank multiplied by the number of banks. For example, the 16F690 has four banks of 128 locations, equivalent to 512 addresses. The first 32 addresses in each bank are assigned as SFRs (total 128), but there are only 256 bytes in total of unique RAM locations (20h−7Fh, A0h−EFh and 120h−16Fh). The remaining

128 registers are duplicates or unimplemented. The number of variables and temporary data storage blocks required can be totaled when the program has been developed, perhaps adding an allowance for future expansion or changes to the specification. If non-volatile data storage is needed, the EEPROM size must also be checked.

12.2.4. Internal Oscillators

To save on external components, many PICs now include an internal oscillator. In the '690, this runs at 8 MHz, or lower frequencies by division, with a default of 4 MHz selected if the OSCCON register is not initialized and the internal oscillator selected in the configuration word. More recently, 32 MHz internal oscillators have been introduced. The frequency can be calibrated using an internal register if a more accurate clock is needed. However, an external crystal clock will still provide maximum accuracy. In recent chips, multiple clock modes are available to optimize the tradeoff between clock speed, accuracy and power consumption. Many chips now have an additional internal oscillator, an internal 31 kHz clock, which can be connected to Timer 1 as an independent timebase and drives the power-up timer system.

12.2.5. Clock Speed

Clock speed is the primary factor in the performance of any microprocessor system, and is critical in some applications. For example, in the motor control example previously described, the higher the clock speed, the more precise the control can be, as the shaft speed can potentially be measured more accurately. Most of the flash PICs currently available operate at up to 20 MHz (12 and 16 series) or 40 MHz (18 series). This gives an instruction cycle time of 200 or 100 ns (nanoseconds) and an execution rate of 5 or 10 MIPS (million instructions per second). All PICs use a fully static design, which means that they can operate down to zero frequency. The clock rate is limited by the time taken by the internal signals to rise and fall, so correct performance is only guaranteed up to the maximum rated speed. The maximum speed is also limited by power dissipation and the consequent heating effect.

12.2.6. Power Consumption

Power consumption is generally proportional to clock speed in complementary metal oxide semiconductor (CMOS) devices, since most of the power is consumed when the transistors switch on and off. This is illustrated by the current consumption curve for a typical device, shown in Figure 12.2. For external crystal operation at clock frequencies between 4 and 20 MHz, high-speed (HS) mode must be selected when programming the chip, and below 4 MHz crystal (XT) mode. The power consumption at high speed may necessitate additional cooling measures to keep a chip within its temperature limits, especially in the larger PICs. A heat sink or even a fan, as found on the processor in a typical PC motherboard design, could be

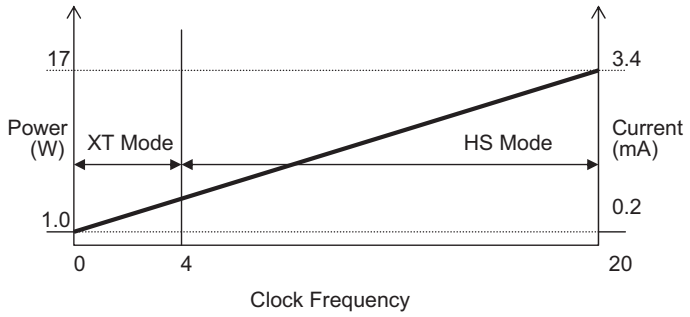


Figure 12.2
PIC power consumption against crystal clock rate

used. For this reason, a major development effort has recently been applied to reducing power consumption. Extreme-low-power (XLP) devices now operate at low supply voltage (3 V) for battery-powered operation, with a typical battery life of 8 years being claimed. Judicious use of sleep and wake-up modes is also important in minimizing power consumption.

12.2.7. Packaging

Some sample integrated circuit (IC) packages are shown in Figure 12.3. The traditional package for integrated circuits is the plastic dual in-line (PDIP) chip, which has two rows of pins spaced at 0.1 inch intervals. The maximum number of pins that can practically be accommodated in this type of package is 64, so other formats have been adopted for larger

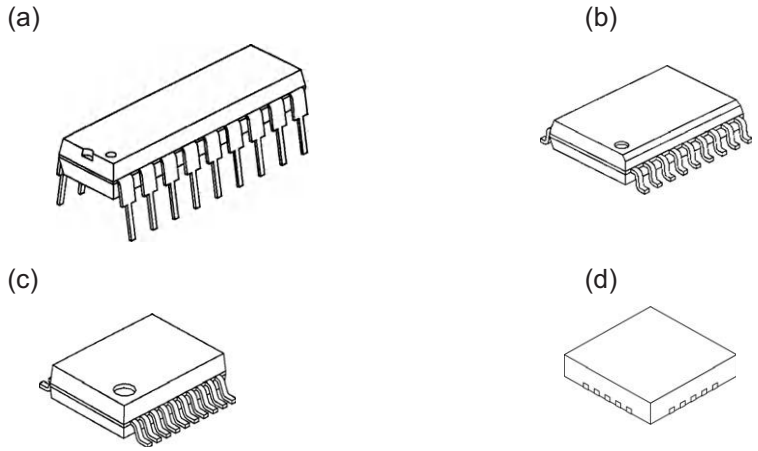


Figure 12.3
MCU packages (18-pin) (not to scale): (a) plastic dual-in-line package (PDIP); (b) small outline integrated circuit (SOIC); (c) shrink small outline package (SSOP); (d) quad flat no lead (QFN)

chips, and to reduce the board area consumed. The plastic leaded chip carrier (PLCC) package has the pins arranged around four sides of a square package, which is designed to fit in a recessed socket. The pin grid array (PGA) has pins arranged in a grid covering one side of the package, with a flat socket mounting.

The actual IC occupies only a small central portion of the dual in-line (DIP) package, so miniaturized packages are possible, if the means to connect them is provided. Surface-mount components are now normally used in commercial products, as chips become larger, circuits more complex, and products themselves miniaturized. The pins of the ICs are not fitted through holes on the board, but soldered onto the surface on flat pads. Surface-mount boards require very precise manufacturing techniques, generally being produced on automatic production systems. Solder paste is applied by printing techniques, components are added by pick-and-place machines, and the whole board is flow-soldered in one operation.

The small outline integrated circuit (SOIC) is a surface-mount DIL package with a pin pitch of 0.05 inches. The smaller shrink small outline plastic package has a pin pitch of 0.026 inches. Quad flat pack (QFP) is a square surface-mount package for larger chips, such as the 44-pin PIC 16F887, with pins on four sides. The larger 8-bit chips can use thin quad flat pack (TQFP), with rows of pins on four sides, or ball grid array (BGA), which has balls of solder attached instead of pins, ready for machine soldering.

12.2.8. Price

The relative cost for each chip shown in Table 12.2 is based on the 'budgetary price' quoted by the manufacturer at the time of writing. Relative cost can be compared, while the actual price will obviously increase in time, and will depend on the volume purchased and third-party supplier prices. The individual price is determined by the complexity of the chip and also the volume of production.

As the range is constantly updated, each design will be superseded by a chip with better features; as the volume builds up, the new device becomes cheaper owing to economies of scale in production and recovery of development costs. New chips may be also sold at a reduced price as a marketing strategy. The older design may become relatively more expensive, as well as having fewer features, before slipping into obsolescence. For example, at the current time the guide price quoted for the original 16F84A is US $3.11, while the pin-compatible replacement, the 16F819, which has analogue inputs and other extra features, is only $1.78, and the 16F690, which has even more features, is $1.20. Therefore, while the older chip has been used as an example because it is less complicated, the reader should consider using the more recent chip in new designs, even if its features are not used to the full. Some devices previously available are no longer produced. In particular, the 16F877A, which had a comprehensive set of features in a 40-pin package, which made it versatile and popular, has been superseded by the 16F887, which has an internal oscillator and other improvements.

12.3. Peripheral Interfaces

In the block diagram of each chip, the peripherals are shown as separate blocks attached to the internal data bus. These provide additional features such as timers, analogue inputs and serial ports. These are set up for use by initializing the related SFRs. The appropriate combination of peripherals is a major factor in chip selection.

12.3.1. Timers and CCP

Hardware timers are used for timing and counting operations, allowing the processor to carry on with some other process while the timer process runs. Basic timer operation has been described in Chapter 6, where a clock input drives a counting register to measure time or count external events. Its functionality can be extended by using additional registers to store timer values, creating a CCP module. CCP stands for capture/compare/PWM.

Capture mode provides input interval measurement (Figure 12.4a). The value in a timer register is captured (stored) when an input changes; the time between the timer start and input change is therefore recorded. In the motor application, for example, the timer could be started when a pulse is received from the shaft sensor, and the time captured when the next pulse arrives, giving the period of the shaft sensor pulse. An interrupt can be enabled to signal this event.

Compare mode provides output interval generation (Figure 12.4b). A value is loaded into a register, which is then continuously compared with a timer register as it runs. When the register values match, an output pin is toggled and an interrupt generated to signal the timeout event. This is a convenient way to generate a timed interval, so that, for example, an output pulse waveform can be generated with set pulse period.

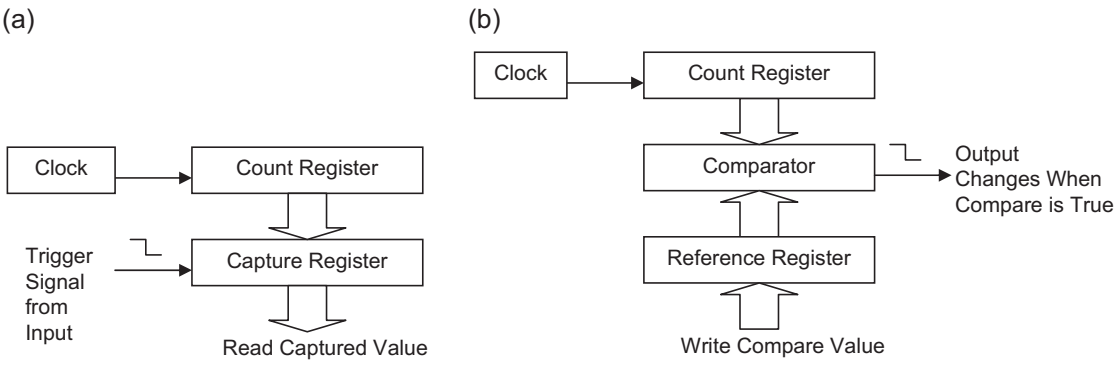


Figure 12.4
Timer CCP operations: (a) capture; (b) compare

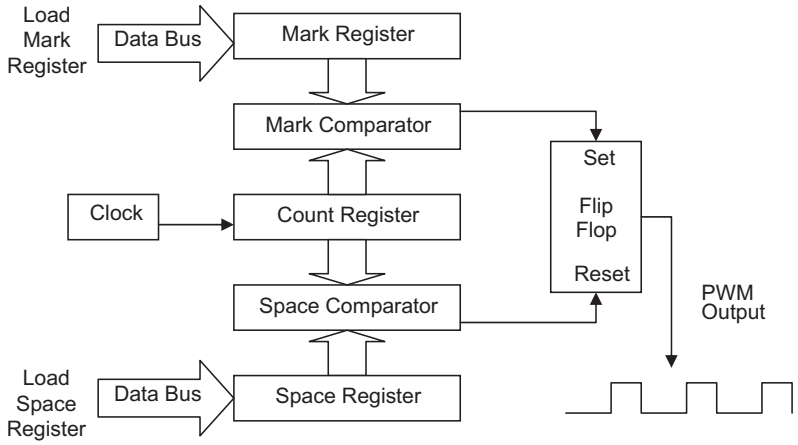


Figure 12.5
Pulse width modulation (PWM)

In PWM mode, preset values are loaded into two registers representing the mark and space period of the PWM output required (Figure 12.5). The timer value is then compared with the mark register and the output toggled after the mark value is reached. The timer is then restarted and compared with the space value as it runs, and the output toggled when the space value matches. The process is repeated, causing the output from the flip-flop to toggle after each mark and space interval, generating a PWM output.

12.3.2. Analogue Comparators

The comparator allows one voltage to be compared with another, and sets or clears its output bit depending on the polarity of the input. Many PICs incorporate comparator inputs as well as analogue/digital (A/D) inputs, with the result bits recorded in a relevant SFR. Often, there are multiple inputs which can be set up to operate in different combinations, and also trigger a range of output events, such as an interrupt. See, for example, the block diagram of the 16F690 C1 comparator module (Figure 8-2 in the data sheet).

12.3.3. Analogue/Digital Inputs

The analogue inputs are used in control systems with input sensors that produce a voltage, current or resistance change in response to an environmental variation or system measurement. For example, in the LPC board, a pot is connected to RA0, which is designated AN0 when used as an analogue input. In the test program, it reads 0−5 V from the pot and uses this value to control the speed of the LED output scan, by copying it into the delay counter as the initial value. The temperature controller described in Chapter 13 is designed to accept inputs from

temperature sensors which give an output change of 10 mV/°C. The PIC then operates outputs to a heater or a cooling fan, which keep the temperature in the target system constant.

Most PICs provide 10-bit conversion. This means that the input voltage is converted to a 10-bit number, giving a resolution of 1 in 1024, or better than 0.1%. This is good enough for all but the most demanding applications. If the full resolution is not required, an 8-bit result can be used by ignoring the two extra bits. Multiple analogue inputs are usually available; the PIC 16F690 has 12 (AN0 to AN11). Code for performing the analogue input conversion is given in the LPC Program 7.1.

The analogue-to-digital conversion (ADC) system is illustrated in Figure 12.6. The port containing the ADC inputs can be set up with a combination of analogue and digital inputs, or all analogue. One of the analogue inputs is selected at a time for conversion, and the converter output is stored in an ADC result register. The maximum voltage level to be converted (reference voltage) can be set externally, or the internal supply voltage (+5 V) can be used. In the temperature controller board, an external voltage reference of +2.56 V is used, because this gives a convenient 0.01 V per bit conversion for an 8-bit result. The converter is driven by the chip clock, but a divider must be set up to allow the minimum specified conversion time (about

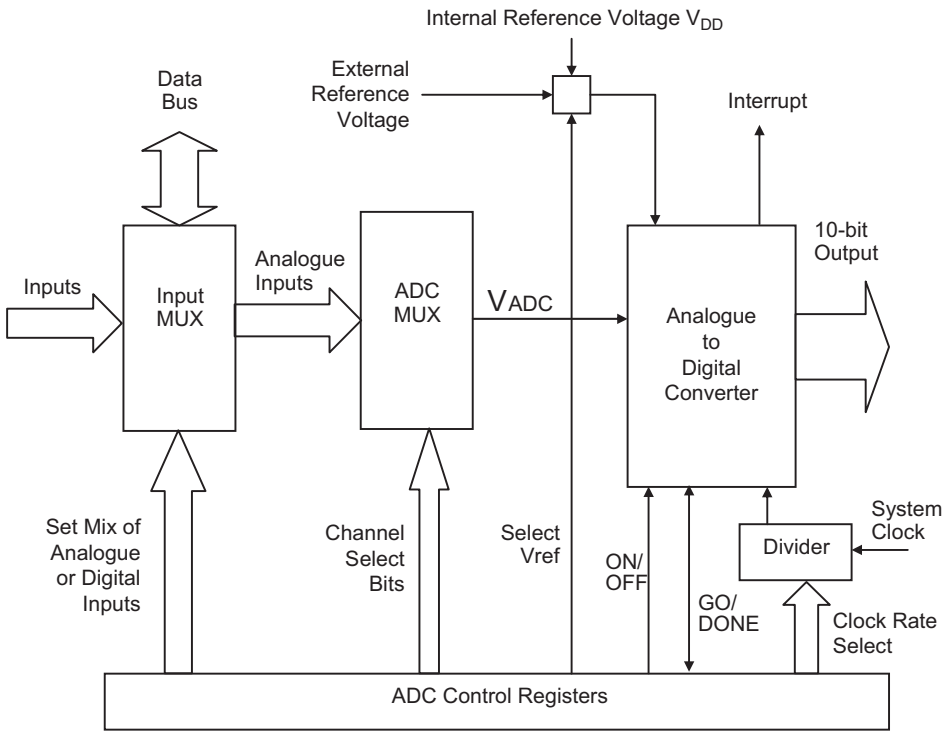


Figure 12.6
Analogue-to-digital converter general block diagram

20 μs); for example, if the chip clock is 20 MHz, divide by 32 must be selected, and at 4 MHz, divide by 8. The GO/DONE bit in the control register is used to start a conversion; the same bit indicates when the conversion is finished.

The ADC works by successive approximation, details of which can be found in standard electronics references. The converter consists of a register, a digital-to-analogue converter (DAC) and an analogue comparator. The register is loaded with the half-range value (512 for 10 bits) and this is converted to an analogue value by the DAC, whose maximum output is set by the ADC reference voltage. The DAC output voltage is compared with the input, and if the input is higher, the comparator value is increased by half of the remaining range ($512 + 256 = 768$, set bit 8). The input is compared again and the register value adjusted up or down, until the value converges on the actual input value within 10 iterations.

12.4. Serial Ports

Serial communication ports allow the PIC to communicate with other MCUs, or exchange data with a master controller, via a single connection. Serial connections may also be made with external memory devices and sensors. There are several protocols available in PICs:

- USART (universal synchronous asynchronous receiver transmitter)
- SPI (serial peripheral interface)
- I^2C (inter-integrated circuit)
- LIN (local interconnect network)
- CAN (controller area network)
- Ethernet
- USB (universal serial bus)

12.4.1. USART

RS232 is an asynchronous communication protocol previously used in the serial (COM) port of the PC for connecting peripherals such as the mouse, before USB was developed. It is low speed, but easy to understand, and has been used as the direct communication between computers, terminals and other systems for many years. It is also still used to download programs to the PIC MPSTART programmer module.

'Asynchronous' means that no separate clock signal is provided with the data, so correct reception of data relies on the sender and receiver operating at the same speed, with reception synchronized using a start bit for each byte. Serial data is sent and received as individual bytes using a pair of shift registers (Figure 12.7a). After each bit is shifted out of the send register onto the line, it must be shifted into the receiver register at the same time. In other words, the receiver must sample the line during the time that the transmitted bit is present. It must then take the next sample after the appropriate interval, which depends on the data rate.

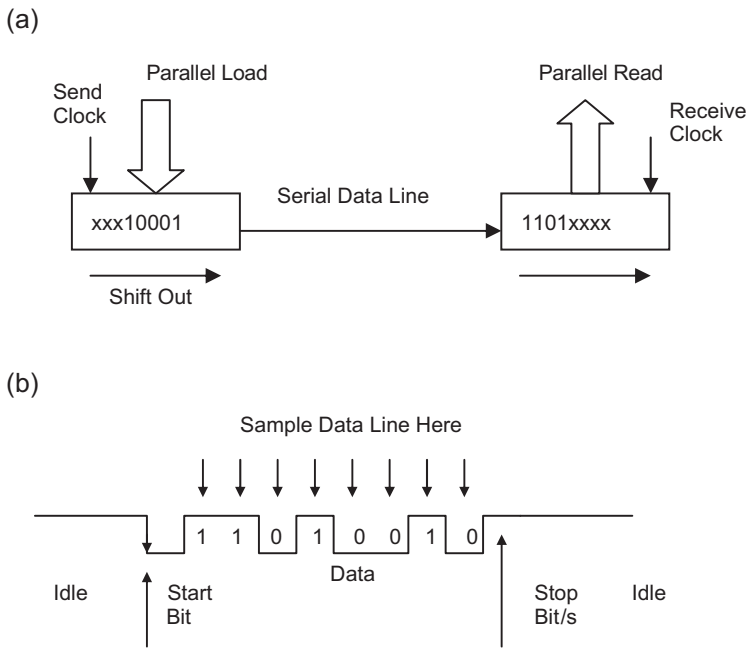


Figure 12.7
USART operation: (a) shift register operation; (b) serial data signal

The 'baud rate' sets this time interval, and there is a set of standard rates of between 300 and 115 200 bits per second. The sender and receiver must be initialized to operate at the same baud rate. At a typical rate of 19 200 baud (about 20 kbits/s), the bit time interval is about 50 μs. The signal is illustrated in Figure 12.7(b). The line is high when inactive; the start of a byte is indicated by the falling edge of a start bit. The receiver then samples the line at the required interval to read each bit, and the sampling is retriggered at the start of each byte.

When the USART is operated in asynchronous mode (Figure 12.8), there is a separate data path for send (TX) and receive (RX). One byte of data is transmitted at a time down the serial line, with start, stop and optional error check (parity) bits. If an error is detected, a retransmission can be requested. A synchronous mode is also available, when the TX pin is used instead to

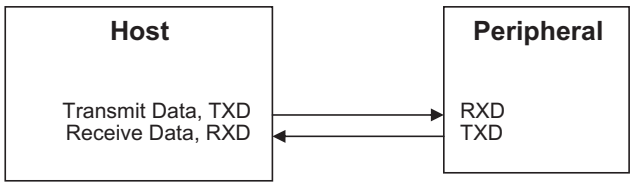


Figure 12.8
USART connections

carry a clock (CK) signal. This is sent alongside the data signal to clock the receiver, making the process more reliable. In this mode, the device can still send and receive, but only in one direction at a time.

The RS232 data signal produced by the PIC USART is output at TTL (transistor–transistor logic) levels. Most terminals, such as the PC, will produce a signal that is transmitted at higher bipolar voltage, typically ± 12 V, to allow the signal to travel further on the line (up to about 100 m). If the PIC is to communicate with such a terminal, the signal must be passed through a line driver, which will boost the voltage and shift the level as required.

12.4.2. SPI Bus

The SPI system uses three pins on each system device:

- Serial data out (SDO)
- Serial data in (SDI)
- Serial clock (SCK).

It is a single-master, multi-slave system, using hardware slave selection (Figure 12.9). To exchange data with a slave, the master selects it by taking the slave select input low (!SS). Synchronous 8-bit data is then exchanged via SDI or SDO, with a clock pulse to strobe in each bit to the destination register. Owing to the hardware selection requirements, this system is most suitable for communication between devices on the same board. Data can be transmitted and received at the same time, at a clock rate of up to 5 MHz with a 20 MHz chip clock.

12.4.3. I^2C Bus

The I^2C (pronounced eye squared see) system needs only two pins on each system device:

- Serial data (SDA)
- Serial clock (SCL).

This system also uses synchronous master–slave communication, but with a software- rather than a hardware-based addressing system (Figure 12.10). As in a network, the destination address is transmitted on the same line (SDA) before the data. A 7- or 10-bit address can be used (up to 1023 slaves), which must be preprogrammed into an address register in each slave. The slave then only picks up the messages with its own address. The clock can operate at up to 1 MHz. I^2C is suitable for communication between separate microcontroller boards, since no slave selection hardware connections are needed. Compare with SPI, the hardware is simpler, but the software is more complex. Note that in the hardware diagram, the lines are pulled up to +5 V, giving active low, wired-OR operation on the serial bus and clock line.

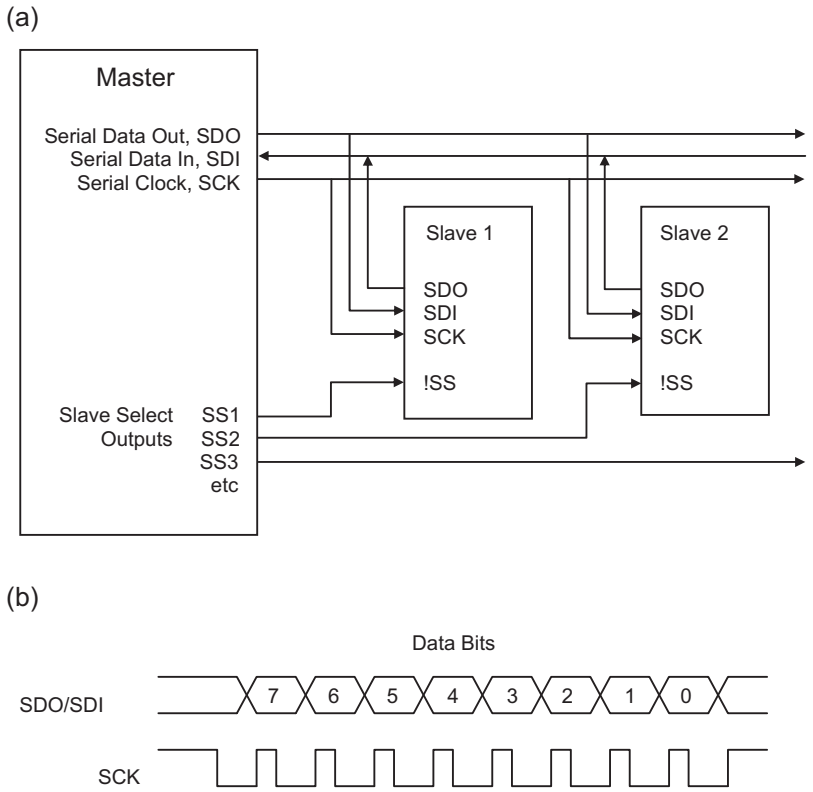


Figure 12.9
Serial peripheral interface (SPI) communication: (a) SPI connections; (b) SPI signals

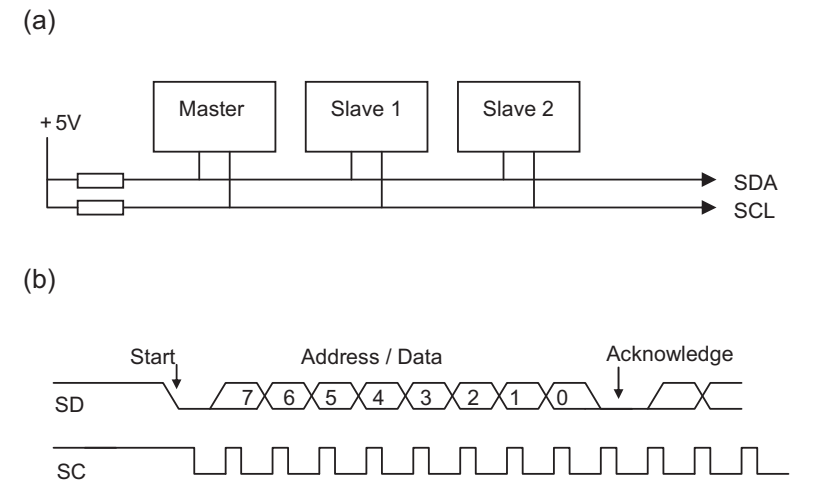


Figure 12.10
Inter-IC bus (I^2C): (a) PC connections; (b) PC signals

12.4.4. LIN Bus

The LIN bus is a mixture of the I^2C and RS232 protocols (Figure 12.11). The PIC interface is designated EUSART (extended USART) to indicate that it supports this additional bus option. It is a single-master, multi-slave protocol using a single bidirectional signal wire operating at 9−18 V (12 V) with open collector output bus transceivers and pull-up resistors. The transceivers are connected to the TX and RX pins as per RS232. Data and control bytes are transmitted in asynchronous mode with start and stop bits in the same way as RS232, but sent as message blocks with synchronization, identifier and up to 8 bytes of data, and terminated by an error check byte in the same way as a network data frame. It is primarily designed for automotive systems, where a reasonably simple and robust protocol is needed to integrate distributed controllers into a network. See Microchip Application Note AN729 for details.

12.4.5. CAN Bus

The CAN system (Figure 12.12) is also designed for transmitting signals in electrically noisy environments, such as motor vehicle control, using differential current drivers operating at 5 V. It is a multi-master system, meaning that any of the system nodes (electronic control units) can send a message at any time. This consists of a data frame containing identifier bits, up to 8 data

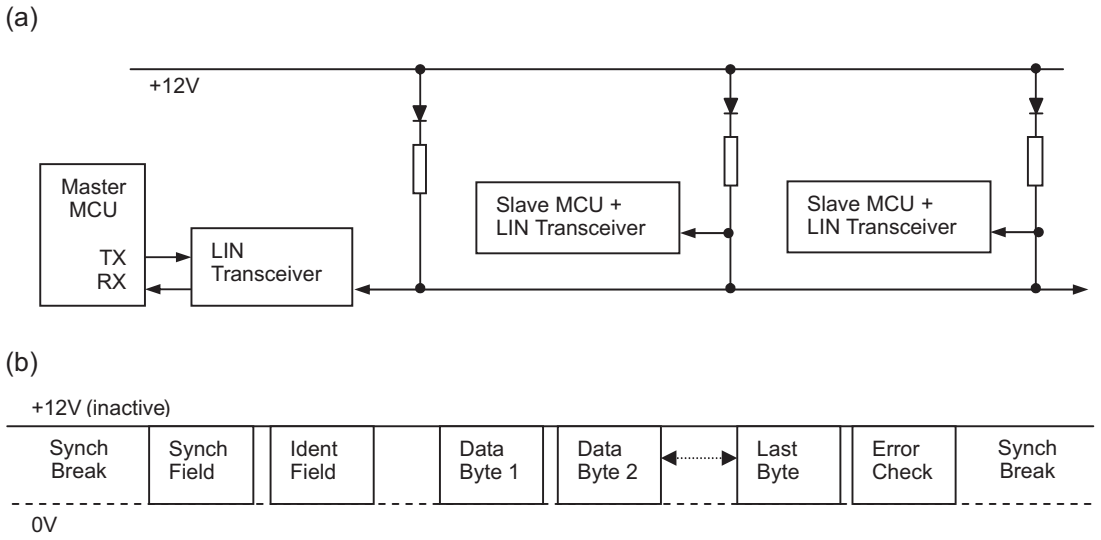


Figure 12.11
LIN controller network: (a) hardware connections; (b) signal

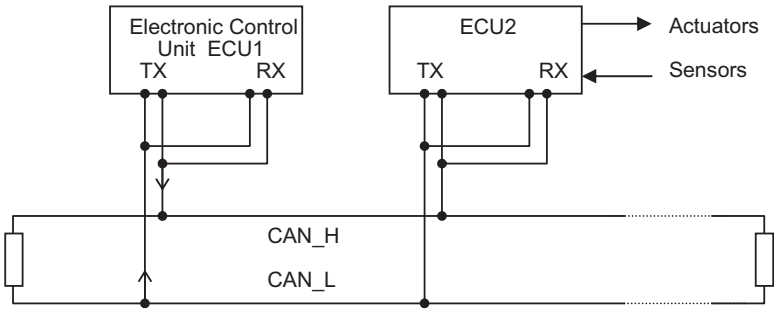


Figure 12.12
CAN bus system

bytes, error checking and acknowledge bits. Collisions are resolved by each message having a priority code within its identifier code. CAN bus is only currently available in selected high-performance PIC 18 series devices.

12.4.6. Ethernet and USB

Ethernet and USB interfaces are now being added to some of the more powerful PIC MCUs, so they can be connected directly to standard peripherals. Some PIC32 (32-bit) devices support 100 Mb/s Ethernet, PIC24 (16-bit) chips do not, while some PIC18 (8-bit) chips offer 10 Mb/s Ethernet. All groups include chips with a USB 2.0 interface. Both interfaces require significant additional hardware and firmware capabilities to support these complex communication protocols.

More details on these communication interfaces is provided in *Interfacing PIC Microcontrollers: Embedded Design by Interactive Simulation* (Newnes 2006) by this author.

Questions 12

1. Summarize the key differences between the PIC 10, 12, 16 and 18 series of microcontrollers. (16)
2. State two advantages of in-circuit serial programming. (4)
3. From the table of PIC flash microcontrollers (Table 12.2), select for minimal cost and name:
   (a) a device that has eight analogue inputs in the smallest package
   (b) a device that could control two PWM motor outputs, has EEPROM, runs at 40 MHz and can be programmed in C. (4)
4. Explain the essential difference between capture and compare timer operations. (3)
5. Describe the essential difference between SPI and I^2C addressing. Which has more complex hardware requirements? (3)

Answers on pages 424–5. (Total 30 marks)

Activities 12

1. Download the data sheet and study the summary page for the PIC 12F675, 16F690 and 18F8720. Summarize the features of each, and suggest a typical application for each device.
2. A robot has four axes to be controlled by a PIC MCU. Each has a PWM speed controlled motor and an incremental encoder with three digital outputs providing the position feedback. A block of EEPROM is needed to store up to 128 programmed positions, requiring a 16-bit code for each axis for each position. Select the most suitable chip from the Microchip website that could be used as a controller for the robot positioning system. Download the data sheet and draw a block diagram for the system, identifying the pins that should be connected to the motors and encoders. Outline how the controller will move the robot between programmed positions. Refer back to Chapter 11 if necessary.
3. Sketch a block diagram for an alternative implementation of the robot controller in Activity 2 above, using a separate controller for each axis connected to a master SPI controller. Select suitable chips for the master and slave controllers, and list the connections required. Compare the cost of each system and suggest an advantage of the master—slave system over the single controller solution proposed in Activity 2.

CHAPTER 13

More PIC Applications

Chapter Outline

Chapter Points

- The PIC 16F887 has a full range of peripheral interfaces, with analogue inputs, serial ports, CCP and PWM.
- The TEMCON2 temperature controller board has four sensors, three power outputs, a keypad and two-digit display.
- TEMCON2 is programmed to maintain the temperature in a heating/cooling system within a set range.
- A similar application is implemented without the keypad using the 16F818.
- A similar application is implemented without the display using the 12F675.
- The 18F4580 is similar to the 16F887, but can be programmed in 'C' and runs at twice the speed.

PIC Microcontrollers. DOI: 10.1016/B978-0-08-096911-4.10013-8

In Chapter 12, the features of the main groups of PIC® flash microcontrollers (MCUs) were outlined. The 16 series provides an intermediate range of features, with between 13 and 33 I/O pins and 1k−8k of program memory. The miniature group (10F/12F) of eight-pin chips have six I/O pins and 1k instructions. The 18F high-performance group provides up to 64k program memory and 70 I/O pins. In this chapter, an application for the 16F887 will be described in some detail, along with how similar applications can be designed using smaller devices, with more limited features.

13.1. TEMCON2 Temperature Controller

The 16F887 has a comprehensive set of peripheral features, including 14 analogue inputs, three timers, four ECCP (Enhanced Capture Compare PWM) channels, extended universal synchronous asynchronous receiver transmitter (EUSART), MSSP (Master Synchronous Serial Port) and in-circuit debugging (ICD). The '887 is a pin-compatible replacement for the 16F877A, with some extra features such as additional analogue inputs, comparators and pulse width modulation (PtWM) outputs, an internal oscillator and local interconnect network (LIN) connectivity.

The temperature controller application board TEMCON2 uses most of the available I/O, matching the PIC selected to the application fairly closely. The 8k memory should be sufficient for most application programs that might be developed for this hardware. A demonstration program is provided which will exercise the simulation design and hardware for test purposes.

13.1.1. System Specification

A heating and ventilation system is required to control the environment in a space such as a greenhouse where the temperature must be kept within set limits (0−50°C). A basic block diagram is shown in Figure 13.1.

The unit will be programmed to accept a maximum and minimum temperature, or a set temperature and operating range. The system will operate on the average temperature reading from four sensors, to give a more accurate representation of the overall temperature. Using more than one temperature sensor also allows the system to tolerate a fault in one sensor, if the application firmware includes a check to see if any sensor is out of range. The temperature is maintained by a relay switched heater and vent, and a fan which can be speed controlled via a field effect transistor (FET) output. The fan is fitted to the heater, so that it can be used for forced heating or cooling, depending on whether the heater is on. The FET interface allows speed control of the fan by PWM. The system should operate as specified in Table 13.1.

Figure 13.2 shows the interfacing requirements for the application. Four temperature sensors are used to monitor the temperature at different points in the target system. Fortunately, this

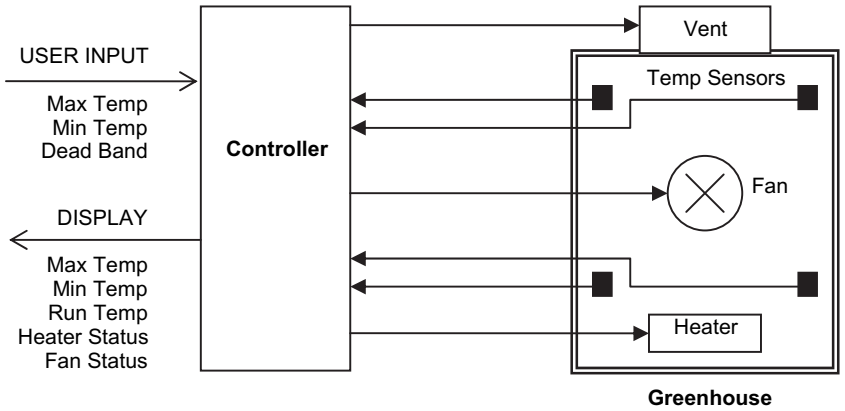


Figure 13.1
Temperature control system

Table 13.1: Temperature controller function table

Measured temperature	Heater	Vent	Fan	Action
Temp << Min	ON	OFF	ON	Forced heating
Temp < Min	ON	OFF	OFF	Heating
Min < Temp < Max	OFF	OFF	OFF	Correct temperature
Temp > Max	OFF	ON	OFF	Cooling
Temp >> Max	OFF	ON	ON	Forced cooling

application does not need much analogue signal conditioning on the input side, other than capacitor decoupling to inhibit noise, and the temperature sensors can be connected directly to the PIC. Their readings can be averaged, or processed with a weighting factor for each, to give a representative value for the measured temperature.

The outputs are controlled via two different interfaces. Relays are used for the heater and vent, assuming that on/off control is suitable. These interfaces are implemented as normally open switched relays, so that an external power supply can be used. The fan output demonstrates the alternative solid-state interface, using a general purpose power FET. This allows proportional control, but the external circuit must be operated at 5 V. This interface would need to be elaborated for a speed-controlled fan in a full-scale system, and the output could be reallocated to one of the PWM outputs on the 16F887.

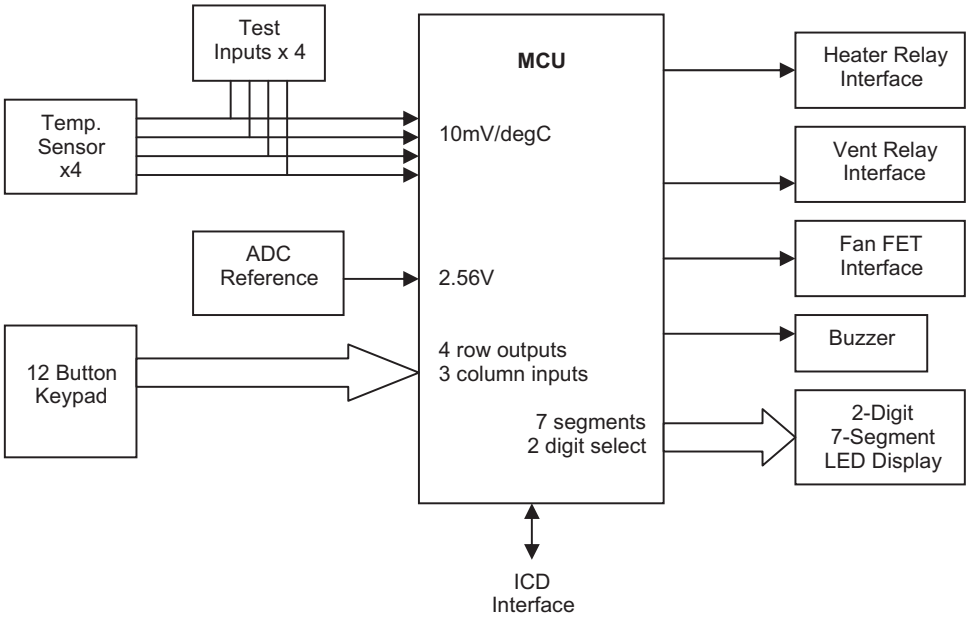


Figure 13.2
Temperature controller interfacing

13.1.2. Input/Output Allocation

The I/O functions provided by the PIC 16F887 are detailed in Table 13.2. These were mapped against the requirements of the application, and the most convenient grouping was decided, giving the I/O allocation in Table 13.3.

13.1.3. Circuit Description

Figure 13.3 shows the schematic for the temperature controller. Each interface will be described separately. The default internal 4 MHz clock is used, to give a convenient instruction execution time, although no timing critical operations are required in this application.

Analogue Inputs

The four temperature sensors are connected to port A, with four pots providing dummy inputs. It is expected that standard calibrated sensors (LM35 or similar) with an output of 10 mV/°C will be used (0°C = 0 mV). The controller is designed to operate at up to 50°C, at which temperature the sensor output is 500 mV. This low voltage is acceptable if the sensors are not connected on long leads, which could pick up electrical interference. For more remote operation, a dc amplifier should be used at the sensor end of the connection to increase the voltage to, say, 5.00 V at 50°C, and the analogue-to-digital converter (ADC) rescaled accordingly. Screened leads should also be used. The inputs are protected from noise and

Table 13.2: 16F887 pin functions (40-pin DIP package)

Pin No.	Pin Label	
12/31	V_{SS}	Ground (0 V)
11/32	V_{DD}	Positive supply (+5 V nominal)
		PORT A (8 BITS)
2	RA0/AN0/ULPWU/C12IN0-	Digital I/O or analogue input or wake up input or comparator input
3	RA1/AN1/C12IN-	Digital I/O or analogue input or comparator input
4	RA2/AN2/Vref-/Cvref/C2IN+	Digital I/O or analogue input or negative reference voltage for ADC or comparator 2 input or reference output
5	RA3/AN3/Vref+/C1IN+	Digital I/O or analogue input or positive reference voltage for ADC or comparator input
6	RA4/T0CKI/C1OUT	Digital I/O or input to timer 0 or comparator 1 output
7	RA5/AN4/SS/C2OUT	Digital I/O or analogue input or slave select input (SPI mode) or comparator 2 output
14	RA6/OSC2/CLKOUT	Digital I/O or external clock circuit or instruction clock output
13	RA7/OSC1/CLKIN	Digital I/O or external clock circuit or clock input
		PORT B (8 BITS)
33	RB0/AN12/INT	Digital I/O (interrupt on change) or analogue input or external interrupt input
34	RB1/AN10/C12IN3-	Digital I/O (interrupt on change) or analogue input or comparator input
35	RB2/AN8/P1B	Digital I/O (interrupt on change) or analogue input or PWM output
36	RB3/AN9/PGM/CI2IN2-	Digital I/O (interrupt on change) or analogue input or programming mode or comparator 1 input
37	RB4/AN11/P1D	Digital I/O (interrupt on change) or analogue input or PWM output
38	RB5/AN13/T1G	Digital I/O (interrupt on change) or analogue input or timer 1 gate input
39	RB6/ICSPCLK	Digital I/O (interrupt on change) or in-circuit serial programming clock input
40	RB7/ICSPDAT	Digital I/O (interrupt on change) or in-circuit serial programming data input
		PORT C (8 BITS)
15	RC0/T1OSO/T1CKI	Digital I/O or timer 1 oscillator output or timer 1 clock input
16	RC1/T1OSI/CCP2	Digital I/O or timer 1 oscillator input or capture 2 input or compare 2 output or PWM2 output
17	RC2/P1A/CCP1	Digital I/O or capture 1 input or compare 1 output or PWM1 output
18	RC3/SCK/SCL	Digital I/O or synchronous serial clock input or output in SPI and I^2C modes
23	RC4/SDI/SDA	Digital I/O or SPI data input or 1^2C data I/O
25	RC5/SDO	Digital I/O or SPI data output

(*Continued*)

Table 13.2: Continued

Pin No.	Pin Label	
26	RC6/TX/CK	Digital I/O or USART asynchronous transmit or USART synchronous clock output
27	RC7/RX/DT	Digital I/O or USART asynchronous receive or USART synchronous data output
		PORT D (8 BITS)
19	RD0	Digital I/O
20	RD1	Digital I/O
21	RD2	Digital I/O
22	RD3	Digital I/O
27	RD4	Digital I/O
28	RD5/P1B	Digital I/O or PWM output
29	RD6/P1C	Digital I/O or PWM output
30	RD7/P1D	Digital I/O or PWM output
		PORT E (4 BITS)
8	RE0/AN5	Digital I/O or analogue input
9	RE1/AN6	Digital I/O or analogue input
10	RE2/AN7	Digital I/O or analogue input
1	RE3/MCLR/Vpp	Digital I/O or master clear input or programming voltage input

overvoltage by a 1k/1nF low pass filter; the input impedance at the ADC is high enough for this to have a negligible effect on the input voltage measurement.

The ADC normally operates at 10-bit resolution, giving output values in the range 0–1023. It needs reference voltages to set the maximum and minimum values for the input conversion. These can be provided internally as V_{DD} and V_{SS} (supply values), but V_{DD} does not give a convenient conversion factor. Therefore, an external reference value was provided from a

Table 13.3: Temperature controller I/O allocation

Device	Function	16F887 Pin	Initialization
Temperature sensors	0–512 mV = 0–51.2°C	RA0, RA1, RA2, RA5	AN0, AN1, AN2, AN4
ADC reference voltage	2.048 V	RA3	VREF+
Heater	Switched output	RE0	Digital output
Vent	Switched output	RE1	Digital output
Fan	Switched output	RE2	Digital output
4 × 3 keypad	Read column	RD0, RD1, RD2	Digital output
	Scan row	RD3, RD4, RD5, RD6	Digital input
2 × 7-segment display	Segments	RC1–RC7	Digital output
	Digit select	RB1, RB2	Digital output
Buzzer	Audio alarm	RB0	Digital output
ICPD interface	Program & debug	RB3, RB6, RB7	N/A

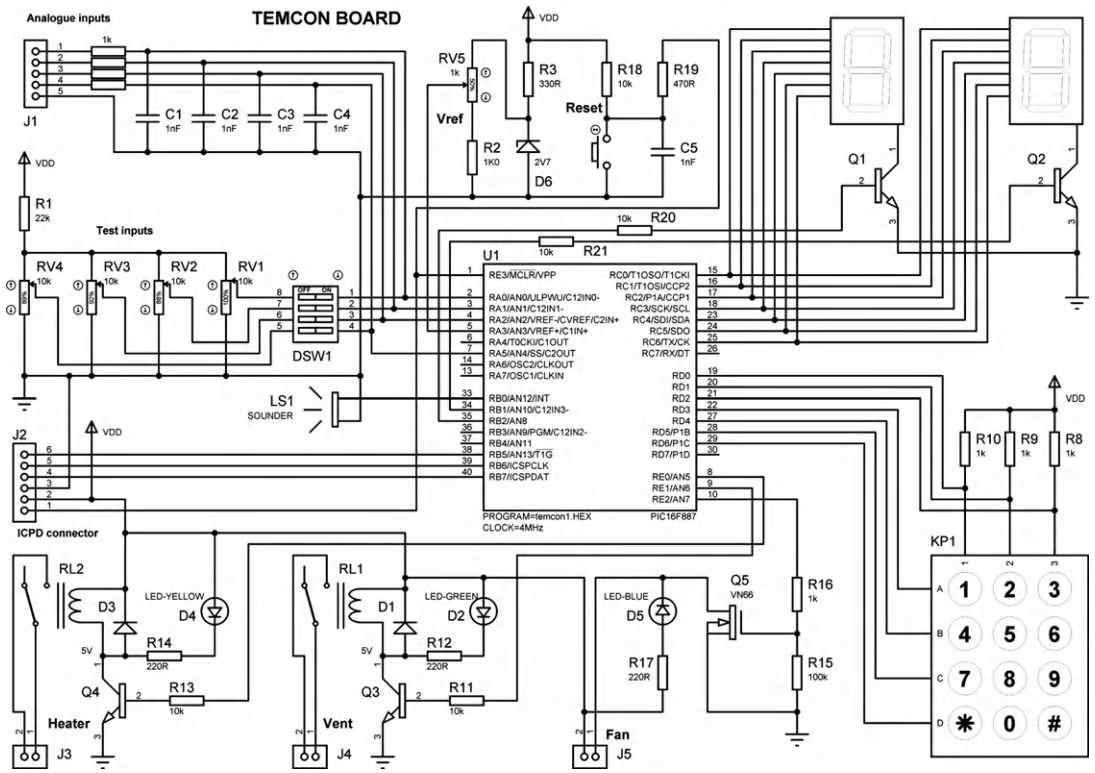


Figure 13.3
TEMCON2 temperature controller schematic

2.7 V Zener diode and potential divider, giving V_{ref+} adjusted to 2.048 V. This gives a conversion factor of 2048/1024 = 2 mV per bit. To simplify the software, and to cover the correct range, only the low 8 bits of the ADC result will be used in the test program, with a maximum value of 255. At 50°C, the input will be 500 mV/2 mV = 250, giving a resolution of 0.2°C per bit. The test pots allow the input to set manually, to check the operation of the software without having to heat and cool the target system. These can be switched in and out as required via a bank of dual in-line package (DIP) switches.

Outputs

Two types of output device are provided: relay and FET. The relay allows the load circuit to be isolated (electrically separate) from the controller. The external circuit operates with its own supply, so the load (heater in this case) can be powered from a single or three-phase supply. The relay is easy to use, but its changeover is relatively slow, and the contacts may wear out in time. The FET interface is more reliable, as it is solid state. The disadvantage in this design is that the load has to operate from the same supply as the FET, the 5 V board supply. It also does not

provide full electrical isolation between the controller and the load. However, the FET interface can be switched at high frequency, allowing PWM speed control, and could be modified for isolated operation. All the outputs include an on-board status light-emitting diode (LED).

Keypad

The 12-button keypad allows the user to input the required temperature and other operating parameters as required by the application program. The target temperature, upper and lower limits, alarm levels and so on would be input as two digits. The keypad is simply a set of switches connected in a row and column arrangement, and accessed by a scanning routine. If the row inputs (A, B, C, D) are all set high initially, and no button is pressed, all the column outputs (1, 2, 3) will be high (pulled up to 5 V). A '0' is then output on each row in turn, and, if a button is pressed, that '0' will appear at the column output and can be detected by the MCU. The combination of the active row and column identifies the key.

Display

A seven-segment display is used, as it is relatively easy to drive compared with the liquid crystal display (LCD), and is self-illuminating. The encoding has been covered in Chapter 10; a look-up table (Table 10.2) provides the output combination for each displayed digit. In this case, two digits are required, but they can both be operated from the same set of outputs by multiplexing. The digits are switched on alternately via Q1 and Q2, at a rate that is fast enough for the digits to appear to be on at the same time, albeit at reduced brightness. Since this effectively halves the average current, the current limiting resistors otherwise needed in the display outputs are unnecessary. Since the switching transistor is acting as a constant current source, and this current will be shared among those segments that are lit, there may be some variation in brightness depending on the digit displayed. This could be improved by using a constant voltage source to control the common terminal current.

Other Interfaces

A buzzer is fitted to provide an audible alarm output. This can be used to signal system failure or, for example, the temperature being too low for too long. Audible feedback from keystrokes is also desirable. A manual reset is provided, so that the program can be restarted without powering down. This will be useful for testing as well as in normal operation. In-circuit programming and debugging are provided via the ICPD connector. An ICD module must be connected between the host PC and the application board.

13.1.4. Hardware Development

The circuit was developed using Labcenter™ ISIS schematic capture software, a component of Proteus VSM, which provides animated drawing objects for integrated software and hardware testing (see Appendix E for details). When the circuit had been tested by simulation,

a stripboard implementation was devised (Figure 13.4). This layout was designed for an earlier version of the board using the pin-compatible PIC 16F877A chip, with a common anode display, so slight modifications would be needed to implement the TEMCON2 circuit in Figure 13.3.

A demo target system was constructed of this original design, with two filament lamps as the heaters, operating from a high-current 5 V supply. A 5 V central processing unit (CPU) fan was fitted as the cooling element, and the temperature sensors were arranged symmetrically inside the enclosure. The wiring of the target hardware is shown in Figure 13.5. Note that there is a sensor output on the fan that could be used to monitor the actual fan speed, if a suitable interface were added to convert the fan sensor pulse to TTL (transistor−transistor logic) levels. The vent was not physically implemented in this test hardware.

The photograph of the finished system (Figure 13.6) shows the simulator at the right of the picture, with the ICD module (enclosed in ABS box) connected to the ICPD connector on the TEMCON board. Five-volt power supplies and a host PC would complete the system. When final hardware testing was completed, an application board was created using Labcenter ARES™ printed circuit board (PCB) layout software, shown in Figure 13.7. This incorporated an on-board +5 V supply for operation from a mains adapter.

13.1.5. Temperature Controller Test Program

Program 13.1 was written to exercise the hardware and to get started in developing applications for the TEMCON2 system, using hardware built to this design or the simulation download.

The program will read in the analogue inputs (select the test inputs by setting the dual in-line (DIL) switches on) and display the raw data on the displays. Pressing a key on the keypad will select an analogue input for display, key ‘1’ for input 1, and so on to ‘4’, then repeating for keys 5−8. An apparently random pattern results, which changes if the test pots are varied, indicating that the hardware input and display interfaces are working. Key 9 will sound the buzzer, while ‘*’,‘0’ and ‘#’ will operate the heater, vent and fan, respectively. A full header has been included with as much information as possible: details of the target system, program description, register initialization, port allocation and so on.

The routine to read in an analogue input is based on the model routine provided in the data sheet, with 20 µs settling time. The conversion is started by setting the GO bit in the ADC control register, and then waiting for it to be cleared by the ADC to indicate that the conversion is complete. In this program, only 8 of the 10 bits of the ADC result are used, so the result is ‘right justified’ to place the least significant 8 bits in the ADRESL register for output to the display. Note that ADRESL is in bank 1 in the 16F887, requiring this bank to be selected and deselected as necessary. In a working program, the analogue input value would be converted into a two-digit decimal value for the display. Using the conversion scaling calculated above,

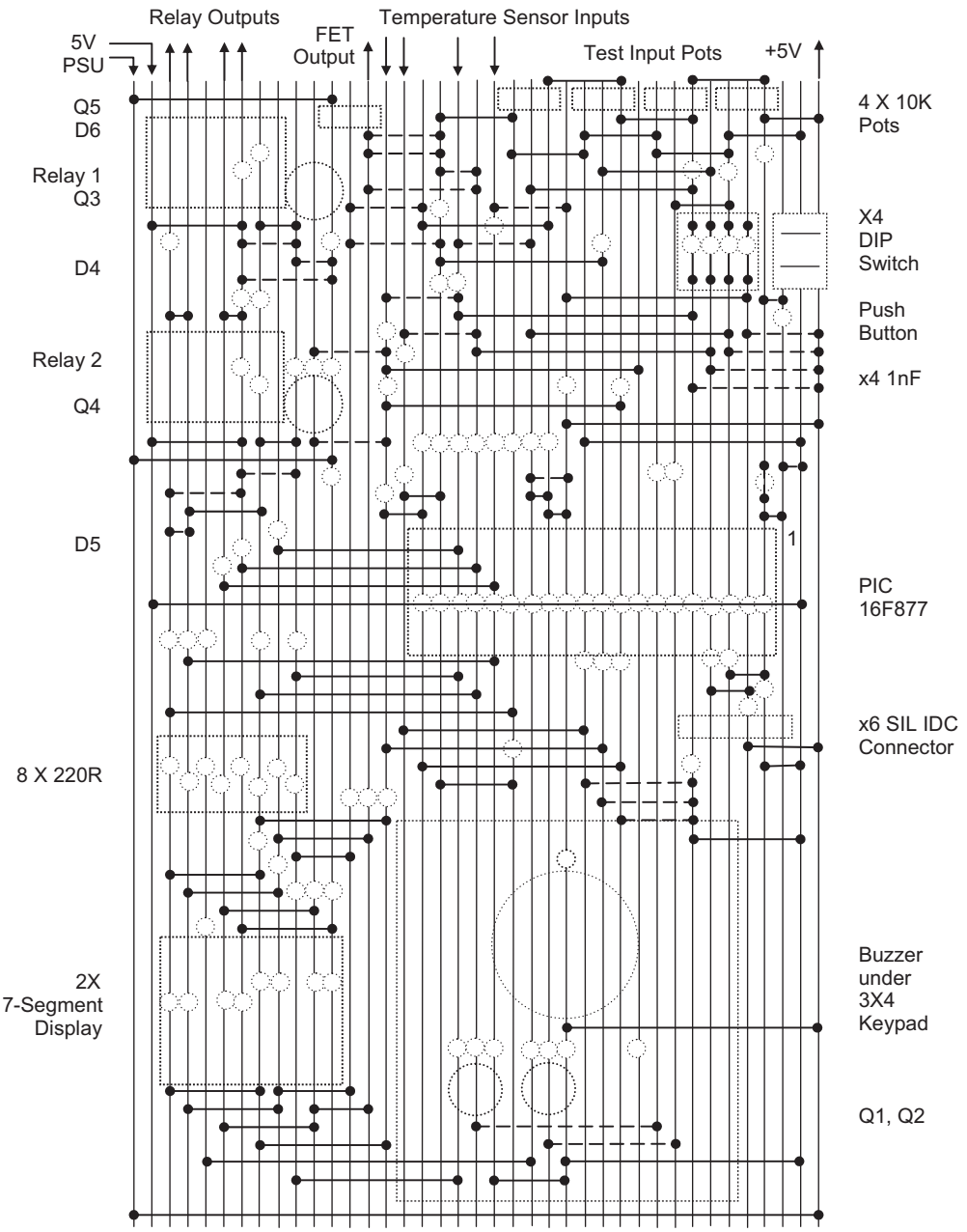


Figure 13.4
Stripboard layout for original TEMCON board

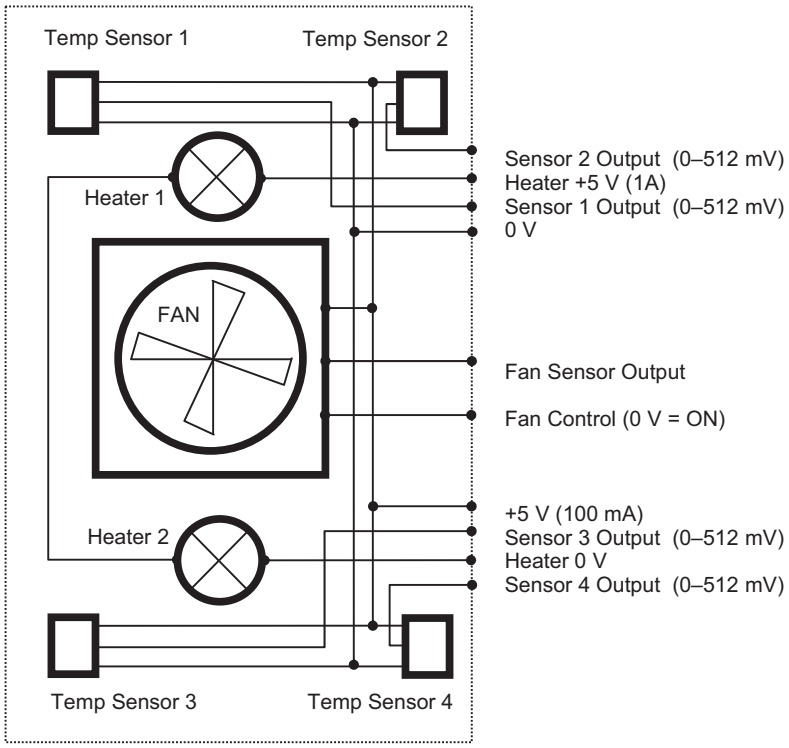
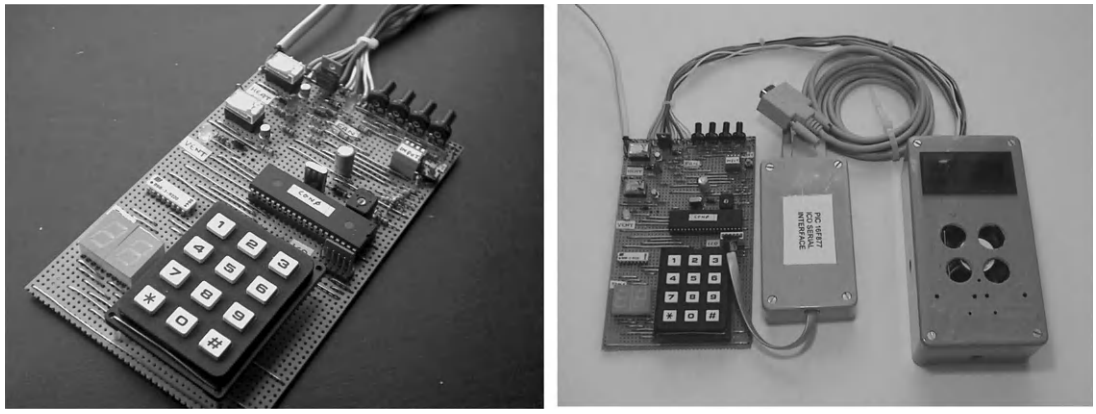


Figure 13.5
Greenhouse simulator wiring



(a) (b)

Figure 13.6
TEMCON temperature controller hardware: (a) stripboard version of TEMCON board; (b) TEMCON board with ICD module and dummy load

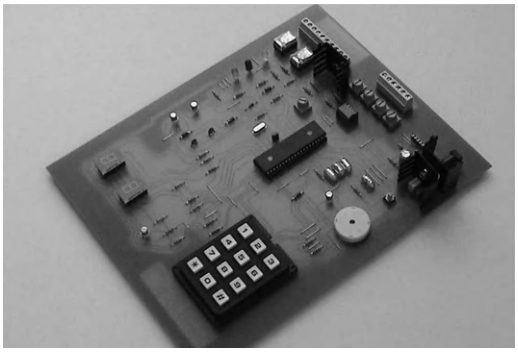


Figure 13.7
TEMCON temperature controller board

a temperature of 50°C would give a result of 250 (in binary) in ADRESL, with the result right justified. Only a quarter of the ADC range is then being used, because the input range is only 500 mV. This result can then be converted into the corresponding display digits '5' and '0', and so on down to zero.

The keyboard scanning routine uses a simple method to check if each key in each row has been pressed, then calls the required action. A more elegant and compact keyboard scanning method is possible when reading in numerical values. A full working program would allow the user to enter the maximum and minimum values for the target temperature, and then go into run mode, where the temperature would be controlled within the set range by operation of the heater, vent and fan. An outline of this application is shown in Program 13.2.

13.1.6. Application Enhancements

It is always useful to consider how an application design could be improved, even if it is ultimately decided that the design effort or extra hardware costs are not justified by the improved performance. As mentioned above, the PWM module could be used to control the speed of the fan. The display could be upgraded to an alphanumeric LCD, so that more information could be displayed and the operating parameters shown to a greater degree of precision. A serial communication port could send the temperature data to a master controller, and receive new operating parameters. If a PC were acting as the host, an external USART/USB converter would be needed. The PC could display the operating data, perhaps in graphical form, or as a plot of temperature variation over time. This data could then be saved on disk, and sent via a network to a supervisory system.

Other references on interfacing which cover the range of techniques needed for input and output in more detail include *Interfacing PIC Microcontrollers: Embedded Design on Interactive Simulation* by the author (Newnes 2006).

```
;*****************************************************************
;          Source File:          TEMCON1.ASM
;          Design & Code:        MPB
;          Date:                 13-1-11
;          Version:              1.0
;
;          Target Hardware:      TEMCON Board
;          Simulation:           Proteus VSM Ver7.7
;          ISIS Design File:     TEMCON.DSN
;
;*****************************************************************
;
;          Test program for PIC887 Controller Board
;
;          Circuit description:
;          PIC 16F887 flash microcontroller receives 4 analogue
;          inputs from temperature sensors (or test input pots)
;          to control Heater, Vent and Fan in a target system.
;          Target temp will be set up using keypad input and
;          displayed on 2-digit multiplexed LED display.
;
;          Test program:
;          Checks all inputs and outputs for correct operation
;          - Press keypad buttons 1-4 to display input pots
;          - Buttons 5-8 ditto
;          - Button 9 to sound buzzer
;          - Button * to operate HEATER output
;          - Button 0 to operate VENT output
;          - Button # to operate FAN output
;
;          Configuration:
;          -Internal clock mode (4MHz, 1us per instruction)
;          -Power-up timer enabled
;          -Watchdog timer disabled
;          -Code Protection off
;
; I/O ALLOCATION  ************************************************
;
;          Analogue temp sensors input (10mV/degC)   RA0,RA1,RA2,RA5
;          ADC reference volts input (2.048V)        RA3
;          Buzzer output                             RB0 = toggle
;          7-segment display    Select lo digit      RB1 = 1
;                               Select hi digit      RB2 = 1
;                               Segments             RC1 - RC7 = 1
;          Keypad column detect input (active low)   RD0 - RD2
;          Keypad row select output                  RD3 - RD6 = 0
;          Relay interfaces     Heater               RE0 = 1
;                               Vent                 RE1 = 1
;          FET interface        Fan                  RE2 = 1
;
;          Data direction codes:
;          TRISA = 11111111
;          TRISB = 11111000
;          TRISC = 00000001
;          TRISD = 00000111
;          TRISE = 00000000
;
```

Program 13.1
Test program for TEMCON board

```
;         RB3, RB6, RB7 reserved for ICD operation
;
; ADC SETUP ************************************************
;
;         ADCON0     Bits 76    01 = A/Dclock = f/8)
;                    Bits 5432  Channel Select (AN0 - AN13)
;                    Bit  2     Go=1 / Done=0
;                    Bit  0     A/D module enable = 1
;         ADCON0 = 01xxx001 depending on channel required
;
;         ADCON1     Bit  7     1 = RIGHT justify result
;                    Bits 3210  0010 = RA0-RA5 analogue
;                                      RE0-RE2 digital
;         ADCON1 = 00000010
;
; ASSEMBLER DIRECTIVES *************************************
;
          PROCESSOR 16F887               ; Select processor
          INCLUDE "P16F887.INC"          ; Include file
;         0x00E4 = CONFIG CODE 1
;         0XFFFF = CONFIG CODE 2
;         ..................................................
;
count     EQU       020        ; assign GPR1 for counter
;
;         ..................................................
;
;         Set origin at address 000:
          org       0x000
;
; START PROGRAM ********************************************

          nop       ; No op. required at 000 for ICD mode

; Initialise control registers ..................................

          banksel   ANSEL                ; Select Bank 3
          movlw     b'00011111'          ; Set Port A anlogue
          movwf     ANSEL                ; ..& Port E digital
          clrf      ANSELH               ; Set Port B digital

          banksel   TRISB                ; Select Bank 1
          movlw     b'11111000'          ; Set up..
          movwf     PORTB                ; display select
          clrf      TRISC                ; display outputs
          movlw     b'10000111'          ; Set up..
          movwf     PORTD                ; keypad I/O
          clrf      PORTE                ; relay/FET outputs
          movlw     B'10000000'          ; A/D right justified,
          movwf     ADCON1               ; ..& internal ref
          movlw     0FF
          movwf     ADRESL

          banksel PORTC                  ; Select bank 0
          clrf      PORTC                ; Clear outputs
          movlw     B'01000001'          ; A/D Fosc/8, AN0,
          movwf     ADCON0               ; ..& enable
```

Program 13.1: (*continued*)

```
; Initialise outputs     ....................................

          banksel   PORTA               ; select bank 0
          clrf      PORTB               ; switch off outputs
          clrf      PORTD
          clrf      PORTE
          movlw     0FF                 ;
          movwf     PORTC               ; all segments on
          goto      start               ; jump over subroutines

; Subroutine to wait about 0.8 ms ..............................

del8      clrf      count               ; Load time delay
again     decfsz    count               ;
          goto      again               ;
          return                        ;

; Subroutine to get analogue input .............................

; Wait 20us ADC aquisition settling time ..

getAD     movlw     007                 ; 3us per loop
          movwf     count               ;
down      decfsz    count               ;
          goto      down                ;

; Get analogue input ..

          bsf       ADCON0,GO           ; Start A/D conversion
wait      btfsc     ADCON0,GO           ; Wait to complete
          goto      wait                ; by testing GO/DONE bit
          return                        ; with result in ADRESL

; Subroutine to show ADC result ................................

show      nop
          banksel   ADRESL
          movf      ADRESL,W            ; move ADC result
          banksel   PORTC
          movwf     PORTC               ; to display
          return

; Subroutines to process keys ..................................

proc1     movlw     b'01000001'         ; Select channel 0
          movwf     ADCON0              ; and
          call      getAD               ; and get analogue input
          return                        ; for next key

proc2     movlw     b'01000101'         ; Select channel 1
          movwf     ADCON0              ; and
          call      getAD               ; and get analogue input
          return                        ; for next key

proc3     movlw     b'01001001'         ; Select channel 2
          movwf     ADCON0              ; and
          call      getAD               ; and get analogue input
          return                        ; for next key

proc4     movlw     b'01010001'         ; Select channel 4
          movwf     ADCON0              ; and
          call      getAD               ; and get analogue input
          return                        ; for next key
```

Program 13.1: (*continued*)

```
proc9     bsf       PORTB,0           ; Toggle buzzer on
          call      del8              ; delay about 0.8ms
          bcf       PORTB,0           ; Toggle buzzer off
          call      del8              ; delay about 0.8ms
          return                      ; for next key

procs     bsf PORTE,0                 ; switch on heater
wait0     btfss PORTD,0               ; * button released?
          goto wait0                  ; no, wait
          return                      ;

proc0     bsf PORTE,1                 ; switch on vent output
wait1     btfss PORTD,1               ; * button released?
          goto wait1                  ; no, wait
          return                      ;

proch     bsf PORTE,2                 ; switch on fan output
wait2     btfss PORTD,2               ; * button released?
          goto wait2                  ; no, wait
          return                      ;

; Routine to scan keyboard ....................................

scan      movlw     0FF               ; Deselect...
          movwf     PORTD             ; ...all rows on keypad

; scan row A of keypad ..........

          bcf       PORTD,3           ; select row A

          btfsc     PORTD,0           ; test key 1
          goto      key2              ; next if not pressed
          call      proc1             ; process key 1

key2      btfsc     PORTD,1           ; test key 2
          goto      key3              ; next if not pressed
          call      proc2             ; process key 2

key3      btfsc     PORTD,2           ; test key 2
          goto      key4              ; next if not pressed
          call      proc3             ; process key 3

; scan row B of keypad ..........

key4      bsf       PORTD,3           ; deselect row A
          bcf       PORTD,4           ; select row B

          btfsc     PORTD,0           ; test key 4
          goto      key5              ; next if not pressed
          call      proc4             ; process key 4

key5      btfsc     PORTD,1           ; test key 5
          goto      key6              ; next if not pressed
          call      proc1             ; process key 5

key6      btfsc     PORTD,2           ; test key 6
          goto      key7              ; next if not pressed
          call      proc2             ; process key 6

; scan row C of keypad ..........

key7      bsf       PORTD,4           ; deselect row B
          bcf       PORTD,5           ; select row C
```

Program 13.1: (*continued*)

```
          btfsc     PORTD,0             ; test key 4
          goto      key8                ; next if not pressed
          call      proc3               ; process key 4

key8      btfsc     PORTD,1             ; test key 5
          goto      key9                ; next if not pressed
          call      proc4               ; process key 2

key9      btfsc     PORTD,2             ; test key 2
          goto      keys                ; next if not pressed
          call      proc9               ; process key 3

; scan row D of keypad ..........

keys      bsf       PORTD,5             ; deselect row C
          bcf       PORTD,6             ; select row D

          btfsc     PORTD,0             ; test key *
          goto      key0                ; next if not pressed
          call      procs               ; process key *

key0      btfsc     PORTD,1             ; test key 0
          goto      keyh                ; next if not pressed
          call      proc0               ; process key 0

keyh      btfsc     PORTD,2             ; test key #
          goto      done                ; next if not pressed
          call      proch               ; process key #

; all done ......................

done      bsf       PORTD,6             ; deselect row D
          clrf      PORTE               ; clear outputs
          return                        ; to main loop


; Main program ***************************************************

start     bcf       PORTB,2             ; switch off high digit
          bsf       PORTB,1             ; and low digit on
          call      scan                ; and read keypad
          call      show                ; and display

          bcf       PORTB,1             ; switch off low digit
          bsf       PORTB,2             ; and high digit on
          call      scan                ; and read keypad
          call      show                ; and display

          goto      start               ; repeat main loop

          END                           ; of source code .........
```

Program 13.1: (*continued*)

```
TEMCONAPP1
      Initialise
            Ports
                  Port A = Temp sensor inputs(4)
                  Port B = Display digit select(2),ICP/D(3)
                  Port C = Display segments(7)
                  Port D = Keypad(4 outputs, 3 inputs)
                  Port E = Heater, Vent, Fan outputs

            ADC   Right justify, 4 channels
                  ADC frequency Fosc/8, select input AN0


      GetMaxMin

            Scan keyboard
            Store & display first digit of maxtemp
            Scan keyboard
            Store & display second digit of maxtemp
            Convert to byte MaxTemp (0-200)

            Scan keyboard
            Store & display first digit of mintemp
            Scan keyboard
            Store & display second digit of mintemp
            Convert to byte MinTemp

      Cycle
            Read tempsensor1 AN0
            Read tempsensor2 AN1
            Read tempsensor3 AN2
            Read tempsensor4 AN5

            IF sensor out of range
                  replace with previous value
            Calculate AverageTemp

      Display AverageTemp
            MSD = AverageTemp/10
            Get 7-seg code & display MSD
            LSD = Remainder
            GEt 7-seg code & display LSD

            IF AverageTemp > Mintemp
                  switch heater OFF
                  ELSE switch heater ON
            IF AverageTemp > Maxtemp
                  switch vent ON
                  ELSE switch vent OFF
            IF AverageTemp > Maxtemp + 4
                  switch fan ON
                  ELSE switch fan OFF

      GOTO Cycle
```

Program 13.2
Outline of temperature controller application

RA2/AN2/Vref-	1	18	RA1/AN1
RA3/AN3/Vref+	2	17	RA0/AN0
RA4/AN4/T0CKI	3	16	RA7/OSC/CLKI
RA5/!MCLR/Vpp	4	15	RA6/OSC2/CLKO
Vss	5	14	Vdd
RB0/INT	6	13	RB7/T1OSI/PGD
RB1/SDO/CCP1	7	12	RB6/T1OSOT1CKI/PGC
RB2/SDO/CCP1	8	11	RB5/!SS
RB3/CCP1/PGM	9	10	RB4/SCK/SCL

Figure 13.8
PIC 16F818 pin-out

13.2. Simplified Temperature Controllers

We will now briefly look at using a couple of other chips to create simplified versions of the temperature controller.

13.2.1. 16F818 Temperature Controller

The PIC 16F818 is a replacement part for the 16F84A. It has a compatible pin-out (Figure 13.8), and additional features at a lower cost. Sixteen I/O pins are available, including five analogue inputs. It has 1k words of program memory; if extra memory is needed, the 16F819 has the same features but 2k program memory. As usual, each pin has multiple functions, other than the two supply pins. Analogue inputs can be selected on RA0–RA4, or external reference voltages. There is a capture, compare and PWM (CCP) module and a synchronous serial port offering serial peripheral interface (SPI) or inter-integrated circuit (I^2C) modes. Other special features are a variety of power-saving modes in addition to the usual 'sleep', an internal oscillator which obviates the need for external clock components, and in-circuit programming and debugging.

This chip can be used in the temperature controller if the keyboard is eliminated, and the set temperature is input from a pot via one of the analogue inputs (Figure 13.9). A fixed control range might be acceptable, or other analogue inputs assigned for setting maximum and

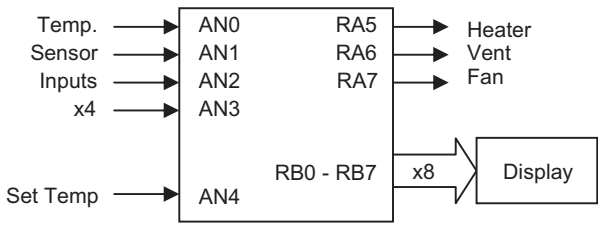


Figure 13.9
16F818 temperature control block diagram

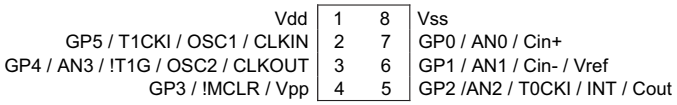


Figure 13.10
12F675 pin-out

minimum temperatures. The display digit selection can be reconfigured to use only one output, or binary coded decimal (BCD) displays used which need only four outputs each. The application then only needs 16 I/O pins. Operating data could be transferred via the serial interface if the display is omitted (RB1, RB2 and RB4).

13.2.2. 12F675 Temperature Controller

The 10 and 12 series of PIC mini-chips offer a minimal set of features in eight-pin packages. The pin-out for the 12F675 illustrates the point (Figure 13.10). It can be configured with six plain digital I/O pins, but also offers two timers, an analogue comparator or four analogue input channels. The 12F629 is the same, except that it does not include the ADC and is therefore a little cheaper. An internal oscillator and in-circuit programming are also available.

A temperature controller could be implemented using this chip if only two analogue inputs are used (Figure 13.11). It could operate with a fixed set temperature, or another analogue input could be used as a set temperature input. With no display, a dial on the set temperature pot may be useful.

13.3. PIC C Programming

The 18 series are the most powerful of the 8-bit PIC microcontrollers. The group offers a large selection of different combinations of advanced 8-bit features, and the larger memory size means that 'C' can be used for application programming, instead of assembler. The instruction set of 75 16-bit instructions is designed to support this option. New low-power consumption devices are being added to this range all the time.

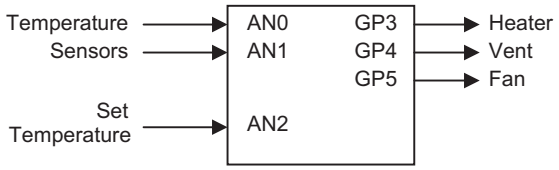


Figure 13.11
12F675 temperature controller

13.3.1. Comparison of 16 and 18 Series PICs

A small selection of the available 18F devices is included in Table 12.2. The architecture is somewhat more complex than the 14-bit devices, with extra blocks for multiplication, a hardware data table access, additional file select registers and other advanced features. However, the data bus is still 8 bits. Taking the 18F4580 as an example, in terms of peripheral features it is comparable to the 16F887 described in Section 13.1, so a comparison of the two devices will be made to illustrate the differences and similarities of the two groups (Table 13.4).

As can be seen, the 18 series device has some advantages: 40 MHz clock rate, 16k program memory and more data memory. However, bear in mind that a program written in C will not be as code efficient as an assembly language equivalent, so these advantages may or may not translate into faster performance, depending on the application and the way that it is structured. The main advantage is that more complex operations, such as mathematical functions, are easier to program in C. For example, the conversion of binary temperature readings to two-digit BCD will be much easier in the temperature controller above. The 18 series PIC has a richer instruction set, including instructions such as multiply, compare and skip, table read, conditional branch and move directly between registers, so still has advantages even when programmed in assembly language.

13.3.2. PIC C Programming

For those readers unfamiliar with C programming, a simple example is shown in Program 13.3. The program will give the same output as BIN1.ASM assembly language program. The program must be converted to PIC 16-bit machine code using the MPLAB C18 Compiler,

Table 13.4: Comparison between the 16F877 and the 18F4580

Feature	16F887	18F4580
Total pins	40	40
Input/output pins	33	36
Ports	A, B, C, D, E	A, B, C, D, E
Clock	20 MHz	40 MHz
Instruction bits	14	16
Program memory (instructions)	8k	16k
Instruction set size	35	75/83
Data memory (bytes)	368	1536
EEPROM (bytes)	256	256
Interrupt sources	14	20
Timers	3	4
Capture, Compare, PWM modules	2	2
Serial communications	MSSP, USART	MSSP, USART, CAN, LIN
Analogue inputs	14 × 10 bits	11 × 10 bits
Resets	POR, BOR	POR, BOR, Stack, Programmed

```
/*      BIN1.C          M Bates       Version 1.0

        Program to output a binary count to Port B LEDs

*************************************************************************/

#include <p18f4580.h>                 /* Include port labels for this chip */
#include <delays.h>

int counter                           /* Label a 16-bit variable location  */

void main(void)                       /* Start main program sequence       */
{
        counter = 0;                  /* Initialise variable value         */
        TRISB = 0;                    /* Configure Port B for output       */

        while (1)                     /* Start an endless loop             */
        {
                PORTB = counter;      /* Output value of the variable      */
                counter++;            /* Increment the variable value      */
                Delay10KTCY(100);     /* Wait for 100 x 10,000 cycles      */
        }
}                                     /* End of program                    */
```

Program 13.3
A simple PIC 'C' program

which is supplied as an add-on to the development system. This compiler recognizes ANSI (American National Standards Institute) C, the standard syntax for microcontrollers. The C compiler must be selected in the development mode dialogue when building the application.

The main elements of the program are as follows:

`/* comment */`

Comments in C source code are enclosed between /* and */ and can be run over several lines. A semicolon is used in assembler.

`#include<p18f4580.h>`

The 'include' is a compiler directive that calls up a header file named 'p18f4580.h', which has the same function as the include file in assembler, in that it contains predefined register labels for that particular processor and the corresponding addresses.

`int counter`

This assigns a label to a register and declares that it will store an integer, or whole number. A standard integer in C is stored as a 16-bit number, requiring two data random access memory (RAM) general purpose register (GPR) locations. EQU provides the equivalent operation in assembler.

```
void main(void)
```

This rather peculiar syntax simply indicates, as far as we are concerned here, the start of the main program sequence. The following brace (curly bracket) starts the main program block, with a corresponding brace at the end. These are lined up in the same column and the main program tabbed in between them, so that they can be matched up correctly.

```
counter = 0;
```

A value of 0 is initially placed in the variable location (low byte). The equivalent in assembler is MOVLW, followed by MOVWF.

```
TRISB = 0;
```

A value 0 is loaded into the data direction register of port B to initialize the port bits for output to the LEDs.

```
while(1)
```

This starts a loop, which will run endlessly. A condition is placed in the brackets that control the loop. For example, the statement could read 'while(count<256)', in which case the following group of statements within the curly brackets (braces) would execute 255 times, counting up to the maximum binary value and stopping. The value 1 means the condition is 'always true', so the loop is endless, until reset. This translates into GOTO in assembler, with DECFSZ providing the conditional test.

```
PORTB = counter;
```

The value in counter is copied to port B data register for display on the LEDs (assembler equivalent: MOVxx).

```
counter++;
```

The variable value is incremented each time the loop is executed. This causes the output to be incremented the next time (assembler equivalent: INCF).

```
Delay10KTCY(100);
```

This calls a predefined block of code, which provides a delay, so that the LED output changes are visible. At a maximum clock rate, the processor instruction cycle time is 0.1 μs, so the delay works out to 0.1 s (10 000 × 100 cycles). The overall count cycle will then take 25.6 s. The delay function is an example of a function call, corresponding to a subroutine in assembler. We know that this is implemented in assembler as a software delay loop or hardware timer operation.

The layout of the program, with tabs, is important for understanding the program and checking the syntax if there are logical errors. However, the layout does not affect the program function,

only the sequence of characters. Nevertheless, the statements must be all on one line; line returns are not allowed within a statement.

Each complete statement is terminated with a semicolon. Note that some are not complete in themselves, and do not have a semicolon. For example, 'while(1)' is not complete without the loop statements, or at least the pair of braces. The close brace terminates the 'while' statement. The whole of the main loop, and any functional subblock, must be enclosed between braces.

13.3.3. Advantages of C Programming

The C compiler converts the program into PIC 16-bit machine code. Most of these C statements translate into more than one machine code instruction. This can be confirmed by studying the list file, which is produced by disassembling the machine code.

The pseudocode for the temperature controller above (Program 13.2) can probably be more easily translated into C than assembly language. For example, the conditional control operations defined using IF…THEN statements will translate directly, whereas, in assembler, they have to be implemented by suitable combinations of 'Bit Test and Skip' with 'Goto' or 'Call'. The comparison of the 'average temperature' with the set values can be done in one statement in C, but in assembler needs a subtract or compare prior to a bit test, which is much more complicated. On the other hand, checking bit inputs is not as easy in C as in assembler, as ANSI C contains no individual bit operations. Bit status in a register has to be checked by using a logical or numerical range check.

There are many references on C programming. To program a microcontroller in C, only the basic set of statements and simple data structures will probably be needed, so if the reader has some knowledge of C already, using it to develop PIC applications should not be too difficult. For further details, see *Programming 8-Bit PIC Microcontrollers in C with Interactive Hardware Simulation* by this author (Newnes 2008). This uses the CCS C compiler, which has a complete set of ready-made functions that simplify the C code, especially I/O handling and mathematical functions.

Questions 13

1. (a) What interfacing modifications are recommended for the LM35 temperature sensor if the connections are over 1 m long? (4)
   (b) For the TEMCON2 system, calculate the output of the LM35 sensor at 25°C, and the decimal value that would be found in the ADRESL on completion of an A/D conversion of this input, if the result is right justified. (4)
2. State one advantage of (a) the relay output and (b) the FET output as used in the temperature controller. (4)

3. Describe briefly how a multiplexed seven-segment LED display works, and its advantage in terms of I/O requirements. (4)
4. Suggest two reasons why the PIC 16F818 would be preferred over the 16F84A in a temperature control application. (4)
5. Compare PIC assembler and ANSI C programming, outlining the advantages of each. (5)

Answers on page 425. (Total 25 marks)

Activities 13

1. Devise an alternative keypad scanning routine to that in Program 13.1 using the rotate instruction, such that the binary value for the keys 0—9 are stored in a suitable register.
2. Design and implement the fully functional program for the temperature controller based on the pseudocode provided in Program 13.2. The user will enter an upper and lower temperature limit, and set the controller to run mode, where the outputs are operated to maintain the temperature between those limits. The system should tolerate a fault in one sensor which puts the output outside the normal operating range. Develop a full design and performance specification for the controller. Test by simulation.
3. Design a temperature-controlled enclosure with heaters, fan and vent, which will allow a fully functioning temperature control program to be tested. Investigate the design of an interface for the fan sensor, so that the fan speed could be controlled by PWM with feedback. Investigate the set-up required to use the PWM output of the 16F887, and redesign the hardware to connect the fan to a PWM output.
4. Implement the minimal temperature controller proposed above using the 12F675 chip, operating as specified in Table 13.1. Create a schematic, simulate the application (interactively if possible), design a layout, implement and test.
5. Study relevant C programming references and the Microchip manual 'MPLAB C18 C Compiler, Getting Started', and modify the program BIN1.C such that the output can be stopped, started and reset by push-button inputs at RA0 and RA1. Why is reading the inputs more difficult in C?

CHAPTER 14

More Control Systems

Chapter Outline

Chapter Points

- Alternative microcontroller designs are derived from the Intel 8086 and the Motorola 68000 architectures.
- Simple sequence control can be implemented using electromechanical relays.
- PLCs have built-in interfacing and user-friendly programming methods to simplify sequence control system design.
- The PC is a universal administrative, design, computing, control and networking hardware platform.
- The control system designer needs to select the most suitable technology for any given application.

PIC Microcontrollers. DOI: 10.1016/B978-0-08-096911-4.10014-X

In this chapter, we will look at some other technologies used to build controller systems, in order to put microcontrollers (MCUs) in context and to evaluate different technologies. Other makes of microcontroller will be briefly considered, and their features compared with the PIC®. Conventional microprocessor systems, with separate processor, memory and input/output (I/O) chips, may offer a more effective solution for complex digital systems, although such discrete designs are increasingly rare. The programmable logic controller (PLC) provides a self-contained device, containing a microcontroller with built-in interfacing. It is frequently connected into a system under PC supervision so that control information can be transferred over a network. In this way, complex production systems can be centrally operated and monitored.

14.1. Other Microcontrollers

The PIC currently dominates the 8-bit microcontroller market, but a comparison with other controllers is still useful, particularly as the alternatives are generally based on historically significant conventional architectures using complex instruction sets, which provide a useful contrast with the PIC reduced instruction set computing (RISC) architecture.

14.1.1. Intel® 8051 Microcontroller

The Intel PC architecture has dominated the desktop/laptop computer market for many years, and the first widely used general purpose microcontroller was based on this architecture. First introduced in 1980, the Intel 8051 was derived from the then standard PC microprocessor, the 8086. As can be seen in the block diagram (Figure 14.1), the original design had multiple parallel ports, timers and interrupts, and a serial port. The 8051 can be used as a conventional processor, as well as a microcontroller. It can access external memory using port 0 and port 2, which act as multiplexed data and address lines. Some of port 1 and port 3 pins also have a dual purpose, providing connections to the timers, serial port and interrupts. The program memory was erasable programmable read-only memory (EPROM), which had to be erased under ultraviolet light and reprogrammed out of circuit.

The 8051 had a conventional architecture, where the same data bus was used to transfer the program code and the internal data. This makes it inherently slower than the PIC Harvard architecture, which has a separate program and data paths operating concurrently. The 8051 also had a complex instruction set (CISC), which provided more options when programming, but reduced execution speed.

14.1.2. Atmel® AVR Microcontrollers

European manufacturer Atmel offers a range of CISC microcontrollers derived from the 8051 architecture and instruction set. The AT89 family are updated 8051 type MCUs. The AVR

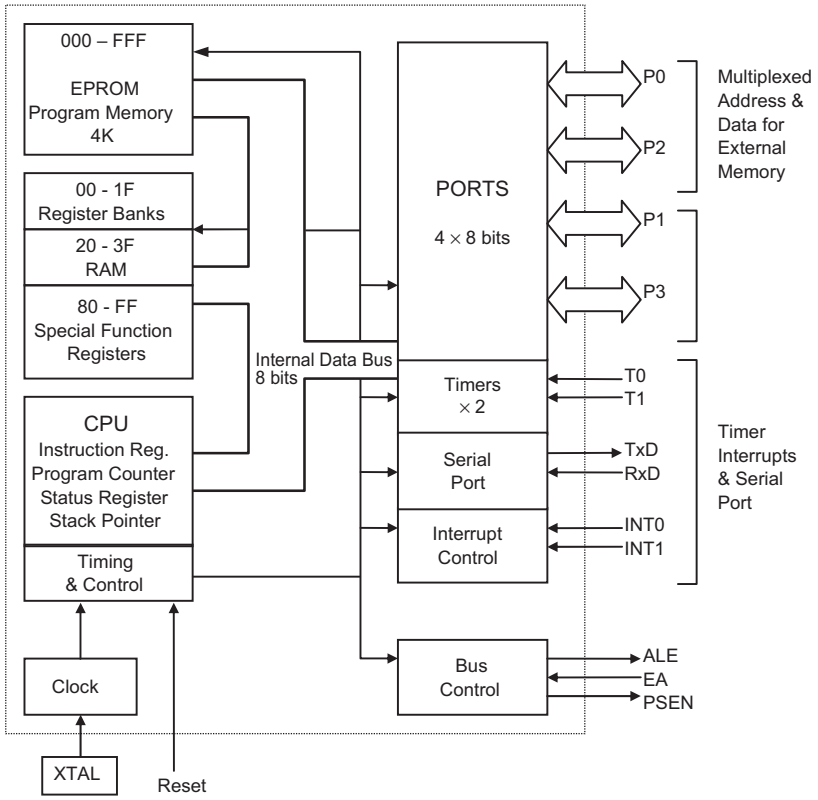


Figure 14.1
Block diagram of Intel 8051 microcontroller

range includes 8-bit ATtiny and ATmega devices, and 32-bit AT32 devices. The AT91SAM group are also 32-bit MCUs, but are based on the high-performance ARM architecture.

The internal architecture of a representative chip, the ATtiny20 MCU, is shown in Figure 14.2. The program execution section is similar to the PIC, in that it has a separate instruction bus (Harvard architecture). It also uses a two-stage pipeline, overlapping the fetch and execution cycles. It can therefore execute instructions in one clock cycle, at a maximum clock rate of 12 MHz. It has a similar range of features to the equivalent PIC, that is, an 8-bit and 16-bit timer, serial ports and eight multiplexed 10-bit analogue-to-digital converter (ADC) inputs.

Unlike the PIC, the AVR chip has 16 general purpose registers that contain the current data, compared with the single working register of the PIC. In addition, it has a separate random access memory (RAM) block for data storage, whereas the PIC has a single integrated RAM block containing special function registers (SFRs) and general purpose registers (GPRs). The timers and other SFRs are addressed explicitly in the instruction set rather than as RAM addresses. The stack is implemented as a selected set of RAM locations, making it more

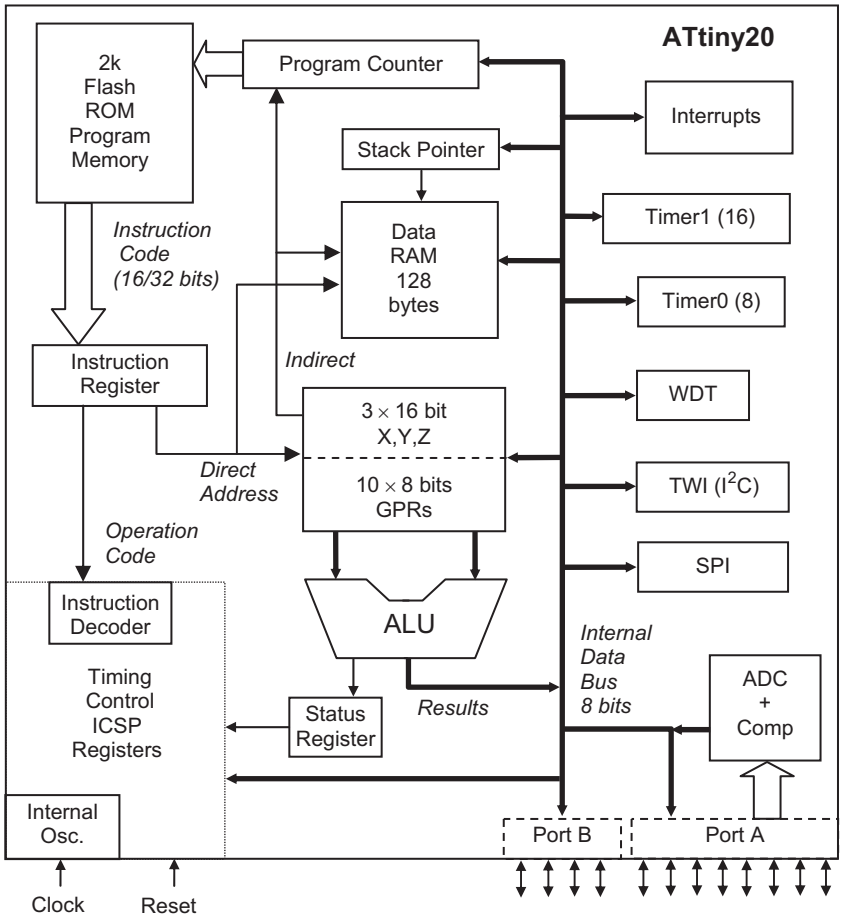


Figure 14.2
AVR internal architecture

flexible in operation but less well protected from corruption by incorrect code. The AVR also incorporates multiple interrupt vectors.

The instruction set is more extensive, comprising 54 instructions with multiple addressing modes. For example, there are several conditional branching instructions, and data movement requires different instructions for load (LD, LDI, LDS), store (ST, STI), move (MOV), input (IN) and output (OUT). This provides some advantages to the experienced programmer who can make best use of the available options, but is more complex to learn initially.

14.1.3. Other Microcontrollers

At the time of writing ST Microelectronics produces a range of microcontrollers with similar features to the PIC16, but with a complex instruction set and conventional architecture.

Freescale Semiconductor Inc. offers a range of microcontrollers based on the architecture and instruction set of the standard Motorola 68000 microprocessor. The current offering concentrates on high-end microcontrollers with 16- and 32-bit cores. Similarly, Texas Instruments Inc. and NXP Semiconductors NV (formerly a division of Philips) offer a power MCU range, including the ARM/CortexM3 32-bit MCUs running at 50 MHz.

14.2. Microprocessor Systems

The main elements of the microcontroller were originally developed as separate devices before being integrated into one chip to produce the microcontroller. The PC, as outlined in Chapter 1, is an example, where the individual central processing unit (CPU), memory and I/O devices are linked together by system address, data and control buses. The M68000 CPU was used in many different microprocessor systems from the 1980s onwards, including home computers, training systems, industrial controllers and instrumentation. It was the first, and most popular, 16-bit microprocessor. The original Apple® Macintosh®, main rival to the Intel-based PC, was designed around it.

The advantage of the conventional microprocessor system is that it can be designed to suit the application more precisely. It includes only those peripherals that are actually needed, and memory capacity as required. Obviously, the system is more complex to design and build, and so this type of system tends to be used for larger applications, where for example, extensive data storage is required. Although now largely obsolete, the 68000 remains a useful example of conventional CISC system architecture, because its regular architecture makes it easier to understand than current microprocessors which have a multi-level bus hierarchy and advanced design features added to the original CPU to improve performance.

14.2.1. M68000 Hardware

A typical development and training system based on the M68000 is shown in Figure 14.3. The target board incorporates separate CPU, EPROM, RAM and port chips. It can be connected to an applications board, which has a range of peripheral transducers, such as switches, light-emitting diodes (LEDs) and a pulse width modulation (PWM)-controlled motor and shaft opto-sensor. This is controlled by the 68000 CPU via a standard 68230 parallel interface/timer (PI/T) chip, which has three 8-bit ports, of which port A provides data transfer and port B the individual control and data lines. The operation of this type of system is described further in Appendix C, Section C.9.

The M68000 target board is shown in Figure 14.4, with a block diagram of the system in Figure 14.5, which can be compared with the PIC internal architecture. Notice that in the PIC MCU block diagram, the internal architecture of the processor is visible, whereas in the

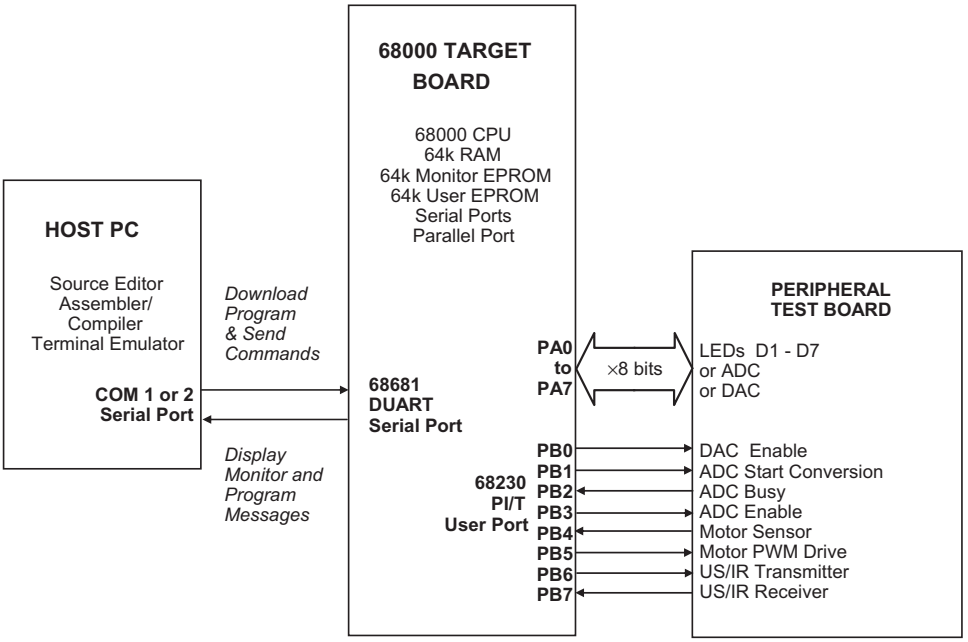


Figure 14.3
M68000 microprocessor demonstration system

68000 system, it is concealed within the CPU. To design a microprocessor system, the CPU signal timing specification must be carefully studied, but this is not necessary in the microcontroller system, a major advantage of designing around an MCU. Another is that the microcontroller can be simulated as a whole, whereas in the microprocessor

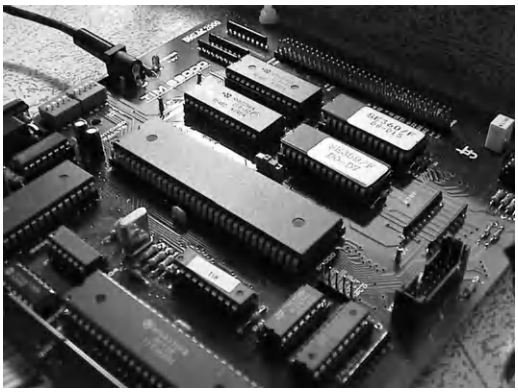


Figure 14.4
Motorola 68000 microprocessor target board

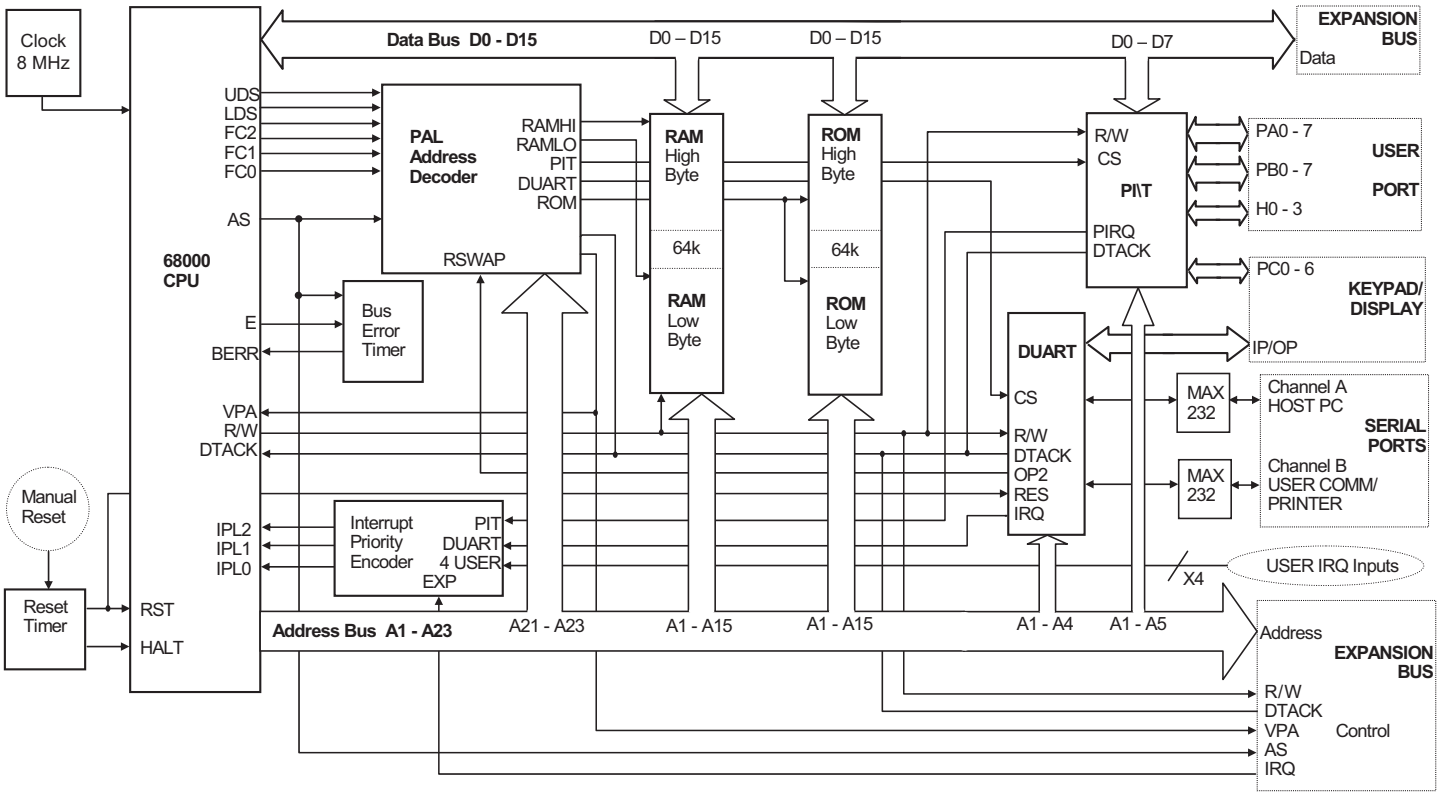


Figure 14.5
Block diagram of M68000 microprocessor board

```
*       OUT3.ASM        MPB     27/8/97
*
*       A demo program using general purpose system
*       initialization file TIM.INI
*
*  ------------------------------------------------------

        use     tim.ini         Initialise system

        move.b  #$ff,DircA      Port A data direction code

again   move.b  d0,PortA        Output data to LEDS
        addq    #1,d0           Increment output value

        move.w  #$0fff,d1       Initialize delay count
delay   subq.w  #1,d1           Decrement and
        bne     delay           Loop until zero

        bra     again           Repeat forever..........
```

Program 14.1
Simple program for 68000 board

system only the CPU can be modeled, unless a system simulator such as Proteus VSM is used.

14.2.2. M68000 Program

A simple program for the 68000 system is shown in Program 14.1, so that the syntax for a complex instruction set (CISC) processor can be compared with PIC assembly language. The program has a similar function to the PIC program BIN2, outputting a binary count to LEDs with a delay. The syntax is analyzed below.

Comments

The comments are delimited with a star.

```
use  tim.ini
```

This is equivalent to the include directive in the PIC – it incorporates a file 'tim.ini' which contains standard register labels, PortA and DircA. Port A is the 8-bit port data register and DircA the data direction register (DDR).

```
move.b  #$ff,DircA
```

Move the literal FF into the DDR to set all bits as output. The '.b' means this is a byte operation (16- and 32-bit words can be moved in the 68000). '#' means this is a literal (immediate data in

68000 speak). '$' indicates a hex number. Note that in the 68000, a '1' in the DDR sets that bit as output – this is the opposite to the PIC.

```
again  move.b  d0,PortA
```

The label 'again' represents a source code line address, 'd0' is the first data register in a set of eight (d0–d7) and PortA is the output register to which the LEDs are connected.

```
addq  #1,d0
```

This means add 1 to (increment) d0. Surprisingly, the 68000 does not have an increment (or decrement) instruction. 'addq' means 'add quick', used for adding a small number to a register.

```
move.w  #$0fff,d1
```

Move a 16-bit word (w) into d1 to initialize the delay loop.

```
delay  subq.w  #1,d1
```

Start of delay loop. Decrement (subtract 1 from) the counter register 'd1'.

```
bne  delay
```

This means 'branch if not zero' to the label delay. The program jumps back and repeats the decrement until the result of the previous operation (decrement) is zero. This is available in the 14-bit PIC only as a pseudo-operation.

```
bra  again
```

This is an unconditional jump equivalent to the GOTO label in PIC programs, to make the program repeat endlessly.

It can be seen that the 68000 syntax is more complex because, first, there are more instructions and, second, there are more registers and addressing modes.

14.3. Control Technologies

Microprocessors and microcontrollers are part of a wide range of technologies used in control systems. These include:

- Electromechanical relays
- Programmable logic controllers
- Microcontroller-based boards
- Dedicated microprocessor designs
- PC-based controllers
- Networked control systems.

To complete an overview of controllers, and to allow comparison with PIC microcontroller-based systems, the essential features of these are outlined below.

14.3.1. Electromechanical Control

A familiar example of an electromechanical sequence controller is found in traditional domestic washing machines (non-digital types). A motorized rotary switch slowly operates multiple contacts in the required sequence to open valves (filling), and switch on motors (washing, spinning and pumping) and heaters. Switched sensors (level, temperature) and safety interlocks (door switch) are connected to the same rotor. In this way, purely electromechanical components can be used to make a robust sequence controller, where the environment is hostile to delicate electronics. However, switches and relays are inherently unreliable because of the moving parts and wear on the contacts due to arcing and mechanical forces.

14.3.2. Relay Control

The relay was the first control system component to be invented, originally to boost telephone signals. It is an electromechanical switching device, which allows a high power load to be controlled by a small input current, using an electromagnetic coil to operate a set of changeover switches. Relays can be wired up to operate in sequence, with time-delayed switching if required, to operate as a process controller. Before the development of transistors and digital logic, even before the development of valves, relays could be used to make simple industrial controllers. For example, a relay can be used to switch on a machine tool, using on and off push buttons and safety interlocks to make its operation safer.

The main components of a relay are shown in Figure 14.6(a). The small input current through the coil creates an electromagnetic field that attracts a steel yoke, which operates a set of contacts, which in turn switch a load (motor, heater, pump, etc.) on and off. The coil typically operates at 12 V or 24 V dc, but a 5 V coil allows the relay to be connected directly to a digital or microcontroller system (see the temperature controller, Section 13.1, Chapter 13).

A relay circuit for controlling a machine tool is illustrated in Figure 14.6(b). The system is designed to provide push-button operation and to prevent the main motor starting unless the machine guard is closed and the cutting fluid pump is on. There is also a thermal torque overload sensor, which disables the machine if the tool jams or the motor is stalled for some other reason. The relays operate in latched mode, and the system will ‘fail safe’ if the power goes off. Relay 2 (motor) is controlled from relay 1 (control), operated at 24 V. The motor and pump are connected to a 240 V supply via contacts in relay 2 and 1, respectively.

14.3.3. PLC Control

Programmable logic controllers are often used for sequential control in industrial systems. The PLC is a self-contained sequence controller, built around a microprocessor or microcontroller,

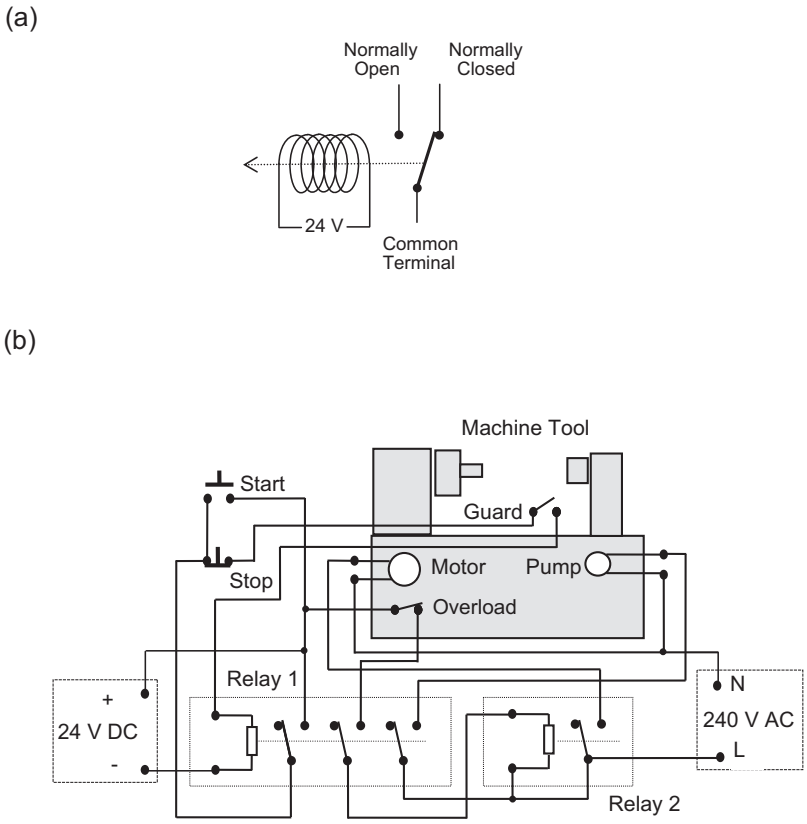


Figure 14.6
Relay control systems: (a) relay operation; (b) wiring diagram of relay machine control system

but with all the interfacing built in. PLCs also use more user-friendly programming techniques, such as ladder logic. A small Mitsubishi PLC is shown in Figure 14.7.

The PLC can be programmed to act like a set of relays to give a particular output sequence in response to switched inputs, which can be manual inputs or derived from sensors. It is suitable for controlling systems where motors, heaters, valves and other loads must be switched directly from a power supply. The same machine tool seen in the previous example is now shown under PLC control in Figure 14.8.

The PLC has inputs labeled X0, X1, X2 and X3. These are detected as 'on' when connected to 24 V via an external switched sensor or control input. The PLC is programmed to operate the outputs, labeled Y0 and Y1, according to the input sequence. The outputs are also simple switched contacts, as in the normally open contact of a relay, which operate a load circuit with an external supply. They are typically designed to handle high power loads operating with mains voltage, or three-phase supplies. If necessary, the PLC outputs can control external

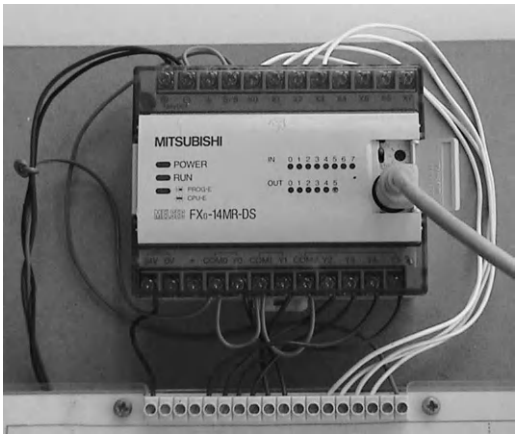


Figure 14.7
Programmable logic controller

contactors (load relays) if the load current exceeds the PLC output contact rating. The control and load circuits are electrically isolated from each other, for safety, reliability and ease of use. The PLC inputs use opto-isolators, where the on/off signal is passed as infrared light, giving complete electrical isolation between the input and controller internal circuits.

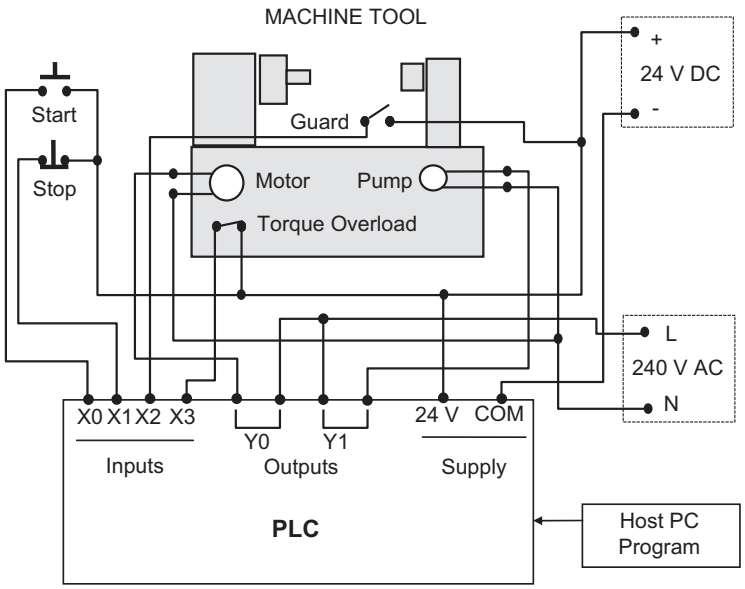


Figure 14.8
Wiring diagram of PLC machine control system

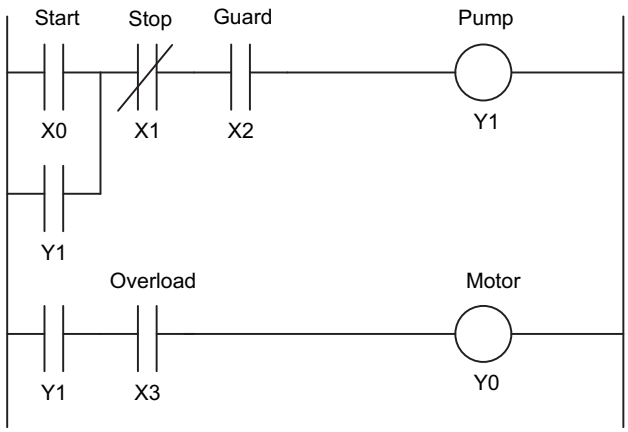


Figure 14.9
Ladder program for machine tool control

The program for the PLC can be created in 'ladder logic' form (Figure 14.9), which allows the control program to be defined as if the PLC contained the relay system shown in Figure 14.6. The graphical program corresponds to the wiring diagram of the low-voltage (control) side of the relay system. Ladder logic uses a basic set of three symbols: normally open contacts, inverted contacts and output coils. These are associated by label with a physical input (Xn) or output (Yn). The normally open contacts represent external normally open contacts connected to the corresponding input; when the real contact closes, the contact in the program is closed. An inverted contact (X1) simply reverses the polarity of the external switch. The sides of the ladder correspond to the 24 V supply rails in the real circuit, so an output goes on when there is a closed path through the contacts in that rung of the ladder to switch on the coil, which operates the associated output in the PLC. The graphical program is entered on a host PC and converted to a machine code program, which is downloaded to the microcontroller in the PLC, in the same way as an assembler program.

In the ladder diagram, the system will come on when the 'Start' input is pressed, if the 'Stop' button is open and the 'Guard' switch is closed (guard closed). The 'Stop' button itself is normally open, but is inverted in the ladder program so it operates as if normally closed. The contact labeled Y1 (pump) closes because the virtual circuit is complete. The associated contact Y1 therefore also closes, which 'holds on' the output, even when the start button is released. A second Y1 contact then switches on the motor, as long as the overload cut-out is closed (no overload). The machine is then running. If the motor is overloaded, the thermal cut-out operates and switches off the motor, but the pump stays on to maintain coolant feed. If the guard is opened or the stop button pressed, then both motor and pump go off. Output Y1 corresponds to relay 1 coil in the relay-controlled system, and Y0 to relay 2 coil.

Ladder programming was designed as a user-friendly method for creating this type of sequential control program, for engineers used to dealing with hard-wired relay systems. It was the first graphical programming method, of which there are now many, such as Flowcode for the PIC.

14.3.4. Microcontroller

For comparison with other control technologies, Figure 14.10 shows the same machine tool operated by a microcontroller. As we know, the microcontroller uses signal levels around 5 V, so the input switches have to be connected with pull-up resistors. The microcontroller is programmed to operate the output loads via suitable interfaces, which allow its outputs to switch the high-power motors. These could be relays or three-phase contactors, but high-current field effect transistors (FETs) are useful here, as they can operate with 5 V inputs and have no moving parts. The microcontroller can be programmed in its native assembler

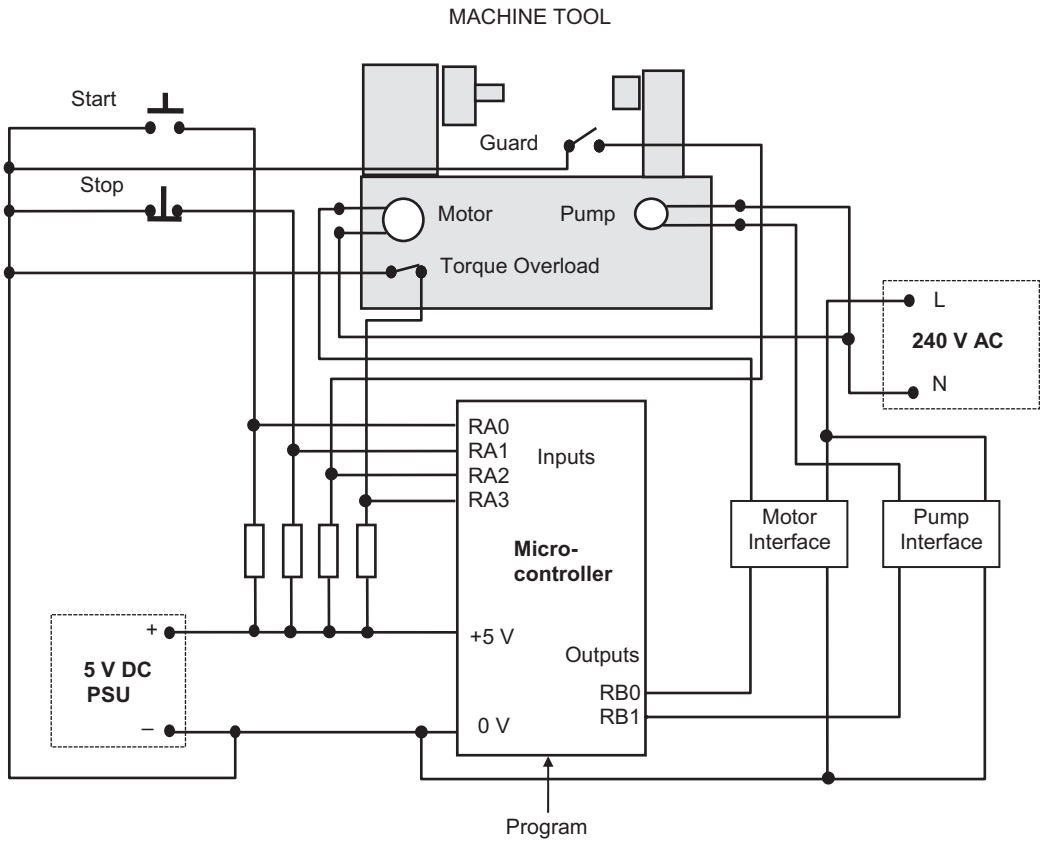


Figure 14.10
Microcontroller machine control system

```
;       MAC1.ASM
;       M Bates  5/12/03  Ver 1.0
;       Program to operate a simple machine tool

; Assembler directives ..............................

        PROCESSOR 16F84A

PortA   EQU     05
PortB   EQU     06

; Initialize ........................................

        MOVLW   B'11111100'     ; Initialize outputs
        TRIS    PortB           ; to Motor & Pump

; Start main loop ...................................

alloff  CLRW                    ; Switch off
        MOVWF   PortB           ; Motor & Pump

start   BTFSC   PortA,0         ; Check Start button
        GOTO    start           ; & wait is not pressed

stop    BTFSC   PortA,1         ; Check for Stop
        GOTO    alloff          ; & restart if pressed

guard   BTFSC   PortA,2         ; Check for Guard in place
        GOTO    alloff          ; & restart if not safe

        BTFSC   PortA,3         ; Check for Overload
        GOTO    coolit          ; & switch off Motor if true

        BSF     PortB,0         ; Motor ON
        BSF     PortB,1         ; Pump ON
        GOTO    stop            ; and loop

coolit  BCF     PortB,0         ; Motor off, keep Pump on
        GOTO    coolit          ; and wait for reset

        END
```

Program 14.2
PIC machine control program

language (Program 14.2), or C, both of which take time to learn. This is why ladder logic was developed for programming PLCs, and the built-in interfacing in the PLC makes this the usual choice for such control applications.

14.3.5. Production Systems

There are two main types of production system. Manufacturing systems include materials and component handling technologies such as conveyors and robots, which work with machine

Figure 14.11
Demonstration flexible manufacturing system

tools and assembly subsystems to produce discrete product items, such as motor vehicles. Process control systems are those that supervise continuous flow production, such as an oil refinery, where the product is a liquid, gas, powder, granules or similar material. This typically involves sequence control of pumps and valves, controlling storage tanks and pipework installations, with inputs from flow, level, temperature sensors, etc., to form a closed loop system. All these systems will contain microprocessor-based controllers, both within the PLCs that control the component subsystems, and within the dedicated controllers built into machines, tools and robots.

A flexible manufacturing system (FMS) workcell has machines that can be reprogrammed to produce a variety of similar products. Typically, it consists of pick-and-place robots working alongside machine tools to manufacture components and assemble them into a finished product. A basic demonstration system is shown in Figure 14.11. It consists of a milling machine, a hydraulic assembly rig and a component handling robot. It is designed to machine and assemble a simple product consisting of three components: a printed circuit board (PCB) in a milled plastic enclosure with a press-fit cover. The robot places a plastic blank in the mill, which machines the casing; the robot then retrieves the enclosure, places it in the assembly rig and inserts the PCB, and the cover is fitted by the hydraulic press.

A block diagram (Figure 14.12) shows how the subsystems of the workcell interconnect. The digital signals in the system operate at 24 V, the higher voltage providing better noise immunity than TTL (5 V) levels. The various controllers signal to each other with, usually, individual active low signals, to control the sequence of operations. For example, when the mill has finished, it asserts (sets active) a ‘Mill Ready’ signal to the robot controller, which triggers the robot program to pick up the finished workpiece. The robot slide, the press rig and the mill are all controlled by their own PLCs, with the main PLC in charge of the overall system. The robot

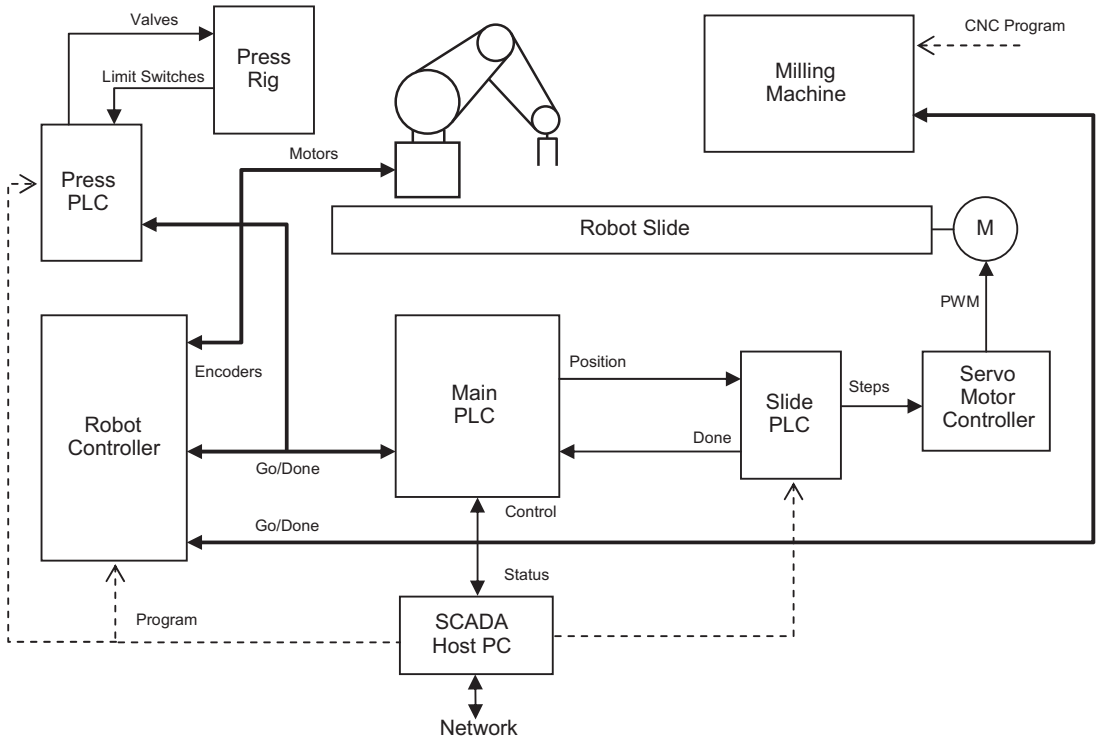


Figure 14.12
Demo flexible manufacturing system block diagram

controller needs a reasonably powerful processor system, because of the complex calculations required for the robot movement. This and the PLCs are programmed from a PC, via a serial port. The main system PLC remains connected to the PC, which then operates as a supervisory control and data acquisition (SCADA) system host when the system is running. It provides a virtual control panel and graphical status display of the system as it runs, reading status bits and writing control bits in the main PLC, and modifying the display accordingly.

Both manufacturing and process control systems can be managed by a SCADA network to provide integrated, centralized control and performance monitoring. A typical display is shown in Figure 14.13. Powerful software suites support the communication and presentation of the information, principally using on-screen interactive mimic diagrams and dynamic database management to give a complete overview of the system operation.

In the industrial environment, the subsystems need to be mounted and connected together using physically robust methods. A typical control cabinet is shown in Figure 14.14. The vulnerable parts of the control system, such as PLCs, microcontroller boards, terminal blocks, power supplies, communication modules and keypads, are protected in a steel cabinet. Another important feature is the emergency stop buttons.

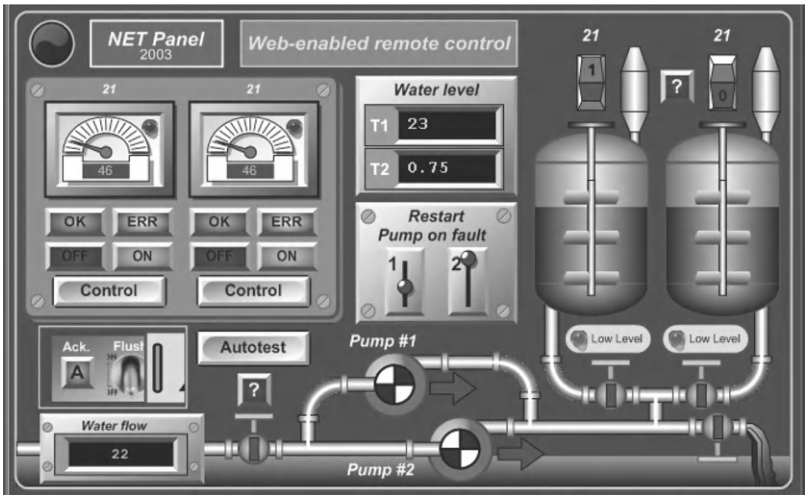


Figure 14.13
SCADA screen

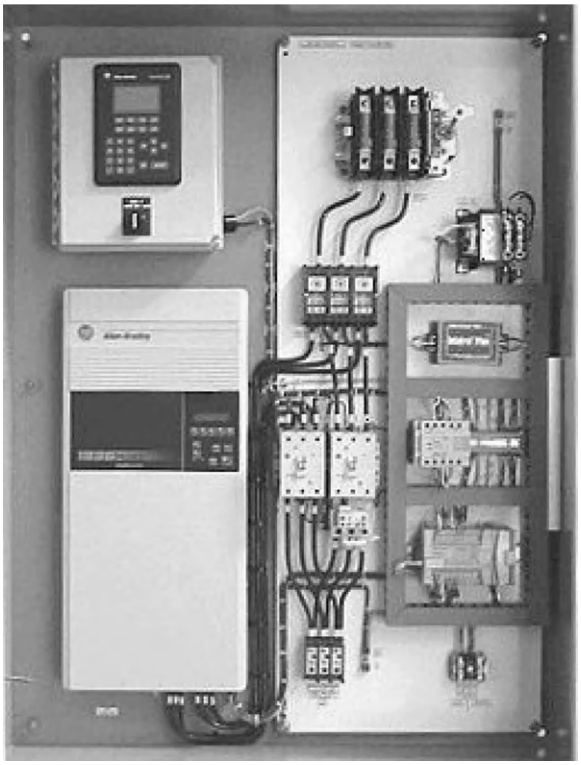


Figure 14.14
Industrial system control cabinet

Table 14.1: Comparison of control technologies

Technology	Advantages	Disadvantages	Typical applications
Relays	Simple to design No programming needed No electronics skills needed Good electrical isolation	Slow Unreliable High power consumption Not suitable for complex systems	Machine safety interlocks Simple process control High power systems Contactors for heavy loads
PLC	Minimal interfacing needed Easy to program Easy to install	Limited processing functionality Limited interfacing flexibility	Machine control Process control systems Flexible manufacturing systems
Microcontroller	Flexible hardware design Large choice Suitable for small embedded applications	Hardware design skills needed Programming skill needed Limited memory Interface design required	Smart cards Consumer goods Instrumentation Dedicated controllers
Microprocessor system	Flexible controller design Suitable for larger systems Expandable memory, I/O, etc.	Expert hardware design skills Good programming skill needed Overall expensive	Automatic machines Specialist control systems Large computers
PC Host	Available off the shelf Built in data storage and comms Graphical programming and display Basic interfacing skills only needed Standard operating system	High cost of basic unit Large physical size Mainly used as front end for other control systems	Machine tool host SCADA host Large systems user interface Instrumentation systems host Networks and distributed systems

14.4. Control System Design

The microcontroller or microprocessor forms the basis of most control systems. A dedicated microprocessor design allows the memory and interfacing to be designed separately, but, because of the range of microcontrollers now available and the additional system design work required, this is now much less likely to be a cost-effective solution. In either case, the peripheral interfacing must also be designed at component level.

By contrast, the PLC offers an off-the-shelf hardware package, requiring no external electronics to interface it. PLCs are generally designed around dedicated microcontrollers, with a built-in proprietary operating system. The program is traditionally written in ladder logic and is compiled automatically to machine code. Additional programming tools are also usually supplied by the individual manufacturer to meet more complex specifications and project management requirements. Overall, the PLC is robust, and easy to install and program, with a variety of communication interfaces to support system integration.

In industrial systems, the PC can function as a general purpose administrative computer, programming host, design workstation, SCADA display, network client or server. As a system controller, the PC is most often connected to client PLCs, robots and machine tools via a network, with the PLCs controlling the target hardware, as in the FMS above. The PC can also act separately as a programming terminal for the different programmable devices, then SCADA host when the system is operational. It can also act as a computer-aided design or electronic computer-aided design (CAD/ECAD) workstation, component database server or just a plain old word processor!

Table 14.1 provides a comparison of the advantages and disadvantages of the different forms of system control outlined above. A reasonable working knowledge of all the options is required in order to select the most appropriate technology for any given application. The microcontroller is central to all these technologies.

Questions 14

1. Outline the differences between the Intel 8051 microcontroller and an equivalent PIC MCU in terms of their general internal architecture and consequent performance. (4)
2. State two advantages and two disadvantages of the conventional processor system over the microcontroller in designing a system to meet a particular specification. (4)
3. Explain briefly the advantages of using a PLC compared with a microprocessor system in control applications. (4)
4. Draw a flowchart for Program 14.2 to show the control sequence clearly. (7)
5. List six possible functions of the PC in a production system. (6)
6. Match up the controller type with the most appropriate programming language or technique:

(a) Small microcontroller	1. ‘C’
(b) CISC Microprocessor	2. None

(c) Relay system 3. Mimic
(d) PLC 4. Assembler
(e) SCADA 5. Ladder Logic. (5)

Answers on pages 425–6. (Total 30 marks)

Activities 14

1. Log on to the Atmel website. Select a microcontroller from the list of available flash devices that is most similar to the 16F690 and compare its features and instruction set. Identify any advantages that the AVR microcontroller may have over the PIC.
2. Study the relay-based machine controller. Devise a circuit to switch a motor on and off using push buttons and a single relay. Why is this safer than using a simple mains switch?
3. Modify the PLC machine tool controller in Figure 14.8, and its program, to operate an alarm output if the machine overloads. The alarm is wired as another output.
4. Devise a block diagram of a domestic washing machine, controlled by a microcontroller. Show interface blocks between the switched actuators and sensors and the microcontroller. Write a description of the operating sequence of the machine, and devise a flowchart for the control sequence, constructed so that it could be implemented in PIC assembly language.
5. By reference to the temperature controller design in Chapter 13, design the hardware interfacing for PIC implementation of the system shown in Figure 14.10. Select a suitable device according to the I/O and memory requirements, test Program 14.2 in the MPLAB simulator and implement the design using the most readily available construction techniques. Devise a target system to simulate the machine tool, and confirm correct operation in hardware.

APPENDIX A

Binary Numbers

Chapter Outline

Digital computers, microprocessors and microcontrollers (MCUs) store their working data as binary numbers. The storage is built from electronic switching circuits. The state of the output of each stage of the circuit can be 0 or 1, which can be represented by 0 V and +5 V. Input data tends to be in the form of decimal numbers and text characters, so these have to be converted to binary for storage and processing, and converted back for output and display. This data is transmitted as digital pulses on the connections between the inputs, storage, processor and output, which are routed between these devices using the same switching transistors (see Appendices B and C for more details).

A.1. Number Systems

All arithmetic is based on number systems, which use a set of characters to represent numerical values. In microcontrollers, decimal, binary and hexadecimal are the most important. We will start with decimal, as this is our reference system used in manual calculations. Binary is the native language of microcontrollers, and hexadecimal is simply a convenient way of representing binary.

A number system can have any base you like, but some are more useful than others. For instance, base 12 is still in widespread use – think of clocks (time), boxes of eggs (dozens) and measurement of angles (degrees). In the past it was even more common, e.g. in old English money (12 pence = 1 shilling). Base 12 is useful because 12 is divisible by 2, 3, 4 and 6, giving lots of useful fractions: one-half, one-third, one-quarter and one-sixth.

A.1.1. Decimal (Base 10)

The name of each number system refers to the 'base' of the number system, which corresponds to the number of symbols used in representing values. In decimal, 10 symbols are used, with which you are familiar:

0 1 2 3 4 5 6 7 8 9

The reason for using 10 is simple: humans have 10 fingers that can be used for counting, so the decimal system was developed as a way of writing this down and doing calculations on paper instead of on our fingers or an abacus. It is no coincidence that the term 'digits' can refer to fingers as well as numbers.
Assuming that we know how to count and write down numbers in decimal, let us analyze their structure. Take, for example, the number 274: in words, two-hundred and seventy-four. This means: take two hundreds, seven tens and four units and add them together. The position of each digit in the number is literally significant; each column has a weighting that applies to the digit in that column. As you know, the least significant digit is conventionally placed at the right, and the most significant at the left. More digits are added at the left-hand end as the number size increases. In decimal, the columns have a weight 1, 10, 100, etc. Note that these correspond to a power series of 10, the number system base. Another example is examined in more detail in Table A.1.

A.1.2. Binary (Base 2)

We can understand the binary system by comparing it with decimal – the basic rules are the same for any number system. In binary, the base is 2, so the column weighting is a power series of 2, as shown in Table A.2. With a base of 2, only the digits 0 and 1 are available, so the numbers tend to have a lot of digits. For instance, a 32-bit computer uses 32-digit binary numbers to represent its data. An example with eight digits is given, showing what the digits

Table A.1: Structure of a decimal number

Column weight	1000	100	10	1
Power of base	10^3	10^2	10^1	10^0
Digits	3	6	5	2
Sum	(3×1000)	$+(6 \times 100)$	$+(5 \times 10)$	$+(2 \times 1)$
Total	$= 3652$			

Table A.2: Structure of a binary number

Bit Significance	MSB							LSB
Column weight	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal weight	128	64	32	16	8	4	2	1
Example number	1	0	1	0	0	0	1	1
Sum	128	+ 0	+32	+0	+0	+0	+2	+1
Total	= 163							

MSB: most significant bit; LSB: least significant bit.

represent and how to convert the value back to decimal. Note that any number to the power zero has the value 1.

The decimal equivalent in all number systems can be calculated by multiplying the digit value by its weighting in decimal, and then adding the resulting column products. In binary, because the digit value is 1 or 0, the result can be obtained by simply adding the digit weight where the digit value is a '1', because any number multiplied by zero is zero. When decimal data is entered into a computer, the values are converted to binary using this process.

The range of a binary number is the number of different codes possible, corresponding to a count from zero up to the maximum possible with the number of bits available. It is calculated as 2^n, where n is the number of digits. For example, for a 4-bit number, the range is $2^4 = 2 \times 2 \times 2 \times 2 = 16$. The maximum value can be calculated as $2^n - 1$. For example, for an 8-bit number, the maximum value is $2^8 - 1 = 255_{10}$.

A.1.3. Hexadecimal (Base 16)

Because binary numbers have many digits, they are not very easy to understand when written down or printed out. Conversion to decimal is not particularly straightforward, so hexadecimal is used as a way to represent binary numbers in a compact way, while allowing easy conversion back to the original binary.

Hexadecimal (base 16), or 'hex' for short, uses the same digits as the decimal system from 0 to 9, then uses letters A to F, as a single character representation for numbers ten to fifteen. Thus, characters that are normally used to make words are used as numbers in hex, not least because the symbols are already available on the keyboard. A binary number can be easily converted to hex by writing it down in groups of four bits, and then converting each group to its equivalent hex digit, as in Table A.3.

The base of the number can be shown as a subscript where necessary to avoid confusion. All number systems use the same set of characters, so if the base of the number given is not obvious from the context, it can be specified. For example, the number 100 (one, zero, zero)

Table A.3: Hexadecimal digits

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

could have the decimal value four in binary, one-hundred in decimal or two-hundred and fifty-six in hexadecimal. Later, we will see other ways of indicating numerical type when programming. Some examples of equivalent values are given in Table A.4.

The base-8 number system, octal, has sometimes been used in computing, with the digits 0—7 representing 3-bit binary numbers, but is not often seen now. It was useful in some early mainframe computers which operated with 12-, 24- or 36-bit numbers.

A.1.4. Counting

A list of equivalent numbers, counting from zero, is given in Table A.5, with some comments on important values. This table also defines memory capacity in microprocessor systems; for example, '1k' of memory is 1024 locations. Notice that $1024 = 2^{10}$. This is worth remembering as a starting point in calculating memory capacity.

Table A.4: Examples of equivalent values

Decimal	Binary	Hexadecimal
16_{10}	100_2	10_{16}
31_{10}	$1\ 1111_2$	$1F_{16}$
100_{10}	$110\ 0100_2$	64_{16}
169_{10}	$1010\ 1001_2$	$A9_{16}$
255_{10}	$1111\ 1111_2$	FF_{16}
1024_{10}	$100\ 0000\ 0000_2$	400_{16}

Table A.5: Significant equivalent numbers

Decimal (Base 10)	Binary (Base 2)	Hex (Base 16)	Comment
0	0	0	All the same
1	1	1	All the same
2	10	2	[2^1] Uses 2nd column in binary
3	11	3	Maximum 2 bit count
4	100	4	[2^2] Uses 3rd column in binary
5	101	5	[2^2] Uses 3rd column in binary
6	110	6	[2^2] Uses 3rd column in binary
7	111	7	Maximum 3 bit count
8	1000	8	[2^3] Uses 4th column in binary
9	1001	9	Decimal and hex same until 9
10	1010	A	Uses letters in hex
11	1011	B	Uses letters in hex
12	1100	C	Uses letters in hex
13	1101	D	Uses letters in hex
14	1110	E	Uses letters in hex
15	1111	F	Maximum 4 bit count
16	1 0000	10	[2^4] Uses 2nd column in hex
17	1 0001	11	Uses space to clarify binary
18	1 0010	12	
19	1 0011	13	
20	1 0100	14	
21	1 0101	15	
22	1 0110	16	
23	1 0111	17	
24	1 1000	18	
25	1 1001	19	
26	1 1010	1A	
27	1 1011	1B	
28	1 1100	1C	
29	1 1101	1D	
30	1 1110	1E	
31	1 1111	1F	Maximum 5 bit count
32	10 0000	20	$= 2^5$
33	10 0001	21	
34	10 0010	22	
..	.. ....	..	
62	11 1110	38	
63	11 1111	39	Maximum 6 bit count
64	100 0000	40	$= 2^6$
65	100 0001	41	
..	... .	..	
127	111 1111	79	Maximum 7 bit count
128	1000 0000	80	$= 2^7$
129	1000 0001	81	
..	....	..	
254	1111 1110	FE	

(*Continued*)

Table A.5: Continued

Decimal (Base 10)	Binary (Base 2)	Hex (Base 16)	Comment
255	1111 1111	FF	Maximum 8 bit count
256	1 0000 0000	100	$= 2^8$
..	. ...	..	
511	1 1111 1111	1FF	Maximum 9 bit count
512	10 0000 0000	200	$= 2^9$
..	.. ..	...	
1023	11 1111 1111	3FF	Maximum 10 bit count
1024	100 0000 0000	400	$= 2^{10} = 1$k
..	....	...	
2047	111 1111 1111	7FF	Maximum 11 bit count
2048	1000 0000 0000	800	$= 2^{11} = 2$k
..	....	...	
4095	1111 1111 1111	FFF	Maximum 12 bit count
4096	1 0000 0000 0000	1000	...
..	. ...	..	
65535	1111 1111 1111 1111	FFFF	Maximum 16 bit count

The rules for counting in any number system are:

1. Start with all digits set to zero.
2. In the right digit position (least significant digit), count up from zero to the maximum digit available (1 in binary, 9 in decimal, F in hexadecimal).
3. If a column value is at its maximum, reset it to zero, and increment (add 1 to) the next column to the left.

There is a fixed number of digits in microprocessor registers or memory locations. These tend to come in multiples of 8 bits (1 byte): 8, 16, 32, 64, 128 or 256. This determines the maximum value that can be stored. This is calculated as $2^n - 1$, where n is the number of bits. The lower values appear in the list of equivalent numbers; for example, a 16-bit register holds the maximum value $2^{16} - 1 = 65\,535_{10}$. Obviously leading zeros must be used to fill the empty positions, because each register bit must be either 1 or 0, and leading zeros do not alter the value.

A.1.5. Bits, Bytes and Words

One binary digit represents a 'bit' of information. A group of eight bits is called a 'byte', and larger binary codes are called 'words'. As we now know, in hexadecimal four bits are represented by one hex digit, so a byte is 2 hex digits, a word 4 and so on. Memory blocks in microprocessor systems tend to be addressed as 8-bit locations, even if multiple bytes are

used, so the contents are typically displayed as 2 hex digits (8 bits). Registers may have from 8 to 256 bits, but still in multiples of 8. The 14-bit program codes in the PIC® 16 are represented with 4 hex digits, the most significant only having values from 0 to 3. The two most significant two bits are assumed to be zero.

A.2. Numerical Conversion

Conversion between numerical types is often required in microprocessor systems. Data may be input in ASCII, processed in binary and output in binary coded decimal (BCD). Machine code is normally displayed in hexadecimal. We therefore need to know how to perform these conversions.

A.2.1. Binary to Decimal

The structure of binary numbers has been described above. Thus, the value of a number is found by multiplying each digit by its decimal column weight and adding. The weighting of the digits in binary is, from the least significant bit (LSB), 1, 2, 4, 8, 16 … or 2^0, 2^1, 2^2, 2^3 … that is, the base of the number system is raised to the power 0, 1, 2, 3, etc. Another example is shown in Table A.6. The same principle could be applied for converting a number of any arbitrary base to decimal.

A.2.2. Decimal to Binary

This conversion is achieved by successive division by two, the base of the required number format. The decimal number is divided by two, the remainder recorded as a binary bit value, and the result divided by two again, until the result is zero. The binary result is obtained by transcribing the column of remainder bits from the bottom up (most significant bit (MSB) to LSB) (Table A.7). Again, the same process could be applied to obtain a number in any required base.

A.2.3. Binary and Hex

Binary to hexadecimal conversion is simple; that is why hex is used. Each group of four bits is converted to the corresponding hex digit, starting with the least significant four, and

Table A.6: Binary to decimal conversion

Column weight	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Binary digit	1	0	0	1	0	1	1	0
Weight × digit	128 × 1	64 × 0	32 × 0	16 × 1	8 × 0	4 × 1	2 × 1	1 × 0
Result	128	+0	+0	+16	+0	+4	+2	+0
Sum	150_{10}							

Table A.7: Decimal to binary conversion

Decimal number = 150				
Divide by 2		**Result**	**Remainder**	**Order**
150/2	=	75	0	LSB
75/2	=	37	1	
37/2	=	18	1	
18/2	=	9	0	
9/2	=	4	1	
4/2	=	2	0	
2/2	=	1	0	
1/2	=	0	1	MSB
Binary equivalent = 10010110				

padding with leading zeros if necessary (Table A.8). The reverse process is just as trivial, where each hex digit is converted to a group of four bits, in order. The result can be checked by converting both to decimal, using the process described above in Table A.6. Hex to decimal is shown in Table A.9.

A.2.4. BCD

Binary coded decimal (BCD) uses only the binary codes from 0 to 9 to represent decimal digits (Table A.10). A numeric keypad, for example, has numbers 0–9, so this format could be used for input. When multi-digit numbers are input, the keys are pressed in the sequence from the highest significant digit to the lowest. To obtain the binary equivalent of the complete number, the value of each keystroke must be added to a binary sum after the number has been

Table A.8: Binary to hexadecimal conversion

1001	1111	0011	1101	$= 9F3D_{16}$
9	F	3	D	

Table A.9: Hex to decimal conversion

Hex Digits	**9**	**F**	**3**	**D**
× Column weight	9×16^3	15×16^2	3×16^1	13×16^0
Results	36 864	+3840	+48	+13
Total	$40\,765_{10}$			

Table A.10: Binary coded decimal (BCD) codes

Decimal	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

entered. The least significant digit is added to the total unweighted, the next multiplied by 10 and added, the next by 100, and so on to give the total value in binary.

After processing in binary, results may need to be displayed in BCD format on, for example, a seven-segment display. Binary to BCD conversion for output may be implemented as the inverse process to that described above. If the maximum value is, say, 9999, the value must be divided in turn by 1000, 100 and 10. The division result gives the BCD code for each digit, with the remainder being passed to the next stage. See Section A.3.3 for an outline of multiplication and division in binary.

A.2.5. ASCII

ASCII (American Standard Code for Information Interchange) is a binary code for representing alphanumeric characters, as found on the computer keyboard. The basic code consists of seven bits. For example, capital (upper case) 'A' is represented by binary code 100 0001 (65_{10}), 'B' by 66, and so on to 'Z' $= 65 + 25 = 90 = 1011010_2$. Lower case letters and other common keyboard characters such as punctuation, brackets and arithmetic signs, plus some special control characters, also have a code in the range 32 to 126. The numerical characters are included, for example '9' $= 011\ 1001_2$, so it sometimes needs to be clear if the code is the binary equivalent (1001_2) or the ASCII code ($011\ 1001_2$). Notice that 30h is added to the binary number to get the ASCII code for that decimal number. Table A.11 shows selected characters from the set that would be found on a simple calculator keypad.

The microcontroller program instructions are stored as binary codes. The source code itself is stored as ASCII, so the source code mnemonics (text strings) must be converted to binary machine code by the assembler (see Chapter 3) to create the MCU hex program.

Table A.11: Selected ASCII codes

Decimal Value	Hex Value	Binary Value	ASCII Character
35	23	010 0011	#
42	2A	010 1010	*
43	2B	010 1011	+
45	2D	010 1101	−
47	2F	010 1111	/
48	30	011 0000	0
49	31	011 0001	1
50	32	011 0010	2
51	33	011 0011	3
52	34	011 0100	4
53	35	011 0101	5
54	36	011 0110	6
55	37	011 0111	7
56	38	011 1000	8
57	39	011 1001	9
65	41	100 0001	A
66	42	100 0010	B
67	43	100 0011	C
	etc.		etc.
97	61	110 0001	a
98	62	110 0010	b
99	63	110 0011	c
	etc.		etc.

A.3. Binary Arithmetic

Some form of calculation is needed in most programs, even if it is a simple subtraction to determine whether an input is greater or less than a required level. At the other extreme, a gaming program may carry out millions of operations per second when animating a three-dimensional graphic scene. Here, we will cover just the basics so that simple arithmetic operations can be programmed.

A.3.1. Addition

A simple calculation, adding two numbers whose result is 255 or less (the maximum for an 8-bit location), is shown in Example A.1. The addition of each bit pair is carried out, from right to left, according to the rules:

```
0 + 0 = 0

0 + 1 = 1

1 + 1 = 0 carry 1 to the next column
```

	Binary	Conversion	Decimal
	0111 0100	64+32+16+4	116
+	0011 0101	32+16+4+1	+53
=	1010 1001	128+32+8+1	169
Carry bits	11 1		

Example A.1
Adding with result less than 256

The conversion to decimal of each binary number is also shown to confirm that the result is correct.

When implemented by the ADDWF or similar instruction in the PIC microcontroller, the carry bits are internally handled, until there is a carry-out from the MSB. This occurs when the result is higher than 255, shown in Example A.2. This carry-out is recorded in the Carry bit of the status register. It is held there so that, for example, it can be added to the next most significant byte of a multi-byte number. In this case, the carry bit from the low byte addition must be added to the LSB of the next byte to obtain the right result. The sample calculation for this case is also shown in Example A.3.

```
                0111 0010    = 116
            +   1001 0000    = 144
Carry out   1   0000 0100    = 260
```

Example A.2
Adding with result greater than 255

```
                0111 0101 0101 0111   =  7557
            +   0001 1000 1100 1011   =  18CB
                1000 1110 0010 0010   =  8E22
Carry bits      111- --11 1-11 111-
Carry from low to high byte ^
```

Example A.3
Adding multiple bytes

A.3.2. Subtraction

Subtraction is straightforward if one number is subtracted from a larger one. Pairs of binary digits are processed individually from the least significant to the most significant according to the rules:

0 − 0 = 0

1 − 0 = 1

```
1 − 1 = 0

0 − 1 = 1 (borrow 1, then 2 − 1 = 1)
```

A borrow is taken from the next column if necessary, having a weight of 2 in the current column. This may cause a further borrow from the next significant column. If the overall result is positive (no borrow into the MSB required), no further processing is needed, as in Example A.4.

```
Borrow   -22- ----
          1100 1011  =  203
      -   0110 0010  =  -98
          0110 1001  =  105
```

Example A.4
Subtraction with a positive result

When executing SUBLW, for example, in the PIC, the carry flag provides the borrow bit into the MSB. Therefore, the carry flag must be set before a subtract operation so that a 1 is available to borrow. If the borrow is taken, the carry flag is cleared, indicating a negative result. This is a negative number represented by the '2s complement' form of that number (Example A.5).

```
Carry     222222--
    1     00100001  =   33
      -   00101101  =  -45
    0     11110100  =  -12

Complement    00001011
       +1  =  00001100  =   12
```

Example A.5
Subtraction with a negative result

Negative numbers are generated when the register is decremented below zero, as shown in Table A.12. Thus, in 8-bit 2s complement form, −1 is represented by FF, −2 by FE and so on. To convert this form back to the equivalent negative integer, invert all the bits and add 1. For example, to convert the 2s complement value FC into its equivalent, −4:

```
FC = 1111 1100 → 0000 0011 + 1 = 4 → −4
```

This would be necessary if the result were to be displayed as BCD or ASCII digits. The 2s complement conversion is shown in Example A.5 to check the result.

A.3.3. Multiplication and Division

A simple algorithm for multiplication is successive addition. For example:

```
3 × 4 = 4 + 4 + 4
```

Table A.12: Negative binary numbers

Decimal	Binary	Hex
+3	0000 0011	03
+2	0000 0010	02
+1	0000 0001	01
0	0000 0000	00
−1	1111 1111	FF
−2	1111 1110	FE
−3	1111 1101	FD
−4	1111 1100	FC
etc.		

This can be implemented by initializing a register to zero, then adding four, three times. The outline for a multiply program is shown below:

```
MULTIPLY BY ADDING
     Clear a Result register
     Load a Count register with Num1
     Loop
          Add Num2 to Result
          Decrement Count
     Until Count = 0
```

After the procedure, the result of the multiplication remains in the Result register. If the result is greater than 255, the carry-out must be handled as part of a multi-byte addition. An alternative method uses shift and add, which is more efficient for larger numbers. Divide is the inverse of multiply, so can be implemented using successive subtraction for small numbers. Specific multiply and divide instructions are typically provided in higher performance MCUs, where the arithmetic and logic unit (ALU) contains suitable hardware.

A.3.4. Floating Point Numbers

On a scientific calculator, large and small numbers are represented by a decimal number (mantissa) and exponent, for example 1.2345×10^6. Computers can also handle numbers in this format, typically as 32-bit numbers with 23 bits representing the decimal part and 8 bits the exponent, with the remaining bit recording the sign (positive or negative). This format would only be used in high-powered microcontrollers programmed in C.

APPENDIX B

Microelectronic Devices

Chapter Outline

This appendix describes the basic circuit elements used in microcontrollers, for those who have not previously studied microelectronic devices. This should allow the reader to interpret the logic circuits and block diagrams that appear in the PIC® data sheet.

B.1. Digital Devices

The binary codes that make up the program and data in the microcontroller are stored and processed as electronic signals. The binary numbers are nominally represented as follows:

$$\text{Binary } 0 \;=\; 0 \text{ Volts}$$

$$\text{Binary } 1 \;=\; +5 \text{ Volts}$$

A +5 V power supply unit (PSU) operated from the mains is the traditional method of powering digital circuits. It must be able to provide sufficient current for the processor circuits at a voltage between 4.75 and 5.25 V for standard TTL (transistor–transistor logic). The power

consumption is the product of the supply voltage and current drawn at the power supply pins of the chip:

$$P = VI \text{ Watts } (I = \text{chip current})$$

By replacing I using Ohm's law ($I = V/R$), we get:

$$P = V^2/R \text{ Watts } (R = \text{input resistance of chip})$$

This power is dissipated as heat, which is undesirable from both the point of view of efficiency and the fact that all chips have a maximum operating temperature. If necessary, cooling must be fitted, as is seen in the typical PC motherboard where the central processing unit (CPU) has a fan attached to its heatsink.

Assuming the input resistance of the chip is constant, we can see that the power consumption is proportional to the square of the supply voltage. That means that if the voltage is halved, for example, the power consumed will be reduced to a quarter of the original value, so any reduction in supply voltage is highly desirable.

A supply of 3.3 V is now commonly used to reduce power consumption in large chips. The power consumption will then be $3.3^2/5^2 \times 100 = 44\%$, or less than half, compared with a 5 V supply. This also allows the chip to run faster, since power consumption is broadly proportional to clock speed.

In the original design of small-scale chips, bipolar transistors were used to form TTL gates operating at +5 V. These have relatively large power dissipation, and run at a correspondingly high temperature. This limits the number of gates that can be operated on one chip, so very large-scale integrated (VLSI) circuits normally use field effect transistor (FET) logic gates, because of their lower power consumption. These are used in complementary metal oxide semiconductor (CMOS) chips, like the PIC, and can run at lower voltages, saving even more power. Low power technology is essential for battery-powered applications, such as laptops and mobile phones. There is continuing development of logic technologies, to obtain higher speed, lower cost and lower power dissipation in increasingly complex chips. Microchip is currently extending its range of XLP (extra low power) chips, which operate down to 1.8 V.

B.1.1. FET Logic

The FET is the basic device upon which microcontroller logic circuits are based. It works as a current switch: current flow through a semiconductor channel is controlled by the voltage at the input gate. An individual FET is shown in Figure B.1(a).

Current flows through the channel when it is switched on by applying a positive voltage between the gate and 0 V. When the input voltage is zero, the channel has a high resistance to current flow,

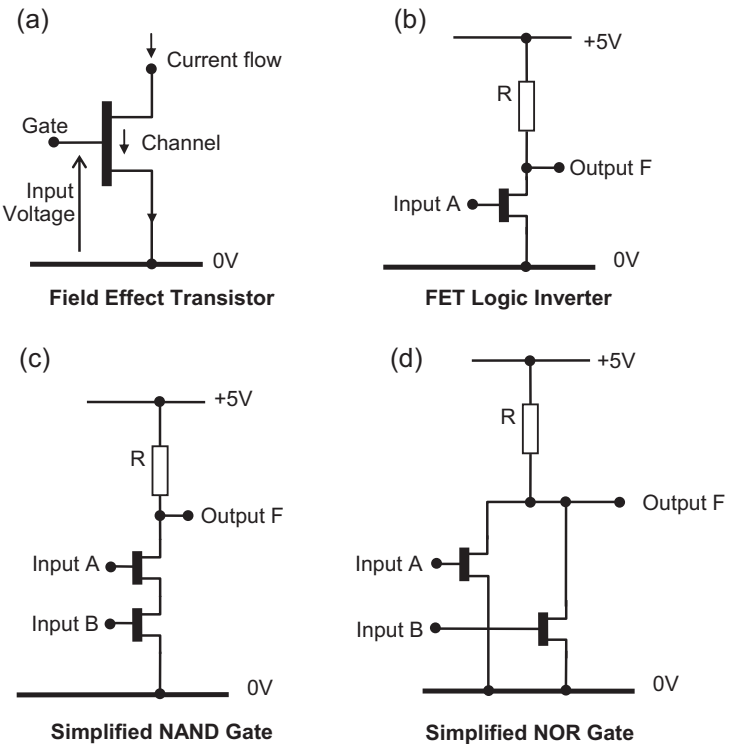


Figure B.1
Field effect transistor logic gates: (a) field effect transistor; (b) FET logic inverter; (c) simplified NAND gate; (d) simplified NOR gate

and the device is off. The current flows from the drain to the source terminal in the N-channel FET. N-channel means there is a surplus of negative charge carriers (electrons) in the semi-conductor channel to conduct the current. P-channel FETs work in the inverse fashion, and N and P types are often used in pairs, where one is on and the other off at any one time.

A logical invert operation is implemented by the FET circuit in Figure B.1(b). Assume that the FET is switched on with +5 V at input A. The channel will then have a low resistance allowing current to flow through the load resistor, R, causing a voltage drop across it. This means that the voltage at F must fall, and for correct switching operation, F must be near zero volts when the FET is on. Thus, the output is near 0 V (logic 0) when the input is +5 V (logic 1). Conversely, the output is 'pulled up' to +5 V (logic 1) by R when the input is low (logic 0). There is then no current flow in the FET channel, and no voltage drop across the resistor. The output must therefore be at the same voltage as the supply, +5 V, giving us the required logic inversion.

The logic operation AND requires the output of a circuit to be *high* only when all inputs are *high* (Table B.1). NAND, the inverse operation, requires that the output is *low* only when all

Table B.1: Logic table for one and two input gates

	Outputs					
Inputs	**NOT**	**AND**	**OR**	**NAND**	**NOR**	**XOR**
0	1	-	-	-	-	-
1	0	-	-	-	-	-
00	-	0	0	1	1	0
01	-	0	1	1	0	1
10	-	0	1	1	0	1
11	-	1	1	0	0	0

inputs are *high*. This operation can be implemented as shown in Figure B.1(c). The output F is only low when *both* transistors are on. The AND function can be obtained by inverting the NAND output, which can be implemented by connecting the inverter circuit to the NAND output.

Similarly, the logic operation OR requires the output of a gate to be *high* when either input is *high* (Table B.1). NOR, the inverse output, requires that the output is *low* when either input is *high*. This operation can be implemented as shown in Figure B.1(d). The output F is low when either transistor is on. The OR function can then be obtained by inverting the NOR output, by connecting the inverter circuit.

In real logic gates, the circuits are a little more complex, but not much. Resistors are not used because they waste too much power; instead, an additional FET is used as an active load, which reduces dissipation. The logic operations in Table B.1 are all we need to make any logic or processor circuit. Digital circuits are based on various combinations of these logic gates, fabricated on a silicon wafer. A microprocessor may contain thousands of gates and millions of transistors.

B.1.2. Logic Gates

The basic set of logic devices comprises the AND gate, OR gate and NOT gate (or logic inverter); standard symbols are shown in Figure B.2. The inputs on the left accept logic (binary) inputs, producing a resulting output on the right.

Three additional gates can be made up from the basic set: the NAND gate, NOR gate and XOR (exclusive OR) gate. The NAND is just an AND gate followed by a NOT gate, and a NOR gate is an OR gate followed by a NOT gate. An XOR gate is similar to an OR gate except that the output is low when both inputs are high (Table B.1).

Table B.1 shows all the possible input combinations for one and two inputs. Obviously, the only possible inputs for the inverter are 1 and 0. With two inputs, there are four possible input

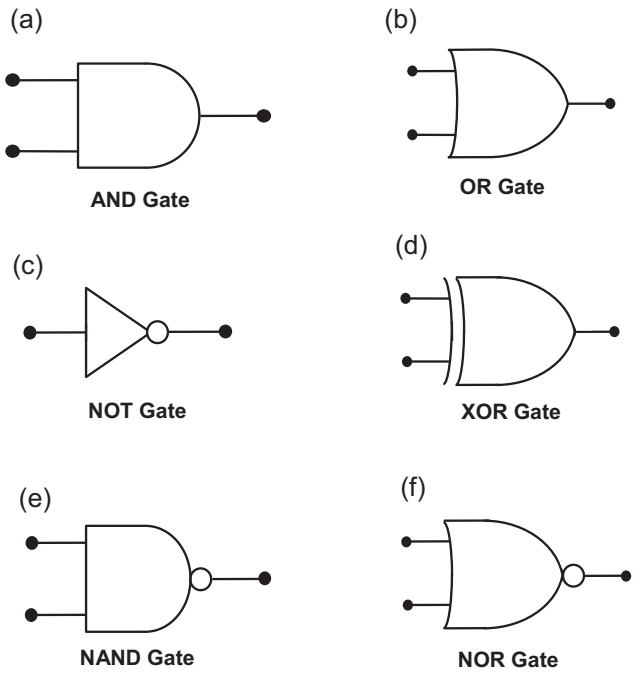


Figure B.2
Logic gate symbols (US Standard): (a) AND gate; (b) OR gate; (c) NOT gate; (d) XOR gate; (e) NAND gate; (f) NOR gate

combinations. Individual gates may have more inputs, but the logical operation will be similar; for instance, a three-input AND gate requires all inputs to be high to give a high output.

Variations on this representation may appear in data sheets. For instance, the circle representing logic inversion may be used at the input to a gate, as well as the output, but it should always be possible to work out the logical operation from the basic logic symbol set. The more detailed analysis and design of discrete logic circuits are described in standard textbooks, and do not need to be covered here. Such discrete design principles are, in any case, less important now for the circuit designer owing to the availability of microcontrollers such as the PIC, which provide a firmware-based alternative to hard-wired logic.

B.2. Combinational Logic

Logic circuits can be divided into two categories: combinational and sequential. Combinational logic describes circuits in which the output is determined only by the current inputs, and the only timing issue is the short delay between input and output changes. In a sequential circuit, the output at any time is determined by current *and* previous inputs.

```
          1  1  1  1    (A)
    +     0  1  1  0    (B)
    =     0  1  0  1    (Sum)
Carry (1) 1  1
```

Figure B.3
Example of binary addition

A timing or clock signal is normally required to trigger changes in a sequential circuit, but not in a combinational one.

B.2.1. Binary Addition

Circuits designed for binary addition provide examples of simple combinational logic. Binary addition is a basic function of the arithmetic and logic unit (ALU) in any microprocessor. A 4-bit binary addition is shown in Figure B.3 to illustrate the process required, as outlined in Appendix A.

The digits in the least significant column are added first, and the result 1 or 0 is inserted in the sum row. If the sum is two (10_2), the result is zero with a carry into the next column. The carry is then added to the sum of the next column, and so on, until the last carry-out is recorded as the most significant bit of the result. The result can therefore have an extra (carry) digit. We can design a logic circuit to implement this process, using a binary adder circuit for each column, feeding the carry bits forward as required.

B.2.2. Binary Adder Circuit

The basic operation can be implemented using logic gates as shown in Figure B.4. Two input bits are applied at A and B, giving the result at F. This circuit is equivalent to a single XOR gate, which can be used as our basic binary adder as the circuit is developed. The circuits that store the inputs and display the outputs will be described later.

B.2.3. Full Adder

To add complete binary numbers, a carry bit must be generated from each bit adder, and added to the next significant bit in the result. This can be done by elaborating the basic adder circuit as

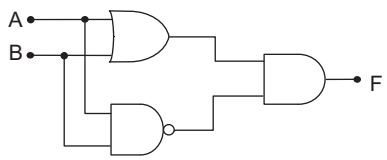
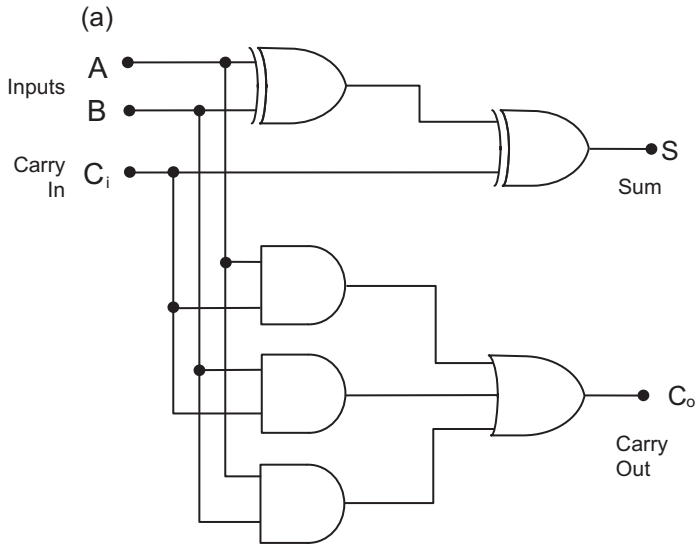


Figure B.4
Binary adder logic circuit



(b)

Inputs			Outputs	
Input A	Input B	Carry In C_i	Carry Out C_o	Sum S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

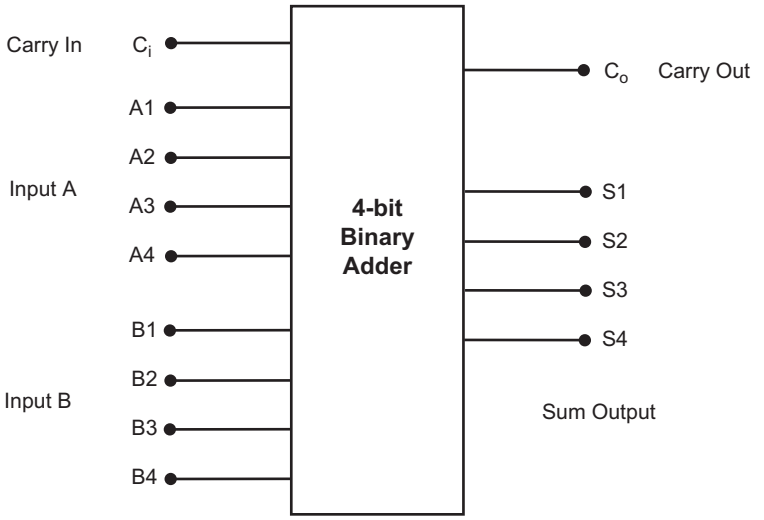
Figure B.5
Full adder circuit and logic table: (a) full adder logic circuit; (b) full adder logic table

shown in Figure B.5(a). The required function of the circuit can be specified with a logic table, as shown in Figure B.5(b). To implement this logic function, the carry-out (C_o) from each stage must be connected to the carry-in (C_i) of the next, so that we end up with four full adders cascaded together. The overall carry-in must be applied to the C_i of stage 1 and the carry-out will then be obtained from C_o of stage 4.

B.2.4. Four-Bit Adder

A set of four full adders can be used to produce a 4-bit adder, or any other number of bits, by cascading one adder into the next. The PIC 16 ALU processes 8-bit data. Since the circuit is now getting a bit complicated, and we are not particularly concerned with exactly how the logic is designed, we can hide it inside a block, and then simply define the necessary logical inputs and the resulting outputs, as shown in Figure B.6.

(a)



(b)

	INPUTS									OUTPUT					
	Input A				Input B						Output Sum				
Row	A4	A3	A2	A1	B4	B3	B2	B1	C_i	C_o	S4	S3	S2	S1	Dec
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1
2	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1
3	0	0	0	0	0	0	0	1	1	0	0	0	1	0	2
4	0	0	0	0	0	0	1	0	0	0	0	0	1	0	2
5	0	0	0	0	0	0	1	0	1	0	0	0	1	1	3
6	0	0	0	0	0	0	1	1	0	0	0	0	1	1	3
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	
etc															etc
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	
509	1	1	1	1	1	1	1	0	1	1	1	1	0	1	30
510	1	1	1	1	1	1	1	1	0	1	1	1	1	0	30
511	1	1	1	1	1	1	1	1	1	1	1	1	1	1	31

Figure B.6
Four-bit binary full adder: (a) 4-bit adder block; (b) logic table for 4-bit adder

All possible input combinations must be processed, and these can be generated by using a binary count in the inputs. The state of the output for each input combination is defined in a logic table. With 2 × 4-bit inputs, plus the carry-in, there are 512 possible input combinations in all, so the logic table only shows the first few and last rows.

In the past, logic circuits had to be designed using Boolean mathematics and built from discrete chips. Now, programmable logic devices (PLDs) make the job easier, as the required operation can be defined with a logic table or function statement. This is entered as a text file into a PC

and converted into programming instructions, which are sent to the PLD to configure the links between each gate.

B.3. Sequential Logic

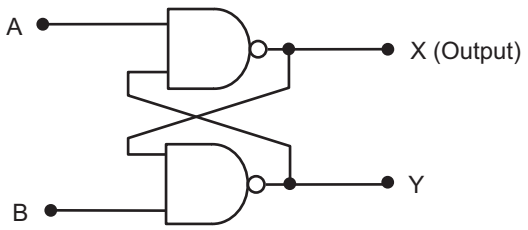
Sequential logic refers to digital circuits whose outputs are determined by the current inputs *AND* the inputs that were present at an earlier point in time. That is, the sequence of inputs determines the output. Such circuits are used to make data storage cells in registers and memory, and counters and control logic in the processor.

B.3.1. Basic Latch

Sequential circuits are made from the same set of logic gates shown in Figure B.2. They are all based on a simple latching circuit made with two gates, where the output of each gate is connected back to an input of the other, as shown in Figure B.7(a).

This latch circuit uses NAND gates, but NOR gates will work in a similar way. When both inputs, A and B, are low, both outputs must be high. This state is not useful here, so is called 'invalid'. When one input is taken high, the output of that gate is forced low, and the other output high. The latch is now set, or reset, depending on which output, X or Y, is being used to feed the

(a)



(b)

Time	INPUTS A	INPUTS B	OUTPUTS X	OUTPUTS Y	Comment
1	0	0	1	1	Invalid
2	0	1	1	0	X = 1
3	1	1	1	0	Hold X = 1
4	1	0	0	1	Reset X = 0
5	1	1	0	1	Hold X = 0
6	0	1	1	0	Set X = 1
7	1	1	1	0	Hold X = 1

Figure B.7
Basic latch: (a) basic latch circuit; (b) sequential logic table for basic latch

next stage. In Figure B.7, X is taken as the output, and is set high. This state is held when the other input is taken high, and this gives us the data storage operation required. The output X can now be reset to zero by taking input B low. This reset state is held when B is returned high.

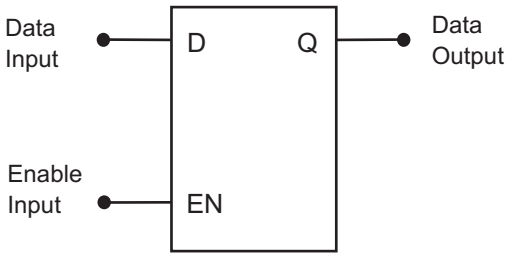
The sequence of events is shown in Figure B.7(b). At time slot 3 a data bit '1' is stored at X, while at time slot 5 data bit '0' is stored. Note that in the time slots when both inputs are high, output X can be high or low, depending on the sequence of inputs *before* that step was reached.

With additional control logic, the basic latch circuit can be developed to give two important logic building blocks: the D-type (data) latch, which acts as a one bit data store, and T-type (toggle) bistable, which is used in counters. Such bistable (two stable states) devices are often referred to as 'flip-flops'. Different kinds of sequential circuits, including the counters and registers as used in the microcontroller, can be constructed from a general purpose device called a 'J-K flip-flop'. Counters and registers are covered in Appendix C.

B.3.2. Data Latch

The data latch is a basic data storage device, shown in Figure B.8(a). The operating sequence can be represented by a logic table (Figure B.8b). When the enable (EN) input is high, the

(a)



(b)

Time	INPUTS D	INPUTS EN	OUTPUT Q	Comment
1	0	0	x	Output unknown
2	0	1	0	Output = Input 0
3	0	0	0	Data 0 Latched
4	1	0	0	Data 0 Held
5	1	1	1	Output = Input 1
6	1	0	1	Data 1 Latched
7	0	0	1	Data 1 Held
8	0	1	0	Output = Input 0

Figure B.8
Data latch: (a) data latch circuit; (b) sequential logic table for data latch

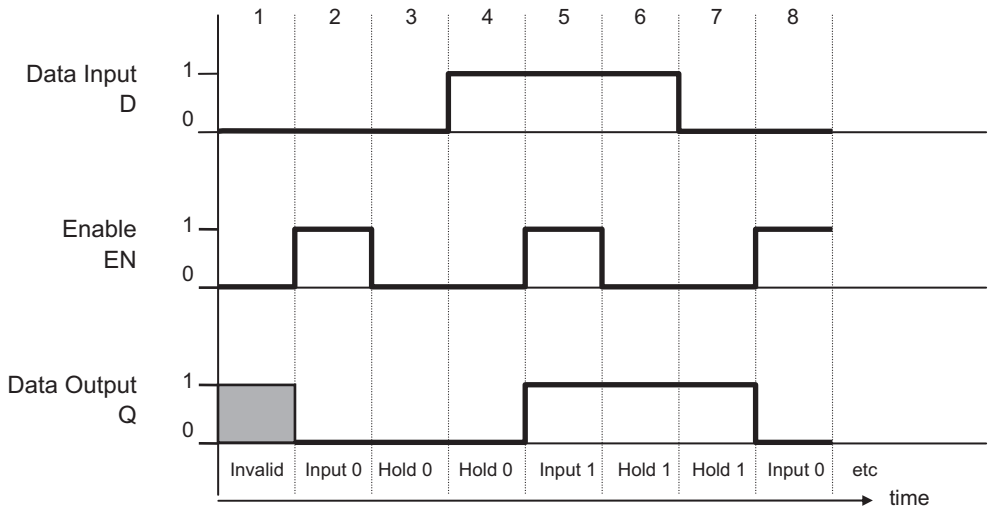


Figure B.9
Data latch timing

output (Q) follows the state of the input (D). When the enable is taken low, the output state is held. The output does not change until the enable is taken high again. This is a transparent latch, because the data goes straight through when the enable is high. A timing diagram (Figure B.9) shows the latch operating sequence in a way that may be easier to interpret than the logic table. An alternative type, the edge-triggered latch, stores the data input at a specific point in time, when the enable (clock) signal changes; this is used in registers and static random access memory (RAM) to store groups of 8 bits.

B.4. Data Devices

All data processing or digital control systems need circuits to carry out the following operations:

- data input
- data storage
- data processing
- data output
- control and timing.

In order to explain the operation of microprocessor and microcontroller systems, the logic devices outlined above will be used to make up a basic data processing circuit. Some additional devices are needed, the tri-state gate (TSG) and line driver, to complete the system (Figure B.10). Note that all the active devices (TSG, latch and driver) need a power supply (nominally 5 V), which is not normally shown explicitly in the digital circuit.

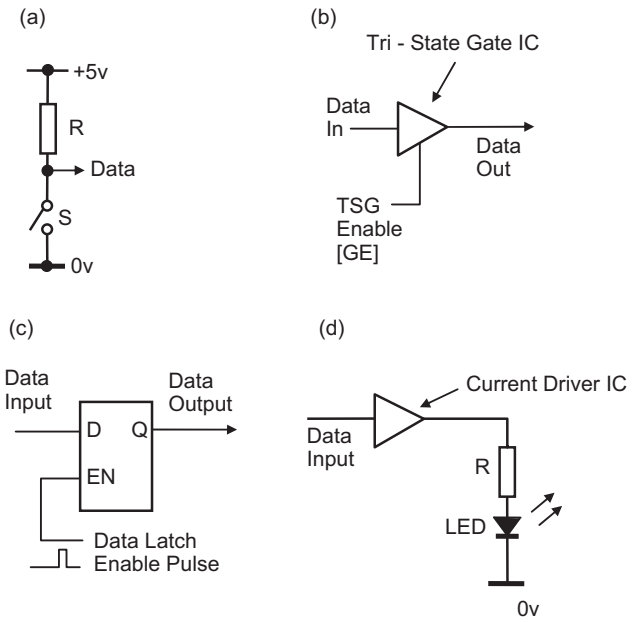


Figure B.10
Data circuit elements: (a) switch input; (b) tri-state gate; (c) data latch; (d) LED output

B.4.1. Data Input Switch

In Figure B.10(a), a switch (S) and resistor (R) are connected across a 5 V supply. If the switch is open, the data output is pulled up to +5 V, via the resistor. If the switch is closed, the logic level at the data output must be 0, as it is connected directly to ground. The resistor is required to prevent a short-circuit between the +5 V and 0 V supplies, while allowing the output to rise to +5 V when the switch is open. This only works if a relatively small current is drawn by the load at the data output. This is usually not a problem, as digital inputs typically draw no more than a few microamps. One practical problem is switch bounce; when the switch is closed, the contacts may bounce open again briefly, which can be detected by any digital circuit attached as multiple switch operations. To provide a single clean transition, a capacitor can be connected across the contacts to smooth the rising voltage, or a latch (see above) connected to the switch output. Data input switches can be used individually as control inputs, or in an array of rows and columns in a keypad (see Chapter 1, Figure 1.9).

B.4.2. Tri-State Gate

The TSG (Figure B.10b) is a digital device that allows electronic switching and routing of signals through a data processing system. It is basically an FET switch (see Section B.1.1 above), where the data passes through the drain-source channel. It is controlled by the Gate

Enable input (GE). When GE is active (in this example high), the gate is switched on, and data is allowed through, 1 or 0. When GE is inactive (low), the data is blocked, and the output goes into a high impedance (HiZ) state, which effectively disconnects it from the input of the following stage. The TSG may have an active low input, in which case the control input has a circular invert symbol. TSGs are used within VLSI circuits such as the PIC microcontroller, where they allow data from different sources onto the internal data bus.

B.4.3. Data Latch

A data latch (Figure B.10c) is a circuit block which stores one bit of data, as described in section B.3.2. If a data bit is presented at the input D (1 or 0), and the latch clocked by pulsing the Latch Enable input (0,1,0), the data appears at the output Q. It remains there when the input is removed or changed, until the latch is clocked again. Thus, the data bit is stored, and can be retrieved at a later time in the data processing sequence. Sets of data latches are used to form the registers in a microcontroller.

B.4.4. LED Data Display

A light-emitting diode (LED) provides a simple data display device. In Figure B.10(d) the logic level to be displayed (1 or 0) is fed to the current driver, which operates as an amplifier that provides enough current (typically about 10 mA) to make the LED light up when the data is '1'. Again, this is essentially an FET switch, which connects the LED to the supply when the input gate is enabled. The resistor value controls the size of the current. Seven-segment, and other matrix displays, use LEDs to display decimal or hexadecimal digits by lighting up suitably arranged LED segments or dots.

B.5. Simple Data System

The way that data is transferred through a digital system using the devices described above is illustrated in Figure B.11. The circuit allows one data bit to be input at the switch (0 or 1), enabled onto the input of the latch, stored at its output and displayed on the LED.

The operational steps are as follows:

1. The data at D1 is generated manually at the switch ('0' = 0 V and '1' = +5 V).
2. When the TSG is enabled, the data becomes available at D2 (while the gate is disabled, the line D2 is floating, or indeterminate).
3. When the data latch is pulsed, level D2 is stored at its output, D3 (D3 remains stored until new data is latched, or the system powered down).
4. While latched, the data at D3 is displayed by the LED (On = '1'), via the current driver stage.

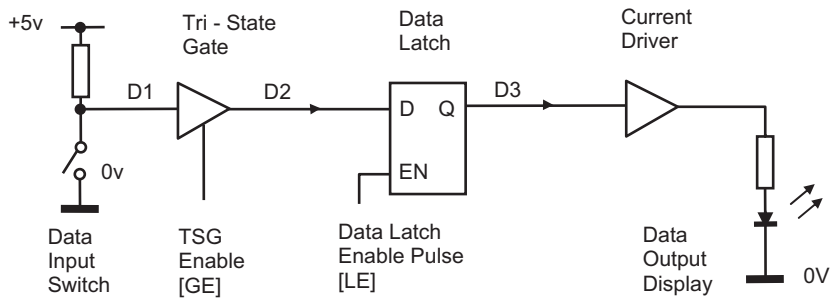


Figure B.11
One-bit data system

Table B.2: One-bit system operating sequence

Operation	Switch	D1	GE	D2	LE	D3
Data Input 1	Open	1	0	x	0	x
Input Enable	Open	1	1	1	0	x
Latch Data	Open	1	1	1	0-1-0	1
Input Disable	Open	1	0	x	0	1
Data Input 0	Closed	0	0	x	0	1
Input Enable	Closed	0	1	0	0	1
Latch Data	Closed	0	1	0	0-1-0	0
Input Disable	Closed	0	0	x	0	0

Table B.2 details the control sequence, with the data states that exist after each operation. Note that 'x' represents 'don't know' or 'don't care' (it could be 1, 0 or floating).

B.6. Four-Bit Data System

Data is usually moved and processed in parallel format within a microprocessor system. The circuit shown in Figure B.12 illustrates this process in a simplified way.

The function of the 4-bit system is to add two binary numbers that are input at the switches. The two numbers A and B will be stored, processed and output on a seven-segment display, which shows the output value in the range 0 to F. The display has a built-in decoder that converts the 4-bit binary input into the corresponding digit pattern on the segments. To obtain the correct result, the two input numbers must add up to 15_{10} or less, or up to 9_{10} for a binary coded decimal (BCD) result.

The common data bus, as is the case in any microprocessor system or microcontroller architecture, is used to minimize the number of connections required. However, this means that

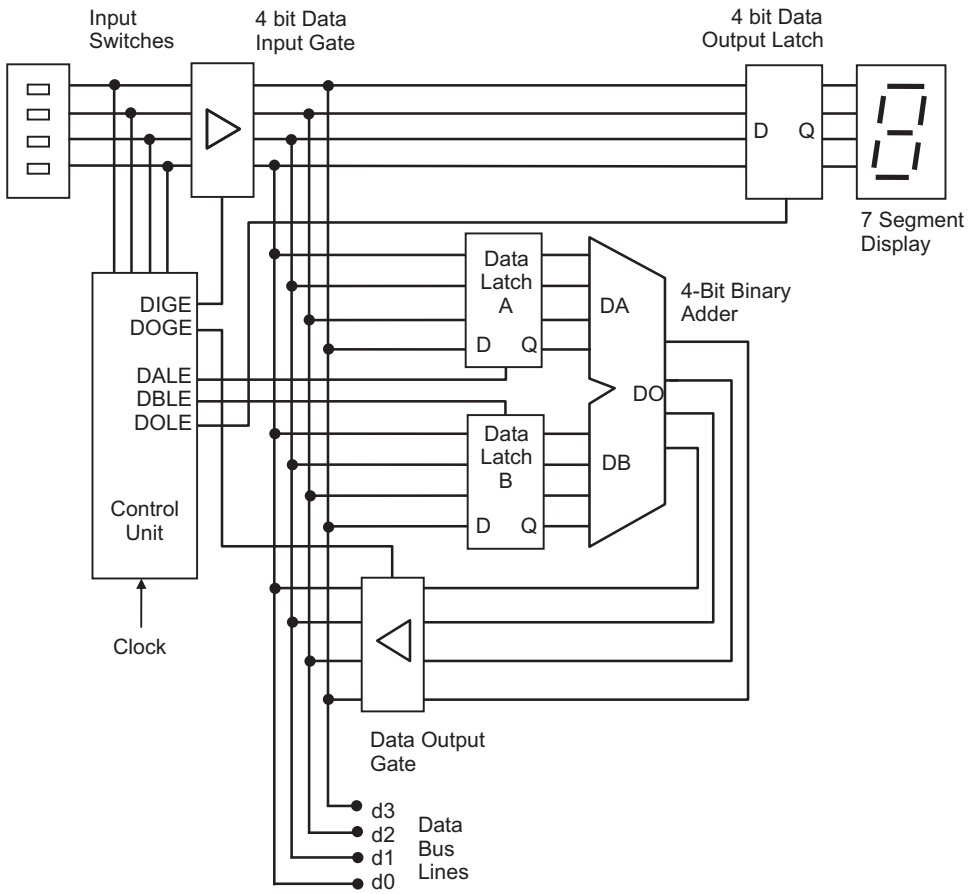


Figure B.12
Four-bit data system

only one set of data must be allowed on the bus at any one time, so only one set of gates must be enabled at a time; otherwise *contention* occurs on the bus and the data is invalid. The data destination is determined by which set of latches is operated when data is on the bus. The gates (data switches) and latches (data stores) must therefore be operated in the correct sequence by the control unit.

For the moment, we will assume that the control signals are generated manually using suitable switches or push buttons. The first number (6) is set up on the input switches, and the data input gate enable (DIGE) set active. This data word is now on the bus, and can be stored in latch A by pulsing the data A latch enable (DALE). Now the input switches are changed to generate the second number (3), which is to be added to the first. This value now appears on the bus and can be stored in latch B by activating DBLE (Table B.3).

Table B.3: Four-bit system operating sequence

Step	Input Switches (4-Bit Binary)	DIGE	DOGE	DALE	DBLE	DOLE	Display Hex 0—F	Data Bus (4-Bit Binary)	Operation
0	xxxx	0	0	0	0	1	X	xxxx	Ready for input
1	0110	0	0	0	0	1	X	xxxx	Set data input number A on switches
2	0110	1	0	0	0	1	6	0110	Enable data A onto bus by switching on input gates
3	0110	1	0	0-1-0	0	1	6	0110	Store data A in latch A by clocking it with a pulse
4	0110	0	0	0	0	1	X	xxxx	Disable input gates — no valid data on bus
5	0011	0	0	0	0	1	X	xxxx	Set data input number B on switches
6	0011	1	0	0	0	1	3	0011	Enable data B onto bus by switching on input gates
7	0011	1	0	0	0-1-0	1	3	0011	Store data B in latch B by clocking it with a pulse
8	0011	0	0	0	0	1	X	xxxx	Disable input gates — no valid data on bus
9	xxxx	0	1	0	0	1	9	1001	Enable result from ALU onto bus
10	xxxx	0	1	0	0	0	9	1001	Store result in output latch by clocking it with a pulse
11	xxxx	0	0	0	0	0	9	xxxx	Result displayed — ready for next input

DIGE: data input gate enable; DOGE: data output gate enable; DALE: data A latch enable; DBLE: data B latch enable; DOLE: data output latch enable.

With the numbers stored at the outputs of the latches DA and DB, the result appears at the output of the binary adder, DO. If the data output gate is enabled (DOGE), the result will appear on the bus. However, the data input gate *must* be disabled first, so that there is no conflict on the bus. The result (9) can then be stored and displayed by operating the data output latch enable (DOLE).

If this operating sequence can be automated, we are on the way to making a microcontroller. The binary operating sequence produced by the control unit must be recorded and played back in some way. This can be done by storing it in a read-only memory (ROM) memory block, along with the data to be input at the switches. Combining the 'instruction codes' (control switch operations) with the 'operands' (input data) gives us the simple 'machine code program' as seen in Table B.4.

The program has three instructions, of 9 bits in length. The instruction/operation code (op-code) is the first 5 bits, and the operand (data) the last 4. The control block needs to be designed such that the op-codes are generated on the control lines in the right sequence, and the data is connected in place of the switch inputs. This could be achieved by addressing the program ROM using a counter so that the control codes are output in turn. A clock signal will drive the system along.

The next step would be to replace the binary adder with a block that could also subtract and carry out logical operations such as increment, shift, AND, OR and so on. Different instruction codes would then set up the circuit to carry out all the required operations. More latches could be added, forming registers within the processor. Better input and output devices such as a keypad and multi-digit display would then give a usable system, as outlined in Chapter 1. The means to program the ROM (a development system) completes our simple processor system. This is how early calculator chips were developed, leading to microprocessors and microcontrollers.

The system outlined above could be implemented in hardware, but the discrete components used may not now be easily obtainable. Since it is designed purely to illustrate the principles of data system operation using a shared data bus, a simulation is probably more useful. A schematic for the 4-bit processing system is shown in Figure B.13 and the design file can be downloaded from www.picmicros.org.uk.

The key component is the ALU 74LS181 (U1), which can operate on pairs of 4-bit binary numbers applied at inputs A0−A3 and B0−B3. The operation can be selected via inputs

Table B.4: Four-bit system 'machine code program'

'Instruction' Code	'Operand'	Hex 'Program'	Operation
1 0101	0110	156	Input and Latch Data A
1 0011	0011	133	Input and Latch Data B
0 1001	0000	090	Latch and Display Result

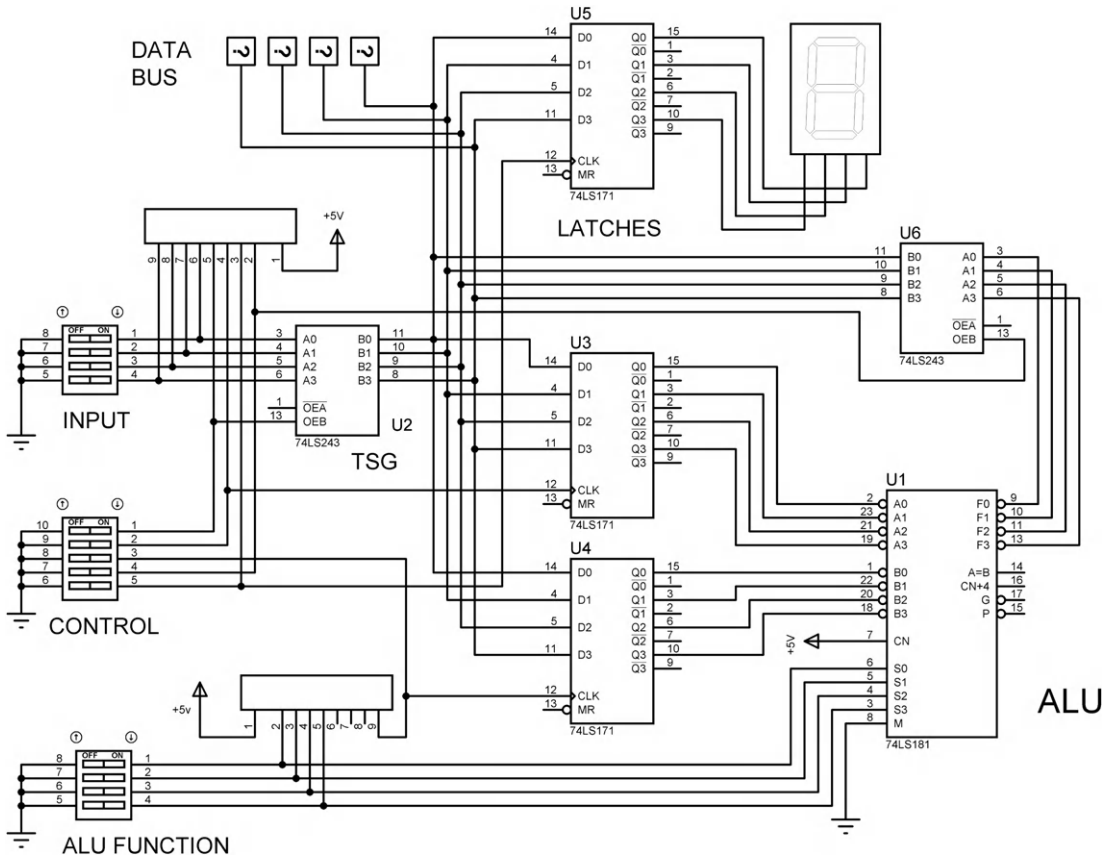


Figure B.13
ISIS schematic of 4-bit data system

S0−S3; to perform an arithmetic addition, these are set to 1001 on the switches. Initially, the control switches should all be switched on, setting all control inputs low to disable the gates and latches.

To start the sequence, the first data is set on the input switches, and allowed onto the data bus via the gates in U2 when enabled via OEB (control switch 1 off). The first nibble (4-bit BCD number) can then be stored in latches U3 by toggling control switch 2 to generate a positive pulse at CLK. The input is then changed and the second nibble stored in U4 by pulsing control switch 3. The sum of these two numbers then appears at the output of the ALU. Switch off the input gates (control switch 1 on) and enable the output gates in U6 (control switch 4 off). The result is now on the bus, and can be stored in the display latches (U5) by pulsing control switch 5 high. The next pair of numbers can now be processed while the previous result is displayed.

APPENDIX C

Digital Systems

Chapter Outline

The basic set of digital devices described in Appendix B is enough to build working data systems; these can be combined into circuit blocks which can in turn be incorporated into more complex digital systems. Some of these system blocks can be built from or supplied as discrete small- and medium-scale integrated circuits, but are now more commonly found integrated within microprocessor system chips and microcontrollers (MCUs). Proteus VSM is ideal for experimenting with these circuits, and no programming is required to test the circuits. Apart from supplying essential background in digital systems, the main purpose of this appendix is to support the interpretation of the hardware diagrams in the PIC® data sheet.

C.1. Encoder and Decoder

A digital encoder is a device that has a number of separate inputs and a binary output, which generates a binary code corresponding to the numbered input that is activated. Therefore, a 3-bit encoder has eight inputs and three outputs. If, for example, input number 4 is set active (usually low), the binary code for 4 (100) is output.

A decoder carries out the inverse logical operation – a binary input code activates the corresponding output. The 3-bit decoder therefore has three inputs and eight outputs. Thus, if the binary code for 5 (101) is input, output 5 of the decoder goes active (low).

An encoder and decoder can be used to operate a keypad, providing a neat example of how they work. This hardware interface reduces the number of input/output (I/O) lines needed to connect

the keypad to a microcontroller, and generates the keycode in hardware. The keypad consists of a set of switches connected in a two-dimensional array, such as that found on a phone or calculator. A simple decimal keypad has 12 keys, 0–9, hash (#) and star (*). A hexadecimal keypad has 16 keys, 0–F, and this type is used in Figure C.1. A calculator keypad has additional keys for the calculator function, and can be scanned in the same way. A standard computer keyboard operates in a similar way, with the keycode transmitted to the main processor in serial form.

A 2-bit decoder has its outputs connected to the four rows of the hex keypad, and a 2-bit encoder receives the input from each column. A key will be detected as a combination of the row select code and column detect code, a total of 4 bits. An additional encoder output indicates when a key has not been pressed.

The four select lines output from the row decoder are normally high. When a binary input code is applied, the corresponding row line goes low (Figure C.1b). A 2-bit binary counter (consisting of two T-type flip-flops; see Section C.5) is used to drive the row decoder, as this will generate each row select code in turn. If a key on the active row is pressed, this low bit will be detected on the column line. These are also normally held high (via pull-up resistors) and are connected to a column encoder. This generates a binary code (Figure C.1c), which corresponds to the input that has been taken low by connection to the active row.

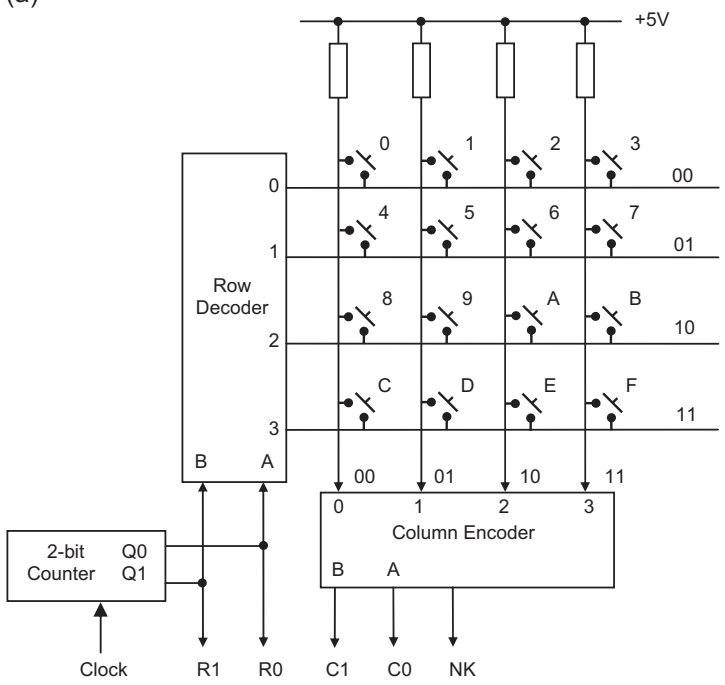
Thus, the combination of the row select binary code (R1, R0) and the column detect binary code (C1, C0) will give the number of the key that has been pressed. For instance, if key 9 is pressed, row 2 will go low when the input code is 10. This will take column 1 low, which will give the column code 01 out. The complete code is then 1001 in binary (9_{10}). A microcontroller connected to the keypad interface can be programmed to generate four clock pulses to the row counter and read the inputs after each pulse, to complete a scan of the keypad. Switch debouncing can be incorporated by reading the inputs after a suitable delay.

Encoders and decoders are combinational logic circuits, which can be designed with any number of code bits, n, and 2^n selected lines. Discrete 3-bit encoders and decoders are available as medium-scale integration (MSI) chips, while the 2-bit encoders needed here would have to be designed from discrete gates. This kind of discrete logic design is covered in numerous standard textbooks. In this case, the counter, decoder and encoder could be designed as a complete subcircuit fitted adjacent to the keypad, perhaps using a programmable logic device (PLD).

C.2. Multiplexer, Demultiplexer and Buffer

A multiplexer is essentially an electronic changeover switch, using a tri-state gate (TSG), which can select data from alternative sources within a data system. This allows two or more

(a)



(b)

Inputs		Outputs			
B	A	0	1	2	3
0	0	0	1	1	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0

(c)

Inputs				Outputs		
0	1	2	3	B	A	G
1	1	1	1	x	x	1
0	1	1	1	0	0	0
1	0	1	1	0	1	0
1	1	0	1	1	0	0
1	1	1	0	1	1	0

(d)

Key Press	Row Decoder Input	Row Decoder Output	Column Encoder Input	Column Encoder Output	
	R1 R0	3 2 1 0	3 2 1 0	C1 C0	NK
None	X X	X X X X	1 1 1 1	X X	1
0	0 0	1 1 1 0	1 1 1 0	0 0	0
1	0 0	1 1 1 0	1 1 0 1	0 1	0
2	0 0	1 1 1 0	1 0 1 1	1 0	0
3	0 0	1 1 1 0	0 1 1 1	1 1	0
4	0 1	1 1 0 1	1 1 1 0	0 0	0
5	0 1	1 1 0 1	1 1 0 1	0 1	0
6	0 1	1 1 0 1	1 0 1 1	1 0	0
7	0 1	1 1 0 1	0 1 1 1	1 1	0
8	1 0	1 0 1 1	1 1 1 0	0 0	0
9	1 0	1 0 1 1	1 1 0 1	0 1	0
A	1 0	1 0 1 1	1 0 1 1	1 0	0
B	1 0	1 0 1 1	0 1 1 1	1 1	0
C	1 1	0 1 1 1	1 1 1 0	0 0	0
D	1 1	0 1 1 1	1 1 0 1	0 1	0
E	1 1	0 1 1 1	1 0 1 1	1 0	0
F	1 1	0 1 1 1	0 1 1 1	1 1	0

Figure C.1

Keypad scanning using an encoder and decoder: (a) decimal keypad operation; (b) 2-bit decoder logic table; (c) 2-bit encoder logic table; (d) keypad logic table

different system devices to share a common signal path (bus line) at different times. In Figure C.2(a), input 1 or 2 is selected by the logic state of the select input. The logic inverter ensures that only one of the TSGs is enabled at a time. Conversely, a demultiplexer (Figure C.2b) splits the signal using the same devices. That is, it can pass data to alternative destinations from the bus. A controller block is needed to provide these coordinated signals.

The bidirectional buffer (Figure C.2c) is used to allow data to pass in one direction at a time along a data path, for example, on a data bus or serial link. To achieve this, the TSGs are connected nose to tail, and are only enabled one at a time, as in the multiplexer. When the

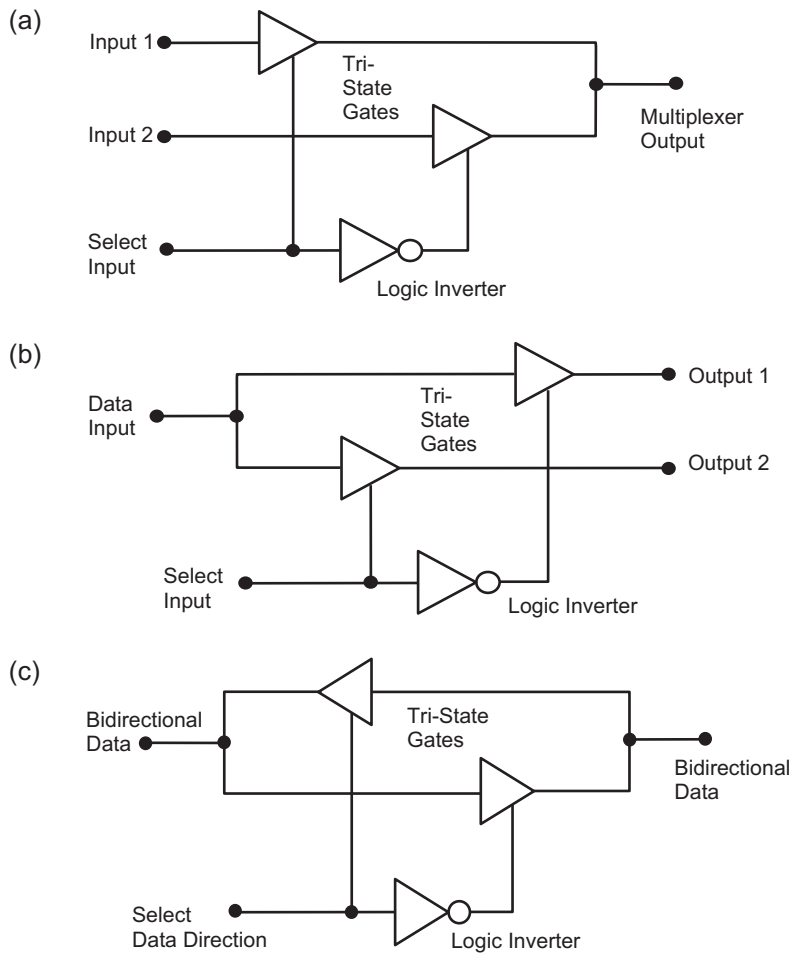


Figure C.2
Multiplexer, demultiplexer and bidirectional buffer: (a) 1-bit multiplexer; (b) 1-bit demultiplexer; (c) bidirectional data buffer

control input is low, the data is enabled through from left to right, and when high, from right to left.

These subcircuits can all be constructed from the same set of gates: two TSGs and a logic inverter. All are important for the operation of bus systems, as outlined in Appendix B. Any data source connected to a common (bus) line needs to be isolated via a TSG. Data receivers do not need isolation, as inputs (data latches) are by definition high impedance (hi-Z). Thus, only one data source should be enabled at a time, whereas the data can be received by multiple devices at once, if required.

C.3. Registers and Memory

We have seen previously how a 1-bit data latch works. If the bidirectional data buffer (Figure C.2c) is added, data can be read from a data line into the latch, or written to the data line from it, depending on the data direction selected. We then have a register bit store. In Figure C.3(a), the Data In/Out line can be connected to the D input or Q output, depending on

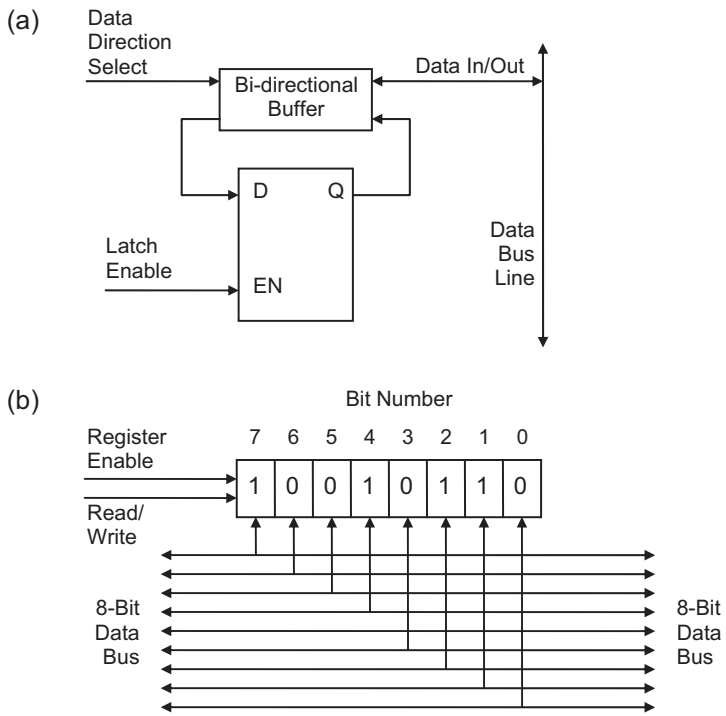


Figure C.3
Register operation: (a) data register bit operation; (b) 8-bit data register operation

the state of the Data Direction Select. If data is to be stored by the latch from the data line, Latch Enable is activated at the appropriate time.

If a set of these register elements is used together, a data word can be stored. The usual data word size is 8 bits (1 byte), with most systems handling data in multiples of 8 bits. An 8-bit register, consisting of eight data latches, is shown in Figure C.3(b). The Register Enable and Read/Write (data direction select) lines are connected to all the register bits, which operate simultaneously, to read and write data to and from the 8-bit data bus.

C.4. Memory Address Decoding

A static random access memory (RAM) location operates in a similar way to a register. The memory device stores a block of 8-bit data bytes, which are accessed as numbered locations (Figure C.4). Each location consists of eight data latches, which are loaded and read together. A read operation is shown in the diagram; the data is being output from the selected location. A 3-bit code is needed to select one of eight locations in the memory block, using an internal address decoder to generate the location select signal. The selected data byte is enabled out via an output buffer, which allows the memory device to be electrically disconnected when another device wants to use the data bus.

The number of locations in a memory device can be calculated from the number of address pins on the chip. In the example above, a 3-bit address provides eight unique location addresses (000_2 to 111_2). This number of locations can be calculated directly as $2^3 = (2 \times 2 \times 2) = 8$.

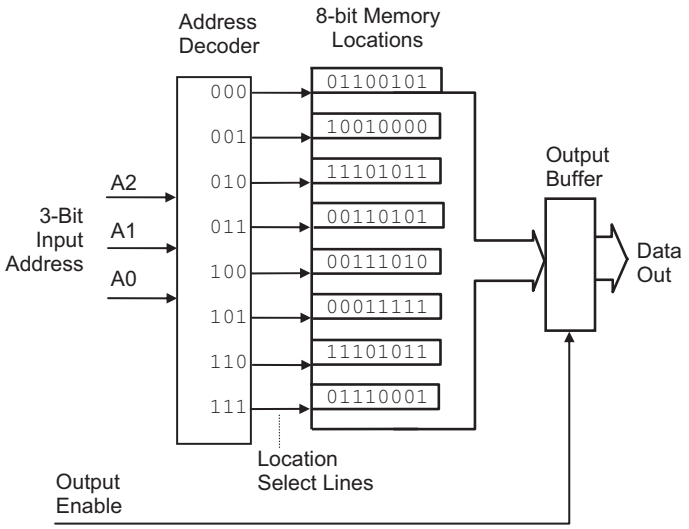


Figure C.4
Memory device operation

Table C.1: Common memory sizes

Address Line	Location (1 Byte Each)	Memory Size
8	$2^8 = 256$	256 bytes
10	$2^{10} = 1024$	1 kb (kilobyte)
16	$2^{16} = 65536$	64 kb
20	$2^{20} = 1048576$	1 Mb (megabyte)
30	$2^{30} = 1073741824$	1 Gb (gigabyte)

The number of locations is therefore calculated as 2 raised to the power of the number of address lines. Some useful values are listed in Table C.1.

C.5. Counters and Timers

A counter/timer register can count the number of digital pulses applied to its input. If a clock signal of known frequency is used, it becomes a timer, because the duration of the count is equal to the count value multiplied by the clock period. Like the data register, the counter/timer register is made from bistable units, but connected in 'toggle' mode, so that each stage drives the next. Each outputs one pulse for every two pulses that are input, so the output pulse frequency is half the input frequency (Figure C.5a). The counter/timer register can therefore be viewed as a binary counter or frequency divider, depending on the application.

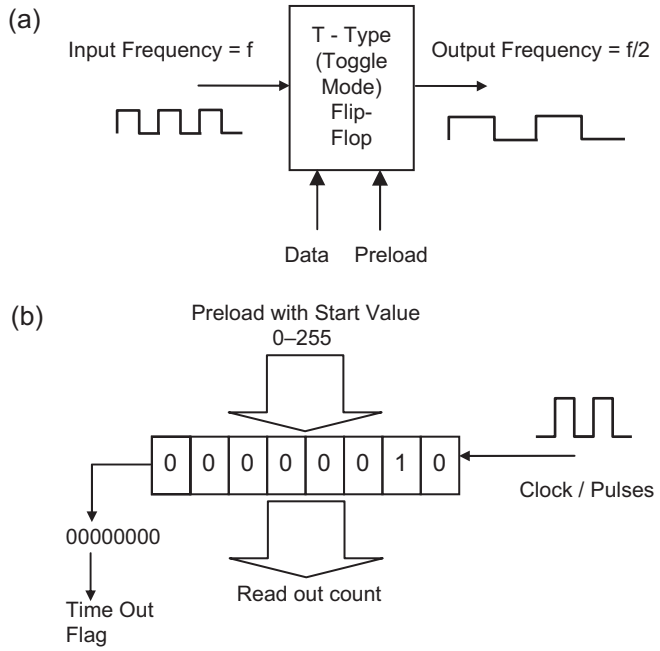


Figure C.5
Counter/timer register operation: (a) counter stage; (b) 8-bit counter register

Figure C.5(b) shows an 8-bit counter/timer, with the input to the least significant bit (LSB) at the right. The binary count seen at the outputs increments each time the LSB is pulsed. Two pulses have been applied, so the counter shows binary 2. After 255 pulses have been applied, the counter will 'roll over' from 11111111 to 00000000 on the next pulse. A signal is output to indicate this, which can be used as a carry-out in counting operations or indicate time out when timing. In a microprocessor system, the time-out signal typically sets a bit in the status register to record this event. Optionally, an interrupt signal may be generated, which forces the processor to carry out an interrupt service routine to process the time-out event.

If the clock pulse frequency is, say, 1 MHz (1 megahertz, 10^6 cycles per second), the period will be 1 μs (1 microsecond, 10^{-6} seconds), and the counter will generate a time-out signal every 256 μs. If the counter is preloaded with a suitable number, we can make it time out after some other number of input pulses. For example, if preloaded with a count of 56, it will time out after 200 μs. In this way, arbitrary time intervals can be generated. In conventional microprocessor systems, the I/O ports often contain timers that the processor uses for timing operations. All PICs have at least one 8-bit counter/timer, with a prescaler that divides the input frequency by a factor of between 2 and 256 in order to extend its range. Many also have 16-bit counters, which allow longer intervals to be generated without a prescaler, and a more accurate count to be recorded. PIC timer/counters are explained in more detail in Chapter 6.

C.6. Serial and Shift Registers

The general purpose data register, as described in Section C.3, is loaded and read in parallel. A shift register is designed to be loaded or the data read out in serial form. It consists of a set of data latches that are connected so that a data bit fed into one end can be moved from one stage to the next, under the control of a clock signal. An 8-bit shift register can therefore store a data byte that is read in one bit at a time from a single data line. The data can then be shifted out again, one bit at a time, or read in parallel. Alternatively, the register could be loaded in parallel and the data shifted out onto a serial output line.

In Figure C.6(a), the 8-bit shift register is fed data from the right. The shift clock has to operate at the same rate as the data arrives, so that the register samples the data at the right time at the serial data input. This means that there must be standard clock rates used to set up the shift register in advance. As each bit is read in, the preceding bits are shifted left to allow the next bit into the LSB. The timing diagram shows the data being sampled and shifted on the falling clock edge – note that only the state of the input at the sampling instant is registered, so the short negative going pulse between bits 6 and 7 is ignored. This type of register is used in microcontroller serial ports, where data is sent or received in serial form (see Chapter 12). In the PC, this could be the modem or network port, the keyboard input or visual display unit (VDU) output.

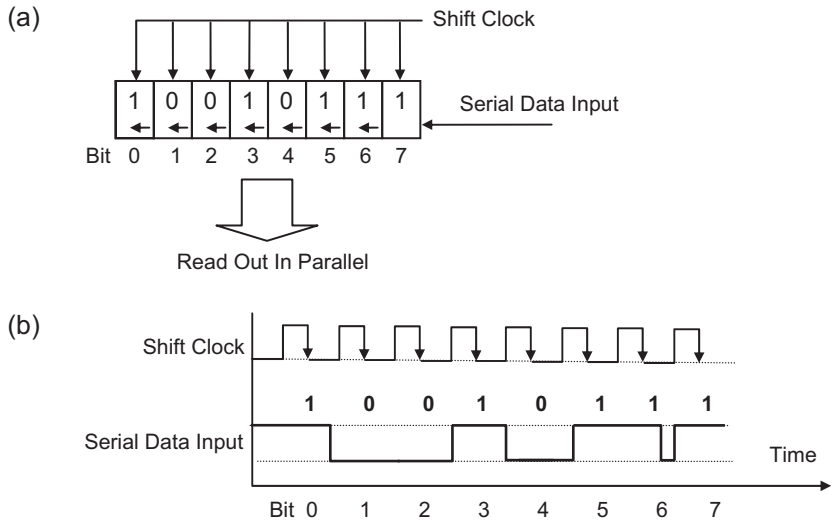


Figure C.6
Shift register operation: (a) shift register; (b) shift register signals

C.7. Arithmetic and Logic Unit

The main function of any microsystem is to process data, for example, to add two numbers together. The arithmetic and logic unit (ALU), shown in Figure C.7, is therefore an essential feature of any microprocessor or microcontroller. A binary adder block has already been described in Appendix B, but this would be just one of the functions of an ALU. The ALU takes two data words as inputs and combines them together by adding, subtracting, comparing and carrying out logical operations such as AND, OR, NOT and XOR. The operation to be carried out is determined by function select inputs. These, in turn, are derived from the

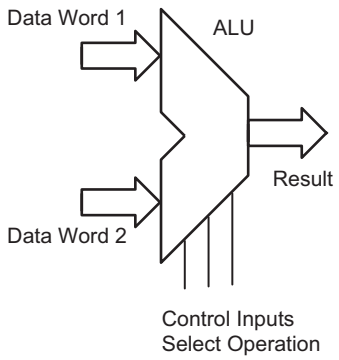


Figure C.7
Arithmetic and logic unit

instruction code in the program being executed in the processor. The block arrows used in the diagram indicate the parallel data paths (8-bit in PIC16) which carry the operands to the ALU, and the result away to the next stage. A set of data registers that store the operands is usually associated with the ALU. In the PIC16, the working register (W) always receives the result, and provides one of the operands. In other processors, multiple data registers can provide the operands and store the result.

C.8. Processor Control

The instruction decoder is a logic circuit in the central processing unit (CPU), which receives the instruction codes from the program to control the sequence of operations. The decoder output lines, which are connected to the registers, ALU, gates and other control logic, are set up for a particular instruction to be carried out (e.g. add two data bytes). The processor control block (Figure C.8) also includes timing control and other logic to manage the processor operations. The clock signal drives the sequence of events, so that after a certain number of clock cycles, the results of the instruction are generated and stored in a suitable register or back in memory.

C.9. CPU System Operation

Although we are mainly concerned with microcontroller architecture, it is worth looking briefly at memory and I/O access in a conventional system, because it explains the process that occurs within the microcontroller chip, and is important for an overview of microprocessor systems. It is a logical extension of address decoding within each memory chip.

The typical microprocessor system consists of memory and I/O devices connected to the CPU by a shared data bus. Only one peripheral chip can use the data bus at any one time, so a system of chip selection is needed whereby the processor can communicate with a particular device. Figure C.9(a) shows the system connections that allow the CPU to read and write

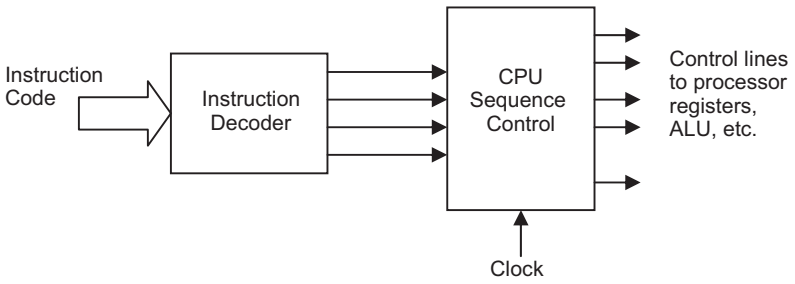
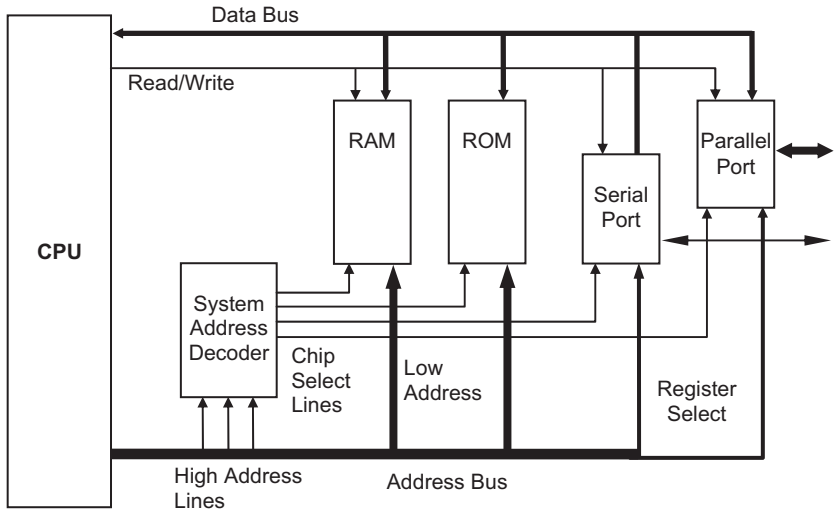
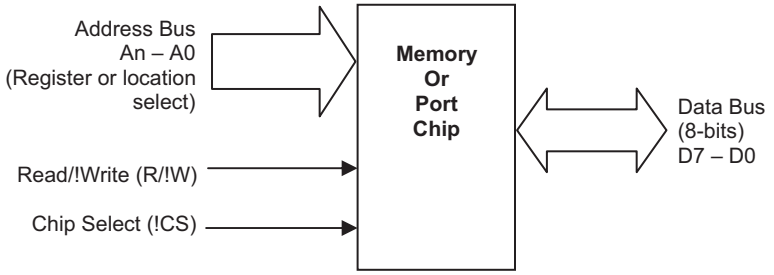


Figure C.8
CPU control logic

(a)



(b)



(c)

Address Range					
Lowest Address	**Highest Address**	**High Bits**	**Decoder Logic**	**Number of Locations**	**Device**
0000	3FFF	00XX	!A15	8000_{16} = 32768_{10}	RAM
4000	7FFF	01XX			
8000	801F	100X	A15.!A14.!A13	$20_{16} = 32_{10}$	Parallel Port Registers
A000	A008	101X	A15.!A14.A13	8	Serial Port Registers
C000	FFFF	11XX	A15.A14	$4000_{16} = 16384_{10}$	ROM

Figure C.9
Microprocessor system operation: (a) block diagram; (b) memory or port chip connections; (c) memory map

data to and from the memory and I/O devices, using an address bus and other control lines. The peripheral chips each have a set of registers or memory locations, and require the following connections (Figure C.9b):

- Data Bus: A set of bidirectional data lines, connected to the CPU system data bus, which feed the data in and out of the memory locations or registers in the chip. In an 8-bit system, they are numbered D0–D7.

- Address Bus: A set of input location or register select lines connected to the internal address decoder; generated in the CPU by the program counter or derived from an instruction operand. A 16-bit address bus, A0−A15, can access 64k memory.
- Read/!Write: Control input selects the data direction at the input bidirectional buffer for the location selected. Generated by the CPU depending on whether a read or write to the location is required. Usually, 1 = read, 0 = write.
- !Chip Select: Active low control input, which enables the output TSGs for a read, and the input latches for a write.

We will assume that in this system the CPU has a 16-bit address bus, allowing 64k (2^{16}) locations to be addressed. The address decoder generates chip select signals derived from the three most significant address lines (A15, A14, A13). It operates according to the table shown in Figure C.9(c). The decoder logic is fairly simple in this case, and can be implemented with a few discrete gates or a small PLD. In fact, the RAM chip select (!A15 = not A15 = low) can simply be connected directly to the address line A15, with no logic required, since the RAM is selected for all addresses with the most significant bit (MSB) = 0.

All data transfers are carried out in the same way, but let us assume that the CPU is reading a program instruction from read-only memory (ROM). The CPU program counter contains the address of the instruction − this is output as a binary code on the address bus. The system address decoder takes, in this system, the 3-bit code on the most significant address lines (ROM = 11X) and activates the ROM chip select line. The lower order address lines are used, as described in Section C.4, to select the required location within the chip. Thus, the location select is a two-stage process, with external (system) and internal (chip) decoding of the address.

When the location has been selected, the data stored there can be read (or written) via the data bus according to the setting of the read/not write (R/!W) line generated by the CPU. To read from memory, the TSGs at the output of the selected device, say ROM, are enabled, while all others connected to the bus are disabled, allowing the ROM data onto the bus lines. The data can then be read off the bus by the CPU, and copied into a suitable register (instruction register in this case). Note that ROM cannot be written and therefore does not need the R/W line connected. The I/O ports only have a few addressable locations, their control registers and a data register for each channel, so only a few of the address lines are needed for these devices.

The design of the decoding logic determines the allocation of the memory and I/O locations to specific ranges of addresses. In this case, the memory space is divided into eight parts by the upper three bits of the 16-bit address, and each chip is assigned to one or more ranges. Notice that not all the available addresses in some ranges are used, especially in those assigned to the ports, as they have only a small number of registers. On the other hand, RAM fully occupies its available space (32k) across the first four of the eight blocks.

Unlike the microcontroller, the CPU system can be tailored to a specific application by the hardware designer, with just the right amount of memory and I/O. An example of this type of system is shown in more detail in Figure 14.5, the block diagram of an M68000 microprocessor system, which has a 24-bit address bus and 16-bit data bus, with 64k of RAM and 64k ROM. This system has linear address space, where all locations are accessed using a continuous addressing sequence.

A similar process is used to select RAM locations in the PIC16, except that the bank selection bits have to be adjusted explicitly in the program using the BANKSEL command. This means the RAM address space is not linear, but paged. Similarly, program ROM is divided into pages of 256 instructions, with PCLATH controlling page selection.

C.10. PIC16 MCU Operation

Figure C.10 reproduces the internal block diagram of the PIC 16F84A, which has the same core as all PIC16 chips, but is the simplest. It incorporates many of the hardware concepts described in this appendix. The data paths between each block show the data word size and the possible data transfer routes. The data bus itself is 8 bits wide, the address bus (program counter output) 13 bits. The control lines that are used by the instruction decoder and control block to manage the data transfers are not shown, because it would make the diagram too complicated. They connect to all parts of the processor, enabling data outputs via TSGs at the source end, and operating data latches at the receiving end, for all data transfers.

- Memory Blocks: The MCU contains a flash ROM program memory block (1k $\times$ 14 bits), RAM file register block (68 $\times$ 8 bits) and EEPROM (64 $\times$ 8 bits). The program counter generates the program memory address (13 bits). The instruction register receives the instruction code from the selected program memory address. The RAM address multiplexer (Addr Mux) can source the RAM address from the instruction literal or file select register (FSR).
- ALU: The ALU multiplexer (MUX) selects the data source as an instruction literal or data bus. The ALU result is stored in the working register (W) or a file register. The result affects individual bits in the status register, e.g. zero flag.
- Ports: The ports A and B are located in the file registers, addresses 05 and 06. The data direction registers, which control the bidirectional multiplexers at the outputs, are at addresses 85 and 86 in bank 1. In other PIC chips, serial port data and control registers are also in the RAM block.
- Timers: TMR0 is an 8-bit timer/counter (RAM address 01), which can be clocked from pin RA0 (counter) or internally from the instruction clock (timer). The control block also contains system timers (power-up, oscillator and watchdog).

For more details on the operation of the PIC16 MCU, refer to Chapter 5.

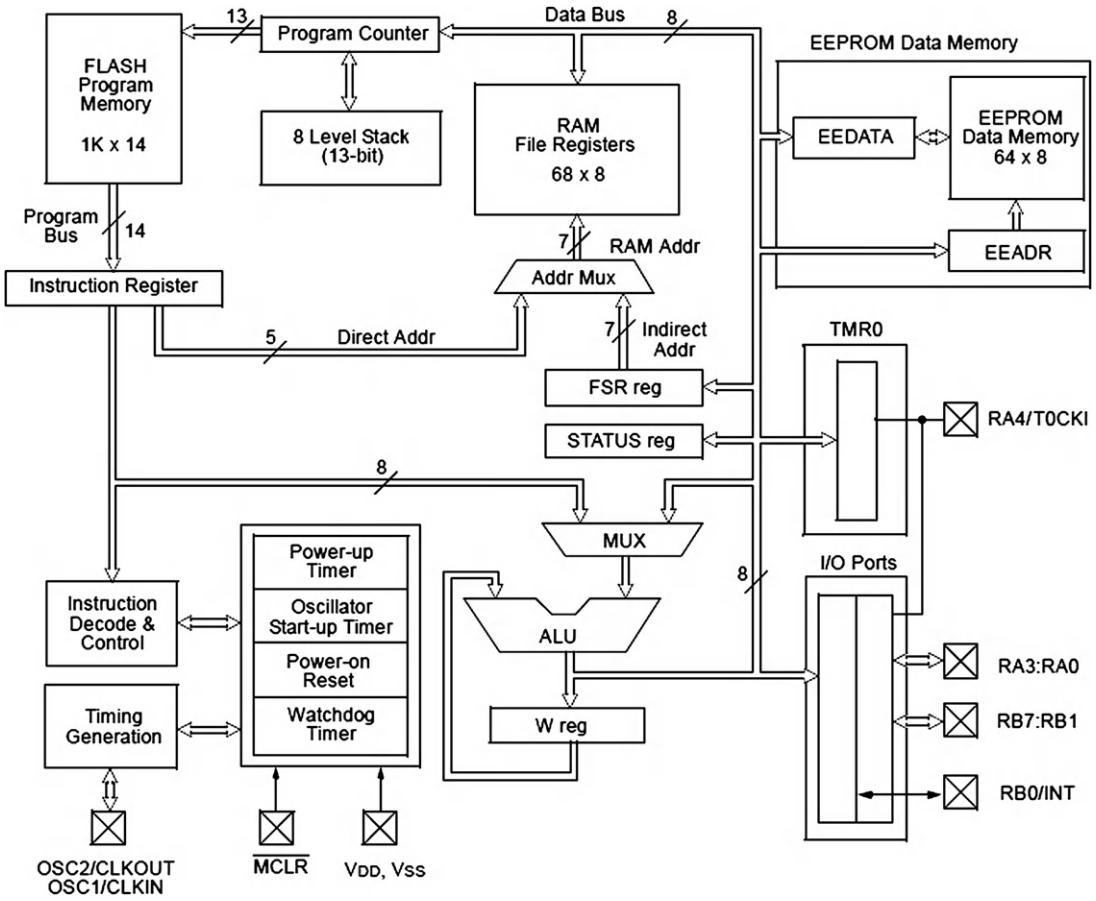


Figure C.10
PIC 16F84A internal architecture (courtesy of Microchip® Technology Inc.)

APPENDIX D

Dizi84 Demo Board

Chapter Outline

This appendix will outline the procedure for producing a demonstration board using stripboard, which will allow simple PIC® programs to be tested in hardware. The Dizi84 (16F84A with DIsplay, buZzer & Interrupt) board has push-button and switched inputs, a seven-segment light-emitting diode (LED) display and a buzzer. This hardware was introduced in Chapter 10, and here the development of the board will be described in more detail, with an application that uses most of its features – an electronic lock. The finished hardware is shown in Figure 10.11.

An in-circuit programming connector is not provided – the chip must be programmed separately and then physically transferred to the target system. The enhanced Dizi84 board described in this appendix incorporates an on-board battery supply, a finger pot to provide an analogue input and hardware switch debouncing to improve the reliability of the push-button operation. The demo programs in Chapter 10 can be tested on this hardware.

D.1. Circuit Design

The circuit is to be built on a 100 mm × 100 mm section of stripboard, a prototyping method that produces a reliable circuit with no special tools or design software. It has parallel copper tracks for making the component connections on a standard 0.1 inch grid. The design includes a 2 × 1.5 V battery pack on the board, so the circuit operates at a voltage of 2–3 V. The power is switched on via a non-latching push button so that it cannot be left on accidentally, and thereby exhaust the batteries; it must be held on manually while the circuit is in operation.

A circuit diagram is shown in Figure D.1. A seven-segment display allows decimal and hexadecimal digits to be displayed. A range of applications with a numerical output can be demonstrated, for example, the electronic DICE in Chapter 10 and the LOCK application detailed below (Section D.5). Port B has eight input/output (I/O) bits; seven are used for the

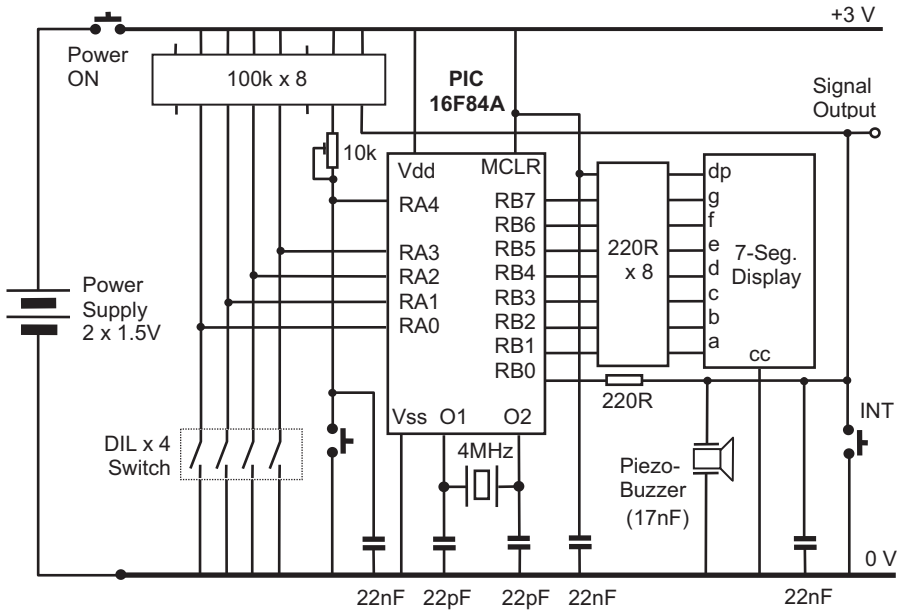


Figure D.1
Dizi84 test board circuit diagram

LED display, leaving RB0 free for use as both an audio output and a push-button (interrupt) input. A small audio transducer, a piezoelectric buzzer, provides a simple and effective way of monitoring audio output frequencies or generating status signals.

Port A has five pins. RA4 can be used as an input to the TMR0 counter, so this was allocated as another push-button input. A 4-bit switch bank is useful for setting coded inputs, for example binary coded decimal (BCD) inputs for the LOCK application, so RA0 to RA3 were allocated for this purpose. The switch and push-button inputs have 100k pull-up resistors, and the push buttons have 22 μF debouncing capacitors. These are fitted to such inputs because, when a switch closes, the metal contacts can bounce open again several times before finally closing. The CR network prevents multiple transitions on the logic signal input to which the switch is connected, because after the capacitor has quickly discharged on the first contact, it must recharge via the 100k. This takes a relatively long time, preventing the voltage from jumping back to a high level when the contacts reopen. By the same process, the CR network also ensures a smooth transition from low to high logic levels when the push button is released. In addition, the CR network on RA4 was modified with a potentiometer (pot) connected as a variable resistance in series with the 100k, so that it could also be used to demonstrate analogue measurement using a digital input.

The layout of a printed circuit board (PCB) or prototype circuit is derived from the circuit diagram. The pins on dual in-line (DIL) chips are spaced 0.1 inch apart, so the circuits must be

laid out on a 0.1 inch grid. When the pin-out of each component has been established by reference to the data sheet or catalogue information, the connections can be mapped out on a square grid on paper. Alternatively, it is not too difficult to use the basic drawing tools in a word processor to do the same job. The layout and parts list for the Dizi84 board is shown in Figure D.2.

The board is viewed from the front (component) side, with the tracks on the back shown vertically. The chips are all placed in the same orientation, so that pin 1 is bottom left. The integrated circuits (ICs) must be fitted across the tracks, so that their pin connections can be separated by cutting the track between each pair of pins on the DIL connector. The PIC chip must be fitted in a socket so that it can be removed for programming.

Horizontal links of tinned copper wire (TCW) complete the connections required. A solder joint is shown as a solid black dot. The broader solid lines indicate a continuous link across the tracks on the back of the board, where a set of adjacent tracks must be connected. Where required, the tracks are cut with a hand drill, as shown on the layout (where visible).

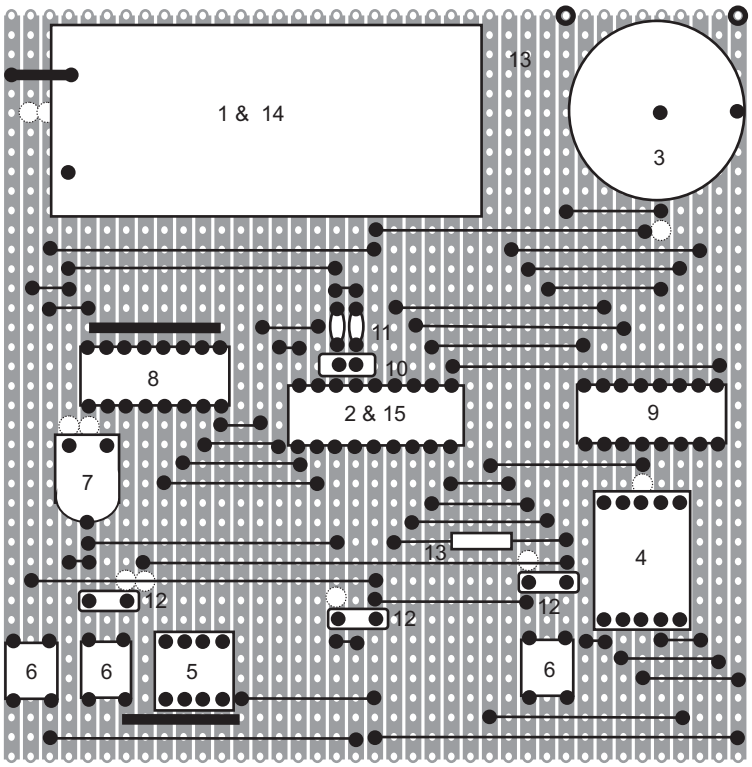
A computer drawing method allows component positioning to be easily adjusted so that the minimum area of stripboard is used. However, with experience, the circuit may be built directly onto the board without necessarily drawing the layout, perhaps with some additional wastage of board area.

The components on the board are numbered according to the parts list in Figure D.2(b). Ideally, this should specify the exact component to be ordered from a suitable supplier, with a catalogue number. If a particular part becomes unavailable, the layout might be affected, as the pin-out might be different. One advantage of using electronic computer-aided design (ECAD) to create a schematic and PCB layout is that the parts list is generated automatically.

D.2. Construction and Testing

When the layout has been checked against the circuit diagram, the main components can be inserted in the board and retained by, if necessary, slightly bending the pins outwards. All the pins should then be soldered to the tracks using the minimum amount of solder necessary, while ensuring that the joint is covered evenly with no cavities. At the same time, the soldering iron should be in contact with the joint for the shortest possible time, to avoid component overheating. The TCW links can also be retained before soldering by bending the ends towards each other. If a very neat job is required, one end can be soldered and the link stretched slightly before fixing the other end, to ensure that the link has no kinks in it, and that adjacent links do not touch; insulated TCW may be used on longer links if necessary. The tracks should then be cut where necessary, and the track side brushed with a small stiff brush to clear any debris. Rake lightly between the tracks with a small screwdriver or knife to ensure that there are no short-circuits left between adjacent tracks and solder joints.

(a)



(b)

Layout	*Description*	
1	Battery Box, 2 x AA cells, PCB mounting	
2	Microcontroller, PIC 16LF84-04	
3	Piezo Electric Sounder, PCB mounting	
4	Seven-Segment LED Display, 0.5 inch, Common Cathode	
5	Piano DIL Switch, 4-way	
6	Tactile Switch, PCB mounting (3 of)	
	Caps for above:	Red
		Blue
		Yellow
7	Preset Potentiometer, 10k, H-mount	
8	DIL Isolated Resistor Network, 100k x 8	
9	DIL Isolated Resistor Network, 220R x 8	
10	Quartz Crystal, General Purpose, 4MHz	
11	Capacitor, 22pF, Ceramic (2 off)	
12	Capacitor, 22nF, Polyester (3 off)	
13	Stripboard, SRBP 3939 100mm x 100mm	
14	Batteries, 1.5V, size AA, Duracell (2 off)	
15	18-pin DIL IC Socket	

Figure D.2
Dizi84 prototype design: (a) stripboard layout; (b) parts list

Thoroughly reinspect the board for correct connections, and check that there is no remaining debris, solder splashes or whiskers, or dry joints. With the batteries not yet fitted, check with a multimeter that there is no short-circuit between the power supplies. Fit the batteries, but not the PIC chip, and hold down the power button. The display decimal point should light. Check the supply voltages on the supply tracks and PIC socket pins: Pin 5 = 0 V and Pin 14 = +3 V. Check that the voltages at the PIC inputs change correctly as the switches are toggled. A digital multimeter or an oscilloscope is required for this test, because of the high impedance of the pull-up resistors. Connect a temporary link between Pin 14 (+3 V) on the PIC IC socket and each PIC output in turn, RB0−RB7. The piezo-buzzer should produce an audible 'tick' and the LED segments should light.

To complete testing of the Dizi84 board, a program needs to be blown into the PIC that exercises all the hardware, while remaining as simple as possible so that there is no question of the software being faulty. A suitable program is listed as Program D.1, which allows functional checks to be carried out as specified in the test procedure. If faults are found, it is quite possible that there are still hardware faults on the board. Check also that all the tracks have been cut as required, and that all connections and components are correct.

The test program source code can be downloaded from www.picmicros.org.uk. A programmer module (e.g. PICSTART Plus) is needed to program the chip (see Chapter 4).

D.3. Analogue Conversion

Most PIC chips contain a 10-bit analogue-to-digital converter (ADC), which can be connected to a selected input. If this is not available (e.g. when using a 16F84A), an external CR network can provide an alternative, if the voltage measurement does not have to be too accurate.

The components connected to RA4 are shown in Figure D.3. The PIC chip is a complementary metal oxide semiconductor (CMOS) device, so the voltage level at which an input changes from logic 0 to 1, the threshold voltage, is around half the supply voltage, 1.5 V. The time taken to reach this level is estimated at 1.5 ms. This could be calculated more accurately from the formula for the charging of a capacitor, but as long as the circuit operation is consistent, it is not necessary for this application where the analogue pot is simply a convenient way of inputting decimal numbers.

The resistance, R, varies between 100k and 110k, depending on the position of the pot. The variation in the pot value will produce a corresponding variation in the rise time of the circuit. The rise time can be measured by discharging the capacitor and then counting while the voltage rises back towards the threshold. The capacitor is discharged by setting RA4 as an output and then setting the port data bit to zero. RA4 is then reconfigured as an input and checked at fixed time intervals while a register is incremented. The count is stopped when RA4 goes high.

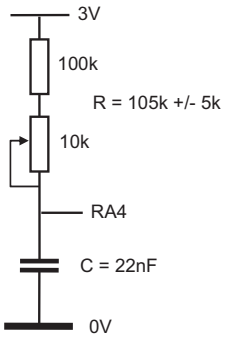
(a)

```
; diz1.asm

        PROCESSOR 16F84A

; Test DIZI hardware ......................

        GOTO    inter         ; jump over delay

; Delay Subroutine ........................

delay   MOVLW   0FF           ; Load FF
        MOVWF   0C            ; into counter
down    DECFSZ  0C            ; and decrement
        GOTO    down          ; until zero
        RETURN

; Check Interrupt Button ..................

inter   BTFSC   06,0          ; Test Button RB0
        GOTO   inter          ; until pressed

; Check Display ...........................

        MOVLW   00            ; Set PortB bits
        TRIS    06            ; as outputs
        MOVLW   0FF           ; Switch on all
        MOVWF   06            ; display segments

; Check Input Button ......................

input   BTFSC   05,4          ; Test Button RA4
        GOTO    input         ; until pressed

; Check DIP Switches and Buzzer ...........

again   MOVF    05,W          ; get DIL input &
        MOVWF   06            ; send to display
        RLF     06            ; rotate bits left

        BSF     06,0          ; set buzzer high
        CALL    delay         ; delay about 1ms
        BCF     06,0          ; reset buzzer low
        CALL    delay         ; delay about 1ms

        GOTO    again         ; and keep going..
        END                  ; End of code
```

(b)

Step	Test	Result
1	Power Button On	Decimal Point ON
2	Button B Pressed & Released	All display segments ON
3	Button A Pressed & Released	Buzzer sounds
4	Operate DIL Switches	Segments a,b,c,d change

Program D.1
Test program for Dizi84 board: (a) source code; (b) test procedure



Estimation of Charging Time

Capacitor value
= 22nF = 22 x 10^{-9} F
Resistor value with 10k pot
set midway = 100k + 5k
= 105k = 105 x 10^{3} Ω
∴ Network Time Constant = C.R
= 22 x 105 x 10^{6} = 2.31 ms
Estimated Charging Time
≈ 1.5 ms

Figure D.3
CR conversion network

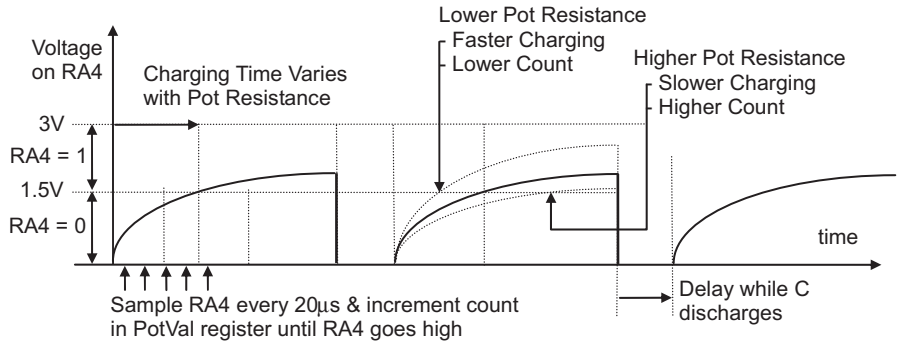
The waveform that will be seen at RA4 is illustrated in Figure D.4(a), and the CR ADC conversion process is outlined in Figure D.4(b). In the LOCK program (Program D.2), a counter register labeled PotVal is incremented, and RA4 checked, within a loop taking 20 μs to execute. An adjustable delay routine allows the timing to be modified to suit the application and CR component values.

The result of the process is that a count is obtained which represents the setting of the pot. This could be converted to a resistance value if required, but in the LOCK program, all we need is a variation in the displayed digit between 0 and 9, to allow the user to input a decimal combination. Therefore, the delay associated with the count was adjusted to give one decade on the display with one turn of the pot. Only the low four bits of the count were required, so any decade of values could be used. The upper end of the 4-bit range, hex numbers A to F, is displayed as '-'. These could be used as 'hidden' digits for extra security, if required.

D.4. EEPROM Storage

Non-volatile read and write memory is useful because data input by the user or acquired by the processor during its operation can be retained while the power is off. One application area is data security and encryption, while another is storage of frequency settings in digital tuners. The LOCK application illustrates this feature of the PIC 16F84A by using electrically erasable programmable read-only memory (EEPROM) to store a four-digit security code. The chip has 64 bytes of EEPROM, with addresses 00−3F. The memory is accessed via EEDATA and EEADR in the special function register (SFR) set. The EEPROM address is loaded into EEADR, and the data byte to be stored in EEDATA. A write initialization sequence is then executed to write the data to the EEPROM memory, using EECON1 and EECON2 page 1 SFRs. The sequence is designed to reduce the possibility of an accidental write to the EEPROM, because a high level of reliability is required for security applications. This code

(a)



(b)

```
Set RA4 as Output
Clear RA4 to 0V to discharge C
Clear Counter Register
Set RA4 as Input
       Test RA4 while C charges through R:
              Increment Counter Register
              Delay 20us
       Until RA4 = 1
Convert Count to Resistance or Pot Position
```

Figure D.4
CR ADC conversion: (a) waveform at analogue input; (b) conversion procedure outline

sequence is given in the data sheet and LOCK program listing. The read sequence, for retrieving the data, is more straightforward. Using EECON1, the data in the address pointed to by EEADR is returned in EEDATA. For accessing sequences of locations, EEADR can be incremented directly. See Chapter 6 for more details.

D.5. LOCK Application

In this demonstration application, a sequence of four decimal digits is stored in the PIC EEPROM memory from the DIL switch inputs. This sets the combination for the lock. To 'open' the lock, the pot is rotated, and the input decimal digits are displayed and entered. This simulates the rotary action of mechanical combination locks. If the sequence of four input digits matches that previously stored in EEPROM, a siren sound is made to indicate the opening of the lock.

```
LOCK PROGRAM OUTLINE              MPB  29/8/99
************************************************
Hardware: DIZI PIC 16F84 Demo Board
************************************************
General Purpose Register Labels:
       0C = Period = Delay Period Preload Value
       0D = Count  = Delay Counter
       0E = PotVal = Count from ADC conversion
       0F = DigVal = Low 4 bits of PotVal
User Bit Labels:
       butA (RA4 input) - Normally 1
       butB (RB0 input) - Normally 1
       buzO (RB0 output)
See Data Sheet for SFR Labels and addresses

{Power Button On}
INIT: Initialise Port B *************************

       Port A defaults to inputs
       RA0 - RA3 = DIL Switches = 4-bit input
       RA4 = Input = butA = INP Button
       RB0 = Input = butB = INT Button
       RB1 - RB7 = Output = 7 Seg Display

MAIN: Select Set or Check Combination ***********

select {Press Button A or B}
               If (butA)=0, GOTO [stocom]
               If (butB)=0, GOTO [checom]
               GOTO [select]

SEQ1: Store 4 digits in EEPROM, beep after each *

stocom {Release Button A}
               CALL [delay] with (W)=FF
               GOTO [stocom] UNTIL (butA)=1

               Clear (EEADR)
getdil {Set DIL Switches or Press A}
               Read (PORTA) into (W)
               Calc (W) AND 0F
               Store (W) in (EEDATA)
               CALL [codtab] with (W)=00-0F
               {Returns with '7SegCode' in (W)}
               Output (W) to (PORTB)
               GOTO [getdil] UNTIL (butA)=0

waita  {Release Button A}
               GOTO [waita] UNTIL (butA)=1
               Store (EEDATA) in (EEADR)
               CALL [beep]
               Increment (EEADR) from 00 to 04
               GOTO [getdil] UNTIL (EEADR)=4
               CALL [beep]
               CALL [beep]
               GOTO [done]
SEQ2: Check 4 digits from pot for match *********

checom {Release Button B}
               CALL [delay] with (W)=FF
               GOTO [checom] UNTIL (butB)=1

               Clear (EEADR)
potin  {Adjust Pot or Press Button B}
               CALL [getpot] for (DigVal)
               {Returns with (DigVal)=00-0F}
               GOTO [potin] UNTIL (butB)=0
               Read (EEDATA) at (EEADR)
               Compare (EEDATA) with (DigVal)
               If (Z)=0 GOTO [done]

waitb  {Release Button B}
               GOTO [waitb] UNTIL (butB)=1
               CALL [beep]
```

Program D.2
LOCK program outline

```
                Increment  (EEADR)
                GOTO [potin] UNTIL EEADR=4
                GOTO [siren]

END1: Sequences matches, sound siren*************

        siren   CALL [beep]
                GOTO [siren]

END2: Digit compare failed, finish **************

        done    Clear (PORTB)
                Sleep

SUBROUTINES ***********************************

SUB1: Get Display Code
          Receives: Table Offset in W
          Returns:  7-Segment Display Code in W

codtab  Add (W) to (PCL)
                RETURN with '7SegCode' in (W)

SUB2: Variable Delay
          Receives: (Count) in W

delay   Load (Count) from (W)
                Decrement (Count) UNTIL (Count)=0
                RETURN

SUB3: Outputs one cycle of sound output
           Receives: (Period)

beep    Load (Period) with FF
                Set RB0 as Output
cycle   Set (BuzO)=1
                CALL [delay] with (Period) in W
                Set BuzO=0
                CALL [delay] with (Period) in W
                Decrement (Period) from FF to 00
                GOTO [cycle] UNTIL (Period)=0
                Reset RB0 as Input
                RETURN

SUB4: Get Pot Value using CR ADC method
          Returns: (DigVal)=00-0F

getpot  Set RA4 as Output
                Clear (RA4)
                CALL [delay] with (W)=FF
                Reset RA4 as Input
                Clear (PotVal)
check   Increment (PotVal) from 00 to XX
                CALL [delay] with (W)=3
                GOTO [check] UNTIL (RA4)=1
                (DigVal) = (PotVal) AND 0F
                CALL [codtab] with (DigVal)=00-0F
                RETURN


END OF LOCK PROGRAM ****************************
```

Program D.2: (***continued***)

In the actual application, a solenoid-operated lock mechanism would be activated from this output, by replacing the siren sequence with an instruction to set an output bit. A suitable current driver interface for the solenoid would be required (see the motor interface in Chapter 8). Only the Power button, Enter button, Digit Select pot and display would be accessible to the

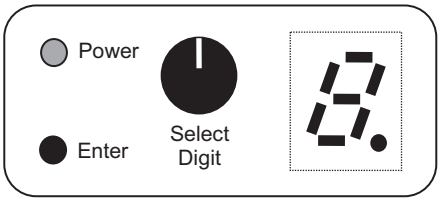


Figure D.5
LOCK user interface

user in the final design. The hardware would need to be reconfigured so that the unit would appear as in Figure D.5 to the user. The DIL switch bank and its button for setting the entry code would be concealed.

The application program contains the following blocks:

1. Declaration of Register and Bit Labels
2. Initialization of Registers
3. Sequence 1 – Store Combination
4. Sequence 2 – Check Combination
5. End 1 – Continuous siren output
6. End 2 – Sleep
7. Subroutine 1 – Display Code Table
8. Subroutine 2 – Variable Delay
9. Subroutine 3 – Output One Tone Cycle
10. Subroutine 4 – Get Digit from Pot.

The program has two main sequences, for inputting and checking a combination, and two alternative endings. The processor goes to sleep after completion of the input sequence, or an incorrect digit match. The DIZI board must be repowered to try again, as there are no other interrupts enabled to restart it. If the combination is correct, the siren sounds, continuing until the power goes off.

The application code is outlined in Program D.2. The pseudocode is developed in a word processor or the program source code editor until the statements are detailed enough to be converted into assembly code statements. It should be written in a form that allows it to be readily converted to PIC assembly language. In this way, the program structure and logic can be worked out before attempting to write the source code itself.

The conventions used in the pseudocode are as follows:

```
Block Structure applied
Target Hardware specified
Register & Bit Labels defined
```

```
User Inputs included in the sequence
GOTO [deslab]
   - Jump to destination address label
CALL [subnam]
   - Call subroutine at address label
   - values passed to and received from subroutine defined
GOTO [addlab] UNTIL (condition)
   - implemented using Bit Test, Skip & GoTo operation
   - (regname) = contents of register labeled 'regname'
Program block type defined:
   INIT = Initialize
   MAIN = Main Program
   SEQn = Sequence ending with GOTO
   ENDn = End operation
   SUBn = Subroutine, optionally receiving and/or returning values
```

The source code file uses the following conventions:

- full details of hardware and operation of application in source code
- SFR, user and bit labels defined in separate blocks for clarity
- block and line comments in source code
- lower case for address labels
- upper case for instruction mnemonics and SFR labels
- capitalization of user register labels
- identification and separation of block types.

The list file for the LOCK program (Program D.3) contains the source code, machine code and memory allocation. The source code can be downloaded from www.picmicros.org.uk.

```
00001 ; ********************************************************************
00002 ; LOCK.ASM                                                 MPB   17/8/99
00003 ; ********************************************************************
00004 ;
00005 ; Four digit combination lock simulation demonstrates the hardware
00006 ; features of the DIZI demo board and the PIC 16F84.
00007 ;
00008 ; Hardware:     DIZI Demo Board with PIC 16F84 (4MHz)
00009 ; Setup:        RA0-RA3         DIL Switch Inputs
00010 ;               RA4             Push Button Input / Analogue Input
00011 ;               RB0             Push Button Input / Audio Output
00012 ;               RB1-RB7         7-Segment Display Output
00013 ; Fuses:       WDT off, PuT on, CP off
00014 ;
00015 ; Operation ----------------------------------------------------------
00016 ;
00017 ; To set the combination, a sequence of 4 digits is input on the DIL
00018 ; piano switches; this is retained in the EEPROM when power is off.
00019 ; To 'open' the lock, a sequence of 4 digits is input via
00020 ; the potentiometer. These are compared with the stored data, and
00021 ; an audio output generated to indicate the correct sequence.
00022 ; The processor halts if any digit fails to match, and the program
00023 ; must be restarted.
```

Program D.3
LOCK program list file

```
                00024 ;
                00025 ;       To set a combination:
                00026 ;       1.      Hold Power On Button
                00027 ;       2.      Press Button A
                00028 ;       3.      Set a digit on DIL switches and Press A - beeps
                00029 ;       4.      Repeat step 3 for 3 more digits
                00030 ;       5.      Release Power Button
                00031 ;
                00032 ;       To check a combination:
                00033 ;       1.      Hold Power On Button
                00034 ;       2.      Press Button B
                00035 ;       3.      Set a digit on pot and Press B - beeps if matched
                00036 ;       4.      Repeat step 3 for 3 more digits
                00037 ;                - if digits all match, siren is sounded
                00038 ;                - if any digit fails to match, the processor halts
                00039 ;       5.      Release Power Button
                00040 ;
                00041 ; *****************************************************************
                00042         PROCESSOR 16F84         ; Processor Type Directive
                00043 ; *****************************************************************
                00044
                00045 ; EQU:  Special Function Register Equates.........................
                00046
  0002          00047 PCL     EQU     02      ; Program Counter Low
  0005          00048 PORTA   EQU     05      ; Port A Data
  0006          00049 PORTB   EQU     06      ; Port B Data
  0003          00050 STATUS  EQU     03      ; Flags
  0008          00051 EEDATA  EQU     08      ; EEPROM Memory Data
  0009          00052 EEADR   EQU     09      ; EEPROM Memory Address
  0008          00053 EECON1  EQU     08      ; EEPROM Control Register 1
  0009          00054 EECON2  EQU     09      ; EEPROM Control Register 2
                00055
                00056 ; EQU: User Register Equates.....................................
                00057
  000C          00058 Period  EQU     0C      ; Period of Output Sound
  000D          00059 Count   EQU     0D      ; Delay Down Counter
  000E          00060 PotVal  EQU     0E      ; Analogue Input Value
  000F          00061 DigVal  EQU     0F      ; Current Digit Value 00 to 09
                00062
                00063 ; EQU: SFR Bit Equates...........................................
                00064
  0005          00065 RP0     EQU     5       ; STATUS - Register Page Select
  0000          00066 RD      EQU     0       ; EECON1 - EEPROM Memory Read Byte Initiate
  0001          00067 WR      EQU     1       ; EECON1 - EEPROM Memory Write Byte Initiate
  0002          00068 WREN    EQU     2       ; EECON1 - EEPROM Memory Write Enable
  0002          00069 Z       EQU     2       ; STATUS - Zero Flag
                00070
                00071 ; EQU: User Bit Equates..........................................
                00072
  0004          00073 butA    EQU     4       ; PORTA  - RA4 Input Button
  0000          00074 butB    EQU     0       ; PORTB  - RB0 Input Button
  0000          00075 buzO    EQU     0       ; PORTB  - RB0 Output Buzzer
                00076
                00077 ; *****************************************************************
                00078
                00079 ; INIT: Initialize Port B (Port A defaults to inputs)
                00080
0000 3001       00081 start   MOVLW   001             ; RB0 = Input, RB1-RB7 = Outputs
0001 0066       00082         TRIS    PORTB           ; Set Data Direction
0002 0086       00083         MOVWF   PORTB           ; Clear Data
0003 286D       00084         GOTO    select          ; Select Combination Read or Write
                00085
                00086 ; SUBROUTINES ***************************************************
                00087
                00088 ; SUB1: 7-Segment Code Table using PCL + offset in W
                00089 ;       Returns digit display codes, with '-' for numbers A to F
                00090
0004 0782       00091 codtab  ADDWF   PCL             ; Add offset to Program Counter
0005 347E       00092         RETLW   B'01111110'     ; Return with display code for '0'
0006 340C       00093         RETLW   B'00001100'     ; Return with display code for '1'
0007 34B6       00094         RETLW   B'10110110'     ; Return with display code for '2'
0008 349E       00095         RETLW   B'10011110'     ; Return with display code for '3'
0009 34CC       00096         RETLW   B'11001100'     ; Return with display code for '4'
000A 34DA       00097         RETLW   B'11011010'     ; Return with display code for '5'
000B 34FA       00098         RETLW   B'11111010'     ; Return with display code for '6'
000C 340E       00099         RETLW   B'00001110'     ; Return with display code for '7'
000D 34FE       00100         RETLW   B'11111110'     ; Return with display code for '8'
000E 34DE       00101         RETLW   B'11011110'     ; Return with display code for '9'
000F 3480       00102         RETLW   B'10000000'     ; Return with display code for '-'
0010 3480       00103         RETLW   B'10000000'     ; Return with display code for '-'
0011 3480       00104         RETLW   B'10000000'     ; Return with display code for '-'
0012 3480       00105         RETLW   B'10000000'     ; Return with display code for '-'
0013 3480       00106         RETLW   B'10000000'     ; Return with display code for '-'
0014 3480       00107         RETLW   B'10000000'     ; Return with display code for '-'
                00108
```

Program D.3: (***continued***)

```
                    00109 ; ------------------------------------------------------------------
                    00110 ; SUB2: Delay routine
                    00111 ;       Receives delay count in W
                    00112
0015 008D           00113 delay   MOVWF   Count           ; Load counter from W
0016 0B8D           00114 loop    DECFSZ  Count           ; and decrement
0017 2816           00115         GOTO    loop            ; until zero
0018 0008           00116         RETURN                  ; and return
                    00117
                    00118 ; ------------------------------------------------------------------
                    00119 ; SUB3: Output One Beep Cycle to BuzO
                    00120
0019 30FF           00121 beep    MOVLW   0FF             ; Load FF into
001A 008C           00122         MOVWF   Period          ; Period counter
                    00123
001B 3000           00124         MOVLW   B'00000000'     ; Set RB0
001C 0066           00125         TRIS    PORTB           ; as output
                    00126
                    00127 ; Do one cycle of rising tone....
                    00128
001D 1406           00129 cycle   BSF     PORTB,buzO      ; Output High
001E 080C           00130         MOVF    Period,W        ; Load W with Period value
001F 2015           00131         CALL    delay           ; and delay for Period
                    00132
0020 1006           00133         BCF     PORTB,buzO      ; Output Low
0021 2015           00134         CALL    delay           ; and delay for same Period
0022 0B8C           00135         DECFSZ  Period          ; Decrement Period
0023 281D           00136         GOTO    cycle           ; and do next cycle until 0
                    00137
                    00138 ; Set RB0 to input again.........
                    00139
0024 3001           00140         MOVLW   B'00000001'     ; Reset RB0
0025 0066           00141         TRIS    PORTB           ; as input
0026 0008           00142         RETURN                  ; from tone cycle
                    00143
                    00144 ; ------------------------------------------------------------------
                    00145 ; SUB4: Get pot value (Rv) using rise time due to C and R on RA4
                    00146 ;       Returns with digit value (0-F) in DigVal
                    00147
                    00148 ; Discharge external capacitor on RA4
                    00149
0027 300F           00150 getpot  MOVLW   B'00001111'     ; Set RA4
0028 0065           00151         TRIS    PORTA           ; as output
0029 1205           00152         BCF     PORTA,4         ; and discharge C setting output low
002A 30FF           00153         MOVLW   0FF             ; Delay for about 1ms
002B 2015           00154         CALL    delay           ; to ensure C is discharged
002C 301F           00155         MOVLW   B'00011111'     ; Reset RA4
002D 0065           00156         TRIS    PORTA           ; as input
                    00157
                    00158 ; Increment a counter until RA4 goes high due to charging of C
                    00159
002E 018E           00160         CLRF    PotVal          ; Clear input value counter
002F 0A8E           00161 check   INCF    PotVal          ; increment counter
0030 3003           00162         MOVLW   03              ; Set delay count to 3
0031 2015           00163         CALL    delay           ; and delay between input checks
0032 1E05           00164         BTFSS   PORTA,4         ; Check input bit RA4
0033 282F           00165         GOTO    check           ; and repeat if not yet high
                    00166
                    00167 ; Mask out high bits of count value, and store & display
                    00168 ; 4-bit digit value, 0-F
                    00169
0034 080E           00170         MOVF    PotVal,W        ; Put count value in W
0035 390F           00171         ANDLW   00F             ; and set high 4 bits to 0
0036 008F           00172         MOVWF   DigVal          ; Store 4-bit value
0037 2004           00173         CALL    codtab          ; Get 7-segment code, 0-9
0038 0086           00174         MOVWF   PORTB           ; and display
                    00175
0039 0008           00176         RETURN                  ; with DigVal from setting of pot
                    00177
                    00178 ; MAIN SEQUENCES ************************************************
                    00179
                    00180 ; SEQ1: Store 4 Digits in non volatile EEPROM
                    00181 ;       Beep after each digit, and twice when 4 done
                    00182
                    00183 ; Complete Button A input operation
                    00184
003A 30FF           00185 stocom  MOVLW   0FF             ; Delay for about 1ms
003B 2015           00186         CALL    delay           ; to avoid Button A switch bounce
003C 1E05           00187         BTFSS   PORTA,butA      ; Wait for Button A
003D 283A           00188         GOTO    stocom          ; to be released
                    00189
                    00190 ; Read 4-bit binary number from DIL switches into EEDATA and display
                    00191
003E 0189           00192         CLRF    EEADR           ; Zero EEPROM address register
003F 0805           00193 getdil  MOVF    PORTA,W         ; Read DIL switches
0040 390F           00194         ANDLW   0F              ; and set high 4 bits to 0
```

Program D.3: (***continued***)

```
0041 0088          00195         MOVWF   EEDATA          ; Put DIL value in EEPROM data
                   00196
0042 2004          00197         CALL    codtab          ; Display DIL input as decimal
0043 0086          00198         MOVWF   PORTB           ;
                   00199
0044 1A05          00200         BTFSC   PORTA,butA      ; Check if Button A pressed
0045 283F          00201         GOTO    getdil          ; If not, keep reading DIL input
                   00202
                   00203 ; Store the current DIL input in EEPROM at current address
                   00204
0046 1683          00205 store   BSF     STATUS,RP0      ; Select Register Bank 1
0047 1508          00206         BSF     EECON1,WREN     ; Enable EEPROM write
0048 3055          00207         MOVLW   055             ; Write initialization sequence
0049 0089          00208         MOVWF   EECON2          ;
004A 30AA          00209         MOVLW   0AA             ;
004B 0089          00210         MOVWF   EECON2          ;
004C 1488          00211         BSF     EECON1,WR       ; Write data into current address
004D 1283          00212         BCF     STATUS,RP0      ; Re-select Register Bank 0
                   00213
004E 1E05          00214 waita   BTFSS   PORTA,butA      ; Wait for Button A to be released
004F 284E          00215         GOTO    waita           ;
0050 2019          00216         CALL    beep            ; Beep to indicate digit write done
                   00217
                   00218 ; Check if 4 digits have been stored yet, if not, get next
0051 0A89          00220         INCF    EEADR           ; Select next EEPROM address
0052 1D09          00221         BTFSS   EEADR,2         ; Is the address now = 4?
0053 283F          00222         GOTO    getdil          ; If not, get next digit
0054 2019          00224         CALL    beep            ; Beep twice when 4 digits stored
0055 2019          00225         CALL    beep            ;
0056 2874          00226         GOTO    done            ; Go to sleep when done
                   00228 ; -------------------------------------------------------------------
                   00230 ; SEQ2: Check PotVal v EEPROM
                   00231
0057 30FF          00232 checom  MOVLW   0FF             ; Delay for about 1ms
0058 2015          00233         CALL    delay           ; to avoid Button B switch bounce
0059 1C06          00234         BTFSS   PORTB,butB      ; Wait for Button B to be released
005A 2857          00235         GOTO    checom          ;
                   00236
                   00237 ; Read the value set on the input pot
                   00238
005B 0189          00239         CLRF    EEADR           ; Zero EEPROM address
005C 2027          00240 potin   CALL    getpot          ; Get a digit value set on pot (Rv)
005D 1806          00241         BTFSC   PORTB,butB      ; Check in Button pressed again
005E 285C          00242         GOTO    potin           ; If not, keep reading the pot
                   00243
                   00244 ; Get a digit value from EEPROM and compare with the pot input
                   00245
005F 1683          00246         BSF     STATUS,RP0      ; Select Register Bank 1
0060 1408          00247         BSF     EECON1,RD       ; Read selected EEPROM location
0061 1283          00248         BCF     STATUS,RP0      ; Re-select Register Bank 0
0062 0808          00249         MOVF    EEDATA,W        ; Copy EEPROM data to W
                   00250
0063 068F          00251         XORWF   DigVal          ; Compare the input with EEPROM data
0064 1D03          00252         BTFSS   STATUS,Z        ; If it does not match, go to sleep
0065 2874          00253         GOTO    done            ;
                   00254
                   00255 ; If digit match obtained, check if 4 done and do next if not
                   00256
0066 1C06          00257 waitb   BTFSS   PORTB,butB      ; Wait for Button B to be released
0067 2866          00258         GOTO    waitb           ;
0068 2019          00259         CALL    beep            ; Beep to confirm successful match
                   00260
0069 0A89          00261         INCF    EEADR           ; Select next EEPROM location
006A 1D09          00262         BTFSS   EEADR,2         ; 4 digits checked yet?
006B 285C          00263         GOTO    potin           ; If not, do the next
006C 2872          00264         GOTO    siren           ; When 4 digits done, run siren
                   00265
                   00266 ; ******************************************************************
                   00267
                   00268 ; MAIN: Select Set or Check Combination
                   00269
006D 1E05          00270 select  BTFSS   PORTA,butA      ; Button A pressed?
006E 283A          00271         GOTO    stocom          ; If so, store a combination
006F 1C06          00272         BTFSS   PORTB,butB      ; Button B pressed?
0070 2857          00273         GOTO    checom          ; If so, check a combination
0071 286D          00274         GOTO    select          ; repeat endlessly
                   00275
                   00276 ; ******************************************************************
                   00277
                   00278 ; END1: When combination successfully matched, make siren sound
                   00279
0072 2019          00280 siren   CALL    beep            ; Do a tone cycle
0073 2872          00281         GOTO    siren           ; and repeat endlessly
                   00282
                   00283 ; -------------------------------------------------------------------
                   00284
```

Program D.3: (***continued***)

```
                    00285 ; END2: When a digit check fails, go to sleep, and try again
                    00286
0074 0186           00287 done    CLRF     PORTB          ; Switch off display
0075 0063           00288         SLEEP                   ; Processor halts
                    00289
                    00290 ; *************************************************************
                    00291         END                     ; of program source code


SYMBOL TABLE
  LABEL                             VALUE

Count                               0000000D
DigVal                              0000000F
EEADR                               00000009
EECON1                              00000008
EECON2                              00000009
EEDATA                              00000008
PCL                                 00000002
PORTA                               00000005
PORTB                               00000006
Period                              0000000C
PotVal                              0000000E
RD                                  00000000
RP0                                 00000005
STATUS                              00000003
WR                                  00000001
WREN                                00000002
Z                                   00000002
__16C84                             00000001
beep                                00000019
butA                                00000004
butB                                00000000
buzO                                00000000
check                               0000002F
checom                              00000057
codtab                              00000004
cycle                               0000001D
delay                               00000015
done                                00000074
getdil                              0000003F
getpot                              00000027
loop                                00000016
potin                               0000005C
select                              0000006D
siren                               00000072
start                               00000000
stocom                              0000003A
store                               00000046
waita                               0000004E
waitb                               00000066

MEMORY USAGE MAP ('X' = Used,  '-' = Unused)
0000 : XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX
0040 : XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXX----------

All other memory blocks unused.
```

Program D.3: (***continued***)

APPENDIX E

Dizi690 Demo Board

Chapter Outline

This appendix describes the process of developing a test board based on the PIC® 16F690 microcontroller (MCU), as used in the Microchip LPC (low pin count) demonstration board. The design stages are:

- schematic capture with Labcenter ISIS
- interactive simulation with Labcenter Proteus VSM
- printed circuit board (PCB) design using Labcenter ARES
- PCB manufacture using Galaad Percival CNC software.

Proteus VSM has been used throughout this book as a design and debugging tool for PIC microcontroller designs. Currently, a demonstration version may be downloaded which only allows sample applications to be tested. For developing new designs, or testing those described in this book, a suitable bundle containing PIC 16 models must be purchased from Labcenter Electronics. For current product and demonstration software availability, refer to www.labcenter.com. The design files for this test board can be downloaded from www.picmicros.org.uk.

E.1. Circuit Design

A target system was required which allowed basic PIC programming principles, as well as the hardware development process, to be demonstrated. The following features are required:

- two push-button inputs (tactile switches, two inputs)
- 4-bit binary switched input (DIP switch, four inputs)

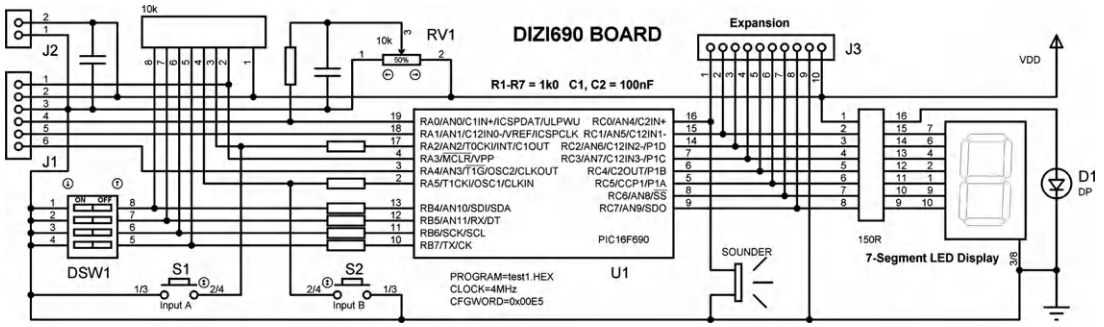


Figure E.1
Dizi690 demo board schematic

- one analogue input (manual pot, one analogue input)
- one-digit seven-segment display (LED, eight outputs including decimal point)
- sounder output (piezo transducer, one output).

In total, 16 input and output pins are required to meet this specification, and the number and type can be matched by the 16F690 MCU. The input/output (I/O) allocation and peripheral connections can be seen in the schematic (Figure E.1).

The six-pin programming connector (J1) is connected to port A, as in the LPC board, with the push buttons (active low) connected to the other pins of this port. The binary input is connected to port B, and the display is on port C. The light-emitting diode (LED) segments require current-limiting resistors (RN1, 150R). The separate LED represents the decimal point which is part of the actual seven-segment display, but this device is not available as a simulated component. The piezo sounder, having a high resistance, can be connected direct to the PIC output. RP1 provides a bank of pull-up resistors for the switch inputs, with the pot connected to RA0. When testing, power is supplied via the programming connector, and an additional power supply connector is provided for independent operation.

E.2. Schematic Edit

The schematic in Figure E.1 was created using Labcenter ISIS, a component of Proteus VSM. On opening this application, an edit screen appears (Figure E.2). The main operations are:

- Select components from library.
- Place and connect up components.
- Write a test program for the MCU.
- Attach to the MCU and simulate.
- Debug the program if necessary.
- Modify the hardware if necessary.

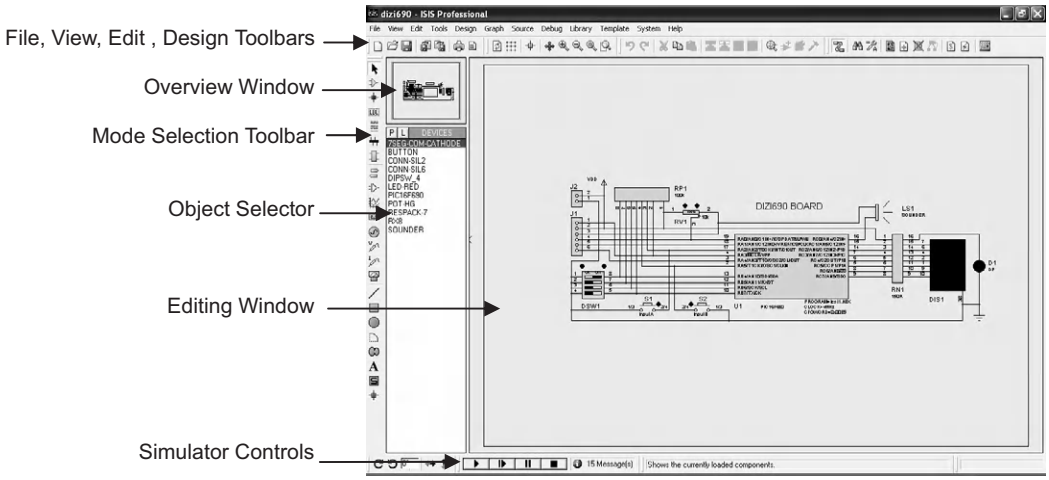


Figure E.2
ISIS schematic capture screenshot

- Export netlist to ARES to make a PCB.

Before starting to use the editor, create an application folder with a suitable project name, such as 'DIZI690', which will hold the design and program files. Open ISIS, and Save As: DIZI690.DSN. The design title, version and author may now also be entered via the Design menu, Edit Design Properties dialogue.

To start creating the schematic, click on the P button at the top of the Object Selector window and the Pick Devices window opens (Figure E.3). The components are organized by category; for example, select Microprocessors ICs, subcategory PIC 16 Family, and the devices are listed in numerical order. Note that the integrated circuit (IC) package is also displayed, which we will need later. If the component number is known, it may be quicker to enter its number in the Keywords search box. Double-click to add the component to the Devices list. Repeat this procedure for the required components in the different categories.

If the design does not need to be tested by simulation, specific components can be selected with the correct pin-out already specified. This appears in the PCB Preview window in the Pick Devices dialogue. If we do wish to test the design by simulation, we need to select active components for the schematic and, if necessary, attach a suitable pin-out when the component is finally specified for the PCB layout (see Section E.5 below).

With all the components included in the devices list, select the PIC16F690 and left-click on the edit screen. The component can be positioned and dropped on the circuit with a second click. Place the other devices in order of size to get a rough arrangement on the diagram, and line up connections where possible, for example, the display inputs with port C outputs. Finish

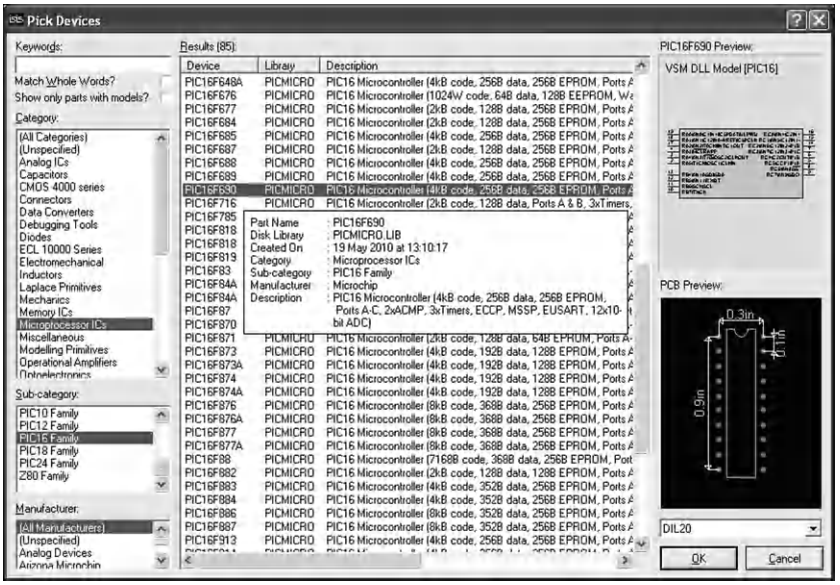


Figure E.3
Pick Devices dialogue

connecting up and adjust the component positions for the neatest arrangement. The components can be flipped and rotated as necessary, either before placing or after using the edit window buttons or component menu. In more complex circuits, labeled connectors can be used to clarify the schematic by reducing the number of explicit connections and allowing connection between separate sheets of the drawing.

In component mode, devices can be moved by single-clicking to highlight, and dragging. Single right-click opens a component edit menu, and double right-click deletes the device. Wiring is completed by clicking on the component connections and dropping the other end by clicking on the destination pin. Bends can be placed by clicking or the connection moved after highlighting. Wiring junctions are indicated by dots, but take care, as it is possible for a junction to be not connected, even if it appears that it is. Move the component labeling (highlight and drag) or hide it in the properties dialogue if it interferes with neat connections and provides no essential information. In the Template, Set Design Defaults menu, it is often desirable to uncheck the Show Hidden Text option to minimize text clutter on the schematic.

When finished and saved, the schematic can be printed or exported as a bitmap DIZI690.BMP file, or a screenshot taken, for insertion into a document. Figure E.1 is an example. To save the bitmap, select File, Export Graphics, Export Bitmap. Adjust the resolution to maximum (600 dpi) and the orientation if necessary, and check the bitmap file box. A bill of materials

```
Bill Of Materials
=================
Design:   Dizi690 Board
Doc. no.: 1
Revision: 1.0
Author:   M Bates
Created:  29/11/10
Modified: 06/01/11

QTY  PART-REFS           VALUE
---  ---------           -----
Resistors
---------
7    R1-R7               1k

Capacitors
----------
1    C1                  100nF
1    C2                  1nF

Integrated Circuits
-------------------
1    U1                  PIC16F690

Diodes
------
1    D1                  DP

Miscellaneous
-------------
1    DIS1                7SEG
1    DSW1                DIPSW_4
1    J1                  CONN-SIL6
1    J2                  CONN-SIL2
1    J3                  CONN-SIL10
1    LS1                 SOUNDER
1    RN1                 150R
2    RP1,RV1             10k
1    S1                  Input A
1    S2                  Input B
```

List E.1
ISIS bill of materials

(list of components) (List E.1) can also be produced, which will be needed later for ordering components.

E.3. Program Edit

To test the design by interactive simulation, a hex file needs to be attached to the MCU. The application program can be designed using flowcharts (Chapter 4) or a text outline, which must then be converted into source code. Assembler code can be created using a built-in editor, so MPLAB editor is not essential. To edit the source code, in the Source menu, select Add/ Remove Source Files, New, enter a filename such as TEST1.ASM and create the file.

Alternatively, for this design, download the program shown as Program E.1, TEST1.ASM, place it in the application folder and attach it in the source file dialogue.

In the source code, the processor is specified so that the assembler can check that the statements are suitable for that processor, for example, that only the available registers are used. The CONFIG code specifies the configuration bits in the MCU, including selection of the default internal clock speed, 4 MHz. Note that the oscillator selection bits also affect the function of RA4 and RA5 (which must be digital I/O). The include file P16F690.INC provides the special function register and bit labeling. This file is downloaded with the MPLAB IDE file set, and must be copied from the 'Microchip/MPASM Suite' folder into the project folder.

The PIC 16F690 must be initialized for most of the pins to operate as digital I/O, so the SFRs ANSEL and ANSELH are cleared initially, and then the analogue input RA0 is enabled by setting ANSEL,0. Note the bank selection syntax required to access the special function registers (SFRs), accessing them in descending order so that bank 0 remains selected for the main program. The analogue setup is described in more detail in Chapter 7, as applied to the LPC board. The data sheet for the 16F690 will need to be downloaded from www.microchip.com to support the program development.

The main loop of the test program provides two main functions, selected via the push buttons: a scan of port B, which also tests the sound output (hold input A), and a hex display (input B). The programming techniques used are described in Chapter 6, including subroutines, software delay loop and data table. The program is initially assembled using the Build All command in the Source menu. The assembler MPASMWIN.EXE is supplied with Proteus VSM. The assembler should be selected automatically when a PIC device is placed on the schematic, but this can be checked in the Source File dialogue.

E.4. Circuit Simulation

The circuit is stored as a set of components and the connections between them in a netlist (List E.2). It is simply a list of the components, the unique numbered nodes (1000−1033, 0 = 0 V) to which the pins are connected, and the component value or label.

The standard simulation engine for analogue networks, which is used in VSM, is called SPICE, a system that has been developed and refined over many years. Essentially, the network is analyzed as a set of simultaneous equations based on the netlist and the component mathematical models, which is solved for given inputs, to predict the resulting outputs. In VSM, these are presented as output voltages and current, or graphs (time and frequency response, etc.), or displayed on virtual instruments.

Digital elements are represented as the logical relationship between device inputs and outputs, and can be combined with the analogue analysis to provide a mixed-mode simulation. The

```
;*************************************************************
;       TEST1.ASM      MPB  Ver 1.0
;       Test program for DIZI690 demo board
;       Status: Tested OK 6-1-11
;
;*************************************************************

        PROCESSOR 16F690       ; Specify MCU for assembler
        __CONFIG 00E5          ; MCU configuration bits
                               ; PWRT on, MCLR enabled
                               ; Internal Clock (default 4MHz)
        INCLUDE "P16F690.INC"  ; Standard register labels

Loco    EQU     20             ; Low count register
Hico    EQU     21             ; High count register
Hexnum  EQU     22             ; Hex table offset

; Initialise registers.....................................

        BANKSEL ANSEL          ; Select Bank 2
        CLRF    ANSEL          ; Ports digital I/O
        BSF     ANSEL,0        ; except AN0 Analogue input
        CLRF    ANSELH         ; Ports digital I/O

        BANKSEL TRISC          ; Select Bank 1
        CLRF    TRISC          ; Port C for output
        MOVLW   B'00010000'    ; A/D clock setup code
        MOVWF   ADCON1         ; A/D clock = fosc/8

        BANKSEL PORTC          ; Select bank 0
        CLRF    PORTC          ; Clear display outputs
        MOVLW   B'00000001'    ; Analogue input setup code
        MOVWF   ADCON0         ; Left justify, Vref=5V,
                               ; Select RA0, done, enable A/D

; Main loop ...............................................

buta    BTFSC   PORTA,2        ; Check button A
        GOTO    butb           ; ..and button B if off
        CLRF    PORTC          ; Clear display
        BSF     PORTC,0        ; Switch on LSB

rotdis  RLF     PORTC          ; Rotate segment
        CALL    vardel         ; Run variable delay
        BTFSS   PORTA,2        ; Check button A again
        GOTO    rotdis         ; ..and continue if on

butb    BTFSC   PORTA,5        ; Else check button B
        GOTO    buta           ; ..and repeat if off
        CALL    hexdis         ; Get display code
        MOVWF   PORTC          ; Show it
        GOTO    buta           ; and repeat always

; Delay subroutine.........................................

vardel  BSF     ADCON0,1       ; start ADC..
wait    BTFSC   ADCON0,1       ; ..and wait for finish
        GOTO    wait
        MOVF    ADRESH,W       ; Store result high byte
        MOVWF   Hico
        INCF    Hico           ; Avoid zero count

slow    CLRF    Loco           ; use result in delay
fast    DECFSZ  Loco
        GOTO    fast
        DECFSZ  Hico
        GOTO    slow
        RETURN                 ; done
```

Program E.1
Test program for Dizi690 board

```
; Hex display subroutine ..................................

hexdis  MOVF    PORTB,W         ; Get DIP switches
        ANDLW   B'11110000'     ; Mask low bits
        MOVWF   Hexnum
        SWAPF   Hexnum          ; Swap hi-lo bits
        MOVF    Hexnum,W

        ADDWF   PCL             ; Hex code table
        RETLW   B'01111110'     ; 0
        RETLW   B'00001100'     ; 1
        RETLW   B'10110110'     ; 2
        RETLW   B'10011110'     ; 3
        RETLW   B'11001100'     ; 4
        RETLW   B'11011010'     ; 5
        RETLW   B'11111010'     ; 6
        RETLW   B'00001110'     ; 7
        RETLW   B'11111110'     ; 8
        RETLW   B'11001110'     ; 9
        RETLW   B'11101110'     ; A
        RETLW   B'11111000'     ; b
        RETLW   B'01110010'     ; C
        RETLW   B'10111100'     ; d
        RETLW   B'11110010'     ; E
        RETLW   B'11100010'     ; F

        END     ; Terminate assembler.......................
```

Program E.1: (***continued***)

behavior of an MCU is controlled by an associated machine code program (HEX file), with the whole network then being described as a co-simulation model. Active components (switches, pots, LEDs, etc.) allow inputs to be generated and outputs displayed in a more natural, interactive way.

The hex file for a PIC MCU (generated from user source code by the assembler; Chapters 3 and 4) must be attached via its properties dialogue. Double-click on the PIC and, in the Edit Component dialogue, click on the Program File tab to open the Select File dialogue. For this application, select and open the TEST1.HEX file. While the Edit Component dialogue is open, set the MCU clock to 4 MHz, and the Configuration Word to 0x00E5, to match the source code configuration statement.

We are now ready to run the simulation by clicking on the Run button at the lower left of the edit window. If all previous steps have been carried out correctly, the schematic will go live, indicated by the logic levels being displayed on each pin (red = high, blue = low) and the status bar showing the simulation time elapsed and host processor load (Figure E.4).

If the central processing unit (CPU) load exceeds 100%, the animation rate drops below real time. This will happen if the circuit is complex or the host CPU is too slow. The properties of the simulated components must be taken into account if the simulation speed is reduced. For example, the switches have a default delay of 1 ms before closing to simulate the mechanical properties. This may be significant if this translates into long real-time delay in the simulation, and the switch does not appear to be working correctly.

```
*C:\PIC Micros 3\Programs\diz690\design\dizi690.DSN
C1 1036 0 100NF
C2 1008 0 1NF
D1 1016 0 DP
DIS1 1004 1003 0 1002 1001 1000 0 1005 1006 7SEG
DSW1 0 0 0 0 1020 1019 1018 1017 DIPSW_4
J1 1022 1008 0 1025 1032 1027 CONN-SIL6
J2 0 1008 CONN-SIL2
J3 1024 1009 1010 1011 1012 1013 1014 1015 0 1008 CONN-SIL10
LS1 0 1024 SOUNDER
R1 1036 1025 1K
R2 1033 1023 1K
R3 1028 1021 1K
R4 1029 1017 1K
R5 1030 1018 1K
R6 1031 1019 1K
R7 1026 1020 1K
RN1 1008 1009 1010 1011 1012 1013 1014 1015 1006 1005 1004 1003 1002
1001 1000 1016 150R
RP1 1008 1022 1023 1021 1020 1019 1018 1017 10K
RV1 0 1008 1036 10K
S1 0 1023 0 1023 INPUT A
S2 0 1021 0 1021 INPUT B
U1 1008 1028 1027 1022 1013 1012 1011 1014 1015 1026 1031 1030 1029
1010 1009 1024 1033 1032 1025 0 PIC16F690
```

List E.2
ISIS schematic netlist for Dizi690 schematic

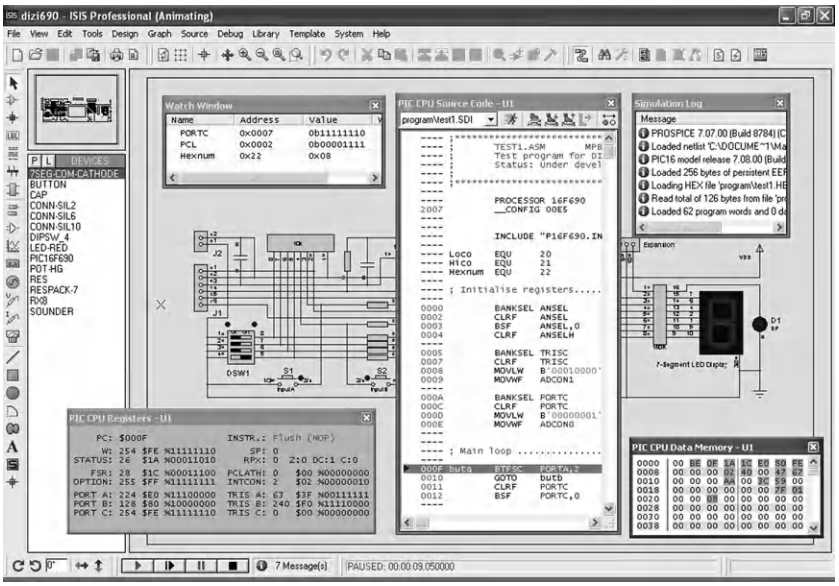


Figure E.4
ISIS simulation screenshot

Stop resets the program simulation and Pause allows access to the debug menu options. The CPU options are the most frequently used; the CPU source code, registers, data memory and watch window are most useful (Figure E.4).

The source code window provides run, step and breakpoint controls. Step Into means execute all instructions in sequence, Step Over means run subroutines below the current level at full speed and Step Out allows us to leave a subroutine at full speed. Breakpoints are set by simply double-clicking on the source code line, and the program will stop there. Using these controls, the program sequence can be checked and particular sections scrutinized.

Register changes can be tracked via the register, data or a watch window; the latter has the advantage that it remains visible during program execution at full speed. Registers to be monitored are selected by right-clicking on the watch window. SFRs can be selected by name, and user registers by number (source code labels, unfortunately, are not recognized by the debugger).

The animated schematic buttons can be operated in transient or toggle mode by clicking on the button or the red dot control. The dual in-line (DIL) switches similarly can be controlled individually or collectively. The pot is adjusted up and down via the + and − dots. The sounder outputs via the host sound card, and the display illuminates to demonstrate correct program operation.

There are some components that have no simulation model, for example the connectors J1 and J2. An error message will be displayed to this effect. The solution is to exclude them from the simulation via their properties dialogue. Similarly, the discrete LED can be excluded from the PCB layout because its function will be performed by the decimal point of the hardware version of the component.

The main purpose of the simulation is software debugging. The relevant techniques are outlined in Chapter 9. Normally, the source code window will be kept open during debugging to make any changes necessary. After editing and saving any corrections, the program is reassembled automatically when the Run button is pressed.

The interactive components provide a much more intuitive interface than the tabular output in MPLAB, while the range of debugging and project management tools is fewer in Proteus VSM. Normally, the hardware will be designed based on the application specification, then the software, but it may be necessary to revisit the hardware in the light of program debugging.

E.5. PCB Design

When the circuit design has been completed in ISIS and firmware has been successfully tested, the schematic netlist can be exported to the PCB layout package ARES.

To create a layout, a physical pin arrangement is needed for each component. The electrical connections in the schematic also have to be mapped onto the physical pins of the selected component. This means that the real component must be selected and its pin-out specified both electrically and physically. Some major components may need to be selected from a supplier's catalogue at this stage and the data sheet studied for this information.

Some pin-outs are already attached because the choice of package is limited to the available component packages selected from the library. For example, the PIC 16F690 is only available in a standard dual in-line (DIL), through-hole package, or one of three surface-mount packages. For prototyping, the DIL package will be used, with a view to converting the PCB to a surface-mount layout when fully functional. The finished prototype through-hole layout is seen in Figure E.7 on page 410.

Other devices have a greater choice of package, and the pin-out of the hardware component selected must be created to suit. For example, a generic seven-segment display has been used in the simulation which has no layout package attached in the Proteus library. A real component is available with a decimal point, which can be used as a power indicator, replacing the discrete LED in the schematic. The data sheet for this was saved in the application folder and a pin-out created in ARES (this is also good practice in using the layout editor). The special pin-out created can then be assigned to the display using the packaging tool in ISIS. The resulting package can be seen on the layout.

E.6. Package Assignment

The device selected was the Kingbright SC52-11 single-digit numerical display. Dimensioning and connection information from the data sheet is shown in Figure E.5. There are two rows of pins with a standard spacing of 0.1 inch, separated by 0.6 inch. Pin 1 is bottom left, which will be identified by a square pad, with the rest circular.

To create the package for the display component, open ARES and save the workspace as DISPLAY.LYT. In the View menu, deselect Metric mode (i.e. select Imperial measure) and set the Snap to 50th (fifty-thousandths of an inch, 0.05 inch). A grid of this resolution is displayed. Click on the Square Through-hole Pad Mode button in the mode toolbar on the left of the screen and place a DILSQ pad on the center marker in the edit window. Then select the Round Through-hole Pad Mode and place four DILCC pads at intervals of 0.1 inch to the right of the square pad. Complete the pin-out with five round pads 0.6 inch (12 squares) above this row. The cursor position is displayed in the status bar below. This should correspond to the positions of pins 1−10 in the dimensioned component front view seen on the data sheet.

Now draw a rectangle (2D Graphics Box Mode) around the pins to represent the outline of the component and select the whole device by looping around this outline in Selection Mode.

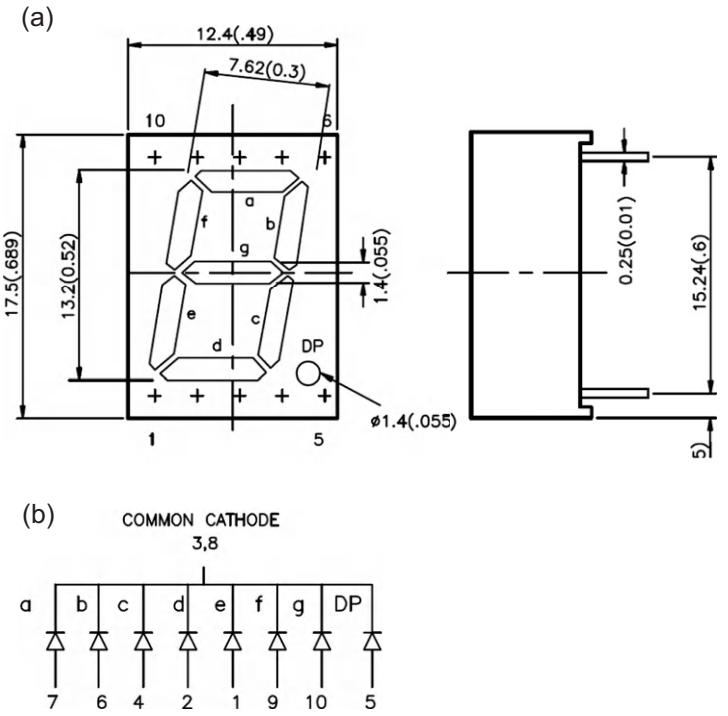


Figure E.5
Numerical display data: (a) display dimensions; (b) LED connections

Right-click on the highlighted package and Make Package. Give it a suitable name in the dialogue (e.g. 7SEGDIS), and select the Package Category: Miscellaneous and Package Type: Through Hole. It will be placed in the USERPKG library.

Now switch back to ISIS and assign packages to the other components where necessary. In the case of the PIC, it is pre-assigned in the Proteus library. In the case of others, such as the pot and tactile (push-button) switches, study the physical properties in the component data sheet. A suitable pin-out may, or may not, be found in the libraries, but it is relatively simple to make one as described above for the display. The pin-out for the pot is fairly standard for a finger pot, but the package for the switches had to be created.

If a component is placed in ISIS and the pins are already numbered, it implies that a package is already attached (e.g. U1, J1, J2, J3, DSW1, RP1, RN1). If not, a package must be assigned, or manufactured in ARES. It is attached by right-clicking on the component and selecting the Packaging Tool (Figure E.6).

In this window, add the required package (7SEGDIS) to the Packagings list from the Pick Packages dialogue. The device pins are listed by function, and the user must assign a pin

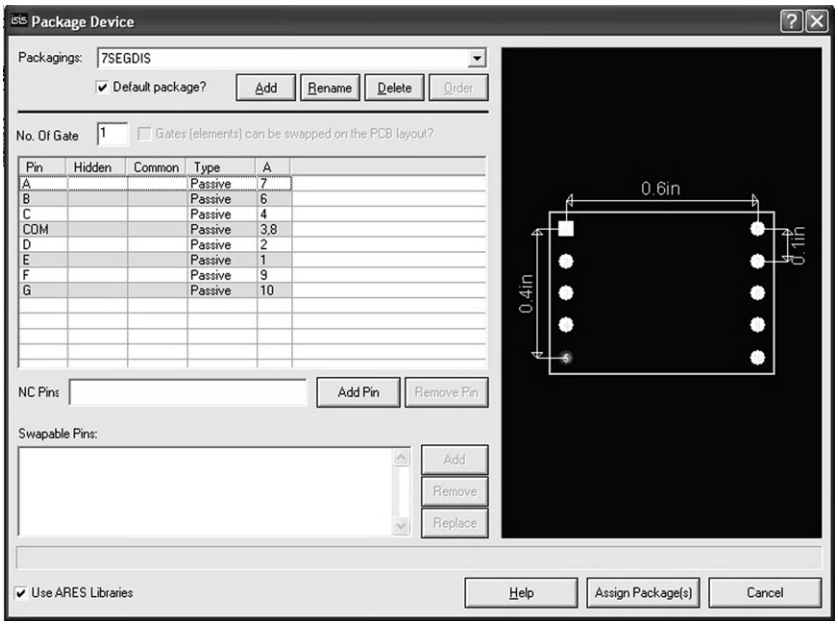


Figure E.6
Package device dialogue

number in column A. In this case the segment connections are copied from the numerical display data sheet common cathode connection diagram, for example, segment A = pin 7. When done, hit the Assign Package(s) button.

When all the components have been assigned a package, the netlist can be exported to ARES PCB layout (ARES button).

E.7. Layout Edit

When the netlist is imported into ARES, the components are listed in the device window, if component mode is selected. These can then be selected and dropped onto the edit window, with the connections initially shown as a rat's nest, directly between pins. Ensure that the imperial measurement is enabled, with the snap set to 50th. There is an automatic placement tool, but it is suggested that in this case the components be placed manually, with the MCU centrally positioned first, followed by the larger components. Try to minimize connection length and cross-overs, with the manually operated inputs, outputs and connectors conveniently placed together, from left to right. The programming connector must be at the edge of the board, with the pins horizontal to connect to PICkit2/3.

Assuming a single-sided board is to be designed, with topside links, open the Design Rule Manager dialogue and select the Net Classes tab. For the Power class, change Pair 1 (Hoz) to

'none', with Trace Style T30. For the Signal class, also change Pair 1 (Hoz) to 'none', with Trace Style T20. A layout with a bottom copper layer only will be produced.

The auto-router may now be invoked, which will place as many tracks as possible, leaving rat's nest connections for those left over. Some manual tidying will be needed, and the remaining tracks should be connected on the copper layer if possible, but using links if necessary. There are nine of these on the sample layout, with a through-hole (via) at each end.

Tracks are laid in copper (default blue) by clicking on the pin at each end. The rat's nest connection then disappears. If necessary, a double-click drops a via on the layout and the track goes red, representing the top layer or a wire link on a single-sided board. Double-click again and another via is dropped and the track reverts to bottom copper blue for final connection to the destination pin. Bends can be dropped by clicking, or by reselecting the track after completion and moving it about. Right-angle bends should be avoided by using a short section at 45° to facilitate current flow and avoid hotspots, especially on the supply tracks which may carry the maximum current. Some experimentation is necessary to achieve optimum results, with the components logically arranged with minimum links required.

Finally, a graphics box should be drawn around the layout; right-click on the boundary line and change the layer to Board Edge. Leave a margin for positioning a mounting screw, standoff or adhesive pad at each corner. The final layout is shown in Figure E.7.

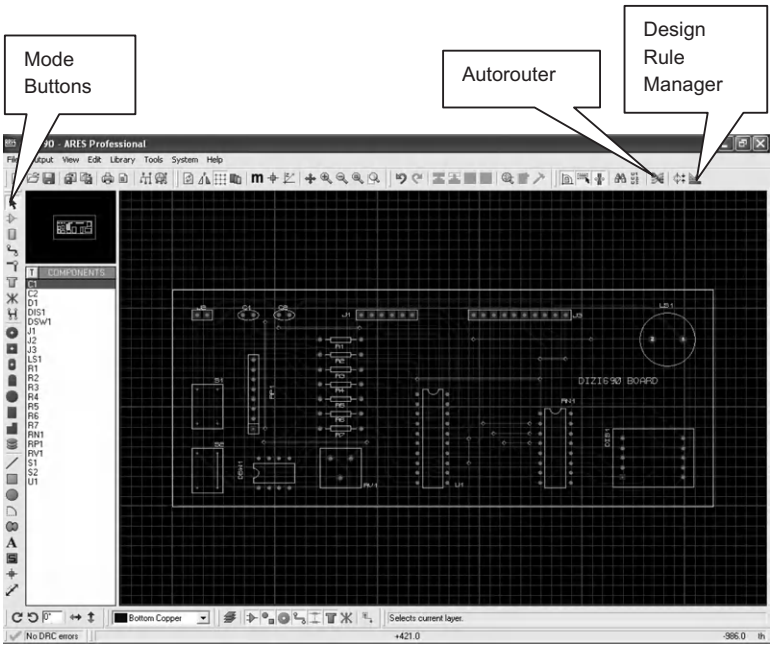


Figure E.7
ARES PCB layout editor screenshot

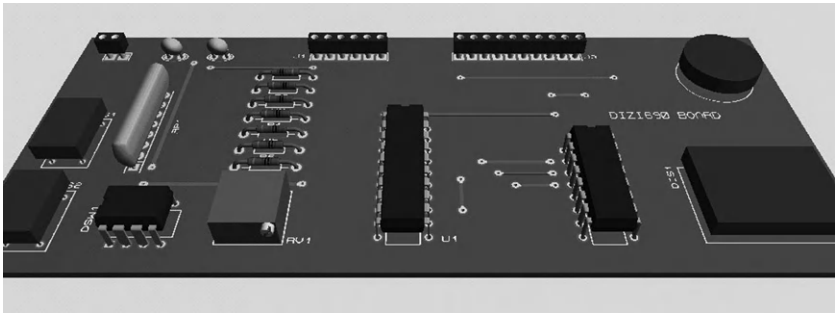


Figure E.8
3D visualization of PCB

When finished, the PCB can be displayed in 3D by ARES by selecting Output, 3D Visualization (Figure E.8) for final checking.

E.8. Output Files

The layout can now be exported as text files which are used as input to the PCB production process. In ARES, select Output, Gerber/Excellon Output. In the output dialogue, uncheck the Top Copper, Top Resist and Bottom Resist options, leaving the Bottom Copper, Top Silk, Drill and Edge options checked. Four files are then produced:

- `CADCAM  Bottom  Copper.TXT`
- `CADCAM  Drill.TXT`
- `CADCAM  READ-ME.TXT`
- `CADCAM  Top  Silk  Screen.TXT`

The first is a Gerber file, which comprises a list of commands for moving a plotter or machine between positions on the board defined by *X* and *Y* coordinates, creating the pads and traces. The format uses the same command codes as a standard CNC machine (G-codes). The first few lines are shown in List E.3 as an illustration, the whole file being much longer.

There are three main types of statement, examples of which are shown below (with added spaces):

```
X+18400  Y+13120  D01* = Trace

X+18800  Y+12720  D02* = Move

X+15500  Y+10520  D03* = Pad
```

The first means move to another position while plotting a trace (track), indicated by the drawing command D01 at the end. The second means move to a position without plotting, indicated by the command D02 at the end. The third means make a pad at this position, D03.

```
G04 PROTEUS RS274X GERBER FILE*
%FSLAX24Y24*%
%MOIN*%
%ADD10C,0.0200*%
%ADD11C,0.0300*%
%ADD12C,0.0500*%
%ADD13R,0.0500X0.0500*%
%ADD14R,0.0600X0.0600*%
%ADD15C,0.0700*%
%ADD16C,0.0800*%
%ADD17C,0.0400*%
%ADD18C,0.0900*%
%ADD19R,0.0800X0.0800*%
%ADD70C,0.0080*%
G54D10*
X+40000Y+8020D02*
X+40000Y+8020D01*
X+18800Y+12720D02*
X+18400Y+13120D01*
X+18400Y+17920D01*
X+19500Y+19020D01*
X+7600Y+11520D02*
X+7600Y+15120D01*
X+8000Y+15520D01*
X+13600Y+15520D01*
Etc
```

List E.3
Gerber file extract

The initialization commands at the top of the program specify the pad sizes and shapes, which are selected for each set of pads using the select tool command, for example, 'G54D12'.

This file is most frequently used to control a photo-plotter, which creates a transparency (Figure E.9) for making multiple boards photographically. The blank copper board is coated

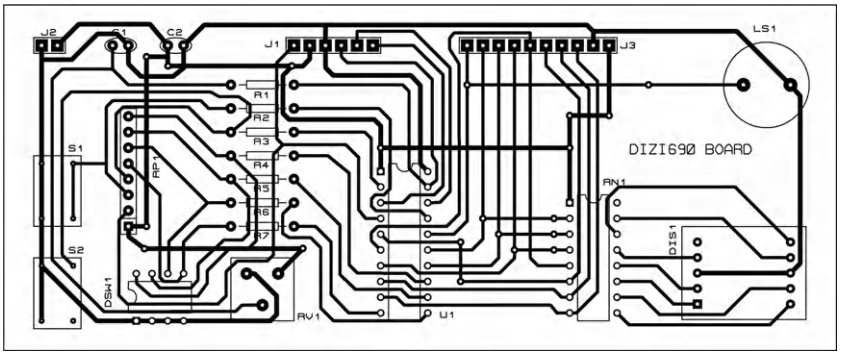


Figure E.9
PCB layout produced in ARES

with an etch-resistant layer, which is photosensitive. When exposed to ultraviolet light through the transparency, the exposed etch resist can be dissolved away, and the board developed by etching away the copper in an acid solution where it has been exposed. This leaves the copper traces and pads behind.

A cheaper way of achieving the same result is to send the layout as a bit map to a laser printer, and print onto translucent drafting film. This can then be used as the photo-resist masking image for the chemical etching process. This is the traditional method for hobbyists and educational purposes. Note that the layout must be reversed for exposure.

An alternative process is to use the same file to control a milling machine, which will isolate the tracks by machining around them with an engraving tool. This has the advantage of being chemical (hazard) free, but is not as precise. However, it does have the major advantage that the drilling operations (through holes) and milling (board edge) can be carried out on the same machine by replacing the engraving tool with a drill or milling tool. The machining option is probably best suited to making prototypes and small numbers of less complex boards. Such machines are now available at reasonable cost for the hobby and education markets.

The Excellon format drill file is similar to the Gerber file, consisting principally of a list of X,Y coordinates for each drill hole, with the drill sizes specified in the initialization code. The Top Silk file also contains a list of coordinates, which control the plotting of the alphanumeric characters to form the labeling of the components and connectors on the component (top) side of the board. The Read-Me file has information about the file list, formatting options and design information, since there is some variation in file formats and machine compatibility. Since all these files are plain text, they can be opened in any text editor and the control codes analyzed.

E.9. PCB Fabrication

Low-cost 2.5D milling machines are now available which make the mechanical route to a prototype PCB more attractive. They are basically an inexpensive drill attached to an $X-Y$ plotter. Limited vertical movement (Z) is also needed to lift the engraving tool and implement drilling operations. The software used here is Galaad Percival (www.galaad.net), designed to control a CIF mill (Figure E.10). A demo version of the software can be downloaded for experimentation; a dongle must be purchased to enable the machine interface.

Percival is loaded onto the host computer, which is attached to the mill via an RS232 link. It takes the ARES output files, displays the layout and calculates the engraving tool path required to make the PCB connections, and carry out the additional milling and drilling operations. This will leave most of the copper in place, as a ground plane. Centering holes can be added at each corner for mounting the board, if required.

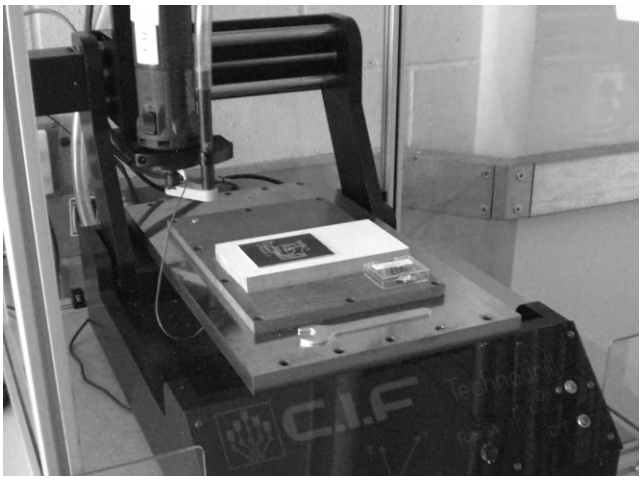


Figure E.10
CIF PCB mill

The main tools used will typically be an engraving bit with 60° point angle to separate the tracks from the rest of the copper layer, a 0.8 mm drill for the component lead holes and a 3 mm slot drill used as a milling tool to bore the mounting holes and other larger apertures, and finally cut round the board edge.

To process the board files in Percival layout editor and mill controller, we select Open, New Circuit and select the file CADCAM Bottom Copper.TXT, and the layout should appear in the edit window. Select File, Flip Horizontal to reverse the layout and to view from the copper side. Using Parameters, Selected Tools, open the Active Tools dialogue and assign the engraving, drilling and milling tools. Now select Machine, Calculate Contours and the toolpath will be calculated and displayed in yellow around the tracks. Display, Final Rendering and the engraved copper side will be displayed (Figure E.11).

The mill must be connected to the PC, and the Machine Parameters set up accordingly. When communication has been successfully established, the mill control panel is used to set up the tool in the initial (reference) position, setting up the reference plane to coincide exactly with the surface of the copper, which in turn must be completely level in relation to the X and Y axes. The engraving depth must be set up to provide a suitable track width. Some experimentation will be necessary to achieve acceptable results, for example, setting up suitable drilling depth, milling depth and speed. The PCB blank must be fitted on a sacrificial board, which will be drilled and milled along with the board. Hazardous copper and epoxy dust should be vacuum extracted from around the tool point while the machine is operating.

When successfully manufactured, the PCB can be populated and tested in the usual way.

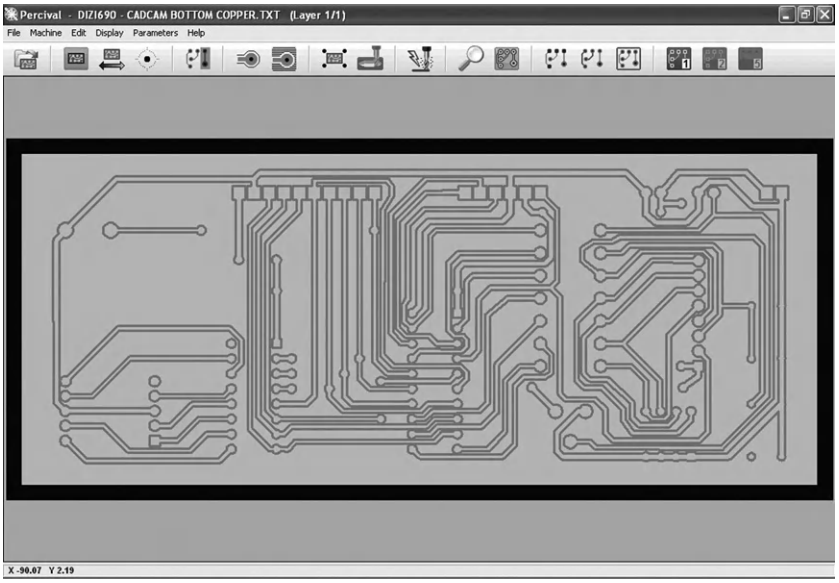


Figure E.11
Percival copper side layout

Answers to Questions

Chapter 1

1. Two of each required:
   Input devices: keyboard, mouse, scanner, microphone. (2)
   Output devices: screen/display, printer, speaker/headphone. (2)
   Storage devices: hard disk, memory stick, CD/R/W. (2)

2. It contains start-up code in non-volatile memory, required before the main operating system code has been transferred to RAM from hard disk, which takes some time. (4)

3. There are too many connections required between the chips for each to be made separately. (2)

4. Two of these:
   Can be designed to meet different requirements (e.g. games, business, education). (2)
   Can be repaired more easily with replacement modules. (2)
   Can be upgraded easily. (2)
   Is a very flexible design which has stood the test of time. (2)

5. ROM is non-volatile, so the program starts without delay in the MCU. (2)
   RAM is more compact, and therefore has greater data capacity, but is volatile, so it needs loading initially, which delays the start of the PC system. (2)

6. (a) CPU: controls the system and performs calculations. (2)
   (b) ROM: non-volatile memory contains program code. (2)
   (c) RAM: volatile memory contains user data and downloaded code. (2)
   (d) Address bus: carries address code for location required by CPU. (2)
   (e) Data bus: carries data to and from memory (RAM) and I/O registers. (2)
   (f) Address decoder: generates memory/I/O select signal. (2)
   (g) Program counter: holds the address of the current instruction (or next). (2)
   (h) Instruction register: holds the current machine code program instruction. (2)

7. The microprocessor system has the main elements (CPU, ROM, RAM, I/O) as separate chips connected together with address, data and control buses. The MCU has all these in one chip. It is used in smaller applications where not too much data memory is needed, such

as control systems. The microprocessor system, e.g. PC, has more memory and is used as a general-purpose computer for personal, business and educational use, for word processing, database management, design, etc. (8)

8. The hardware is designed and built. The MCU program is written in a text editor on a host computer, debugged, assembled to a hex file, tested in simulation, debugged again, downloaded and tested in hardware. (6)

(Total 50 marks)

Chapter 2

1. The program is stored as a list of binary machine codes in program memory, created by the assembler from the user source code. The instructions are executed in turn, copied to the instruction register and decoded to set up the processor control logic accordingly. The program counter stores the address of the current instruction, and is incremented automatically to the next. If a jump is required, the destination address is given as the instruction operand, which is placed in the program counter. The MCU clock drives the program execution along at four clock cycles per instruction. (5)

2. Original data = 01101010
   (a) 00000000, (b) 01101011, (c) 01101001, (d) 10010101, (e) 00110101, (f) 11010100, (g) 01001010, (h) 01101011 (1 each)

3. Original data = 01001011, 01100010
   (a) 01001011, (b) 10101101, (c) 01000010, (d) 01101011, (e) 00101001 (1 each)

4.

PC	Stack	
….	….	
02F2	XXXX	Instructions
02F3	XXXX	Before Call
016F	02F4	Subroutine Start
0170	02F4	Subroutine
0171	02F4	Instructions
0172	02F4	Return
02F4	XXXX	Instructions
02F5	XXXX	After Call
….	….	

(5)

5. Allocate registers A,B,C
   Clear register A
   Move 4 into register B
   Move 3 into register C

```
Loop1   Add B to A
        Decrement C
        Test C for zero
Jump back to 'Loop1' if C not zero
Finished with product in A                                    (7)
```

(Total 30 marks)

Chapter 3

1. 0A86. (2)
2. 30. (2)
3. Jump Destination. (2)
4. Labels go in first column. (2)
5. EQU, END, they are directions for the assembler, not part of the program. (3)
6. 00, 03. (2)
7. Memory address, machine code, line number, address label, instruction mnemonic, operand label. (1 each)
8. (a) Source code file, assembler program entered by user. (2)
   (b) Machine code file, generated by assembler. (2)
   (c) List file, shows all files combined. (2)

(Total 25 marks)

Chapter 4

1. e, c, b, f, a, d. (6)
2. ASM: Text file entered using instruction mnemonics. (2)
   HEX: Machine code created by assembler from source code. (2)
   LST: List file contains source and machine code plus label and memory allocation. (2)
3. Avoids repetition of code, saving on program memory. (2)
   Reusable within program and as a separate file. (2)
4. BTFSS or BTFSC. (2)
5. MOVLW 00 or CLRW. (2)
6. Replace count value 0FF with 07F or 080. (2)

7. A stimulus workbook table is created with the input pin selected in the first column and the operating mode (e.g. toggle) in the second. (4)

8. Watchdog timer off, code protection off, power-up timer on, RC clock. (4)

(Total 30 marks)

Chapter 5

1. Program memory stores the machine code in flash memory as a binary list. (2)
   The program counter stores the current address and is incremented for each instruction. (2)
   The instruction decoder converts the instruction code into a control line setup. (2)
   The multiplexer selects the data source for the ALU, literal or register. (2)
   The working register stores the current data for and receives results from the ALU. (2)

2. The data direction bits in the TRIS register default to 1 for input. (2)

3. The arithmetic and logic unit carries out operations such as increment, rotate and bit set on individual registers and add, subtract, AND and OR on pairs of registers. It receives one byte from the working register, and the other from either the instruction literal or a file register. (5)

4. The stack stores the return address automatically so the program can return to the next instruction after the subroutine has finished. (2)

5. PORTA is the data register for the port pins RA0−RA3. (1)
   TRISA is the data direction register for PORTA. (1)
   TMR0 is the timer/counter register. (1)
   PCLATH is the program counter high byte, bits 8−12. (1)
   GPR1 is the first general purpose register in RAM. (1)

6. STATUS,2 is the zero flag, Z, which is set when result zero occurs in a register. (2)
   INTCON,1 is the RB0 interrupt flag, INTF, set when RB0 changes to force an interrupt. (2)
   OPTION,5 is the Timer0 clock source select bit, T0CS, internal or RA4. (2)

(Total 30 marks)

Chapter 6

1. (a) 4 clock cycles, (b) 1 instruction cycle, (c) 2 instruction cycles. (3)

2. Instruction cycle = 40 μs. (2)

3. 1 ms = 1000 μs, clock = 1 μs, count = 1000/4 = 250 = max count 256 − 6, preload = 6. (3)

4. TRISB 3,4,5,6 = 0 and INTCON 3,7 = 1. (2)

5. RC: Adjustable. (1)
   XT: Accurate. (1)
   HS: Maximum clock speed. (1)
   INTOSC: No external components needed. (1)

6. END. (2)

7. Subroutine: block of code only assembled once and called many times, uses less memory. (2)
   Macro: block of code that is inserted each time it is needed, faster. (2)

(Total 20 marks)

Chapter 7

1. Pin 1: !MCLR Master Clear/V_{pp} programming voltage. (1)
   Pin 2: V_{DD} positive supply, 5 V nominal. (1)
   Pin 3: V_{SS} negative supply, 0 V. (1)
   Pin 4: ICSPDAT, In-circuit serial programming data. (1)
   Pin 5: ICPSCLK, In-circuit serial programming clock. (1)

2. The port pins default to analogue inputs. (2)

3. Maximum value = 1024/4 = 256 mV, 1 mV per bit. (4)

4. Testing by simulation allows logical errors to be eliminated more quickly than downloading and testing in hardware. (3)

5. The button input on !MCLR is by-passed by the programmer connected to the same pin, so it can only be operated from the host computer. (3)

6. Three of the following: more I/O pins (18 vs 33), LEDs (8 vs 4), separate timer clock, external crystal clock, in-circuit debugging. (3)

(Total 20 marks)

Chapter 8

1. See Figure 8.1. (2)
   The average current in the load can be controlled by switching it on and off over a fixed cycle and varying the mark/space ratio. (2)

2. The block diagram is an outline of the hardware showing the main circuit blocks and signal flow between them, which can be converted into a circuit diagram. (2)

The flowchart is an outline of the software showing the main operations and the sequence of execution of these operations, using a small set of symbols for terminators, processes, input, output and branches, which can be converted into source code. (2)

3. Pseudocode – a text outline – can be converted directly into source code in the editor. (2)
   Structure cart – a block diagram – shows the program hierarchy in complex programs. (2)

4. (a) Assigns a label to the port bit connected to the motor output. (2)
   (b) Tests the active low input bit which enables the motor to run, and wait if not pressed. (2)
   (c) Follows the decrementing of the speed variable to correct register roll-under to prevent change from zero to maximum speed. (2)
   (d) Complements speed control count to generate off delay, so that overall period is constant. (2)

(Total 20 marks)

Chapter 9

1. Syntax error: incorrect source code instruction, detected by the assembler, reported in the error messages. (2)
   Logical error: program sequence or output effect is incorrect in relation to the specification, detected by source code debugging. (2)

2. Single stepping: one instruction at a time with a break after each to check sequence. (1)
   Breakpoint: stop the program at selected points to check function, or start single stepping. (1)
   Pin stimulus: changes to single inputs in the simulator, asynchronous or programmed changes specified in the workbook. (1)
   Watch window: selected registers can be displayed and updated continuously. (1)

3. 0005: source code line number. (1)
   1A05: machine code in hex. (1)
   start: jump destination label. (1)
   btfsc: assembler instruction mnemonic. (1)
   0x5: register to be operated on. (1)
   0x4: register bit to be operated on. (1)

4. Two of:
   Inputs can be operated on screen. (1)
   Outputs can be seen immediately. (1)
   Animated schematic with voltage/logic levels. (1)
   Integrates with schematic editing and hardware layout. (1)

5. Two of:
   Inspect the board for correct components, build faults, etc. (2)
   Check power supply to MCU pins before fitting. (2)
   Check for overheating in components. (2)
   (Total 20 marks)

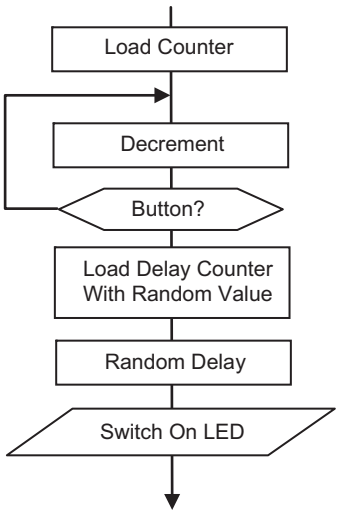
Chapter 10

1. (a) Breadboard: quick and easy to construct. (1)
   unreliable. (1)
   (b) Stripboard: reliable connections. (1)
   not suitable for volume production. (1)
   (c) Simulation: does not require hardware. (1)
   cannot fully represent the final hardware. (1)
   (Or other answer given in the text.)

2. (a) 0000 000x. (2)
   (b) 1011 101x. (2)

3. Clock = 4 MHz, timer clock = 1 MHz, period = 1 μs.
   Required frequency = 1 kHz, period = 1 ms.
   Half cycle = 500 μs, set prescale to 2.
   Preload timer with 6.
   Wait for timeout flag after 250 cycles.
   Toggle output.
   Reload timer and repeat. (5)

4. (5)



(Total 20 marks)

Chapter 11

1. Pulse width modulation switches the drive to a motor on and off rapidly using a pulse waveform. The ratio of the on and off time is varied to control the average current in the motor, hence the speed. (2)
   A power transistor is needed to switch the current in the motor, and a microcontroller to generate PWM. The required speed can be input as an analogue or digital value. (2)

2. A slotted wheel or similar device is attached to the motor shaft, which produce pulses in a sensor. The pulses can be counted to monitor the position, and frequency or period can be measured to calculate the speed. (4)

3. The microcontroller can control the speed of the motor using PWM. It can measure the actual speed using an encoder attached to the motor shaft, and modify the duty cycle of the output until the feedback matches the required speed. (4)

4. (a) Number of steps per output rev $= 90 \times 200 = 18\,000$.
   Resolution/accuracy $= 360/18000 = 0.02°$. (4)
   (b) Circumference of circle $= 2\pi r = 2 \times 3.14 \times 0.5 = 3.14$ m.
   Arc of circle $= 0.02/360 \times 3.14$ m $= 0.174 \sim 0.2$ mm. (4)

(Total 20 marks)

Chapter 12

1. The PIC10 range is the smallest of the range, with only four I/O pins in an eight-pin package. They have only one 8-bit timer, and two analogue inputs, with an 8 MHz internal oscillator. Program memory is 0.5k, with a maximum of 23 RAM locations. (4)
   The PIC12 range have six I/O pins in the same package, with an additional 16-bit timer and four analogue inputs. The internal oscillator can be replaced by a 20 MHz external crystal oscillator. Program memory is slightly greater (1k) with more RAM (256 bytes). (4)
   The PIC16 range is much more extensive, with chips up to 64 pins and 55 I/O. The largest have nine timers and 30 analogue input channels. Program memory is up to 16k instructions, and 1536 RAM locations. Various serial ports are available. (4)
   The PIC18 range has a more complex and extensive 16-bit instruction set, up to 64k program capacity and 4k RAM. The maximum speed is doubled to 40 MHz, with a much greater choice of chips than the other groups. A USB serial port is added. (4)

2. The MCU does not have to be removed, avoiding damage and saving time. (2)
   The chip can be reprogrammed quickly and easily. (2)

3. (a) 16F676. (2)
   (b) 18F4580. (2)

4. Capture mode uses an external signal to capture the count in a timer register, which allows a time interval to be measured. Compare mode waits for the timer count to match a preset value in a reference register and thereby generate a timed output pulse.(3)

5. SPI uses hardware slave selection using an extra signal input to the target peripheral. I^2C uses software slave selection, with the address given in the data frame. SPI therefore has more complex hardware requirements. (3)

(Total 30 marks)

Chapter 13

1. (a) A dc amplifier to increase the voltage range and screening. (4)
   (b) 250 mV, 125. (4)

2. (a) The relay provides complete electrical isolation of the load. (2)
   (b) The FET switches more quickly, allowing PWM. (2)

3. Multiplexing means the displays are switched on alternately using the same set of outputs to operate them, reducing the I/O requirement. (4)

4. Two of: internal oscillator, analogue inputs, PWM output, serial ports. (4)

5. Assembler relates directly to the internal architecture and may be more straight forward for simple applications, and uses less memory. However, for more complex applications, C source code is shorter and simpler, provides better features for data handling, etc., and is easier to derive from the typical program outline. (5)

(Total 25 marks)

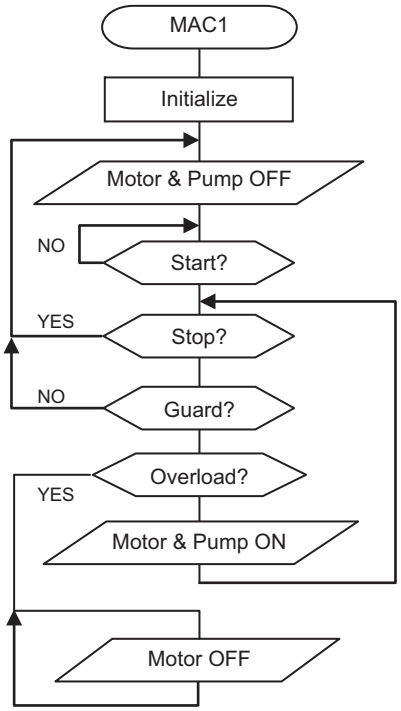
Chapter 14

1. The 8051 was a CISC processor with conventional Von Neumann architecture, i.e. an internal data bus shared by instructions and data, reducing its performance. The PIC MCU has Harvard architecture, which separates data and instructions, and a RISC instruction set, allowing faster execution at the same clock speed. (4)

2. Advantages: Hardware can be tailored to the application. (1)
   Large memory can be handled. (1)
   Disadvantages: Two of:
   More hardware design needed. (1)
   More expensive. (1)
   Simulation not possible. (1)

3. The PLC is self-contained, has built in interfacing, an easier programming method (ladder logic) and features for system integration. (4)

4. (7)



5. Six of: Programming host, SCADA host, database server, network server, network client, word processor, spreadsheet host, CAD workstation. (6)

6. (a) 4, (b) 1, (c) 2, (d) 5, (e) 3. (5)

(Total 25 marks)

Links and Acknowledgements

Links

www.picmicros.org.uk
The author's support site, providing:
- Downloads of application examples for this and related books
- Information about related books
- Current updates from the author

www.microchip.com
Manufacturer of the PIC microcontroller and support products
- PIC microcontroller product information
- Development system free downloads (MPLAB IDE)
- Demo system hardware information

www.labcenter.com
Supplier of Proteus VSM ECAD design software
- ISIS schematic capture and interactive co-simulation
- PIC simulation models and demo versions
- ARES PCB design

www.galaad.net
Supplier of PCB CNC production software

By the same author

M. Bates. *Interfacing PIC Microcontrollers: Embedded Design by Interactive Simulation* (Newnes 2006).

M. Bates. *Programming 8-bit PIC Microcontrollers in C with Interactive Hardware Simulation* (Newnes 2008).

Acknowledgements

The author gratefully acknowledges the use of development software and product information supplied by Microchip Inc. (US), Labcenter Electronics (UK) and Galaad (France).

Demo Files

The following demonstration files can be downloaded from www.picmicros.org.uk. In most cases, a Proteus VSM design file (APPX.DSN) is provided for testing. If necessary, the APPX.HEX file attached to the MCU must be changed to test each application *. If VSM is not available, test in MPLAB IDE.

BINAPPS*

Very simple programs for the BIN board with LED outputs.
Use MPLAB IDE, LED1.DSN simulation design or hardware.

BIN1	Minimal program with no labels outputs a binary count
BIN2	As BIN1 with labels
BIN3	Outputs a binary count with run and reset inputs
BIN4	As BIN3 with delay subroutine
BIN5	Simulation version of BIN4

GENAPPS*

Demonstrate various programming techniques.
Test using MPLAB IDE, LED2.DSN simulation design or hardware.

ASD1	Demonstrates the use of assembler directives
EEP1	Demonstrates the use of EEPROM storage
INT1	Demonstrates the use of interrupts
INX1	Demonstrates the use of indexed addressing
TAB1	Demonstrates the use of a program data table
TIM1	Demonstrates the use of the hardware timer

DIZI84*

Apps for DIZI84 board with digit display and audio output.
Test using DIZI84.DSN simulation design or hardware.

BEL1	Generates a random tone sequence
BUZZ1	Generates a fixed audio tone
DICE1	Displays random numbers 1 to 6
DIZ1	Test program for DIZI board
GEN1	Variable frequency audio output
GIT1	Outputs six guitar string frequencies
HEX1	Displays hexadecimal numbers
LOCK1	Operates as an electronic lock using EEPROM
MESS1	Displays a message on the 7-segment display
MET1	Metronome simulator outputs low frequency ticks
REACT1	Reaction timer measures delay between beep and button
SCALE1	Outputs a scale of 8 tones
SEC1	Displays a one second count 0 to 9

MOTORS

Demonstrate motor control using various hardware.
Test using individual simulation design.

CLS2	Runs a dc motor under closed loop speed control in MOT2 board
DCM1	Runs the dc motor on the MOT2 board forward and reverse
HOB1	Operates a hobby servo using PWM from 16F690 on LPC board
MOT1	Open loop speed control of dc motor simulated in MPLAB using 16F84
POS2	Moves a dc motor to set position in MOT2 board

ADVAPPS

Various demo board designs and test programs.
Test using individual simulation design.

DIZ690	Design, test program and PCB layout for DIZI690 demo board
KEY690	Displays two digits from a keypad using a 16F690 in simulation mode
LPC690	Simulation of Microchip LPC demo board with test program
TEM887	Simulation of temperature controller board running test program

Index

C

D

L

M

R

S

X

Z