PIC Microcontrollers
This Page is Intentionally Left Blank
PIC Microcontrollers
An Introduction to Microelectronics
Second Edition

Martin Bates
Contents

Preface to the First Edition x
Preface to the Second Edition xii
Introduction xiii

PART A MICROELECTRONIC SYSTEMS 1

Chapter 1 Computer Systems 3
  1.1 The PC System 3
  1.2 Wordprocessor Operation 9
  1.3 PC Microprocessor System 11
  1.4 PC Engineering Applications 14
  Summary 18
  Questions 18
  Activities 19

Chapter 2 Information Coding 20
  2.1 Number Systems 20
  2.2 Machine Code Programs 25
  2.3 ASCII Codes 28
  Summary 29
  Answers 30
  Activities 30
## Contents

### Chapter 3 Microelectronic Devices
- 3.1 Digital Devices 32
- 3.2 Combinational Logic 36
- 3.3 Sequential Logic 39
- 3.4 Data Devices 41
- 3.5 Simple Data System 43
- 3.6 4-Bit Data System 44
- Summary 47
- Questions 47
- Activities 48

### Chapter 4 Digital Systems
- 4.1 Decoders and Encoders 49
- 4.2 Multiplexers, Demultiplexers and Buffers 51
- 4.3 Registers and Memories 51
- 4.4 Binary Number Coding 51
- 4.5 System Address Decoding 54
- 4.6 Memory Address Decoding 55
- 4.7 Serial and Shift Registers 56
- 4.8 Arithmetic and Logic Unit 57
- 4.9 Processor Control 58
- Summary 59
- Questions 59
- Answers 60
- Activities 60

### Chapter 5 Microcontroller Operation
- 5.1 Microcontroller Architecture 61
- 5.2 Program Operation 65
- Summary 73
- Questions 74
- Answers 75
- Activities 75

### Part II THE PIC MICROCONTROLLER

### Chapter 6 A Simple PIC Application
- 6.1 Hardware Design 79
- 6.2 Program Execution 83
- 6.3 Program BIN1 85
- 6.4 Assembly Language 87
### Chapter 7: PIC Program Development
- 7.1 Program Design
- 7.2 Program Editing
- 7.3 Program Structure
- 7.4 Program Analysis
- 7.5 Program Assembly
- 7.6 Program Simulation
- 7.7 Program Downloading
- 7.8 Program Testing

### Chapter 8: PIC 16F84 Architecture
- 8.1 Block Diagram
- 8.2 Program Execution
- 8.3 Register Set

### Chapter 9: Further Programming Techniques
- 9.1 Program Timing
- 9.2 Hardware Counter/Timer
- 9.3 Special Features
- 9.4 Numerical Types

### Summary
90

### Questions
90

### Answers
91

### Activities
91

### Chapter 7: PIC Program Development
- 7.1 Program Design
- 7.2 Program Editing
- 7.3 Program Structure
- 7.4 Program Analysis
- 7.5 Program Assembly
- 7.6 Program Simulation
- 7.7 Program Downloading
- 7.8 Program Testing

### Summary
115

### Questions
115

### Answers
116

### Activities
116

### Chapter 8: PIC 16F84 Architecture
- 8.1 Block Diagram
- 8.2 Program Execution
- 8.3 Register Set

### Summary
126

### Questions
127

### Activities
127

### Chapter 9: Further Programming Techniques
- 9.1 Program Timing
- 9.2 Hardware Counter/Timer
- 9.3 Special Features
- 9.4 Numerical Types

### Summary
155

### Questions
155

### Answers
156

### Activities
156
## Contents

**PART D MORE CONTROLLERS** 225

### Chapter 14 More PIC Microcontrollers 237

14.1 Common Features of PIC Microcontrollers 237
14.2 Selecting a PIC 242
14.3 Advanced PIC Functions 251
14.4 Serial Communications 254

**Chapter 15 More PIC Applications and Devices** 256

15.1 16F877 Application 256
15.2 16F818 Application 273
15.3 12F675 Application 274
15.4 18F452 Application 275

### Chapter 16 More Control Systems 280

16.1 Other Microcontrollers 280
16.2 Microprocessor System 282
16.3 Control Technologies 288
16.4 Control System Design 298

### Appendix A PIC 16F84 Data Sheet 302

### Appendix B DIZI-2 Board and Lock Application 347

### Index 367
Preface to the First Edition

The Microchip™ PIC 16F84 microcontroller is an unremarkable looking 18-pin chip – so why write a whole book on it? The answer is that it contains within its ordinary looking plastic case most of the technology that students of microelectronics need to know about in order to understand microprocessor and computer systems. It also represents a significant new development in microelectronics and, importantly, it offers an easier introduction to the world of digital processing and control than conventional microprocessors. The microcontroller is a self-contained, programmable device, and the student, hobbyist or engineer can put it to use without becoming involved in the complexities of microprocessor operation. On the other hand, we can learn a great deal about microelectronics by looking inside.

Studying the PIC chip will give the user a valuable insight into the technology behind the explosion in microprocessor-controlled applications which has occurred in recent years, which has been based on cheap, mass-produced digital circuits. Mobile phones, video camcorders, digital television, satellite broadcasting and stereo receivers – these are not many current electronic products which do not contain some kind of microprocessor. Industrial control systems have seen similar developments, where complex computer control systems have usually increased productivity, quality and reliability. The list of products is the nearest in power of the microprocessor.

The microcontroller is essentially a complete computer on one chip, which can carry out a complex programmed sequence of actions, with the minimum of additional components. As an example, in this book a motor control circuit will be described which allows the motion of a small dc motor to be programmed and controlled by the PIC chip. The only additional major components required are power transistors to provide the required drive to the motor. In the past, such projects would have required many more components, and have much more complicated circuit diagrams and device connections. In the future, microcontrollers will be developed which will be able to control single motor applications in its own right.

When I first came across the PIC chip a few years ago, it was immediately obvious that this would be an ideal device for teaching and learning microprocessor software techniques, especially for students with minimal prior knowledge and skills. It is relatively cheap, and even better, it has user-replaceable program memory that is electrically erasable programmable (Flash ROM). In addition, the manufacturers, Microchip Technologies, had the foresight to make development system software widely available, and the support hardware and software are being added to all the time, by the manufacturer, independent suppliers and enthusiasts. On the other hand, a complete set of more powerful development tools is also available for the professional user.

Both DOS and Windows versions of the PIC development system have been used to prepare the sample applications in this book, and the programs downloaded using the PICSTART-16B
Preface to the First Edition

Martin P. Bates
Lecturer in Microelectronics
Hastings College of Arts & Technology
July 1999

programming unit. However, there are many designs for inexpensive programmers available in magazines and on the Internet, usually with their own software. The current Windows version of the program development package, MPLAB, can be downloaded free of charge from the Internet at 'http://www.microchip.com,' along with data sheets and all the latest product information. The data sheet for the PIC 16F84 is reprinted in full, because it is an excellent document which contains the definitive information on the chip, presented in a clear and concise manner.

The objective of this book is to ensure that any beginner, student or engineer, will quickly be able to start using this chip for their own projects and designs. When I started using it in my teaching, I put together a teaching pack and was expecting a range of suitable reference books to quickly appear. Indeed, the chip soon started to feature in numerous electronics magazine projects and was clearly popular, but all the books that I obtained seemed to assume quite a lot of prior knowledge of microprocessors. I wanted to make the PIC as simple as possible, so I decided to write a suitable book, I would have to do it myself! I hope that the reader finds the result useful.
Preface to the Second Edition

The revisions required in the second edition of this book are mainly due to the rapid development of microcontroller technology. As the PIC family of devices has grown, more features have been incorporated at lower cost. So, while the focus of the first edition was the popular 16F84 chip, and this remains a valuable reference point for the beginner, the scope has been expanded so that a broader understanding of the range of microcontroller types and applications can be gained.

One of the reasons the 16F84 was originally selected was its flash memory, which allows easy reprogramming, making it a good choice for education and training. Flash memory is now available in a wider range of devices, making the choice of chip less obvious. On the one hand the user now has more small 8-pin devices which can be used in simple systems requiring fewer inputs and outputs, as well as a proliferation of more powerful devices incorporating a variety of serial data interfaces, as well as analog inputs and many other advanced features.

For this reason the focus has been shifted away from the 16F84. A wider selection of devices and I/O methods is now discussed, and a more general treatment attempted. Application development software has also moved on, and new methods of programming and debugging introduced. I hope I have been able to reflect these developments adequately without introducing too many complications for the beginner, to whom this text is still firmly addressed.

Part A is a general introduction to microcontroller system technology, and can be skipped if appropriate. Part B, the PIC microcontroller is described in detail from first principles. Part C contains practical advice on implementing PIC projects, with examples. Part D contains new material on the more advanced features of other PIC MCUs (Microcontroller Units) as well as other control system technologies.

I have tried to incorporate a systematic approach to project development, making the design process as explicit as possible. The book will thus support the delivery of the microelectronic systems and project modules of, for example, UK BTEC electronics programmes which incorporate an Integrated Vocational Assignment, which requires the student to develop a specific project and document the process in detail. The PIC is a good choice for producing interesting, but achievable projects which incorporate a good balance between hardware and software. Application development software has also moved on, with new methods of programming and debugging introduced.

Acknowledgements are due to Microchip Technology Inc. for their kind permission to reproduce the PIC 16F84A data sheet, to Microsoft Corporation and Labcenter Electronics for the application software used to produce documents, drawings, circuit schematics and layouts for the book, and to all for use of their trademarks.

Finally, thanks to the following for their help, advice and tolerance: Melvyn Ball (Hastings College), Jason Guest (General Dynamics, Hastings), Chris Garrett (University of Brighton) and, of course, Julie at home; also, to all colleagues who commented on the first edition, and students who bought it!

Martin Bates
December 2003
mbates@hastings.ac.uk
Let's admit one thing straight away - microprocessor systems are quite complicated! However, they are now found in so many different products that all students of engineering need to know something about how they work.

In this book we are going to look specifically at the PIC family of microcontrollers. Microcontrollers have all the essential features of a full-size computer, but all on a single chip. By contrast, conventional microprocessor systems, such as the PC (personal computer), are built with a separate processor, memory, input and output chips. The extra hardware and software required to make these chips work together makes the system much more difficult to understand than one single-chip microcontroller unit (MCU).

As well as being easier to understand, microcontrollers are important because they make electronic circuits cheaper and easier to build. Hand-wired circuits can be replaced with a microcontroller and its software, reducing the number of components required. Importantly, the software element (control program) can be reproduced at minimal cost, once it has been created. For the development costs may be high, but the production costs will be lower in the long run. It is also easier to change software if the product is to be modified. In general, software is increasingly replacing hardware in electronic design. For example, to design a system that needs to work under different conditions, the software can be changed to suit the new conditions or requirements, whereas hardware changes may require redesigning.

So the development costs may be higher, but the production costs will be lower in the long run. It is also easier to change software if the product is to be modified. In general, software is increasingly replacing hardware in electronic design. For example, to design a system that needs to work under different conditions, the software can be changed to suit the new conditions or requirements, whereas hardware changes may require redesigning.

Using the PIC, we will find that we can quite quickly work out some simple, but useful, applications. These will illustrate the universal principles of microprocessor systems that apply to many complete computers and control circuits. As first, however, we do not have to worry too much about exactly how the chip works - we will go back to that later. The big problem with microprocessors and microcontrollers is that in order to fully understand how the system works, 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 reached.

We will approach microcontroller and microprocessor systems (microsystems) step by step, assuming very little prior knowledge. The operation of the PIC will be outlined first, because most students will be familiar with how it works from the user's point of view. We will look at how the hardware and software interact, and the function of the Pentium microprocessor in controlling the input (keyboard, mouse), output (screen) and memory and disks.

Some basic microelectronic system principles will then be covered. One objective is to understand the hardware diagrams in the PIC data sheets, so that external circuits connected to the PIC input/output pins can be designed correctly. Also, it is important to understand the internal hardware configuration of a microcontroller to fully understand the programming of the chip. The clarity and completeness of these data sheets is an important reason for choosing the PIC as our typical microcontroller. We can then start to look specifically at the PIC microcontroller and develop simple applications which will illustrate the essential hardware
features and basic programming ideas. More details will then be added using further application examples.

In the final section, the complete application design process will be described, including use of the PIC development system and hardware design methods. The range of PIC microcontrollers and other control technologies which can carry out similar functions to microcontrollers, such as programmable logic controllers, will also be described.

All reference material can be downloaded from www.microchip.com and other manufacturers’ websites.
Part A
Microelectronic Systems

1 Computer Systems
2 Information Coding
3 Microelectronic Devices
4 Digital Systems
5 Microcontroller Operation
Chapter 1
Computer Systems

1.1 The PC System

The PC hardware is based on the Intel series of microprocessors with Microsoft Windows operating system software. The standard PC hardware comprises a main unit, separate keyboard and mouse, VDU (visual display unit) and possibly a printer and connection to a network.

The circuit board (motherboard) in the main unit contains a group of chips which work together with the software and other hardware devices such as the keyboard and mouse, to perform computing tasks.

The main unit contains a group of chips that work together with the software and other hardware devices such as the keyboard and mouse, to perform computing tasks. These chips include the microprocessor, which is responsible for executing instructions, and memory chips, which store data and instructions. The microprocessor reads instructions from memory and executes them, while the memory chips hold data and instructions that are needed for the execution of the program.

The software, which includes the operating system and applications, provides an interface between the user and the hardware. It allows the user to interact with the computer, perform calculations, and store and retrieve data.

The PC system includes a variety of hardware components, such as the main unit, keyboard, mouse, VDU, printer, and network connection. The main unit contains the microprocessor and memory chips, which work together to perform computing tasks. The keyboard and mouse allow the user to input data and instructions into the system, while the VDU displays the results of the calculations and data.

The printer provides a hard copy of the results, which can be useful in certain situations. The network connection allows the user to communicate with other computer systems over a network, which can be useful for sharing files, accessing resources, and collaborating with others.

We will begin our study of microsystems with something familiar, by looking at how the PC works when running a wordprocessor. Most readers will be familiar with using a wordprocessor and will know more or less how it functions from the user’s point of view. Some basic microsystem concepts will be introduced by analysing how the software operates with the computer hardware, to allow 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.

It is also useful to get some idea of how a PC works because it is used as 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 created and downloaded to the PIC chip using the PC. The PIC development system hardware can be seen connected to the PC in Fig. 1.1(a). A simplified diagram, Fig. 1.1(b), allows us to see the main parts of the system more clearly.

We will then have a quick look at a microcontroller system, set up to operate as a simple equivalent of the microprocessor-based PC system, so we can see how it compares. The microcontroller has a keypad with only 12 keys instead of a keyboard, and a seven-segment display instead of a screen. In practice it is much smaller than the PC, yet it can carry out the same tasks.

The microcontroller can be used in a great variety of circuits. Also, it is much cheaper!
Computer Systems

![Computer System Diagram](image)

Figure 1.1  (a) The PC system (with PIC development system); (b) Diagram of PC system.

(a) Computer System Diagram

(b) Diagram of PC System

A computer system is used 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 processor must have access to software (programs) to allow useful work to be done by the hardware. These are usually stored on a hard disk inside the main unit. This can hold large amounts of data that is retained when the power is off. There are two main types of software required – the operating system (Windows™) and the application (Word™). As well as the operating system and application software, the hard disk stores the data created by the user (document files). Documents can also be stored on floppy disk for backup or portability.
The keyboard is used for data input, and the VDU displays the resulting document. The mouse provides an additional input device, allowing operations to be selected from menus or by clicking on icons and buttons. This is called the graphical user interface (GUI). There may be a network card fitted in the PC to exchange information with other users, download data or share resources such as printers over a local area network (LAN). In addition, a modem can give direct access to a wide area network (WAN), usually the Internet. A CD-ROM drive allows mass storage of reference information stored on optical disk to be accessed, and it also used to load application software.

If we remove the cover from the main unit, the main components can be identified easily. In the photograph, Figure 1.2(a), the power supply is top left, with the hard disk drive below it and the motherboard vertical at the back of the tower case. The disk and video interface cards are visible at the bottom, slotted into edge connectors into the motherboard, with a modem in the cabinet in the dark casing. The connections to the video board and printers are available at the rear, with some of the interfaces having been built into the motherboard, so the whole package is more compact.

Block diagrams are useful for showing the main parts of a complex system, and how they connect together in a simplified form. Block diagrams are used for showing the main parts of a complex system, and how they connect together in a simplified form. Figure 1.2(b) shows the components of the PC system and the directions of the information flow between them. In the case of the disk drives and network it is bidirectional (flowing in both directions), representing the process of saving data to, and retrieving data from, the hard disk or floppy disk.

1.1.1 PC Hardware

Inside the PC main unit, a 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 interfaces are usually on the motherboard. In older designs, the 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. Note that the functional block diagrams do not show any difference between internally and externally fitted peripherals, because it is not relevant to the overall system operation.

The PC is a modular system, which allows the hardware to be put together to meet the individual user’s requirements, and allows subsystems, such as disk drives and keyboard, to be easily replaced if faulty. The modular design also allows upgrading (for instance, fitting extra memory) or changing the type of peripheral modules if some are not used. In this case, the PC can be ‘tradesized’ (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 PC as a universal hardware platform.

1.1.2 PC Motherboard

The main features of a typical motherboard are shown in Fig. 1.3. The heart of the system is the microprocessor, a single chip, which is also called the central processing unit (CPU). The name is derived from the days when the CPU was built from discrete components and could be the size of a washing machine! In Fig. 1.3(a), the CPU is under the cooling fan at the lower right. The CPU controls all the other system components, but most of the functions are now stored in the CPU. The CPU requires program instructions to be stored in memory before it can do anything useful. The blocks of program required are provided by the operating system software and the application software which are downloaded to memory from the hard disk on startup.
The Intel CPU has undergone continuous development since the introduction of the PC in 1981, with the Pentium processor being the current standard. Intel processors are classified as CISC (complex instruction set computer) devices, which means they have a relatively large number of instructions which can be used in a number of different ways. This makes them powerful, but relatively slow compared with more streamlined processors which have fewer instructions.

As stated above, CPU cannot work on its own; it needs some memory and input/output devices for getting data in, storing it and sending it out again. The main memory block is
Figure 1.3  PC motherboard. (a) PC motherboard in the main unit; (b) layout of PC motherboard.
Computer systems are made up of RAM (read and write memory) chips, which are mounted in SIMMs (single in-line memory modules). Higher capacity DIMMs (dual in-line memory modules) are used currently. These can be seen at the top of the photograph in Fig. 1.3(a). Additional peripheral interfacing boards are fitted in the expansion card slots to connect the main board to the disk drives, VDU, printer and network. Spare slots allow additional peripheral interfaces and more memory to be added if required. Each peripheral interface is a sub-circuit which is built around a specific input/output chip (or set of chips) which handles the data transfer.

The integrated support device (ISD) is a chip which provides various system control and memory management functions in one chip, and is designed for that particular motherboard. The motherboard itself can be represented as a block diagram (Fig. 1.4) to show how the components are interconnected.

When the data has been processed and stored, it can be sent to an output peripheral, such as the screen. We will look at how this is achieved in more detail later.

Busses connect all the main chips in the system together, but, because they operate as shared connections, can only pass data to or from one peripheral interface or memory location at a time. This limits the speed at which the processor can transfer data to or from memory. The disadvantage of bus connection is that it slows down the program execution speed because all data transfer is done via 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 typically 16, 32 or more bits wide, and are 16 or 32 connections working together, each carrying one bit of a data word simultaneously. This parallel data connection is faster than a serial connection, such as the keyboard input or network connection, which can only carry one bit at a time. In the microcontroller, these system bus connections are hidden inside the chip, making circuit design easier.

1.1.3 PC Memory

There are two types of memory in the PC system. The main memory block is RAM, whose input data is stored before and after processing in the CPU. The operating system and application programs are also stored in RAM from disk for execution, because access to data in RAM is

![Fig. 1.4 Block diagram of PC motherboard.](image-url)
Wordprocessor Operation

1. Wordprocessor Operation

In order to understand the operation of the PC microprocessor system, we will look at how the wordprocessor application uses the hardware and software resources.

1.1 Starting the Computer

When the PC is switched on, the BIOS ROM program starts automatically. It checks that the system hardware is working properly and displays messages to report the results. If there is a problem, the BIOS program attempts to diagnose the fault, and will display an error message. If all is well, it loads the main operating system software (Windows) from hard disk onto RAM. As you will probably have noticed, this all takes some time; this is an indication of the amount of data transfer required, and the relatively slow access to the hard drive.

1.2 Starting the Application

Windows displays an initial screen with icons and menu which allows the application to be selected using the mouse and on-screen pointer. Word is started by clicking on its icon. Windows controls the selection of a command which reads the executable file (WINWORD.EXE) stored on disk. It loads into the opening system (MSDOS) (Microsoft disk operating system); required files contained are loaded to start the application.

The application program is transferred from disk to RAM, or as much of it as will fit into the available memory. If necessary, application program blocks can be swapped into memory when needed. The wordprocessor screen is displayed and a new document file can be created or an existing one loaded by the user from disk for updating.

1.3 Data Input

The main data input is obviously from the keyboard, which consists of a grid of switches which 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 the keyboard cable. The serial data is a sequence of high and low voltage, which is then interpreted by the CPU and used to control the display on the screen.
which represent a binary code. Each key generates a different code. The keyboard interface
converts this serial code to parallel form by latching 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, the keyboard interface notifies the CPU that a key is ready to be processed. This
interrupt signal is generated by the keyboard interface and sent to the CPU. The CPU then
processes the interrupt signal to determine which key was pressed and sends the appropriate
code to the CPU. This serial-to-parallel data conversion process is required in all the interfaces
that use serial data transfer, namely, the keyboard, VDU, network and modem. Binary coding,
interrupts and other such processes will be explained in more detail later.

In Windows, and other GUIs, the mouse can be used to select commands for managing the
application and its data. The mouse controls a pointer on the screen; when the mouse is moved,
the ball turns two rollers, which have perforated wheels attached. The holes are detected using
an opto-detector, which sends pulses representing movement in two directions. These pulse
sequences are passed to the CPU via the mouse interface and used to modify the position of
the pointer on the screen. The buttons, each to take an action, must also be input to the CPU.

1.2.4 Data Storage

Each character of the text being typed into the wordprocessor is stored as an 8-bit (one byte)
binary code, which occupies one location in RAM. Each bit of data must be stored as a charge
on a small capacitor in the RAM chip. The parallel data is received by the CPU, then sent back
to the same data bus from the CPU to the RAM. The RAM stores the byte data at numbered locations; these address numbers are identified by the CPU using the system address
bus. The data is then transferred to the RAM via the CPU via the
ISD, which provides the additional logic required to handle the data transfer.

1.2.5 Data Processing

In the past, programs running on the DOS operating system required less processing power,
partly because the screen was simpler, being divided up into one space for each character. The
video interface would convert the stored character code into the pattern for the character, and
output it to the correct position on the screen.

The Windows screen is more complicated because the text is displayed in graphics or
terminology mode, at a higher resolution, so that the text size, style and layout appear on the screen as it will
be printed. Graphics, tables and special characters can be embedded in the text. This means the
CPU has far more work to do in displaying the page, and this is one reason why Windows needs
more memory and a more powerful CPU than earlier DOS-based wordprocessors. The processor
must also manage the WIMP (Windows, Icons, Mouse, Pointer) interface, which allows actions
such as selecting on screen. Word 4.0 has many more features than earlier wordprocessors, and
Word 5.0 offers enhanced graphics and features such as callouts, hyperlinks, and above all,
called desktop-publishing (DTP) programs, which provide comprehensive page layout control.

1.2.6 Data Output

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 VDU via the system data bus and video interface. The
display is made up of pixels, which are small dots of light arranged in a grid to form a text
image. Each pixel is addressed by a coordinate system. The shape of the pixel is
a square, and the display is scanned to display the text. The display is a two-dimensional image
made up from a serial data stream which sets the colour of each pixel on the screen at a
fixed rate.
If a file is transferred on a network, it must also be sent in serial form. The characters (letters) in a text file would typically be 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 'A', 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. If you watch an inkjet printer working, you can see the scanning operation take place. In this process, the data is sent in the parallel form, along with control codes, to the printer port. If the printer itself is capable of formatting the final output, only the character code and any formatting codes are needed. For cheaper printers, the computer itself must generate the page layout, and send a 'bit-map' of the page, where one bit (or group of bits) is the code for one coloured dot on the page; this will take longer.

The operation of the wordprocessor can be illustrated using a flowchart, which is a graphical method of describing a sequential process. Figure 1.5 describes only 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.

![Wordprocessor Flowchart](image)

1.3 PC Microprocessor System

As we have seen, the PC working as a wordprocessor carries out the following functions:

- Data input
- Data storage
- Data processing
- Data output
All microprocessor systems perform these same basic functions. To carry them out, the microprocessor system needs a set of supporting chips, with suitable interconnections. The system will therefore typically consist of:

- CPU
- RAM
- ROM
- I/O (Input/Output) ports
- ISD
- XTAL (crystal) clock generator

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 overall charge of the system.

### 1.3.1 System Operation

The PC motherboard components are connected as shown in Fig. 1.6. The address and data busses, control lines and input ports are required to handle the data transfer between the CPU, memory and ports. The clock circuit contains a crystal oscillator as found in watches and clocks, which produces a precise fixed frequency signal which drives the microprocessor. The CPU operations are triggered on the rising and falling edges of the clock signal, allowing their relative timing to be precisely controlled. 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 this timing reference. This is why the CPU should not be operated at a frequency above its rated clock speed – correct completion of each step can no longer be guaranteed and the system could crash. 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 ISD is designed to assist the processor to handle memory and I/O operations within a particular design.

![Figure 1.6](image.png)

**Figure 1.6** Block diagram of PC microprocessor system.
For simplicity, only the keyboard port is shown in the block diagram, as this was sometimes (in older designs) the only I/O device on the main board. However, other ports, such as the printer, modem and so on are connected in the same way, whether they are part of the motherboard or fitted as expansion cards. The signal connections to the plug-in peripheral interfaces will be made to the system buses and the relevant control lines via the expansion bus, which appears on the motherboard as edge connectors. This allows the system to be upgraded by replacing or adding to these cards. In current designs, where upgrading is less likely to be required, the VDU, disk and network tend to be integrated onto the main board. Additional RAM memory may be fitted in a similar way if spare slots are available.

1.3.2 Program Execution

The ROM and RAM memory contain program information and data in numbered locations. The CPU contains address decoding logic which allocates particular memory chips to a range of addresses. 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 system designer, and accessed by the CPU in the same way as memory locations.

A register is a temporary store for a data word within a port chip or the CPU. In the port chip it can hold 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 or not is stored temporarily in the port data register. More of this later!

The wordprocessor 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 load a character code from the keyboard 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 is received.

1.3.3 Execution Cycle

Program execution is illustrated in Fig. 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 location (memory slot) containing the required instruction. This address is kept in the program counter. The sample address is shown in decimal (3724) in Fig. 1.7, but it is output in binary form on the address lines from the processor. The ISD uses the address to select the RAM chip which has been allocated to this address. The address bus also connects directly to the RAM chip to select the individual location.

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 instructions (3). The operands (data 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 operands to the dataprocessing logic (5). Additional data can be fetched from memory (6) if this would be the last data of our
Figure 1.7 Program execution sequence.

Wordprocessor. 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) by the address of the next instruction code. The address of the next instruction is then output and the sequence repeats from step 2.

Operating the wordprocessor program and the text data are stored in different parts of RAM during program execution, and the wordprocessor application program calls up operating system routines as required to read in, process and store the text data. Current CISC processors such as the Pentium series have instructions which are more than 8 bits in size which are stored in multiple locations, and use complex memory management techniques, to speed up program execution. These long instructions and data words are normally multiples of 8 bits, as this is how the memory is organised.

1.4 PC Engineering Applications

The PC can be used as a standard hardware platform in a variety of engineering systems by fitting special interfacing hardware in the expansion slots and programming the PC to control the target system through the I/O interfaces (Fig. 1.8). This type of arrangement is increasingly used in manufacturing systems where the PC might control a machine tool, robot or assembly system, or be used in instruments or data logging applications. The PC provides a
The Microcontroller

The microcontroller can provide a simplified form of all the main elements of the conventional 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 control signals in and out.

1.5.1 A Microcontroller Application

A simple equivalent of the word processing application described above could be built as shown in Fig. 1.9, around an MCU (microcontroller unit).

The basic function of the system shown is to store and display numbers which are input on the keypad. The microcontroller chip can be programmed to scan the keypad and identify any key which has been pressed. The keys are connected in a 3 x 4 grid of rows and columns, so that a row and a column are connected together when the key is pressed. The microcontroller can identify the key by selecting a row and checking the columns for a connection. Thus, four input lines and three outputs are required for connection to the microcontroller. In order to simplify the drawing, these parallel connections are represented by the block arrows.

Seven-segment displays show the input numbers as they are stored in the microcontroller. Each display digit consists of seven light emitting diodes (LEDs) which show as a line segment of the number when lit. Each number from 0 to 9 is displayed as a suitable pattern of lit segments.

Figure 1.9  Microcontroller keypad display system
The basic display program could work as follows: when a key is pressed, the digit is displayed on the right (least significant) digit, and subsequent key presses roll over the previously entered digit 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.

The starting point for writing the program for the microcontroller is to convert the general description given above into a description of the operations which can be programmed into the chip using the set of instructions which are available for that microcontroller. The instruction set is defined by the manufacturer of the device. The precise sequence of the required function is implemented as the program algorithm, which can be graphically represented by a flowchart (Fig. 1.10).

![Flowchart for keypad display program](image-url)

With suitable development of the software and/or hardware, the system could be modified to work as a calculator, message display, electronic lock or similar application. Additional digits could be added to the display as required. Keyboard scanning and display driving are standard operations for microcontrollers, and the techniques required to create the working application will be explained in later chapters.
1.5.2 Programming a Microcontroller

Some microcontrollers have ROM program memory, which is programmed before the chip is fitted into the application circuit, and cannot be changed. One-time programmable (OTP) devices are generally used for larger production volumes where the program is known to be correct.

We will be using PIC chips which have flash program memory, which can be erased and re-programmed many times, which is invaluable when learning. A PIC device is programmed by placing it in a special programming unit which is attached to a host computer (Fig. 1.11). Note the zero insertion force (ZIF) socket which will accept different-sized chips for programming.

The program is written and converted to machine code in the host computer using suitable development system software and downloaded via a serial data link to the chip in the programmer unit. The microcontroller is then placed in the application circuit. The circuit can then be checked for correct operation.

Having introduced some basic ideas concerning microprocessors and microcontrollers, in the next chapter we will review some principles of digital circuits and microprocessor systems. The process of creating microcontroller applications such as the example outlined above can then be tackled.

---

**Figure 1.11** Programming a PIC microcontroller. (a) PIC program downloading; (b) PIC programming unit.
The PC consists of data input, storage, processing and output devices.

The main unit is a modular system, consisting of the motherboard, power supply, disk drives and expansion cards containing interfacing circuits plugged into the motherboard.

The motherboard carries the microprocessor (CPU) chip, RAM memory modules, a BIOS ROM, IDE and keyboard interface.

The CPU communicates with the main system chips via a shared set of address and data bus lines. The address lines select the device and location for the data to be transferred on the data bus.

The microcontroller provides, in simplified form, most of the features of a conventional microprocessor system on one chip.

Questions

1. Name at least two PC input devices, two output devices and two storage devices.
2. Why is the BIOS ROM needed when starting the PC?
3. Why are shared bus connections used in the typical microprocessor system, even though it slows down the program execution?
4. State two advantages of the modular PC hardware design.
5. Why does the PC take so long to start up?
6. Sort these data paths into serial and parallel:
   (a) internal data bus
   (b) keyboard input
   (c) printer output
   (d) modem I/O
   (e) program counter
   (f) instruction register
   (g) data bus

7. State the function, in ten words or less, of the:
   (a) CPU
   (b) ROM
   (c) RAM
   (d) LSI
   (e) program counter
   (f) instruction register

8. Explain the difference between a typical microprocessor and microcontroller.
1. Study the messages which appear on the screen when PC is switched on, and explain their significance with reference to the system operation.

2. Under supervision if necessary, and with reference to relevant manuals, carry out the following investigation. Disconnect the power supply and remove the cover of the main unit of a PC and identify the main hardware subsystems – power supply, motherboard and disk units. On the motherboard, identify the CPU, RAM modules, expansion slots, keyboard interface, VDU interface, disk interface, printer interface. Is there an internal modem or network card? Are there any other interfaces fitted?

3. Run the wordprocessor 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.
Chapter 2
Information Coding

2.1 Number Systems

Mathematics is based on number systems, which use a set of characters to represent numerical values. The characters used are simply symbols, just squiggles on a page, but the number systems they are part of have developed over thousands of years—because they are so useful.

In microprocessors, microcontrollers, and digital electronic systems, numerical processing is carried out using binary codes, a number system which has only one common element with the development of digital computers. We therefore have to understand binary numbering in order to use a microcontroller. Another number system, hexadecimal, is also useful here because it provides a more compact way of representing binary codes.

Much of modern technology is based on the use of mathematical models to represent information and processes in the real world. These mathematical models are used in engineering to help design new systems and products. For instance, the three-dimensional drawing of a suspension arm for a car created on a CAD (computer-aided design) system screen is generated from a digital representation of the shape of the part in the memory of the computer. The advantages of the computer model are fairly obvious—it can be visualized on disk, modified electronically and modeled much more easily than the equivalent information on paper. The component design can also be mathematically analyzed in the computer prior to construction. For example, the stresses and strains to which the component will be subjected in its final position in the suspension assembly can be studied. Further, when a component design is finalized, the design data can be converted directly to a program for a machine tool which will automatically manufacture the part.

The programs for our microcontroller applications will be stored in the same way; therefore, we need to know something about how such data is represented in the computer.
2.1.1 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, ten symbols are used, with which, hopefully, you are familiar:

```
0 1 2 3 4 5 6 7 8 9
```

Why use a particular base number? The reason for using ten is simple – we humans have ten fingers which can be used for counting, so the decimal system was developed as a way of writing that down and doing calculations on paper (or stone) instead of on our fingers. We see the term 'digit' to refer to fingers and numbers and 'digital' to describe binary electrical circuits. The use of written numbers was essential for the original development of industry and commerce.

Assuming that we know how to count and write down numbers in decimal, let's analyse what a typical number means. Take the number 274; in words, it is 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 which 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 detailed in Table 2.1.

A number system can be used for any base you like, but some are more useful than others. For instance, values of the base 12 system are still in use – think of clocks, boxes of eggs and measurement of angles. Base 12 is useful because 12 is divisible by 2, 3, 4 and 6, giving lots of useful fractions – a half, a third, a quarter and one-sixth. However, the decimal system is our standard system, so the analysis of other systems will still be based on decimal for comparison of number values.

2.1.2 Binary: Base 2
Binary is used in digital computer systems because it represents the way that values are stored and processed. The binary digits, 0 and 1, represent two voltage levels used in digital circuits. The number of symbols is two, so the binary system is convenient – it uses just two digits. The basic rules are the same for any number system.

In binary, the base is 2, so the columns weighting is a power of 2, as shown in Table 2.2 (note that any number in the power zero has the value 1). With a base of 2, only the digits 0 and 1 are used:

```
<table>
<thead>
<tr>
<th>Digit</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
</table>
```

<table>
<thead>
<tr>
<th>Column weight</th>
<th>1024</th>
<th>512</th>
<th>256</th>
<th>128</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power of Base</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total value in (1 0 0 0 0 0 0 0 0 0 0 0 0 0)</th>
<th>128 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0</th>
</tr>
</thead>
</table>

Table 2.1: Structure of a decimal number

Table 2.2: Structure of a binary number

Note: The number system base is 10 for decimal and 2 for binary.
Table 2.2: Structure of a binary number

<table>
<thead>
<tr>
<th>Column weight</th>
<th>2^0</th>
<th>2^1</th>
<th>2^2</th>
<th>2^3</th>
<th>2^4</th>
<th>2^5</th>
<th>2^6</th>
<th>2^7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal weight</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
</tr>
</tbody>
</table>

Example number: 10100011

Decimal equivalent: 128 + 0 + 32 + 0 + 0 + 0 + 2 + 1 = 163

0 and 1 are available, so the numbers tend to have lots of digits. For instance, a 32-bit computer uses 32-digit binary numbers. An example with 8 digits is given showing what the digits represent and how to convert the value back to decimal.

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 values are 1 or 0, the result can be obtained by simply adding the digit weights where the digit value is 1. Because any number multiplied by zero is zero. When decimal data is entered into a computer, the values are converted to binary. The program instructions which process input and output data are also stored as binary codes.

### 2.1.3 Hexadecimal: Base 16

Binary numbers have lots of digits, so they are not very easy to understand when written down or printed. 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 10-15. Thus, hexadecimal numbers are made up of the symbols A, B, C, D, E, and F, because symbols are already available. A binary number can then be easily converted to hex by writing a group of 4 bits, and then converting each group to its equivalent hex digit, as in Table 2.3.

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 (zero, zero, zero) could have the decimal value 4 in binary, 100 (one hundred) in decimal or 256 in hexadecimal.

Some examples of equivalent values are given in Table 2.4. The numbers are printed in ‘Courier’ type, as used on old-fashioned typewriters, because each character occupies the same space, or all the digit line up neatly in columns.

### 2.1.4 Counting

A list of equivalent numbers, counting from zero, is given in Table 2.5, 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 1K = 2^10. This is worth remembering as a starting point in calculating memory capacity.
### Table 2.3 Hexadecimal digits

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>

### Table 2.4 Examples of equivalent values

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>10000</td>
<td>10</td>
</tr>
<tr>
<td>31</td>
<td>11111</td>
<td>1F</td>
</tr>
<tr>
<td>100</td>
<td>1100100</td>
<td>64</td>
</tr>
<tr>
<td>169</td>
<td>10101001</td>
<td>A9</td>
</tr>
<tr>
<td>255</td>
<td>11111111</td>
<td>FF</td>
</tr>
<tr>
<td>1024</td>
<td>100000000000</td>
<td>400</td>
</tr>
</tbody>
</table>

### Table 2.5 Significant equivalent numbers

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>

continued...
<table>
<thead>
<tr>
<th>Decimal (Base 10)</th>
<th>Binary (Base 2)</th>
<th>Hex (Base 16)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>F</td>
<td>Maximum 4-bit count</td>
</tr>
<tr>
<td>16</td>
<td>10000</td>
<td>10</td>
<td>Use 2nd column in hex</td>
</tr>
<tr>
<td>17</td>
<td>10001</td>
<td>11</td>
<td>Use space to clarify binary</td>
</tr>
<tr>
<td>18</td>
<td>10010</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>10011</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>10100</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>10110</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>10111</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>11000</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>11001</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>11010</td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>11011</td>
<td>1B</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>11100</td>
<td>1C</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>11101</td>
<td>1D</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>11110</td>
<td>1E</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>11111</td>
<td>1F</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>100000</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>100001</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>100010</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>62</td>
<td>111110</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>111111</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1000000</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>1000001</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>127</td>
<td>1111111</td>
<td>7F</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>10000000</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>129</td>
<td>10000001</td>
<td>81</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2047</td>
<td>11111111111</td>
<td>7FF</td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td>10000000000</td>
<td>800</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>65535</td>
<td>11111111111111</td>
<td>FFFF</td>
<td></td>
</tr>
<tr>
<td>65536</td>
<td>10000000000000</td>
<td>8000</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Maximum 16-bit count
The rules for counting in any number system are given below.

1. Start with all digits set to zero.
2. In the right digit position (LSB), 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.

In microprocessors, there is a fixed number of digits in the registers which store binary numbers (8, 16, 32 bits or more). If the number storage space has a fixed number of digits, 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.

2.2 Machine Code Programs

Microcontrollers store their program code and data in binary form, typically using voltage levels of +5V and 0V to represent binary 1 and 0. The program is initially stored in non-volatile ROM, and is executed by passing each code in turn to a decoding circuit which sets up the processor to carry out that particular instruction. The processor then operates on input or stored data, and produces the output as required.

2.2.1 Data Words

Conventional microprocessors handle the code in 8-bit binary words, or multiples of 8 bits. The data word size has increased with the complexity of the integrated circuits available; some examples are given in Table 2.6.

The first generation of popular UK home computers, such as the Commodore, Apple, and Spectrum used 8-bit microprocessors, that is, the program and data words were all 8-bit numbers. Second generation home games machines such as the Atari and Amiga used 16-bit 68000 chips, which was also the processor used in the Apple Mac, the first mass-produced computer to use a WIMP interface.

The original IBM PC was a business-oriented personal computer using the Intel 8088, which handled 16 bits inside the CPU, but only 8 bits externally. The latest processor is now a 32-bit Pentium processor, and so on to the current generation. At the same time clock speeds increased, leading to the 33 MHz Pentium processor, and so on to the current generation. As the clock rate increases, so the processor complexity developed, so that the data processing capability of the current Pentium PC is massive compared with the original 8-bit machines.
Table 2.6: Comparison of microprocessors and microcontrollers

| Microprocessor/microcontroller | Computer/ 
application | Address bus (bits) | Data bus (bits) | Instruction (bits) | Internal CPU data (bits) |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Zilog Z80</td>
<td>Spectrum</td>
<td>16</td>
<td>8</td>
<td>8/16/24</td>
<td>8</td>
</tr>
<tr>
<td>National 8005</td>
<td>Commodore/BBC</td>
<td>16</td>
<td>8</td>
<td>8/16/24</td>
<td>8</td>
</tr>
<tr>
<td>Motorola 68000</td>
<td>Atari/Amiga/Mac</td>
<td>24</td>
<td>16</td>
<td>16/32/48</td>
<td>16</td>
</tr>
<tr>
<td>Intel 8086/8</td>
<td>PC XT</td>
<td>+4</td>
<td>8/16</td>
<td>16/32/48</td>
<td>16</td>
</tr>
<tr>
<td>Intel Pentium</td>
<td>Pentium PC</td>
<td>32</td>
<td>32</td>
<td>16/32/64</td>
<td>32</td>
</tr>
<tr>
<td>Intel 8051</td>
<td>Industrial/Control</td>
<td>Internal 16</td>
<td>Internal 8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>PIC 16F84</td>
<td>Industrial/Control</td>
<td>Internal 13</td>
<td>Internal 8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

The 8051 was one of the first widely used microcontrollers and is still established in the industrial control market. The PIC family is a more recent challenger for the position of leading microcontroller type. Its manufacturer, Microchip, has succeeded by initially specialising in small, cheap, re-programmable devices which were good for beginners, and then expanding the range, providing free development tools along the way.

2.2.2 Machine Code

Microprocessor machine code is a list of binary codes which are often shown in hexadecimal.

An example of 6502 code is listed in Table 2.7.

The program code is a list of 8-bit binary numbers, stored in numbered memory locations, here starting at 0200₁₆, forming a list of instructions for the microprocessor to execute. The function of the particular program is to load a number given in the program /lparen/ori 55₁₆/rparen/ori into the main data register (called A), and then store it in a memory location /lparen/ori 0300₁₆/rparen/ori. The program shows two instructions, each of which starts with the instruction (operation) code /lparen/ori A9₁₆/comma/ori 8D₁₆/rparen/ori, which are followed by data required by the instruction (a number to load, and a memory address to store it). These are called the operands. Note that in 6502 programs, the complete instruction may consist of 1, 2 or 3 bytes.

Table 2.7: 6502 machine code

<table>
<thead>
<tr>
<th>Memory address</th>
<th>Hex code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0200</td>
<td>A9 8D 03 0</td>
<td>Load the main data register A with the number 55</td>
</tr>
<tr>
<td>0201</td>
<td>55</td>
<td>Store the contents of register A, whose address is 0300</td>
</tr>
<tr>
<td>0202</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0203</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0204</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>0205</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0206</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.2.3 Assembly Language

The 6502 machine code is translated into assembly language using a utility called an assembler.

Assembly language consists of mnemonics which are similar to the machine code but allow the programmer to write instructions in a more human readable form. An example of assembly language code is shown in Table 2.8.

Table 2.8: 6502 assembly code

<table>
<thead>
<tr>
<th>Memory address</th>
<th>Hex code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0202</td>
<td>55</td>
<td>Store the contents of register A, whose address is 0300</td>
</tr>
<tr>
<td>0203</td>
<td>00</td>
<td>Store the contents of register A, whose address is 0300</td>
</tr>
<tr>
<td>0204</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>0205</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0206</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0207</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0208</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0209</td>
<td></td>
<td></td>
</tr>
<tr>
<td>020A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>020B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>020C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>020D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>020E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>020F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The assembler program translates the assembly code into machine code, and then the machine code is loaded into the microprocessor.
2.2.3 8086 Machine Code

The Intel 8086 was the CPU developed for the original IBM PC. It is useful to know something about 8086 machine code (see Table 2.8) because this is the native language of the PC, and it can be studied without access to any other hardware system. As with other processor families, the same basic instructions set has been expanded for later processors, but the basic syntax is the same, so 8086 code should run on a Pentium processor. Backward code compatibility has always been a major feature of the Intel/Microsoft product line.

Table 2.8  PC machine code

<table>
<thead>
<tr>
<th>Address</th>
<th>Segment: Offset</th>
<th>hex 1 2 3 4 5 6 7 8 9 A B C D E F</th>
</tr>
</thead>
<tbody>
<tr>
<td>01234567</td>
<td>89ABCDEF</td>
<td>1B85:0100 OF 00 B9 8A FF F3 AE 47-61 03 1F 8B C3 48 12 B1</td>
</tr>
<tr>
<td>01234567</td>
<td>89ABCDEF</td>
<td>1B85:0110 04 8B C6 F7 0A 0A D0 D3-48 DA 2B D0 34 00 74 1B</td>
</tr>
<tr>
<td>01234567</td>
<td>89ABCDEF</td>
<td>1B85:0120 00 DB D2 D3 E0 03 F0 8E-DA 8B C7 16 C2 B6 01 16</td>
</tr>
</tbody>
</table>

8086 code can be viewed on a PC by selecting the ‘MS-DOS prompt’ from the Windows Start button menu. An MS-DOS window should open with a ‘>’ symbol and flashing cursor. Text commands can then be entered (before Windows, all operating system actions had to be entered this way). Type ‘debug’, which is a text command to the operating system. A ‘-’ prompt appears, to indicate that Debug commands will be accepted. If ‘d’ (dump) is entered, the contents of a block of PC program memory will be dumped to the screen as 2-digit hex codes.

The addressing system in the Intel processor was more complicated than that in most other processors, with the address derived from the combination of a 16-bit segment address and an 8-bit offset. This system was originally devised when the 16-bit 8086 was introduced, to maintain compatibility with older 8-bit systems. The memory was at this stage divided into 10 × 64k segments (64k is the maximum memory space addressable with a 16-bit address). Thus each address is shown in the form ‘SSSS:OOOO’, where SSSS is the four-digit segment address and OOOO is the 4-bit offset. For example, if SSSS = 1B8516 and OOOO = 010016, then the actual address will be 1B850 + 0100 = 1B95016. In theory, this system could address up to 4 gigabytes of memory.

If the Debug command ‘s’ is entered, the source assembly language ‘mnemonics’ are displayed for the current memory range. It can be seen that each instruction can contain 1, 2, 3 or 4 bytes. Assembly language is the programming method used for writing machine code programs, because the assembler mnemonics are easier to remember than the corresponding binary codes. An assembler utility is required to convert the source code mnemonics into executable machine code. This idea will be explained in more detail later, using the PIC instruction set as an example.

2.2.4 PIC Machine Code

The PIC machine code program is easier to interpret than the 8086 code, because it has instructions which are 14-bits long (in the 16F84). In hexadecimal, a 14-bit instruction must be represented with four digits, with the most significant two hex-mnemonic. The default program start address (which is used if the programmer does not specify another), is 0000 (zero). A simple PIC machine code program is shown in Table 2.9.

The machine code for the first PIC program, BIN, that we will be studying later is listed. It consists of five instructions, stored at addresses 0000–0004 in the program memory.
### Table 2.9: Simple PIC machine code program BIN1

<table>
<thead>
<tr>
<th>Program memory address</th>
<th>Program machine code</th>
<th>Meaning of Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>3000</td>
<td>Move the number 00 into the working register</td>
</tr>
<tr>
<td>0001</td>
<td>0066</td>
<td>Copy this code into port B data direction register</td>
</tr>
<tr>
<td>0002</td>
<td>0186</td>
<td>Clear port B data to register to zero</td>
</tr>
<tr>
<td>0003</td>
<td>0A86</td>
<td>Increase the value in port B data register by one</td>
</tr>
<tr>
<td>0004</td>
<td>2803</td>
<td>Jump back to Address 0003</td>
</tr>
</tbody>
</table>

The meaning of each instruction is given, but a fuller explanation will have to wait for now. Each instruction is 14-bits long, but the actual operation code and operand length varies within the fixed total, as shown in Table 2.10.

### Table 2.10: PIC machine code in binary form

<table>
<thead>
<tr>
<th>Memory address</th>
<th>Binary machine code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>11 00 0000 0000</td>
<td>Load W with 0000 (0)</td>
</tr>
<tr>
<td>001</td>
<td>00 0000 0101 0</td>
<td>Copy W to direction register of port 110 (6)</td>
</tr>
<tr>
<td>002</td>
<td>00 0001 1000 0110</td>
<td>Clear data register of port 110 (6)</td>
</tr>
<tr>
<td>003</td>
<td>00 1100 1000 0110</td>
<td>Increment register of port 110 (6)</td>
</tr>
<tr>
<td>004</td>
<td>10 0000 0000 0011</td>
<td>Jump back to address 0000000011 (3)</td>
</tr>
</tbody>
</table>

The operation code part of the 14-bit instruction is shown in bold, while the operand is shown in italics. The operands refer to numbered registers and addresses within the PIC chip. For example, the last instruction operand contains the address of the third instruction, because the program jumps back and repeats from this point.

The PIC machine code can be seen in the programmer software (MPLAB) window prior to downloading, or printed in the source program list file. When the PIC chip is placed in the programmer unit, the binary codes for the program can be sent to its program memory, in serial form, one bit at a time. Each 14-bit code is stored at the address location (0000–0004) specified. When the program is later executed, the codes are interpreted by the processor block in the chip and the action carried out. The meaning and use of the registers will be explained later, when this program will be analysed in more detail.

### 2.3 ASCII Code

ASCII (American standard code for information interchange) is a type of binary code for representing alphanumeric characters, as found on your computer keyboard. The basic code consists of seven bits. For example, capital (or 'upper case') 'A' is represented by binary code 100 0001 (65), 'B' by 66, and so on to 'Z' = 65 + 25 = 90 = 101 1010. Lower case letters and other common keyboard characters such as punctuation, brackets and alphabetic signs, plus some special control characters also have a code in the range 91–127. The numerical characters...
also have a code, for example '9' = 011 1001₂, so you sometimes need to make it clear if the code is the binary equivalent (011 1001₂) or the ASCII code (9₁₀).

We will not be using ASCII codes in great detail in this book, but we need to know of them, as they are the standard coding method for text. When a program is typed in, the computer will convert it into corresponding binary machine-code instructions. If this is confusing, come back to this point when we have looked at programming in more detail!

Summary

- Programs and data in a microprocessor system are stored in binary form, typically as '0' = 0 V and '1' = 5 V.
- The binary codes can be displayed and printed in hexadecimal form, where 1 hex digit = 4 binary bits.
- A microprocessor program consists of a sequence of binary codes representing instructions and data which are decoded and executed by the CPU.
- The microprocessor memory contains a set of locations, numbered from zero, where the program is stored.
- Each program instruction consists of an operation code and (often) an operand.
- Each complete instruction may occupy a fixed number of bits or a variable number of bytes.

Questions

1. Refer to Table 2.5.
   (a) Predict the binary equivalent of 35₁₀, 6₁₀, and 1025₁₀.
   (b) Convert the numbers in (a) from binary to hexadecimal.
   (c) Work out the 8-bit binary code for the 6502 program code in Table 2.7.
   (d) Write down the 16-bit binary code for the hex address 0203₁₆.

2. Write down the hex code, and work out the decimal equivalent number for the binary numbers:
   (a) 101₁₀
   (b) 1101₁₀
   (c) 1011011₁₀
   (d) 0110110011111100₂
   (e) 01101101 1111 1000₁₀

3. Light emitting diodes are often used to display output in simple test systems, where a binary '1' lights the LED. For an 8 bit output, work out the binary and hex codes required:
   (a) To light the LEDs.
   (b) To switch them all off.
1. (a) \( 55_{16} = 101011_2 \)
(b) \( 0203_{16} = 00000010 00000011_2 \)
(c) \( A9_{16} = 10101001_2 \)
(d) \( 03_{16} = 00000011_2 \)

2. (a) \( 101_2 = 5_{10} \)
(b) \( 1100_2 = C_{16} \)
(c) \( 1001\ 1110_2 = 9E_{16} = 158_{10} \)
(d) \( 0011\ 1010\ 1111\ 0000_2 = 3AF0_{16} = 15088_{10} \)

3. (a) \( 1111\ 1111_2 = FF_{16} \)
(b) \( 0000\ 0000_2 = 00_{16} \)
(c) \( 0101\ 0101_2 = 55_{16} \)
(d) \( 00,\ 01,\ 03,\ 07,\ 0F,\ 1F,\ 3F,\ 7F,\ FF_{16} \)
(e) \( 00,\ 01,\ 02,\ 04,\ 08,\ 10,\ 20,\ 40,\ 80_{16} \)

Activities
1. The seven-segment display is a device which we will use later as an output device for the PIC chip. Digits are displayed by illuminating selected segments. A diagram showing the connections to the LED segments is given in Fig. 2.1(a). The segments are identified by letter: ‘a’ for the top segment, ‘b’ is the next clockwise round the outside, and so on up to ‘f’ for the top left segment, with the middle segment called ‘g’.

These are connected as shown to a port data register bits 1–7, with the LSB not connected. Work out the binary and hex codes required to obtain the displayed characters 0–F shown in Fig. 2.1(b), if the display operates ‘active high’, that is, a ‘1’ in the register switches the corresponding segment on. Assume that bit 0 = ‘0’.

2. Debug is a DOS utility which allows you to operate at machine code level in the PC system. At the DOS prompt on a PC, enter the command ‘debug’; a prompt ‘>’ is obtained.
(a) Enter ‘?’ and the debug commands are displayed.
(b) Enter ‘A’ (_always) and the contents of the current memory range are displayed in hex, bytes (2 digits), with the ASCII character equivalent at the right. The 4-digit codes
Figure 2.1 Seven-segment display of two digits.

on the left are the segment address and offset, separated by a colon. The addresses
are displayed at intervals of 16 (10H) locations, since each row shows 32 bytes.
(c) Enter 'A' (assemble), and the assembly code is displayed, one instruction per line.
Note the presence of instructions such as MOV, ADD, INC, and so on. Note also the variable instruction length.
(d) Enter 'V' to view the processor registers. Note that at least one of the
segment registers (CS, DS, SS, ES) contains the segment address, and the instruction
pointer (IP) contains the offset.
(e) Enter 'Y' to trace the machine code execution. The code is executed one instruction
at a time so that you can track the changes in the registers. Note that the system
continues to run normally in background during debugging, so that the screen remains
visible and further editing operations can be edited.
(f) Enter 'Q' to quit.
We have seen in Chapter 2 that a microcontroller program consists of a list of binary codes, stored in non-volatile memory. The instructions are executed in sequence, processing data obtained from the chip registers or inputs. The results are stored back in the registers, in RAM locations, or sent to an output device. We will now look briefly at the basic circuit elements needed to provide these functions. This section aims to explain the operation of basic elements of logic devices, which include microcontrollers, to allow the reader to understand PIC data sheets.

3.1 Digital Devices

The binary codes which make up the program and data in the microcontroller are stored and processed as electronic signals. The binary numbers are conventionally represented as follows:

- Binary 0 = 0 V
- Binary 1 = 5 V

A +5V supply, usually derived from the mains, is therefore required to power the circuits. It must be able to provide sufficient current for the processor circuits, at a voltage which must be between 4.75V and 5.25V for standard TTL (Transistor–Transistor Logic).

The power consumed in operating a digital circuit simply appears as waste heat, which must be dissipated from the chip. This is why a large complex device such as the Pentium processor typically has a heatsink and fan attached. The power consumption is the product of the supply voltage and current drawn at the power supply pins of the chip:

\[ P = V \times I \text{ Watts} \]

where

- \( P \) = chip power
- \( V \) = chip supply voltage
- \( I \) = chip current
- \( R \) = input resistance of chip

We have seen in Chapter 2 that a microcontroller program consists of a list of binary codes, stored in non-volatile memory. The instructions are executed in sequence, processing data obtained from the chip registers or inputs. The results are stored back in the registers, in RAM locations, or sent to an output device. We will now look briefly at the basic circuit elements needed to provide these functions. This section aims to explain the operation of basic elements of logic devices, which include microcontrollers, to allow the reader to understand PIC data sheets.
If the same logic function can be implemented with less power consumed, this problem of power dissipation can be reduced and system efficiency increased. There are two ways to do this: to use a low power transistor type in the circuits, or to reduce the supply voltage on both. A supply of 3.3 V is commonly used to reduce power consumption in large chips. The heating effect is proportional to the voltage squared (see above), so the voltage reduction from 5 V to 3.3 V will reduce heating and power consumption by 66%.

In the original small-scale chips, bipolar transistors were used to form TTL gates. However, these have relatively large power dissipation, and run at correspondingly high temperatures. This limits the number of gates that can be contained on one chip, so VLSI (very large-scale integrated) circuits normally use FET-based (field effect transistor) logic gates, because of their lower power consumption. Also, these chips can use a wider range of supply voltages, so are more suitable for battery-powered applications, such as laptop computers. There is continuing development of logic technologies, to obtain higher speed, lower cost and lower power consumption.

The PIC chip is a CMOS (complementary metal-oxide semiconductor) device, using FETs as digital switches. There is a range of complementary bipolar devices, each using digital gates that can process binary data. For example, we will see how logic gates can be combined to create a binary adder, which is an essential feature of any microprocessor, allowing it to carry out binary arithmetic.

### 3.1.1 FET Logic Gates

The FET is the basic switching device which appears in the PIC data sheet in the equivalent circuits for various functional blocks. It is a transistor which works as a current switch; current flow through the channel is controlled by the voltage at the input 'gate'.

A single FET is shown in Fig. 3.1(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, and the device is off. Some FETs operate with a negative voltage at the input to control the current flow.

A logical 'invert' operation is implemented by the FET circuit in Fig. 3.1(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 volt drop across it. This means that the output, F, must be near zero volts when the input is high. 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) when the input is low (logic 0).

The logic operation 'AND' requires the output of a gate to be HIGH only when all inputs are HIGH (see Table 3.1). 'NAND', the inverse operation, requires that the output is LOW only when all inputs are HIGH. This operation can be implemented as shown in Fig. 3.1(c). The output F is low when both transistors are on. The AND function can then be obtained by inverting the NAND output.

Similarly, the logic operation 'OR' requires the output of a gate to be HIGH when either input is HIGH (see Table 3.1). 'NOR', the inverse operation, requires that the output is LOW when either input is HIGH. This operation can be implemented as shown in Fig. 3.1(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.
Microelectronic Devices

Table 3.1 Logic table for one and two input gates

<table>
<thead>
<tr>
<th>Inputs</th>
<th>NOT</th>
<th>AND</th>
<th>OR</th>
<th>NAND</th>
<th>NOR</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

3.1.2 Logic Circuits

In real logic gates, the circuits are a little more complex. There are no actual resistors used because they would be too much power wasted, instead, other FETs are used as "active loads", which reduces the power which would be dissipated as heat in the resistors. The logic operations in Table 3.1 are all we need to make any logic or processor circuit.
Digital circuits are based on various combinations of logic gates fabricated on a silicon wafer. They can be supplied as discrete gates on small-scale ICs (SSI), or as complete logic circuits on large-scale ICs (LSI). Microprocessors are the most complex of all, containing thousands of gates and millions of transistors, and are called very large-scale ICs (VLSI).

### 3.1.3 Logic Gates

Whichever technology is used to fabricate the gates, the logical operations are the same. The symbols for logic gates used in most data sheets, including the PIC, conform to US standards, because that is where the chips are often designed. The basic set of logic devices are the AND gate, OR gate and NOT gate (or logic inverter), shown in Fig. 3.2.

There are three additional gates which 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 (see Table 3.1). The inputs on the left accept logic (binary) inputs, producing a resulting output on the right. These logic values are typically represented by +5 V and 0 V, as we have seen. These gates, in various combinations, are used to make the control and data processing circuits in a microprocessor, microcontroller and supporting chips. Their functions have been summarised in the logic table, Table 3.1.

<table>
<thead>
<tr>
<th>Logic Gate</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND Gate</td>
<td><img src="image1" alt="AND Gate" /></td>
</tr>
<tr>
<td>OR Gate</td>
<td><img src="image2" alt="OR Gate" /></td>
</tr>
<tr>
<td>NOT Gate</td>
<td><img src="image3" alt="NOT Gate" /></td>
</tr>
<tr>
<td>NAND Gate</td>
<td><img src="image4" alt="NAND Gate" /></td>
</tr>
<tr>
<td>NOR Gate</td>
<td><img src="image5" alt="NOR Gate" /></td>
</tr>
<tr>
<td>XOR Gate</td>
<td><img src="image6" alt="XOR Gate" /></td>
</tr>
</tbody>
</table>

Figure 3.2 Logic gate symbols (US standard).
Microelectronic Devices

The operation of logic circuits is shown in this way in IC data sheets, sometimes with 0 represented by L (low) and 1 by H (high). Note that only two inputs to each gate are shown here, but there can be more than two. The logical operation will be similar, for instance, a 2-input AND gate requires all inputs to be high to give a high output.

Variations 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. It should always be possible to work out the logical operation from the basic logic symbol set. More detailed analysis and design of discrete logic circuits is provided in standard textbooks, and does not need to be covered here. Such discrete design principles are, in any case, less important now for the circuit designer due to the availability of microcontrollers such as the PIC which provide a software-based alternative to hard-wired logic.

3.2 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 not by the inputs at some previous point in time. Circuits for binary addition will be used as 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 Fig. 3.3, to illustrate the process required.

The process of binary addition (Fig. 3.3) is carried out in a similar way to decimal addition. The digits in the least significant column are added first, and the result 1 or 0 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 written down as the most significant bit of the result. The result can therefore have an extra digit, as in our example.

\[
\begin{array}{c}
\text{Carry} \\
\text{Sum}
\end{array}
\]

1 1 1 1 (A) + 0 1 1 0 (B) = 0 1 0 1 (Sum)

Carry: 1
1 1 1 1

Figure 3.3 Example of binary addition.

Having specified the process required, we can now design a logic circuit to implement this process. We will use a binary adder circuit for each column, feeding the carry bits forward as required.

3.2.1 Simple Binary Adder

The basic operation can be implemented using logic gates as shown in Fig. 3.4. The two binary bits are applied at A and B, giving the result at F. Obviously, some additional mechanisms is needed to store and present this data in the inputs, and this will be described later. This circuit is equivalent to a single XOR gate, which can therefore be used as our basic binary adder.

### Figure 3.4

![Simple Binary Adder](image-url)
3.2.2 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 shown in Fig. 3.4(a).

The required function of the circuit can be specified with a logic table, as shown in Fig. 3.4(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 two full adders cascaded together. The overall circuit and logic table are shown in Fig. 3.5(a) and (b), respectively.

---

**Figure 3.4** Binary adder logic circuit.

**Figure 3.5** Full adder circuit and logic table. (a) Full adder logic circuit; (b) Full adder logic table.
carry in must be applied to the C\textsubscript{i} of stage 1 and the carry out will then be obtained from C\textsubscript{o} of stage 4.

### 3.2.3 4-Bit Adder

A set of 4 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 16F84A ALU, for example, processes 8-bit data. As we are not particularly concerned with exactly how the logic is designed, we can think it simple terms of input and output columns.

All possible input combinations must be correctly processed, and these can be specified by using a binary count in the input columns. The state of the output for each possible input

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A4 A3 A2 A1 B4 B3 B2 B1 C\textsubscript{i}</td>
<td>C\textsubscript{o} S4 S3 S2 S1 Dec</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>1 0 0 0 0 0 0 0 0 1</td>
<td>0 0 0 0 1 0 0 0 1 1</td>
</tr>
<tr>
<td>2 0 0 0 0 0 0 0 1 0</td>
<td>0 0 0 1 0 0 0 1 1</td>
</tr>
<tr>
<td>3 0 0 0 0 0 0 0 1 1</td>
<td>0 0 1 0 0 1 0 2</td>
</tr>
<tr>
<td>4 0 0 0 0 0 0 1 0 0</td>
<td>0 0 1 0 0 1 0 2</td>
</tr>
<tr>
<td>5 0 0 0 0 0 0 1 0 1</td>
<td>0 0 1 0 1 0 1 3</td>
</tr>
<tr>
<td>6 0 0 0 0 0 0 1 1 0</td>
<td>0 0 1 0 1 0 1 3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>509</td>
<td>1 1 1 1 1 1 1 0 1 1 1 1 0 1 3</td>
</tr>
<tr>
<td>510</td>
<td>1 1 1 1 1 1 1 1 0 1 1 1 1 0 3</td>
</tr>
<tr>
<td>511</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1 3</td>
</tr>
</tbody>
</table>

**Figure 3.6** 4-bit binary full adder. (a) 4-bit adder block; (b) Logic table for 4-bit adder.
A combination is then defined. 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, as examples.

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 chip, in much the same way that the PIC itself can be programmed.

### 3.3 Sequential Logic

Sequential logic refers to digital circuits whose outputs are determined by the current inputs AND the inputs which 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.

#### 3.3.1 Basic Latch

Sequential circuits are made from the same set of logic gates shown in Fig. 3.2. They are all based on a simple latching circuit made with two gates, where the output of one gate is connected with an input of the other, as shown in Fig. 3.7a. This 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 high:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

When both inputs are high, the output of that gate is forced low, and the other high:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

This circuit is called a 'latch', because it 'catches' the data. Now, the logic table for basic latch:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

When both inputs are high, the output of that gate is forced low, and the other high:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

When both inputs are low, the output of that gate is forced high, and the other low:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 3.7 Basic latch operation. (a) Basic latch circuit; (b) Sequential logic table for basic latch.
3.3.2 Data Latch

A basic sequential circuit block is a data latch, which is shown in Fig. 3.8(a). The input and output sequence can be represented on a truth table, Fig. 3.8(b). When the enable (EN) input is high, the output (Q) follows the state of the input (D). When the enable is taken low, the output state is hold. The output does not change until the enable is taken high again. It is called a transparent latch, because the data goes straight through when the enable is high. There are other types of latches, called edge-triggered, which latch the data input at a specific point in time.

### Figure 3.8

Data latch operation. (a) Data latch; (b) Sequential logic table for data latch.

<table>
<thead>
<tr>
<th>Time</th>
<th>Data input</th>
<th>Enable input</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Output 0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Data 1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>Data 1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>Output 0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>Data 1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>Output 0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>Data 1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>Data 1</td>
</tr>
</tbody>
</table>
time, when the enable (or ‘clock’) signal changes. This type of circuit block is used in registers and data RAM to store groups of typically 8 bits.

A timing diagram (Fig. 3.9) gives a pictorial view of the latch operating sequence, which can be easier to interpret than the logic table. It can also provide information about the precise timing of the signals, if required. This may be important, because there is always a delay between changes at the input and output of a gate. When designing high-speed circuits in particular, these timing characteristics must be carefully considered. However, for simplicity, time delays between the signal edges are not shown in Fig. 3.9.

Figure 3.9  Data latch timing diagram.

These signals in the actual circuit can be displayed using an oscilloscope, if small timing delays are significant. If not, a logic analyser may be used, which operates like a multichannel digital oscilloscope allowing many signals to be displayed simultaneously, but it may not reveal small time delays. The logic analyser works by sampling the signals at intervals, and can display the data in numerical form, as in the sequential logic table.

3.4 Data Devices

All data processing or digital control systems have circuits to carry out the following operations:

- data input
- data storage
- data processing
- data output
- control and timing

Data processing devices must be controlled in sequence to carry out useful work. In a microprocessor system, most of this control logic is built into the CPU and its support chips, but additional control circuits usually need to be designed for each specific system. In order to
3.4.1 Data Input Switch

In Fig. 3.10(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 of the data output must be zero, 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 less than a few microamps. If necessary, a capacitor may be connected across the switch to absorb any of the switch contacts that do not close cleanly, thus ensuring a smooth transition from high to low, and back.

3.4.2 Tri-State Gate

The tri-state gate (TSG) (Fig. 3.10(b)) is a digital device which allows electronic switching and routing of signals through a data processing system. 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 circuit driver IC.

Figure 3.10: Data circuit elements. (a) Switch input; (b) Tri-state gate; (c) Data latch; (d) LED output.
symbol. TSGs can be obtained as individual gates in a small-scale integrated circuit chip and are used as basic circuit building blocks within large-scale integrated circuits such as the PIC microcontroller.

3.4.3 Data Latch

A data latch (Fig. 3.10(c)) is a circuit block which stores one bit of data, as described above. If a data bit is presented at the input D (0 or 1), and the latch is "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.

3.4.4 LED Data Display

An LED can provide a simple data display device. In Fig. 3.10(d) the logic level to be displayed (0 or 1) is fed to the current driver, which operates as a current amplifier and provides enough current (typically about 10 mA) to make the LED light up when the data is a '1'. The resistor value controls the size of the current. Some equipment and other matrix displays use LEDs to display decimal or hexadecimal digits by lighting up various LED segments or digits.

3.5 Simple Data System

The way that data is transferred through a digital system using the devices described above is illustrated in Fig. 3.11. The circuit allows one data bit to be input at the switch (0 or 1), stored at the output of the latch 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 power turned off.
4. While latched, the data at D3 is displayed by the LED (ON = '1'), via the current driver stage.

![Figure 3.11 1-bit data system](image-url)
Note that all active devices (gate, latch and driver) must be connected to the +5V power supply, but these supply connections do not have to be shown in a block diagram or logic circuit. Table 3.2 details the control sequence, with the data states which exist after each operation. Note that ‘x’ represents ‘don’t know’ or ‘don’t care’ (it could be 1, 0 or floating).

3.6 4-Bit Data System
Data is usually moved and processed in parallel form within a microprocessor system. The circuit shown in Fig. 3.12 illustrates this process in a simplified way.

The function of the 4-bit system is to add two numbers which have been 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–F. The display has a built-in decoder which 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.

The common data bus is used to minimise the number of connections required, but it means that only one set of data can be on the bus at any one time; therefore, only one set of gates must be enabled at a time. 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 (See Table 3.3).

The 4-bit adder works as follows; for the moment, we will assume that these operations are carried out manually using suitable switches or push buttons to generate the control signals. 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 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. The result can then be stored and displayed by operating the data output latch enable (DOLE).
If this operating sequence can be automated, and we then have the makings of a microprocessor. 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 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 a 'machine code program' as seen in Table 3.4.

The program described above has three instructions, of 9 bits in length – it is running a simple processing system with a set of codes which are equivalent to the machine code program on a microcontroller. 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. 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 viable system as outlined in Chapter 1. The means to program the ROM (a development system) completes our processor system. This is how early calculator chips were developed, leading to microprocessors and microcontrollers. The system outlined can be built (if you can get the obsolete components)!
Table 3.3  4-bit system operating sequence

<table>
<thead>
<tr>
<th>Operation</th>
<th>Display hex</th>
<th>DoL</th>
<th>DaE</th>
<th>DoG</th>
<th>DaG</th>
<th>DI</th>
<th>4-bit binary</th>
<th>Switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ready for input</td>
<td>3F</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Ready for input (for completeness)</td>
<td>3F</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Set data input number A on switches</td>
<td>0110</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>Enable data A onto bus by switching on input gates</td>
<td>0110</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>Store data A in latch A by clocking it with a pulse</td>
<td>0110</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>Disable input gates – no valid data on bus</td>
<td>0110</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>Set data input number B on switches</td>
<td>0101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>Enable data B onto bus by switching on input gates</td>
<td>0101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>Store data B in latch B by clocking it with a pulse</td>
<td>0101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>Disable input gates – no valid data on bus</td>
<td>0101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>Enable result from ALU onto bus</td>
<td>0101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0111</td>
<td>0101</td>
</tr>
<tr>
<td>Store result in output latch by clocking it with a pulse</td>
<td>0101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0111</td>
<td>0101</td>
</tr>
<tr>
<td>Result displayed – ready for next input</td>
<td>0101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0111</td>
<td>0101</td>
</tr>
</tbody>
</table>

*Note: The 4-bit binary values represent the state of the input switches.*
Table 3.4 4-bit system ‘machine code program’

<table>
<thead>
<tr>
<th>Instruction code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0101</td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td>15 x</td>
</tr>
<tr>
<td></td>
<td>Input and latch data A</td>
</tr>
<tr>
<td>1 0011</td>
<td>0101</td>
</tr>
<tr>
<td></td>
<td>13 x</td>
</tr>
<tr>
<td></td>
<td>Input and latch data B</td>
</tr>
<tr>
<td>0 1001</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>09 x</td>
</tr>
<tr>
<td></td>
<td>Latch and display result</td>
</tr>
</tbody>
</table>

Summary

- MOS digital circuits are based on the field effect transistor acting as a current switch. These are combined to form logic gates on integrated circuits.
- The basic set of logic gates is AND, OR and NOT, from which all logic functions can be implemented. NAND, NOR and XOR form a useful additional set.
- Combinational logic gives outputs which depend only on the current input combination. Sequential logic outputs additionally depend on the prior sequence of inputs.
- Basic data system devices are the data latch used for bit storage and the TSG used to route data. Data input and output devices are also needed. Additional logic circuits provide sequential control.

Questions

1. Why is it necessary for battery-powered digital circuits to operate at a wide range of voltages?
2. Draw a simple logic inverter using a FET and a resistor, and then add to the circuit to provide the AND and OR logic functions.
3. Describe in one sentence the operation of (a) an OR gate and (b) an AND gate.
4. Representing a 1-bit full adder as a single block, with inputs A, B and C, and outputs S and C, draw a 4-bit adder consisting of four of these blocks, with the inputs and outputs shown in Fig. 3.6(a).
5. Construct a timing diagram for the sequential logic table shown in Fig. 3.7(b) to show how a basic latch works.
6. Draw a circuit with two input logic switches whose data can be stored in one of three different D-Type latches. Describe how the control logic must work to allow either input to be stored in any of the latches.
7. Modify the 4-bit system (Fig. 3.12) operating sequence so that only the final result is displayed.
Activities

1. Construct the full adder circuit using the necessary logic chips and check that it works as described. Then simulate it using a suitable schematic capture and simulation package and check that the function is accurately simulated.

2. Investigate the operation of a suitable programmable logic device, and work out how it would be programmed to create a 4-bit adder.

3. Investigate how the basic latch circuit could be used to 'de-bounce' the data input switch.

4. Using suitable digital circuit simulation software, test the 4-bit system operation, if the components are available in the device libraries.
4.1 Encoder and Decoder

A digital encoder is a device which has a number of separate inputs and a binary output. An output binary number is generated corresponding to the numbered input which is active. A decoder is a device which carries out the inverse logical operation: a binary input code activates the corresponding output. Thus, if the binary code for 5 (101) is input, output 5 of the decoder goes active (usually low). An example of encoder and decoder operation is described below, where they are used to operate a keypad.

A set of switches are combined in a two-dimensional array to form a simple keyboard. These may have 12 keys (decimal) or 16 keys (hexadecimal). The decimal pad has digits 0–9, hash (#), and star (*), while the hex keypad has digits 0–F. A hex keypad is used in the example illustrated in Fig. 4.1. To read the keypad, an interface is needed which can detect when one of the buttons has been pressed. A software-based solution is explained later in Chapter 15, so a hardware solution will be described here.

Four select lines are output from a row decoder which are normally high (pull-up resistors can be attached to each line if necessary). When a binary input code is applied, the corresponding

4.1 Encoder and Decoder

A digital encoder is a device which has a number of separate inputs and a binary output. An output binary number is generated corresponding to the numbered input which is active. A decoder is a device which carries out the inverse logical operation: a binary input code activates the corresponding output. Thus, if the binary code for 5 (101) is input, output 5 of the decoder goes active (usually low). An example of encoder and decoder operation is described below, where they are used to operate a keypad.

A set of switches are combined in a two-dimensional array to form a simple keyboard. These may have 12 keys (decimal) or 16 keys (hexadecimal). The decimal pad has digits 0–9, hash (#), and star (*), while the hex keypad has digits 0–F. A hex keypad is used in the example illustrated in Fig. 4.1. To read the keypad, an interface is needed which can detect when one of the buttons has been pressed. A software-based solution is explained later in Chapter 15, so a hardware solution will be described here.

Four select lines are output from a row decoder which are normally high (pull-up resistors can be attached to each line if necessary). When a binary input code is applied, the corresponding
row select line goes low. A 2-bit binary counter can be used to drive the row decoder (see Figure 4.1 for example), which will generate a row select code at each count, continuously. If a switch on the active row is pressed, this low bit can be detected on the column line. The column lines, which are also normally high, are connected to a column encoder. This generates a binary code which corresponds to the input which has been taken low by connection to the row which is low.

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 which has been pressed. For instance, if key 9 is pressed, the row select code would be (2, 1) and the column detect code would be (1, 1). The combination of these codes would be (11), which corresponds to key 9.
4.2 Multiplexer, Demultiplexer and Buffer

These devices can be constructed from the same set of gates: two TSGs and a logic inverter, as shown in Fig. 4.2. All are important for the operation of bus systems as outlined in Chapter 3.

A multiplexer is basically an electronic changeover switch, which can select data from alternate sources within the data system. A typical application is to allow two different signal sources to use a common signal path (bus line) at different times. In Fig. 4.2(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 (Fig. 4.2(b)) splits the signal using the same basic devices as the multiplexer.

The bidirectional buffer (Fig. 4.2(c)) is used to allow data to pass in one direction at a time where a data path, for example, can be unidirectional in one direction and bidirectional in the other. The TSGs are connected nose to tail, and operate alternately as in the multiplexer. When the control input is low, the data is enabled through from left to right, and from right to left.

4.3 Registers and Memory

We have seen previously how a 1-bit data latch works. If the bidirectional data buffer (Fig. 4.2(c)) 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 Fig. 4.3(a), the data read-in line can be connected to the D input or Q output, depending on the state of the select input. The data from the latch is then enabled out to the data line. The latch enable is activated at the appropriate time.

If several of these register elements are wired together, a data word can be stored. A common data word size is 8 bits (1 byte), and most computer hardware uses multiples of 8 bits. A 4-bit register, consisting of 4 data latches, is shown in Fig. 4.3(b). The registers 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.

4.4 Memory Address Decoding

A static RAM memory location operates in a similar way to the register. The memory device typically stores a block of 8-bit data bytes which are accessed by numbered locations (Fig. 4.4). The data access is performed by the address decoder. The selected byte is loaded into the register, and the address decoder is used to select one of the eight locations in the memory block. An internal address decoder selects the location on the basis of the address, and 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 electronically disconnected when another device wants to use the 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\(_2\)–111\(_2\)). This number of locations can be calculated directly as \(2^n\), where \(n\) is the number of address pins. Thus, for 3 address pins, \(2^3 = 8\) locations are provided. Some useful values are listed in Table 4.1. Each memory location normally contains 1 byte. Table 4.1 was derived in a similar way and contains other significant values up to the largest 16-bit number, \(FFFF\). In a microprocessor system, program and data words (binary numbers) are stored in these memory locations, ready for processing in the CPU. The data transfer is implemented using a
Memory Address Decoding

Figure 4.3 Register operation. (a) Data register bit operation; (b) 8-bit data register operation.

Figure 4.4 Memory device operation.
**Table 4.1**

<table>
<thead>
<tr>
<th>Address lines</th>
<th>Memory size</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>256 bytes</td>
</tr>
<tr>
<td>10</td>
<td>1024 bytes</td>
</tr>
<tr>
<td>16</td>
<td>64 kb</td>
</tr>
<tr>
<td>20</td>
<td>1 Mb</td>
</tr>
<tr>
<td>30</td>
<td>1 Gb</td>
</tr>
</tbody>
</table>

Figure 4.5 Microprocessor system addressing.

Note: which is a common set of lines which pass data between memory and registers, with TSGs directing the data to the correct destination (Fig. 4.7).

4.5 System Address Decoding

Although we are mainly concerned with microcontroller architecture, it is worthwhile briefly at memory and I/O access in a conventional system, because it explains the process which occurs within the microcontroller chip and is important for an overview of microprocessor systems. It is a logical extension of the address decoding within each microcontroller chip. We will look at microcontroller systems in more detail in Chapter 1, where we explain the address space for each memory chip. Only one chip can use the data bus at any one time, so a system of chip selection is needed, so that the processor can "talk to" the required peripheral chip.
### Table 4.2 Typical memory map

<table>
<thead>
<tr>
<th>Address range</th>
<th>Number of locations</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 - 7FFF</td>
<td>8000 - 801F</td>
<td>Parallel port registers</td>
</tr>
<tr>
<td>8000 - 801F</td>
<td>20</td>
<td>Serial port registers</td>
</tr>
<tr>
<td>A000 - A008</td>
<td>8</td>
<td>ROM (16k)</td>
</tr>
<tr>
<td>C000 - FFFF</td>
<td>4000</td>
<td>ROM (16k)</td>
</tr>
</tbody>
</table>

Figure 4.5 shows the basic connections in a microprocessor system which allows the CPU to read and write data from the memory and I/O devices. Let us assume that the CPU is reading a program instruction from ROM, although all data transfers are done in the same way.

The CPU program counter contains the address of the instructions that it is output as a binary code on the address bus. The system address decoder takes, in this system, the 2-bit code on the most significant address lines and sets one of four chip select lines active accordingly, which activates the chip to be accessed (ROM). The low order address lines are used, as described in section 4.4, to select the required location within the chip. Thus, the location selection is a two-stage process, with external (system) and internal (chip decoding) of the address.

When the location has been selected, the data stored in it can be read or written to by the data bus. The data in the selected device (ROM) is enabled, while all others connected to the bus are disabled, allowing the ROM data onto the data bus. 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 port only has a few addressable locations, its registers, so only a few of the address lines are needed for this device.

As a result of the design of the decoding system, the memory and I/O devices are allocated to specific ranges of address. The decoder can then be tailored to specific applications by selecting the appropriate ranges of addresses.

A typical memory map for a system with a 16-bit address bus (four hex digits) is shown in Table 4.2. The I/O is memory mapped, so the port registers are placed in the same address space as the memory. Notice that not all the available addresses have to be used.

### 4.6 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 stage outputs one pulse for every two pulses which are input, so the output pulse frequency is half the input frequency for each stage (Fig. 4.6(a)). The counter/timer register can therefore be viewed as a binary counter or frequency divider, depending on the application.
Digital Systems

T-type (toggle mode) flip-flop

Input frequency = f
Output frequency = f/2

00000000

Clock/pulses

0–255

Preload with start value

Time-out signal (a)

Figure 4.6: Counter/timer register operation. (a) Toggle mode stage; (b) 8-stage counter register.

Figure 4.6(b) shows an 8-bit counter/timer, with the input to the LSB at the right. The binary count stored 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 to output is indicated here, which can be used as a 'carry out' in counting operations or 'time out' in timing operations. In a microprocessor system, the 'time-out' signal typically initiates a 4-bit or 8-bit 'status' register to record the 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. Interrupts will be explained fully later.

If the clock pulse frequency is 1 MHz (1 megahertz), the period will be 1/μs (1 microsecond), and the counter will generate a time-out signal every 256 μs. If the counter can be preloaded, 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, known time intervals can be generated. In conventional microprocessor systems, the I/O ports often contain counters that divide the input frequency by a factor of between 2 and 256 in order to extend its range. Many have 16-bit counters, which allow longer intervals to be generated without a prescaler. PIC timers/counters are explained in more detail in Chapter 9.

4.7 Serial and Shift Registers

The general purpose data register, as described in Section 4.3, 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 which 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, which is read one bit at a time from a single data line. The data can then...
4.7 Shift Register Operation

Shift registers are used in microprocessor serial ports, where data is sent or received in serial form. In the PC, this could be the modem or network port, the keyboard input or VDU output.

4.8 Arithmetic and Logic Unit

The main function of any processor system is to process data, for example, to add two numbers together. The arithmetic and logic unit (ALU) shown in Fig. 4.8 is an essential feature of any microprocessor or microcontroller.

A binary adder block has already been described, 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, XOR; the operation to be carried out is determined by function select inputs. These in turn are derived from the instruction code in the program being executed in the processor. The block arrows used in the diagram indicate the parallel data paths, which carry the operands to the ALU, and the result away. A set of data registers which store the operands are usually associated with the ALU, as seen in the 4-bit data system in Chapter 3.
4.9 Processor Control

The instruction decoder is a logic circuit in the CPU which takes 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 (Fig. 4.9) 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 instructions are generated and stored in suitable registers or back to memory.

The block diagram for the PIC 16F84A provided in the Appendix (Fig 1-1) shows a complete system. The data paths between each block show the data word size and the possible data transfer routes. The control lines which are set up by the instruction decoder and control block connect to all parts of the processor enabling data inputs at the source end, and operating data latches at the receiving end, of all data transfers.

We can now see more clearly the main difference between the microcontroller and microprocessor. In the conventional system, the bus system is external to the CPU, while in the microcontroller, the bus system is internal to the CPU, while it is external to the microcontroller.
Summary

- The encoder generates a binary code corresponding to the active numbered input, and the decoder carries out the inverse operation activating the selected output according to the binary input.
- The multiplexer allows a selected input to be connected to a single output line, and the demultiplexer carries out the inverse operation connecting a single input line to a selected output line.
- A register or memory cell stores one bit of data using a data latch and bidirectional buffer.
- Numbered memory locations are accessed by decoding the address to generate a system device select and chip location select.
- Counters and timers use a counting register to count digital pulses, or measure time intervals using a clock input.
- Shift registers convert parallel to serial data, and back.
- The ALU provides data processing operations.
- The processor control signals are generated by the instruction decoder and timing circuits.
- The clock signal provides the timing reference signal for all processor operations.

Questions

1. Describe the process whereby an encoder and decoder could be used to scan a 4 × 32 key computer keyboard.
2. Two 8-bit registers, A and B, are connected to an 8-bit data bus via bidirectional buffers, so as to allow data to be stored and retrieved. Draw a block diagram and explain the sequence of signals required from the controller circuits to transfer the contents from A to B using data direction select and data latching signals.
3. A minimal microprocessor system, configured as shown in Fig. 4.5, has a 16-bit address bus. The two most significant lines, A14 and A15 are connected to the 2-bit decoder, which operates as specified in Fig. 4.1(b), with A14 = input A (LSB) and A15 = input B (MSB). The four select outputs are connected to memory and I/O chip select inputs as follows: 0 = RAM1, 1 = RAM2, 2 = ROM, 3 = I/O. The RAM1 chip is selected in the range of addresses from 0000 to 3FFF (hex). Work out the lowest address where each of the three remaining chips are selected.
4. Calculate the number of locations in a memory chip which has 12 address pins.
5. Calculate the time interval generated by an 8-bit timer preloaded with the value 11001110 and clocked at 125 kHz.
1. In a suitable TTL, Logic device data book or supplier’s catalogue, look up the chip numbers and internal configuration of the medium scale ICs: 3 to 8-line decoder, octal D-latch, octal bus transceiver, 9-bit shift register, 8-bit binary counter. Also identify the largest capacity ROM chip listed in your source.

2. Refer to the PIC 16F8X data sheet, Fig 1-1 (block diagram). State the function of the following features: ROM program memory, RAM file registers 8 x 8, program counter, instruction register, instruction decode and control, multiplexer, ALU, W reg, I/O Ports, TMR0.

3. Refer to the PIC 16F8X data sheet, Fig 4-1 (block diagram of pins RA0-RA7). Identify the following devices in the circuit diagrams (flip-flop gates, TTL, transparent data latch, edge-triggered data latch). Describe how an input data bit would be transferred onto the internal data bus line from the I/O pin.
Chapter 5

Microcontroller Operation

5.1 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 analyse the circuit, which will be very complex, in detail. The PIC data sheet may append a more detailed block diagram of the operation of the chip, however, we will look at a general block diagram of a microcontroller (Fig. 5.1).

The block diagram shows a general microcontroller that can be considered in two parts, the program execution section and the register processing section. This division reflects the PIC architecture, where the program and data are accessed separately. This arrangement increases the overall program execution speed and is known as Harvard architecture.

5.1.1 Program Memory

The control program is normally stored in non-volatile ROM. Microcontrollers which are designed for prototyping and short production runs have traditionally used Erasable Program-able ROM (EPROM) into which the program can be "blown" using a suitable programming equipment. In this chapter we will look at some common elements of microcontrollers and the basic features of machine code programs.
5.1.2 Program Counter

The program counter is a register which keeps track of the program sequence by storing the address of the instruction currently being executed. The default start address of the program is usually zero, that is, where the first instruction in the program will be stored unless the program author specifies otherwise. The program counter is therefore automatically loaded with zero when the chip is powered up or reset. In the PIC 16XXXX chips (any chip starting with 16), the program counter is file register 2.

As each instruction is executed, the program counter is incremented (increased by one) to point to the next instruction. Program jumps are achieved by changing the program counter to point to an instruction other than the next in sequence. For instance, if a branch back by three instructions is required, 3 is subtracted from the contents of the PC.

Sometimes, 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 at specific low addresses. For instance, PIC 16XXXX devices use address 004 to store the ‘interrupt vector’, if interrupts are
In some microcontrollers and microprocessors, this 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.

5.1.3 Instruction Register (IR) and Decoder

To execute an instruction, the processor copies the instruction code from program memory into the instruction register. If can then be decoded 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.

In the PIC, the instruction code includes the operand, which may be a literal value or register address. For example, if a literal 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 (Fig 1-1) in the data sheet.

5.1.4 Timing and Control

This sequential logic block provides overall control of the chip, and from it, control signals go to all parts of the chip to move the data around and carry out logical operations and calculations. A clock signal is needed to drive the program sequence, it is normally derived from an crystal oscillator, which provides an accurate, fixed frequency signal. There is always a maximum frequency of operation specified. PIC 16XXXX chips are specified at an frequency from a maximum of 20 MHz, down to zero.

The reset input can restart the program at any time by clearing the program-counter to zero. If the program runs in a continuous loop, there is no instruction to end the loop, the circuit may be needed. However, it is not essential to connect an active reset input because the program will restart automatically from the start of the program at address 0000 when the power is turned on. A clock signal is needed to restart the program, and a reset switch is not required.

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 16XXXX, the program will restart at address 0000, where the sequence known as the ‘interrupt service routine’ (or a jump to it) must be stored. More details are provided in Chapter 9.

5.1.5 Working Register

In some microcontrollers and microprocessors, this 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. In the PIC,
64

Microcontroller Operation

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 or accumulator works closely with the ALU in the data processing operation.

5.1.6 Arithmetic and Logic Unit

This is a combinational logic block which 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 (for instance, 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/accumulator operations are detailed later in this chapter.

5.1.7 Port Registers

Input and output in a microcontroller is achieved by simply reading or writing a port data register. If a binary code is presented to the input pins of the microcontroller by an external device (for instance, a set of switches), the data is latched into the register allocated to that port when it is read. This input data can then be moved (copied) into another register for processing. If a port register is initialised for output, the data moved to that register is immediately available at the pins of the chip. It can then be displayed, for example, on a set of 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.

In the PIC 16F84A, there are two ports, A and B. Port A has five pins and Port B has eight. Port direction registers are mapped as special function registers 5 and 6, respectively (Bank 0). In larger chips, additional ports may be available. For example, in the PIC 16F877, Ports C, D, and E are available at register addresses 5, 6, and 7, respectively. The port data direction registers are mapped into a second register bank. These are accessed by special instructions which will be explained later.

5.1.8 Special Function Registers (SFRs)

These registers provide dedicated program control registers and processor status bits. In the PIC, these registers are used as part of the main register block. The working register is the only one that is not located in the main register block and is accessed by name, "W", not by number.

A processor will also contain control registers whose bits are used individually to set up the processor operating mode or record significant results from those operations. Processors generally have a status register (SR) which will contain a zero (Z) flag. This bit is automatically set to 1 if the result of any operation is zero in the destination register (the register which receives the result). The carry (C) flag is another commonly used status register bit. 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. In the PIC 16XXXX, the status register is file register 3.

The status register bits are often used to control program sequence by conditional branching. Alternate sections of code are executed depending on the state of the status flag. In the PIC
Table 5.1  Selected PIC special function registers.

<table>
<thead>
<tr>
<th>Register</th>
<th>File</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>TMR0</td>
<td>Timer/counter allows external and internal clock pulses to be counted</td>
</tr>
<tr>
<td>02</td>
<td>PCL</td>
<td>Program counter stores the current execution address</td>
</tr>
<tr>
<td>03</td>
<td>STATUS</td>
<td>Individual bits record results and control operational options</td>
</tr>
<tr>
<td>05</td>
<td>PORTA</td>
<td>Bidirectional input and output bits</td>
</tr>
<tr>
<td>06</td>
<td>PORTB</td>
<td>Bidirectional input and output bits</td>
</tr>
<tr>
<td>0B</td>
<td>INTCON</td>
<td>Interrupt control bits</td>
</tr>
<tr>
<td>85</td>
<td>TRISA</td>
<td>Port A data direction bits</td>
</tr>
<tr>
<td>86</td>
<td>TRISB</td>
<td>Port B data direction bits</td>
</tr>
</tbody>
</table>

5.2 Program Operations

We have seen in Chapter 2 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 element, 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 which the instruction decoder can recognise. These codes could be worked out manually from the instruction set given in the data sheet (Table 7-2). When computers were first developed, this was indeed how the program was entered, using a set of switches or keyboard. This is obviously
time-consuming and inefficient, and it was soon realized that it would be useful to have a compile-able tool which would generate the machine code automatically from a program which was written in a more user-friendly form. Assembly language programming was therefore developed, which allowed for a more efficient way of writing programs. The main disadvantage of assembly language programming is that it is much more difficult to read and write than high-level languages. Assembly language is therefore used for writing programs which need to be very efficient, such as those used in embedded systems.

Assembly language programming is similar to machine language programming, but it is more user-friendly. Assembly language uses mnemonic ("designed to aid the memory") code words. 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 which is specified as the operand. The destination register is specified by number (file register address), such as 0C (the first general purpose register in the PIC 16F84). The complete instruction, with its machine code equivalent, is:

```
008C MOVWF 0C
```

There are two main types of instructions:

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.

5.2.1 Single Register Operations

The processor operates on data stored in registers, which typically contain 8 bits. The data can originate in three ways:

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

This data can be processed using the set of instructions defined for that processor. Table 5.2 shows a typical set of operations which can be applied to a single register. The same binary number is shown before and 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 assembler code to increment a PIC register is:

```
0A06 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
Table 5.2: Single register operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Before</th>
<th>After</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLEAR</td>
<td>0101</td>
<td>0101</td>
<td>0101 1101 Reset all bits to zero</td>
</tr>
<tr>
<td>INCREMENT</td>
<td>0101</td>
<td>0101</td>
<td>0101 1110 Increase binary value by one</td>
</tr>
<tr>
<td>DECREMENT</td>
<td>0101</td>
<td>0101</td>
<td>0101 1100 Decrease binary value by one</td>
</tr>
<tr>
<td>COMPLEMENT</td>
<td>0101</td>
<td>0101</td>
<td>0101 1101 Invert all bits</td>
</tr>
<tr>
<td>ROTATE LEFT</td>
<td>0101</td>
<td>0101</td>
<td>0110 0010 Shift all bits left by one place, replace MSB in LSB</td>
</tr>
<tr>
<td>ROTATE RIGHT</td>
<td>0101</td>
<td>0101</td>
<td>0010 1110 Shift all bits right by one place, losing the LSB</td>
</tr>
<tr>
<td>CLEAR BIT</td>
<td>0101</td>
<td>0101</td>
<td>0101 0101 Reset bit (3) to 0</td>
</tr>
<tr>
<td>SET BIT</td>
<td>0101</td>
<td>0101</td>
<td>0101 1101 Set bit (7) to 1</td>
</tr>
</tbody>
</table>

5.2.2 Register Pair Operations

Table 5.3 shows basic operations that can be applied to pairs of registers. Normally, the result is retained in one of the registers, which is referred to as the destination register. A binary code to be combined with the contents of the destination register is obtained from the source register. The source register contents remain unchanged after the operation.

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

Move

It is the most commonly used instruction in any program and 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 or the processor is reset.

```
080C MOVF 0C, W
```

This instruction moves the contents of register 0C (binary 10101000110) into the working register.

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 are created if necessary by using shift, add and subtract operations.

```
E0E4 ABWF IC, W
```

This instruction adds the contents of W to register IC.
Table 5.3: Operations on register pairs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Source before</th>
<th>Operation</th>
<th>Source after</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE</td>
<td>Source</td>
<td>MOVE</td>
<td>Destination</td>
<td>Overwrite destination with source, leaving source unchanged</td>
</tr>
<tr>
<td>ADD</td>
<td>Source</td>
<td>ADD</td>
<td>Destination</td>
<td>Add source to destination, leaving source unchanged</td>
</tr>
<tr>
<td>SUB</td>
<td>Source</td>
<td>SUB</td>
<td>Destination</td>
<td>Subtract source from destination, leaving source unchanged</td>
</tr>
<tr>
<td>AND</td>
<td>Source</td>
<td>AND</td>
<td>Destination</td>
<td>AND source and destination bits, leaving source unchanged</td>
</tr>
<tr>
<td>OR</td>
<td>Source</td>
<td>OR</td>
<td>Destination</td>
<td>OR source and destination bits, leaving source unchanged</td>
</tr>
<tr>
<td>XOR</td>
<td>Source</td>
<td>XOR</td>
<td>Destination</td>
<td>Exclusive OR source and destination bits, leaving source unchanged</td>
</tr>
</tbody>
</table>

Logical operations act on the corresponding pairs of bits in a literal, or source register, and destination. The result is normally stored 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 Chapter 3).

3901 ANDLW 01

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. This type of operation can be used for bit testing if the processor does not provide a specific instruction.

5.2 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. Most of the time, the PC is simply incremented by one to proceed to the next instruction. However, if a program jump (branch) is needed, the PC must be modified, that is, the address...
of the next instructions must be loaded into the PC, replacing the existing value. This new address can be given as a jump instruction operand (absolute addressing), or calculated from the current address, for example, by adding a number to the current PC value (relative addressing).

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, the program counter 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.

On the question of terminology, 'jump' and 'branch' are two terms for describing sequence control operations, but the ways in which they work are slightly different. A 'branch' is made relative to the current address, by adding to the current value in the PC. A 'jump' uses absolute addressing, that is, the contents of the PC are replaced with the destination address. Because the PC is cleared on reset, the jump instruction will always move to address 0001, but such operations can be created from the available instructions, if required. Therefore, the program counter can be modified directly using register processing operations to create a relative jump or branch.

The jump instructions must have a destination address as the operand. This can be given 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 will see later 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', in the same way that mnemonics are used to represent the binary machine code instructions. 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 Figs 5.2, 5.3 and 5.4.

**Jump**

The unconditional jump (Fig. 5.2) 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. instruction that originates from the same address. Note that the code for GOTO is 28h combined with the destination address 05h, giving the instruction code 2805h (h for hexadecimal).

---

**Program**

<table>
<thead>
<tr>
<th>Address</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Start</td>
</tr>
<tr>
<td>0001</td>
<td>Label</td>
</tr>
<tr>
<td>0002</td>
<td>Next</td>
</tr>
<tr>
<td>0003</td>
<td>Label</td>
</tr>
<tr>
<td>0004</td>
<td>Label</td>
</tr>
<tr>
<td>0005</td>
<td>Jump</td>
</tr>
</tbody>
</table>

**Figure 5.2** Unconditional jump.
Microcontroller Operation

The unconditional jump is often used at the very end of a program to go back to the beginning of the sequence, and keep repeating it.

```
start: fixed 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 label and its reference must match exactly. The label is replaced by the corresponding address by the assembler when creating the machine code for the GOTO instruction.

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.

In Fig. 5.3, the code 1885 tests an input bit of the PIC and skips the next instruction if it is zero ('0'). Instruction YYYY (representing any valid instruction code) is then executed. If the bit is not zero ('1'), 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 an RISC processor, 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

<table>
<thead>
<tr>
<th>Address</th>
<th>Program</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>1885</td>
<td>Test bit 1 of file register 5</td>
</tr>
<tr>
<td>001</td>
<td>YYYY</td>
<td>Execute this jump if bit = 1</td>
</tr>
<tr>
<td>002</td>
<td>2807</td>
<td>Jump to here if bit not zero</td>
</tr>
<tr>
<td>003</td>
<td>ZZZZ</td>
<td>....</td>
</tr>
</tbody>
</table>
Program Operations

71

execution continues either at the instruction following the jump, if the jump is skipped, or at
the jump destination.

In complex instruction set processors, conditional jump instructions which test specific bits
in the CPU status register are usually available. When the CPU operates on some data in a
register, status register bits record certain results, such as whether the result was zero or not. In
this case, if the result of a previous instruction was zero, a 'zero flag' is set to 1 in the status
register.

Pseudocode (structured program description) for a typical use of the conditional jump, a
delay routine, would look like this:

Allocate 'Count' Register

....
again
Load 'Count' register with literal XX
Decrement 'Count' register
If not zero, jump to label 'again'
....

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 calculated.

Subroutines

Subroutines are used to carry out discrete program functions. They allow programs to be
written in manageable, self-contained blocks, which can then be executed as required. 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 placed in the PC, as for the GOTO
instruction. In addition, the address of the next instruction in the main program is saved in
the stack. In the PIC, this is a special block of RAM memory used only for this purpose. In
conventional processors, a part of main RAM memory may be set aside for this purpose. The
return address is 'pushed' onto the stack when the subroutine is called, and 'popped'
back into the program counter at the end of the routine, when the RETURN instruction is
executed.

In Fig. 5.4, 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
the CALL instruction is decoded, the destination address is loaded into the PC, and the
program counter jumps to the subroutine at 0F0. 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 be
jumped back to the program counter at the end of the routine, when the RETURN instruction is
executed.

One advantage of using subroutines is that the block of code is only moved into main
memory once, but called many times during program execution. This can be very
important in optimizing program performance.
Microcontroller Operation

Address Main Program

000 To call subroutine

Program counter loaded with destination address 0F0 and return address 003 pushed onto stack automatically

20F0

YYYY

To return from subroutine

Program counter reloaded with 003 pulled from stack

Figure 5.4 Subroutine call.

A delay loop is written as a subroutine. It is a counting loop which just uses up time to give a delay between status changes, which is called "time slice" in a computer. A delay loop sets an output high, delays, sets the output low, and delays again before repeating the whole process. This same program also creates an example of direct modification of the program counter (labelled PCL) to create a data table.

Pseudocode for a delay loop written as a subroutine would be as follows:

; 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

....

Figure 5.4 Subroutine call.
Program Operations

END of Program

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

delay

; Increment 'Count' register

TEST 'Count' register for zero

If not zero, jump to label 'delay'

RETURN from subroutine

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

Note that the 'delay' routine is called twice, but using a different delay value in the 'Count' register. Thus, the same code can be used to give different delay times. Notice also that we have started using comments in our pseudocode to identify the functional blocks as the program gets more complex.

Summary

• The typical microcontroller contains a program execution section, and a register processing section.

• The program counter steps through the program ROM 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 hold 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.

Questions

1. Outline the sequence of program execution in a microcontroller, describing the role of the program ROM, program counter, instruction register, instruction decoder, and timing and control block.

2. A register is loaded with the binary code 01101010. State the contents of the register after the following operations on this data: (a) clear, (b) increment, (c) decrement, (d) complement, (e) rotate right, (f) shift left, (g) clear bit 5, (h) set bit 0.
1. Microcontroller Operation

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.

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. Write a pseudocode program 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.

Answers

2. (a) 00000000, (b) 01101011, (c) 01101001, (d) 10010101, (e) 00110101, (f) 11010100, (g) 01001010, (h) 01101011.

3. (a) 00100111, (b) 10101101, (c) 01000010, (d) 01101011, (e) 00101001.

4. PC Stack
   0172 XXXX before Call
   0173 XXXX Call subroutine
   0174 XXXX subroutine starts
   0175 XXXX instructions...
   0176 XXXX return
   0177 XXXX after Call

5. Allocate registers A, B, C
   Clear register A
   Move 4 into register B
   Move 3 into register C
   Loop: Add B to A
   Increment C
   Test C for zero
   Jump back to 'loop' if C not zero
   Finished with product in A
1. Study the PIC 16F8X block diagram (Appendix A, Fig. 1-1), and identify the features described in Section 5.1.

2. Study the PIC instruction set (Appendix A, Table 7-2) and allocate the instructions to the following categories: Move, Arithmetic, Logic, Jump and Control. Make a list of instructions, organized into these categories, with a description and an example for each showing the required syntax (one line per instruction). This can then be used as a handy instruction reference when programming.
This Page is Intentionally Left Blank
Chapter 6
A Simple PIC Application

6.1 Hardware Design

A very simple machine code program for the PIC will now be developed, avoiding complicating factors as far as possible. A simplified internal architecture will be used to explain the execution of the program, and the program will then be developed further, with new programming techniques being added at each step. Since the core architecture and programming methods are similar for all PIC microcontrollers, this serves as an introduction to the whole family.

The specification for the application is as follows. The circuit will 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 commence from zero.

6.1.1 PIC 16F84A Pinout

The PIC 16F84A microcontroller is supplied in an 18-pin DIL (dual in line) chip. Simplified pin labelling, taken from the data sheet (see Appendix A), is shown in Fig. 6.1. Some of the pins have dual functions which will be discussed later. The suffix ‘A’ in the chip number indicates the enhanced version of the chip, which, so far as we are concerned, is functionally identical to the original 16F84, except that it can run at 20 MHz.

6.2 Program Execution

6.3 Program BIN1

6.4 Assembly Language
A Simple PIC Application

Figure 6.1 Pin-out of PIC 16F84.

The chip has two ports, A and B. The port pins allow data to be input and output as digital signals, at the same voltage levels as the supply which is connected to Vdd and Vss. CLKIN and CLKOUT are used to connect clock circuit components, and the chip then generates a fixed frequency clock signal which drives all its operations. !MCLR ('NOT Master CLEAR') is a reset input, which can (optionally) be used to restart the program. Note that the active low operation of this input is indicated by a bar over the pin label. An exclamation mark at the beginning of the pin label means the same thing. In many 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.

A summary of the pin functions is provided in Table 6.1. Port B has eight pins, so we will assign these pins to the LEDs and initialise them as outputs. Port A has five pins, of which two can be used for the input switches. A resistor and capacitor will be connected to the CLKIN pin to control the clock frequency. !MCLR must be connected to +5V.

6.1.2 BIN Hardware Block Diagram

The hardware arrangement required for the application can be represented in a simplified form as a block diagram (Fig. 6.2). The main parts of the hardware are shown, together with the direction of signal flow. The names of the signals must be described with labels or illustrated with simple diagrams. 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 through to RA4, and RB0 through to RB7 respectively. The two push button switches will be connected to RA0 and RA1, and a set of LEDs connected to RB0-RB7. These switches will later be used to control the output sequence. RA1 will be programmed to act...
Table 6.1 PIC 16F84 pins arranged by function

<table>
<thead>
<tr>
<th>Pin</th>
<th>Label</th>
<th>Function</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>Vdd</td>
<td>Positive supply</td>
<td>+5V nominal, 3–6 V allowed</td>
</tr>
<tr>
<td>5</td>
<td>Vss</td>
<td>Ground supply</td>
<td>0 V</td>
</tr>
<tr>
<td>4</td>
<td>!MCLR</td>
<td>Master clear</td>
<td>Active low reset input</td>
</tr>
<tr>
<td>16</td>
<td>CLKIN</td>
<td>Clock input</td>
<td>Connect RC clock components to 16</td>
</tr>
<tr>
<td>15</td>
<td>CLKOUT</td>
<td>Clock output</td>
<td>Connect crystal oscillator to 15 and 16</td>
</tr>
<tr>
<td>17</td>
<td>RA0</td>
<td>Port A, Bit 0</td>
<td>Bidirectional Input/Output</td>
</tr>
<tr>
<td>18</td>
<td>RA1</td>
<td>Port A, Bit 1</td>
<td>Bidirectional Input/Output</td>
</tr>
<tr>
<td>1</td>
<td>RA2</td>
<td>Port A, Bit 2</td>
<td>Bidirectional Input/Output</td>
</tr>
<tr>
<td>2</td>
<td>RA3</td>
<td>Port A, Bit 3</td>
<td>Bidirectional Input/Output</td>
</tr>
<tr>
<td>3</td>
<td>RA4</td>
<td>Port A, Bit 4</td>
<td>Bidirectional Input/Output</td>
</tr>
<tr>
<td>6</td>
<td>RB0</td>
<td>Port B, Bit 0</td>
<td>Bidirectional Input/Output</td>
</tr>
<tr>
<td>7</td>
<td>RB1</td>
<td>Port B, Bit 1</td>
<td>Bidirectional Input/Output</td>
</tr>
<tr>
<td>8</td>
<td>RB2</td>
<td>Port B, Bit 2</td>
<td>Bidirectional Input/Output</td>
</tr>
<tr>
<td>9</td>
<td>RB3</td>
<td>Port B, Bit 3</td>
<td>Bidirectional Input/Output</td>
</tr>
<tr>
<td>10</td>
<td>RB4</td>
<td>Port B, Bit 4</td>
<td>Bidirectional Input/Output</td>
</tr>
<tr>
<td>11</td>
<td>RB5</td>
<td>Port B, Bit 5</td>
<td>Bidirectional Input/Output</td>
</tr>
<tr>
<td>12</td>
<td>RB6</td>
<td>Port B, Bit 6</td>
<td>Bidirectional Input/Output</td>
</tr>
<tr>
<td>13</td>
<td>RB7</td>
<td>Port B, Bit 7</td>
<td>Bidirectional Input/Output</td>
</tr>
</tbody>
</table>

Figure 6.2 Block diagram of BIN hardware.

As a ‘run’ input, enabling the binary counter, while RA0 will provide a ‘reset’ input to restart
the counter sequence. However, these inputs will not be used in the first program, BIN1. The
connections required are shown in Table 6.2.

The clock components are the only additional parts needed, and the configuration and values for these are obtained from the data sheet. The

Figure 6.3 Block diagram of BIN hardware.
### Table 6.2 PIC 16F84A pin allocation for BIN application

<table>
<thead>
<tr>
<th>Pin Connection</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vss</td>
<td>0 V</td>
</tr>
<tr>
<td>Vdd</td>
<td>+5 V</td>
</tr>
<tr>
<td>MCLR</td>
<td>+5 V</td>
</tr>
<tr>
<td>CLKIN</td>
<td>Clock circuit</td>
</tr>
<tr>
<td>CLKOUT</td>
<td>Not connected (n/c)</td>
</tr>
<tr>
<td>RA0</td>
<td>Reset switch</td>
</tr>
<tr>
<td>RA1</td>
<td>Count switch</td>
</tr>
<tr>
<td>RA2</td>
<td>n/c</td>
</tr>
<tr>
<td>RA3</td>
<td>n/c</td>
</tr>
<tr>
<td>RA4</td>
<td>n/c</td>
</tr>
<tr>
<td>RB0</td>
<td>LED bit 0</td>
</tr>
<tr>
<td>RB1</td>
<td>LED bit 1</td>
</tr>
<tr>
<td>RB2</td>
<td>LED bit 2</td>
</tr>
<tr>
<td>RB3</td>
<td>LED bit 3</td>
</tr>
<tr>
<td>RB4</td>
<td>LED bit 4</td>
</tr>
<tr>
<td>RB5</td>
<td>LED bit 5</td>
</tr>
<tr>
<td>RB6</td>
<td>LED bit 6</td>
</tr>
<tr>
<td>RB7</td>
<td>LED bit 7</td>
</tr>
</tbody>
</table>

### Figure 6.3 Circuit diagram of BIN hardware

#### 6.1.3 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 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 the 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) to allow the chip to run. Other unused pins can be left open circuit, but unused I/O pins should be programmed as inputs.

6.2 Program Execution

The program for the chip is created using the PIC development system software on a PC and downloaded by a serial data link. This process will be described in more detail later, but for now we will assume that the program is in memory.

A block diagram showing a simplified program execution model for the PIC 16F84 is shown in Fig. 6.4. The binary program, shown 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 (00–0B) being reserved for specific purposes. These are called special function registers (SFRs). The rest may be used for temporary data storage, and are called general purpose registers (GPRs). Only GPR1 is shown in Fig. 6.4.

### 6.2.1 Program Memory

The program memory is a block of flash ROM, which means it is non-volatile but can be easily reprogrammed. The program created on the host computer is downloaded via the port register pins RB6 and RB7 when the chip is placed in “program mode” by supplying +14 V at the !MCLR pin 3. It is possible to write the program directly into the Flash ROM by a process called “in-circuit programming,” but this is not recommended for production applications.

![Figure 6.4 - PIC 16F84 simple program execution model]
A Simple PIC Application

programming software in hexadecimal form, but it is normally created using assembly language. This will be described later.

The 14-bit codes are loaded into memory starting at address 000. When the chip is powered up, the program counter (sets automatically to 000), and the first instruction is fetched from this address, copied to the instruction register in the control block, decoded and executed.

6.2.2 Program Counter: 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, so that it then points to the required jump destination.

6.2.3 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. 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 to or from W, in two stages, since direct moves between the registers are not available in the instructions set.

6.2.4 Port B Data Register: 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 initialised as outputs. The data direction is set as output 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.

6.2.5 Port A Data Register: File Register 05

The least significant five bits of file register 05 are connected to ports RA0–RA4. The other three are unused. This register is used to read the push buttons. If not initialised as outputs, the PIC I/O pins automatically become inputs, i.e. TRISA = xxx11111. We will use this default setting for Port A. However, the state of these inputs will have no effect unless the program uses them; the first program BIN1 will not.

6.2.6 General Purpose Register 1: File Register 0C

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

6.2.7 Bank 1 Registers

The main registers such as the program counter and port data registers are in a 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 them using a simple 'move' instruction.

Unfortunately, this is not the case with bank 1 registers. Special instructions are needed to load them, and there are two ways to do this. The first way is a simple method which we will use initially. It requires the 8-bit code to be loaded to be placed in W first, and then moved into the bank 1 register using the TRIS command. Later, we will use the recommended method, using bank selection.

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.

6.3 Program BIN1

The simple program called BIN1, introduced in Chapter 2, is listed as Program 6.1. The program consists of a list of 14 bit binary machine code instructions, represented as 4-digit hex numbers. If bits 14 and 15 are assumed to be zero, the codes are represented by hex numbers in the range 0000–3FFF. The program is stored at addresses 000–004 (5 instructions) in program memory.

<table>
<thead>
<tr>
<th>Memory address</th>
<th>Machine code instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>3000</td>
<td>Load working register (W) with number as last two digits, 00</td>
</tr>
<tr>
<td>001</td>
<td>0066</td>
<td>Store W in Port B direction code register</td>
</tr>
<tr>
<td>002</td>
<td>0186</td>
<td>Clear Port B data register</td>
</tr>
<tr>
<td>003</td>
<td>0A86</td>
<td>Increment Port B data register</td>
</tr>
<tr>
<td>004</td>
<td>2803</td>
<td>Jump back to address above</td>
</tr>
</tbody>
</table>

6.3.1 Program Analysis

The explanation of the program instructions must be related to the internal hardware of the PIC 16F84, as shown in Fig. 6.4.

Address 0000: Instruction = 3000

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 in zero in this case, can be seen in the code as the last two digits, 00.

Address 0001: Instruction = 0066

This means move (copy) the contents of W to Port B data direction register (TRISB). W must contain 00, which is loaded in the first instruction. This code will set all 8 bits of TRISB to zero, making all bits output. The file register address of Port B is given as the last digit of the code.

These first two instructions are required to initialise Port B for output, using the TRIS command to load the bank 1 register called TRISB in the register set.
A Simple PIC Application

Address 0002: Instruction = 0186

This instruction will clear file register 6 (last digit), which means that it will set all bits in the Port B data register (PORTB) to zero. Operations can be carried out directly on the port data register, and the results will appear immediately on the LEDs.

On start-up, the register bits default to '1', switching the LEDs on. When the 'clear' instruction is executed, they will go out.

Address 0003: Instruction = 0A86

Port B data is modified; the binary value is increased by 1 and this value will be seen on the LEDs.

Address 0004: Instruction = 2803

This is a jump instruction, which causes the program to jump 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.

6.3.2 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 1 each time. In other words, Port B data register will act as an 8-bit binary counter. When it reaches FF, 11111111, a roll over to 00000000 occurs. Increment operation is performed each time the binary count is incremented. RB1 will thus be toggled (inverted) every time RB0 is incremented, and so on, with each bit toggling at half the frequency of the previous bit. The MSB therefore toggles at 1/128 of the frequency of the LSB. The output pattern generated is shown in Fig. 6.5.

An instruction in the PIC takes 4 clock cycles to complete, unless it causes a jump, in which case, it will take 8 clock cycles (two instruction cycles). The repeated loop in BIN1 will

Figure 6.5 Waveforms produced by program BIN1 at Port B.
However, if we consider the PPM frequency generated for the RB0, then it will take $4 + 8 = 12$ clock cycles, and thus it will take 24 cycles for the RB0 to go low and high, giving the output period of the LSB. With the clock component values as indicated on the circuit diagram in Fig. 6.3, the clock can be set to run at 100 kHz, so RB0 will then flash at $100\times 10^3 / 24 = 4.167$ kHz. This is too fast to see unaided, but it can be slowed by increasing the value of the capacitor $C$ in the clock circuit. Alternatively, the outputs can be displayed on an oscilloscope. We will see later how to slow the outputs down without changing the clock. 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 by adjusting the clock rate or using a crystal oscillator. Again, we will see later how to slow the outputs down without changing the clock.

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 by adjusting the clock rate or using a crystal oscillator. Again, we will see later how to slow the outputs down without changing the clock.

6.4 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 recognise or remember.

6.4.1 Mnemonics

For this reason, microcontroller programs are normally written in assembly language, not machine code. Each instruction has a corresponding mnemonic defined in the instruction set in the data sheet. The main task of the assembler program supplied with the chip is to convert a source code program written in mnemonic form into the required machine code. The mnemonic form of the program BIN1 is shown in Program 6.2.

The instructions can now be written as recognisable (when you get used to them!) code words. The program can be typed into a text editor, spaced out as shown, using the tab key to place the codes in the correct columns. Note that the first column (column 0) must be 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.

<table>
<thead>
<tr>
<th>Line number</th>
<th>Column 0</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>MOVLW</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>TRIS</td>
<td>06</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>CLRF</td>
<td>06</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>INCF</td>
<td>06</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>GOTO</td>
<td>03</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>END</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Simple PIC Application

The meaning of the mnemonics is as follows:

0 MOVLW 00 Move Literal 00 into W
1 TRIS 06 Move W into TRISB to set Port B as outputs
2 CLRF 06 Clear file register 06 (Port B)
3 INCF 06 Increment file register 06 (Port B)
4 GOTO 03 Jump to address 03 (back to 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 in order to improve its readability.

6.4.2 Assembly

The source code program could be created using a general-purpose text editor, but is normally
created within a dedicated software package such as MPLAB, the PIC integrated development
environment (IDE), which contains the assembler as well as a text editor. The source code text
is entered and the assembler invoked from the menus. The assembler program analyses the
source code, character by character, and converts it into the binary code required for each instruction.
The terminology can be confusing here; the assembly-language application program (source
code) is created in the text editor, while the software tool which does the conversion is the
assembler program.

The source code is saved on disk as a text file called PROGNAME.ASM, where
'progname' represents any suitable filename. This is then assembled 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
which contains both the source and hex code, which may be useful later on when debugging
(fault finding) the program. Further information on using MPLAB will be given later.

6.4.3 Labels

The mnemonic form of the program with numerical operands can now be further improved. The
operands can be input in a more easily recognisable form, in the same way that the mnemonics
represent the instruction codes. The assembler is designed to recognise labels. A label is a word
which represents an address, register or literal. Examples used below are 'again', 'port', and
'label'.

The jump destinations are defined by label, by simply 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. Register
and literal labels, on the other hand, must be 'declared' at the beginning of the program, and
the assembler will then substitute the numerical operand for the label where it is found in the
source code.

The program BIN1 can thus be rewritten using labels as shown in BIN2 source code
(Program 6.3). The literal value 00 and the port register address 06 have been replaced with
labels which are assigned at the beginning of the program. These are 'equate' statements, which
Program 6.3  BIN2 source code using labels

<table>
<thead>
<tr>
<th>Label</th>
<th>Port A</th>
<th>Port B</th>
</tr>
</thead>
<tbody>
<tr>
<td>allout</td>
<td>BAD</td>
<td>BAD</td>
</tr>
<tr>
<td>portb</td>
<td>BAD</td>
<td>BAD</td>
</tr>
</tbody>
</table>

allow the numbers which are to be replaced in the source code to be declared. In this case, the label 'allout' will represent the Port A data direction code, while the data register address itself, 06, will be 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 instructions 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'.

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

6.4.4 Layout and Comments

A final version of BIN2 (Program 6.4) 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 modification and upgrading (software maintenance). Comments must be preceded by a semicolon (;), which tells the assembler to ignore the rest of the line. Comments and information can then occupy a whole line, or can be added after each instruction in columns 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. Blocks 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 operation of the program easier to understand. In this version, the first block contains the operand label equates, the second the port initialisation, and the third the output sequence. The layout of the program is very important in showing how it works.

We now have a program that can be entered into a text editor, assembled and downloaded to the PIC chip. The exact way of doing this will vary with the version of the PIC software and programming hardware that you use.

<table>
<thead>
<tr>
<th>Label</th>
<th>Port A</th>
<th>Port B</th>
</tr>
</thead>
<tbody>
<tr>
<td>again</td>
<td>BAD</td>
<td>BAD</td>
</tr>
<tr>
<td>BAD</td>
<td>BAD</td>
<td>BAD</td>
</tr>
<tr>
<td>portb</td>
<td>BAD</td>
<td>BAD</td>
</tr>
<tr>
<td>again</td>
<td>BAD</td>
<td>BAD</td>
</tr>
<tr>
<td>BAD</td>
<td>BAD</td>
<td>BAD</td>
</tr>
<tr>
<td>portb</td>
<td>BAD</td>
<td>BAD</td>
</tr>
<tr>
<td>again</td>
<td>BAD</td>
<td>BAD</td>
</tr>
<tr>
<td>BAD</td>
<td>BAD</td>
<td>BAD</td>
</tr>
</tbody>
</table>
A block diagram can be used to outline the hardware, and the circuit designed from it.

The PIC 16F84 program is stored in flash ROM, at addresses from 000. The instructions are decoded and executed by the processor control logic.

The CPU registers are modified according to the program, and the sequence can be modified by the instructions.

The PIC 16F84 has 14-bit instructions, containing both the operation code and operand.

The program is written using assembler mnemonics and labels to represent the machine code instructions and operands.

Layout and comments are used to document the program operation.

Questions

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

Program 6.4  BIN2 source code with comments

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQU allout</td>
<td>Define Data Direction Code</td>
</tr>
<tr>
<td>EQU portb</td>
<td>Declare Port B Address</td>
</tr>
</tbody>
</table>

MOVLW allout  ; Load W with DDC
TRIS portb    ; Set Port B as outputs
CLRF portb    ; Switch off LEDs
AGAIN          ; Increment output
GOTO AGAIN    ; Repeat endlessly

END            ; Terminate source code

Summary

- A block diagram can be used to outline the hardware, and the circuit designed from it.
- The PIC 16F84 program is stored in flash ROM, at addresses from 000. The instructions are decoded and executed by the processor control logic.
- The CPU registers are modified according to the program, and the sequence can be modified by the instructions.
- The PIC 16F84 has 14-bit instructions, containing both the operation code and operand.
- The program is written using assembler mnemonics and labels to represent the machine code instructions and operands.
- Layout and comments are used to document the program operation.
5. Give two examples of a PIC assembler directive. Why are they not represented in the machine code?

6. What are the numerical values of the labels ‘allout’ and ‘again’ in BIN2?

Answers

1. 0A86
2. 30
3. Jump destination
4. Labels go in first column
5. EQU, END
6. 00, 03

Activities

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, work out a program entirely in machine code. Modify the machine code program by deleting the ‘Clear Port B’ operation and changing the ‘Treated Port R’ to ‘Treated Port R’. What would be the effect at the output when the program was run?

2. Construct the circuit shown in Fig. 6.3 using a suitable hardware prototyping method. Refer to Section 12.3 for assistance. A ‘zero state’ must be set for the PIC chip before the machine code for BIN1 is downloaded into the microcontroller. Feed the outputs to a small loudspeaker with a 220R current-limiting resistor in series. Use an oscilloscope to measure the clock and output frequencies. Compare the relationship between the clock frequency and the output frequency. Increase the capacitor value to 220 nF, which should make the MSB flash at a visible rate. Predict the output frequency (Hint: the rate is proportional to the product of RC clock components).

3. Enter the program BIN2, using labels, into the text editor, assemble and test as above. Check that the machine code and function is identical to BIN1.

4. 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. See Table 7.4 for an assembler directive example.
Chapter 7

PIC Program Development

7.1 Program Design
7.2 Program Editing
7.3 Program Structure
7.4 Program Analysis
7.5 Program Assembly
7.6 Program Simulation
7.7 Program Downloading
7.8 Program Testing

We have seen how to start developing PIC application hardware and software, and can now 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 as described in Chapter 6.

This chapter will describe features of the standard PIC development system which is currently available, but hardware and software support to application developers are being continuously developed by Microchip and independent suppliers. The Internet provides ready access to the most recent information on the range of PIC chips and support software available at any given time. The manufacturer’s website can be found at www.microchip.com.

Since the available software tools are continuously updated, a definitive tutorial in using a particular version would soon be out of date. At the time of writing, MPLAB Version 6 is the most recently released version of the PIC IDE, but 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 in general terms. More details on debugging programs are provided in Chapter 11.

The flowchart in Fig. 7.1 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 analysed by the software designer so that the required program algorithm can be derived. This involves using various software design techniques, such as pseudocode and flowcharts, to outline the program processes and their sequence in a consistent way. These must then be converted to source code.
The program source code is developed from the program algorithm. By filling in the details and assigning values to each instruction, the source code is created. The program must be saved on disk as it is developed. It is a good idea to always save copies on different disks (fuzzy, hard disk or network drive) in case of disk failure. Also, keep copies on a network drive or on different disks – this always happens when you least expect it!

The source code text file is called PROGNAME.ASM, where PROGNAME is the application name, such as BIN1. Successive versions of a program can be numbered: BIN1, BIN2, etc.

The program can then be assembled by calling up the assembler utility, this is called MPASM. 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 re-assembled until it is free of syntax errors.

When the logical errors have been removed, the program can be downloaded to the chip and it should work first time, if the hardware has been correctly designed. Final testing can
Table 7.1

<table>
<thead>
<tr>
<th>Software tool</th>
<th>Tool function</th>
<th>Files produced or used</th>
<th>File description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text editor</td>
<td>Used to create and modify source code text file</td>
<td>PROGNAME.ASM</td>
<td>Source code text file</td>
</tr>
<tr>
<td>Assembler</td>
<td>Generates machine code from source code, reports syntax errors, generates list and symbol files</td>
<td>PROGNAME.HEX, PROGNAME.ERR, PROGNAME.LST, PROGNAME.COD</td>
<td>Executable machine code, error messages, list file with source and machine code, symbol and debug information</td>
</tr>
<tr>
<td>Simulator</td>
<td>Allows program to be tested in software before downloading</td>
<td>PROGNAME.HEX, PROGNAME.COD</td>
<td></td>
</tr>
<tr>
<td>Programmer</td>
<td>Downloads machine code to chip</td>
<td>PROGNAME.HEX, PROGNAME.COD</td>
<td></td>
</tr>
</tbody>
</table>

Then compare the finished circuit function with the specification, and, hopefully, no further debugging should be necessary at this stage. The main software tools and files created and used by MPLAB during the development process are listed in Table 7.1.

7.1 Program Design

These are national and company standards for specifying engineering designs which should be applied in commercial work. The design rules for different types of products will vary, but national standards will have been set and the design should be rigorously tested and documented than a commercial one. Our designs here are artificial in that they are intended to illustrate features of the PIC microcontroller rather than meet a user’s requirement. Nevertheless, we can follow the design process through the main steps.

7.1.1 Application Design

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 costs, and timescales can be predicted, as well as establishing the market or customer requirements. For our purposes, the minimal specification given in the text will suffice.

The next step is to design the hardware on which the application program will run. A block diagram which shows the user interface (input and output) requirements is a good starting point. The interfacing of the microcontroller is generally based on a limited number of standard devices, such as push buttons, keypad, LCD display, motor controllers, and so on. The circuit design techniques required will not be covered here, but it is essential that the demands of the microcontroller are within its specification. For example, the maximum current available at the standard PIC output is about 20 mA, sufficient to drive an LED, but not a relay.
The microcontroller itself must be selected by specifying the requirements such as:

- number of inputs and outputs
- program memory size
- data memory size (number of spare file registers)
- program execution speed
- other special interfaces (e.g. analogue inputs, serial ports)

The hardware configuration for the BINx applications has already been described in Chapter 6. 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 way, for example, not enough I/O pins, another microcontroller, or other types of hardware such as a conventional microprocessor system, must be considered. However, it is easier to stay within one family of processor and most manufacturers supply a range of chips, from which the most suitable can be selected.

7.1.2 Program Specification

The operational requirements of the application must be clearly specified in advance. In the commercial environment, a customer may do this, or if the application is more speculative a marketing survey may be undertaken. These requirements must be written in a way that lends itself to conversion into a software product using the language and tools available. Each programming language offers a different combination of features which must be matched to the user requirement as closely as possible. Similarly, the hardware system type must be selected to suit the application, before attempting the detailed circuit design. Choosing the most suitable microprocessor or microcontroller is clearly crucial. To make this choice, one needs a knowledge of the whole range of options. Chapter 14 provides a starting point for investigating and comparing different solutions.

For so-called embedded applications (the controller is built into the application circuit), the main choice of language is between selectable and a high level language (HLL) such as 'C'. 'C' allows such features as screen graphics, file handling and complex calculations to be more easily included in an application. For example, a maths function such as 'Sin x' is provided in 'C'; this would require a much more complex calculation in assembly language. On the other hand, assembly language code is generally faster and requires less memory. Of course, ultimately all languages are converted into machine code to run on the selected processor. The HLL requires more program memory because each statement is converted into several machine instructions. High level languages are normally used to develop applications for conventional microcontroller systems, especially using the Windows/Intel PC as a standard development platform. However, for less complex applications, a suitable microcontroller could be used, programmed in assembly code. The PC provides a convenient range of options, which offer a combination of different features, for example, built-in serial communication ports or analogue to digital converters (ADCs). The most powerful (XXXX series of PICs is typically programmed in 'C' (see Chapter 16).

7.1.3 Program Algorithm

The specification for the demonstration application is as follows. The circuit will 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 counter to remain from zero.

This BIN3 specification is much less demanding than would normally be the case for real software product. The frequency of operation of the output could, for instance, be specified. As the specification is not very specific, it should be easy to meet!

A flowchart is useful for clarifying the algorithm, particularly when learning, as it provides a pictorial representation. A flowchart for BIN3 is shown in Fig. 7.2. The program flow is placed in the start symbol at the top of the flowchart, and the process required 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), inputs and outputs (sloping) and decisions (pointed). The decision box has two outputs, to represent a conditional branch in the program. The decision box should contain a question with the correct yes or no, and the active selection labelled Yes or No as appropriate; only one needs to be labelled. The jump destinations are also labelled; these same labels will be used in the program as address labels. Software design techniques, including flowcharts, will be covered in more detail later.

![Flowchart for BIN3](image)

Figure 7.2 Flowchart for BIN3.

### 7.2 Program Editing

The program is written using the instructions set of the processor selected. This is provided with the hardware data sheet. A summary is provided in Table 7-2 of the data sheet. The source code, that is, the assembly code program, must be entered into a suitable text editor, usually the MPLAB edit window. 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 text editor has limited editing features, because it is not used for creating plain text files. The typeset “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 8
characters for the programs in this book.

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 created. Name the folder with the application
name, e.g. BIN3.ASM at the top of the file and immediately save it in the folder. This ensures that
the required files are checked for correct operation before any further source text is entered.
When saving on disk, there is a high risk of disk failure and possible operating system crash,
resulting in the loss of the edit file. Avoid disaster by keeping at least two copies of the source
code on different drives.

7.2.1 Instruction Set

Table 7.2 is a more user-friendly form of the PIC 16F84 instruction set organised by function.
Each example is given with each instruction so that the exact syntax can be seen. More detailed
information is provided in the data sheet. Other PIC chips have additional instructions, but they
all use the same basic set.

7.2.2 BIN3 Source Code

In program BIN3, the same instructions are used as in BIN2 (Chapter 6), with additional
statements to read the switches and control the output. Program 7.1 is the result.

Firstly, note the general layout and punctuation required. 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
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, toto, porta and portb refer to
the registers 05, 06 and 0C, respectively. 'label' and 'name' are legal labels representing
the parts between the program name 'BIN3.' and the semicolon, respectively. In order to
promote clarity, the instructions are placed in the first three columns, and all comments
are 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.

7.2.3 Syntax

'Syntax' refers to the way that words are put together to form 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, in our case, MPASM.EXE. The assembler
must be provided with source code which it can convert into the required machine code without
ambiguity, that is, only one meaning is possible. This is why the assembler syntax rules are
very strict.

7.2.4 Layout

The program layout should be in four columns, as described in Table 7.3. Each character
then occupies the same space, and the columns are correctly aligned. The label, command and
## PIC 16F84 INSTRUCTION SET BY FUNCTIONAL GROUPS

**F** = Any file register (specified by number or label), example is 0C

**W** = Working register

**L** = Literal value (follows instruction), example is 0F9

* = Use of these instructions not now recommended by manufacturer

### Operation

#### Move

- **Operation**: 
  - MOVF 0C,W
  - MOVWF 0C

#### Register

- **Operation**: 
  - CLRW
  - CLRF 0C
  - DECF 0C
  - INCF 0C
  - SWAPF 0C
  - COMF 0C
  - RLF 0C
  - RRF 0C
  - BCF 0C,3
  - BSF 0C,3

#### Arithmetic

- **Operation**: 
  - ADDWF 0C
  - ADDWF 0C,W
  - ADDLW 0F9
  - SUBWF 0C
  - SUBWF 0C,W
  - SUBLW 0F9

#### Logic

- **Operation**: 
  - ANDWF 0C
  - ANDWF 0C,W
  - ANDLW 0F9
  - IORWF 0C
  - IORWF 0C,W
  - IORLW 0F9
  - XORWF 0C
  - XORWF 0C,W
  - XORLW 0F9

#### Test and Skip

- **Operation**: 
  - BTFSC 0C,3
  - BTFSS 0C,3
  - DECFSZ 0C
  - INCFSZ 0C

#### Jump

- **Operation**: 
  - GOTO start
  - CALL delay
  - RETURN
  - RETLW 0F9
  - RETFIE

---

### Table 7.2 PIC 16F84 instruction set by functional groups

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td>MOVF 0C,W</td>
<td>PICF25</td>
</tr>
<tr>
<td>Move</td>
<td>MOVWF 0C</td>
<td>PICF25</td>
</tr>
<tr>
<td>Register</td>
<td>CLRW</td>
<td>PICF25</td>
</tr>
<tr>
<td>Register</td>
<td>CLRF 0C</td>
<td>PICF25</td>
</tr>
<tr>
<td>Register</td>
<td>DECF 0C</td>
<td>PICF25</td>
</tr>
<tr>
<td>Register</td>
<td>INCF 0C</td>
<td>PICF25</td>
</tr>
<tr>
<td>Register</td>
<td>SWAPF 0C</td>
<td>PICF25</td>
</tr>
<tr>
<td>Register</td>
<td>COMF 0C</td>
<td>PICF25</td>
</tr>
<tr>
<td>Register</td>
<td>RLF 0C</td>
<td>PICF25</td>
</tr>
<tr>
<td>Register</td>
<td>RRF 0C</td>
<td>PICF25</td>
</tr>
<tr>
<td>Register</td>
<td>BCF 0C,3</td>
<td>PICF25</td>
</tr>
<tr>
<td>Register</td>
<td>BSF 0C,3</td>
<td>PICF25</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>ADDWF 0C</td>
<td>PICF25</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>ADDWF 0C,W</td>
<td>PICF25</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>ADDLW 0F9</td>
<td>PICF25</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>SUBWF 0C</td>
<td>PICF25</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>SUBWF 0C,W</td>
<td>PICF25</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>SUBLW 0F9</td>
<td>PICF25</td>
</tr>
<tr>
<td>Logic</td>
<td>ANDWF 0C</td>
<td>PICF25</td>
</tr>
<tr>
<td>Logic</td>
<td>ANDWF 0C,W</td>
<td>PICF25</td>
</tr>
<tr>
<td>Logic</td>
<td>ANDLW 0F9</td>
<td>PICF25</td>
</tr>
<tr>
<td>Logic</td>
<td>IORWF 0C</td>
<td>PICF25</td>
</tr>
<tr>
<td>Logic</td>
<td>IORWF 0C,W</td>
<td>PICF25</td>
</tr>
<tr>
<td>Logic</td>
<td>IORLW 0F9</td>
<td>PICF25</td>
</tr>
<tr>
<td>Logic</td>
<td>XORWF 0C</td>
<td>PICF25</td>
</tr>
<tr>
<td>Logic</td>
<td>XORWF 0C,W</td>
<td>PICF25</td>
</tr>
<tr>
<td>Logic</td>
<td>XORLW 0F9</td>
<td>PICF25</td>
</tr>
<tr>
<td>Logic</td>
<td>BTFSC 0C,3</td>
<td>PICF25</td>
</tr>
<tr>
<td>Logic</td>
<td>BTFSS 0C,3</td>
<td>PICF25</td>
</tr>
<tr>
<td>Logic</td>
<td>DECFSZ 0C</td>
<td>PICF25</td>
</tr>
<tr>
<td>Logic</td>
<td>INCFSZ 0C</td>
<td>PICF25</td>
</tr>
<tr>
<td>Jump</td>
<td>GOTO start</td>
<td>PICF25</td>
</tr>
<tr>
<td>Jump</td>
<td>CALL delay</td>
<td>PICF25</td>
</tr>
<tr>
<td>Jump</td>
<td>RETURN</td>
<td>PICF25</td>
</tr>
<tr>
<td>Jump</td>
<td>RETLW 0F9</td>
<td>PICF25</td>
</tr>
<tr>
<td>Jump</td>
<td>RETFIE</td>
<td>PICF25</td>
</tr>
</tbody>
</table>
Control

SLEEP
Go into Standby Mode to save power

CLRWDT
Clear Watchdog Timer to prevent automatic reset

NOP
No Operation – delay for 1 cycle

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 00–4F. 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 but retained in the source register.

Program 7.1 BIN3 source code

; BIN3.ASM M. Bates 12-10-03
;...................................................................
; Slow output binary count is stopped, started
; and reset with push buttons.
; Processor = 16F84 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
;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

;Initialization Port B (Port B defaults to inputs).............
MOVWF portb ; Port B Data Direction Code
TSS portb ; Load the DDR code into F86
;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

;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

continued...
Table 7.3: Layout of assembler source code

<table>
<thead>
<tr>
<th>Label</th>
<th>Command</th>
<th>Operand(s)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

 operand columns should be set to a width of 8 characters, with the maximum label length of 6 characters, leaving a minimum of two clear spaces between columns. 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.

7.2.5 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. Only comments in blocks (sets of statements grouped together) can be used. Comments should be placed at the beginning of lines, or at the end of comments. The format and type of comments is processor dependent. Some comments may also be used within program blocks.

The table is normally used to place the text in columns, and the tab spacing can be adjusted if necessary.

A standard header block is recommended (see Program 7.1). 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 more complex programs, the processor type, hardware setup and other relevant information.
7.2.6 Creating a Project

MPLAB is designed to work using a named project to keep track of the application files created. A project file can be created which records information about the project such as the location of the application files and the window configuration used for testing. When the project is re-opened the windows re-appear as they were last set up. It is not essential to create a project for our simple applications, which can be assembled by selecting 'Quickbuild' (Ver. 6) or 'Build Node' (Ver. 5) in the 'Build Project' menu. Earlier versions of MPLAB required it, and it will be needed for more complex applications.

7.3 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.

7.3.1 BIN4 Source Code

Program BIN3 (Chapter 6) 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 7.3.

The main difference between BIN3 and BIN4 is that the program now has the delay code 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 re-used in the program as necessary. It can be called as many times as required, which means that the block of code only needs to be written once. It can also be saved as a separate file and re-used in another program.

A program flowchart has been given for BIN3 (Fig. 7.2). The same flowchart describes BIN4 but the delay routine can now be expanded as a separate subroutine (Fig. 7.3). In addition, the delay time is loaded prior to the subroutine execution, so the same delay routine could be used to provide different delay times. The use of flowcharts in program design will be more fully examined in Chapter 10.

7.4 Program Analysis

The program BIN4 will now be analysed in some detail as it was designed to contain examples of most of the basic PIC syntax. A sample instruction is given in each case.

7.4.1 Label Equates

\texttt{timer \textbf{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 assembler directive \\texttt{EQU} is used to assign a label to a number, which can be a literal, file register number or 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 software counter. 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.
PIC Program Development

Program 7.2 BIN source code

; 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 16F84
Hardware: BIN Demo System
Clock: CR ~100kHz
Inputs: Push Buttons RA0, RA1 (active low)
Outputs: LEDs (active high)

***************************************************************************
Register Label Equates..............................................
porta EQU 05 ; Port A Data Register
portb EQU 06 ; Port B Data Register
timer EQU 0C ; Spare register for delay
***************************************************************************
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 Analysis

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

Figure 7.3 Flowcharts for program BIN4. (a) Main routine; (b) Subroutine.
7.4.2 Port Initialisation

TRIS Port B

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 as hex: 00000000. This is useful, especially if the port bits are to be used as a mixture of inputs and outputs; the binary code identifies the data direction for each bit individually. The code is loaded into W using MOVLW, and the TRIS command follows.

The TRIS instruction is still available as a simple way of initialising 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 by the HPASM assembler, as it is easier for beginners.

7.4.3 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 it has been pressed.

7.4.4 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' is therefore 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 button is not pressed, the program waits by jumping back to the 'start' line. The combined effect is that the count runs when the 'run' button is pressed, and the count is reset to zero if the 'reset' button is pressed.

7.4.5 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. 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.

7.4.6 Subroutine Call and Return

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

```
main ...... ; start main program
......
CALL delay ; jump to subroutine
...... ; return to here
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 after the subroutine', so the return address has to be stored on a stack called a '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 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.

7.4.7 End of Source Code

```
END
```

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

7.5 Program Assembly

The assembler program (MPASM) takes the source code text anddecodes it character by character, line by line, starting at the top left. In MPLAB, the correct processor type must first be selected via the configuration menu, as there is some variation in valid syntax between processors. Then, in the project menu, select the option to assemble a single file. It is not necessary, at this stage, to create a project.

When the assembler runs, the corresponding 14-bit binary machine code for each line in the source code is generated until the END directive is detected. The binary code is saved automatically as a file called BIN4.HEX in the same folder as the source code.
7.5.1 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. You must note the messages and line numbers, or print out the error file, BIN4.ERR. Then go back and re-edit the source code and make the necessary changes. The error is sometimes on a previous line to the one indicated, and sometimes one error can generate more than one message. Warning and information messages can usually be ignored. There is more detail about error messages in Chapter 11.

You may receive the following messages:

Warning C:\MPLAB\BOOKPRGS\BIN4.ASM 65: Use of this instruction is not recommended.
Message C:\MPLAB\BOOKPRGS\BIN4.ASM 81: Using default destination of 1 (file).

The first warning will be caused by using the instruction TRIS, which the manufacturer warns may not be supported in future. OPTION is also not recommended. However, it is used here because it simplifies the initialization of the ports. 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 destination in instructions where the result can be placed either in the file register or in the working register (see Section 9.4.1). The assembler assumes that the file register is the destination by default.

When all errors have been eliminated, and the program successfully assembled, the machine code can be inspected by viewing program memory. Note that the source code labels are not reproduced, as the program code has been ‘disassembled’ 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.

7.5.2 List File

A program ‘list file’ BIN4.LST is also produced by the assembler, which contains the source code, the machine code, error messages and other information (see listing Table 7.4). This is useful for analysing the program and assembler operations, and debugging the source code.

The list file headers show the assembler version used and source file details. The column headings are given:

- **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 there is no machine code produced by the lines which are occupied by a full line comment. The actual program starts to be produced at line 00040. The machine code for the first
Table 7.4 BIN4 list file

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>LINE</th>
<th>SOURCE TEXT</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>00001</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00002</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00003</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00004</td>
<td>Processor:</td>
<td>PIC14F16C10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00005</td>
<td>Hardware:</td>
<td>PIC14F16C10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00006</td>
<td>Clock:</td>
<td>CR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00007</td>
<td>Inputs:</td>
<td>RA0, RA1 (active low)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00008</td>
<td>Outputs:</td>
<td>LEDA (active high)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00009</td>
<td>Subroutines:</td>
<td>HILAS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00010</td>
<td>Register Label Equates</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00011</td>
<td>porta EQU 05</td>
<td>Port A Data Register</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00012</td>
<td>portb EQU 06</td>
<td>Port B Data Register</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00013</td>
<td>timer EQU 0C</td>
<td>Spare register for delay</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00014</td>
<td>inres EQU 0 ; 'Reset' input button RA0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00015</td>
<td>inrun EQU 1 ; 'Run' input button RA1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>00016</td>
<td>Defines DELAY subroutine</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00017</td>
<td>delay MOVLW b'00000000'</td>
<td>Delay count literal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00018</td>
<td>MOVWF timer</td>
<td>is loaded into timer register</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00019</td>
<td>reset GOTO</td>
<td>location of start of main program</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00020</td>
<td>LOADER</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00021</td>
<td>Part 3</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00022</td>
<td>Part 4</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00023</td>
<td>Part 5</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00024</td>
<td>Part 6</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00025</td>
<td>Part 7</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00026</td>
<td>Part 8</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00027</td>
<td>Part 9</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00028</td>
<td>Part 10</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00029</td>
<td>Part 11</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00030</td>
<td>Part 12</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00031</td>
<td>Part 13</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00032</td>
<td>Part 14</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00033</td>
<td>Part 15</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00034</td>
<td>Part 16</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00035</td>
<td>Part 17</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00036</td>
<td>Part 18</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00037</td>
<td>Part 19</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00038</td>
<td>Part 20</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00039</td>
<td>Part 21</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00040</td>
<td>Part 22</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00041</td>
<td>Part 23</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00042</td>
<td>Part 24</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00043</td>
<td>Part 25</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00044</td>
<td>Part 26</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00045</td>
<td>Part 27</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00046</td>
<td>Part 28</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00047</td>
<td>Part 29</td>
<td>.................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00048</td>
<td>Part 30</td>
<td>.................</td>
</tr>
</tbody>
</table>

continued...
Table 7.4
PIC Program Development

0005 0B8C 0005 down DECFSZ timer ; Decrement timer register
0006 2805 00051 GOTO down ; and repeat until zero then
0007 0008 00052 RETURN ; return to main program

00053
00054
00055 ; Start main loop ................................
00056
00057
0008 0186 00058 reset CLRF portb ; Clear Port B Data
00059
0009 1C05 00060 start BTFSS porta,inres ; Test RA0 input button
00061 GOTO reset ; and reset Port B
00062
000B 1885 00063 BTFSC porta,inrun ; Test RA1 input button
00064 GOTO start ; and run count if pressed
00065
000D 0A86 00067 INCF portb ; Increment count at Port B
00068 CALL delay ; Execute delay subroutine
00069 GOTO start ; Repeat main loop
00070
00071
00072 END ; Terminate source code

SYMBOL TABLE

<table>
<thead>
<tr>
<th>LABEL</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>__16C84</td>
<td>00000001</td>
</tr>
<tr>
<td>delay</td>
<td>00000003</td>
</tr>
<tr>
<td>down</td>
<td>00000005</td>
</tr>
<tr>
<td>inres</td>
<td>00000000</td>
</tr>
<tr>
<td>inrun</td>
<td>00000001</td>
</tr>
<tr>
<td>porta</td>
<td>00000005</td>
</tr>
<tr>
<td>portb</td>
<td>00000006</td>
</tr>
<tr>
<td>reset</td>
<td>00000008</td>
</tr>
<tr>
<td>start</td>
<td>00000009</td>
</tr>
<tr>
<td>timer</td>
<td>0000000C</td>
</tr>
</tbody>
</table>

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

0000 : XXXXXXXXXXXXXXXX--------- --------- ---------
0 0 4 0:------------ --------- --------- ---------
All other memory blocks unused.

Errors : 0
Warnings : 0
Messages : 2
If we study the machine code, we can see how the labelling works; for example, the last instruction 'GOTO start' is encoded as 2809, and the 09 refers to address 09 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 symbol table also contains the addresses of the variables and constants used in the program.

The amount of program memory used, 16 locations (0000–000F), 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.

7.6 Program Simulation

The BIN4.HEX file can be downloaded to the PIC chip and executed; it would run correctly, because the program given here has already been tested. However, when a program is first developed, it is quite likely that 'logical' errors will be found. Logical errors prevent the program from working correctly; that is, the program executes but it does not necessarily carry out the right operations in the right order.

If this is so, the source code must be analysed again to try to find the errors. In compiling programs, the process might take a long time, especially if the program is large or complex. In this situation, a software simulator can be used to 'run' the program on the host PC, as if it were being executed in the chip, but without having to interface with the actual hardware. It can thus be checked for logical errors and the program can be debugged more quickly and easily.

Using the simulator MPLAB SIM (Fig. 7.4), the program can be run and stopped at will; this allows the effect of the program on the registers 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.

Suppose that in developing BIN4 we had failed to analyse the mixing logic correctly, and that the program had been compiled. If the ‘inres’ button was pressed, the program would jump back to the ‘start’ line instead of going to the ‘second’ line (‘sometime’), to allow the output response to be started. This fault could be detected in the simulator by running the program, stopping at the start sequence and simulating the switch inputs.

To test BIN4 code, proceed as follows: Open MPLAB SIM and make sure the correct program is selected. In the simulation interface, the ‘Step’ button is highlighted. Press the ‘Step’ button; the program will execute one instruction at a time. Press the ‘Step’ button again, the program will execute two instructions. This process can be repeated until the program is stopped at the desired point.
Figure 7.4 MPLAB simulation windows for debugging program BIN4.

The program can be executed one instruction at a time using the 'Step Into' button, and the sequence examined. The program should loop through the reset sequence. The program can be restarted from the top in any time by clicking on the 'Reset' button.

7.6.2 Input Simulation

We now need to simulate the action of the push buttons in the hardware which are used to start and stop the output sequence. The simplest way to do this is to use 'asynchronous input' found under the debugger stimulus menu. This displays a set of screen buttons which can be assigned to any input. Assign a button to RA0 and RA1, and set to toggle mode; the toggle option will make the input change over each time the button is pressed.

The inputs can now be toggled to allow the program to proceed from the reset loop. Set both inputs high initially, taking RA0 low will allow the main loop to proceed and operating RA1 will execute the reset loop. Unfortunately, the state of the input is not indicated in the simulator stimulus window, so file register 05, Port A, must be displayed preferably in binary to monitor changes on the inputs.

7.6.3 Register Display

View the file registers, or special function registers, 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
7.6.4 Step Over

Once the program has entered the delay loop, single stepping is not useful, because the program is simply executing the same simple sequence over and over. A shortcut around the delay subroutine is needed. One way to do this is to use the “Jump Here” button, which sets a breakpoint at the end of the delay subroutine (the line labeled “start”). When the program executes to this line, it stops and the single stepping mode is re-entered.

7.6.5 Breakpoints

Another technique for executing some parts of the program at full speed is the use of breakpoints. For example, if part of a large program can be checked, it can be stepped and single stepping started at some point in the program. In BIN4, a breakpoint can be set at the start of the main loop by right-clicking with the mouse over the line labeled “start,” and selecting “Set Breakpoint” from the menu. This places a breakpoint at the start, and it will stop at the breakpoint. Run again, and a complete loop will be completed at full speed and Port B should increment.

7.6.6 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 processor clock frequency. For BIN4, a CR clock is assumed, operating at 100 kHz. The processor frequency must be set to this value in the simulator. Then run the program to the break point at “start,” stop the clock, and run again. The stopwatch will display the time elapsed for two clock cycles.

Example:

The program contains two main loop cycles with same the two output for RB0 to be toggled up and down once, giving one full output cycle. Therefore, we can double the loop time to get the output period, and calculate the reciprocal to give the frequency at RB0. The period at RB7 will be 128 times longer, and the frequency 128 times higher.

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 × 31.08 ms = 62.16 ms
- Output period at RB7 = 0.06216 × 128 = 7.96 s
- Output frequency at RB0 = 1/0.06216 ≈ 16.1 Hz

This shows that changing the higher order output bits will be clearly visible using this clock frequency at the maximum delay loop count (FF). The frequency at RB1 will be about 8 Hz, RB2 4 Hz, RB3 2 Hz and RB4 1 Hz and so on.

More information about using MPLAB for debugging is given in Chapter 11.
7.7 Program Downloading

After testing in the simulator for correct operation, the machine code program can now be blown into the Flash ROM on the chip. The program is downloaded via a serial or parallel link into RB6 (clock) and RB7 (data) on the chip while a programming voltage of about 14 V is applied to MCLR. There are two methods for program downloading, outlined below.

7.7.1 Programming Unit

In this case, a programming unit must be plugged into the serial port of the PC (COM1 or COM2), power connected and the chip inserted into the socket on the programmer. This is usually a ZIF socket which is opened to allow the IC to be dropped in, and closed to clamp the pins (Fig. 7.5). The chip orientation must be carefully checked, as inversion could probably damage the chip, as the supplies will be reversed. Anti-static precautions should be observed since the PIC is a CMOS device (prone to static damage and latchup events). However, the device has not been found to be overly sensitive.

The programming unit is enabled from the main menu bar. If the programming unit has been correctly connected, the programming dialogue should open with the hex code to be downloaded also visible. If not, the COM port needs to be changed.

Before downloading, the chip configuration options need to be selected (Fig. 7.6).
The main options here are 'RC' and 'XT'. 'RC' must be selected for the oscillator configuration used in the BIN hardware. 'XT' will be used later for crystal-oscillator-clocked applications. The program will not run in the hardware if the wrong type of oscillator has been selected, so check this carefully.

Watchdog Timer (WDT): Off

The watchdog timer 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 getting 'stuck' in a loop, due to an undetected program bug, or an input condition which 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 (PuT): On

Mains-derived power supplies may take some time to reach the correct value (+5 V) when first switched on. The power up timer is an internal timer which delays the start of program execution until the power supply is at the correct voltage and stable. This helps to ensure that the program starts correctly every time. Always enable the power up timer unless there is a good reason not to do so.
114 PIC Program Development

11.4 Code Protection

Code Protect (CP): Off

If this 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; no code protection normally is needed off.

When the configuration bits have been set, click on the program button, and the progress of the program download should be displayed. When complete, the chip can be removed to the application circuit.

7.7.2 In-Circuit Programming and Debugging

The PIC chip can be alternatively be programmed while fitted in the application circuit. This is very useful to minimise the risk of damage, allows the chip to be programmed with different software versions after completion of the hardware, and even allows remote programming via a communication link. All that is needed is an on-board connection to RB6, RB7 and MCLR.

In addition, this connection allows in-circuit debugging (ICD), which is a very useful feature for in-circuit testing. ICD is also a very powerful way to test the application development of the PIC, as it enables the connection to pin MCLR and RB6 of the chip to the application circuit. The program can then be downloaded.

If the chip is set to run in ICD mode, the debugging features available MPLAB (single step, breakpoints, etc.) can be used to control and monitor program execution in the chip itself, rather than in a purely software simulation. This has the great advantage that the interaction with the real hardware can be monitored, and hence the hardware and software verified at the same time. See Chapter 14 for more details.

7.8 Program Testing

The application circuit should be checked for correct on-board connections before inserting the microcontroller and other active devices (this is easy if the chips are in sockets). Static hardware testing comprises of a correctly constructed circuit, and overall it is strongly advisable. Where the hardware has been thoroughly checked, verify the microcontroller (running correct control system) and program.

Prior to testing, the application circuit should be thoroughly tested and correct operations tested and confirmed and recorded. The test procedure should check all possible input sequences, not just the common ones of the design to be tested. It is not unusual to find that complex programs will behave surprisingly if not written correctly. All output connections must be checked and recorded, and input operations tested and correct.

The program should be tested immediately in the circuit. If there are unexpected results, a detailed test report may need to be prepared, as outlined below.

1. Testing Equipment

(a) +5 V on MCLR, Vdd, Vss,
(b) clock signal on CLKIN,
(c) input changes on RB0, RB1.

2. Software Check

(a) software test run,
(b) output correct against the original specification.

(b) software test run,
(c) output correct against the original specification.
Table 7.5  Basic test schedule for BIN4

<table>
<thead>
<tr>
<th>Test Procedure</th>
<th>Correct Operation</th>
<th>Checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check PIC connections</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Power up</td>
<td>LEDs off</td>
<td></td>
</tr>
<tr>
<td>Clock frequency</td>
<td>100 kHz</td>
<td></td>
</tr>
<tr>
<td>Press RUN</td>
<td>Count on LEDs</td>
<td></td>
</tr>
<tr>
<td>Release RUN</td>
<td>LEDs halted</td>
<td></td>
</tr>
<tr>
<td>Press and release RESET</td>
<td>LEDs off</td>
<td></td>
</tr>
<tr>
<td>Power OFF</td>
<td>LEDs on LED from now</td>
<td></td>
</tr>
</tbody>
</table>

(c) WDT off, PuT on, CP off,
(d) program verified.

More suggestions on hardware and software testing are given in Chapter 11.

Summary

- The development process consists of application specification, hardware selection and design, and program development and testing.
- The program is converted from a software design to assembler source code, FILENAME.ASM, using the instruction format defined for the assembler.
- The assembler converts the source code into object code, FILENAME.HEX. Any syntax errors detected must be corrected.
- 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 can be detected and corrected at this stage.
- The program can then be downloaded and tested in the target hardware, using a test schedule developed from the specification.

Questions

1. Place the following program development steps in the correct order: Test in Hardware, Simulate, Assemble, Edit Source Code, Download.
2. Suggest two advantages of using 'C' as the programming language.
3. State two advantages of using subroutines.
4. State the instruction for incrementing the register 0F.

5. In which register must a port data direction code be placed prior to using the TRIS instruction?

6. Explain how a switched input on RA4 of the PIC 16F84 is simulated in MPLAB.

7. Explain how a switched input on RA4 of the PIC 16F84 is simulated in MPLAB.

8. Explain how a switched input on RA4 of the PIC 16F84 is simulated in MPLAB.

Answers

4. INCF 0F

5. W

6. Delay Count = 80h

Activities

1. Download from www.microchip.com or otherwise obtain the supporting documentation for MPLAB. Study the tutorial in the User’s Guide and the help files supplied with MPLAB as necessary to familiarise 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 any comments. Assemble, correct any errors and simulate. Check that the Port B (F6) file register operates as required.

2. Construct a prototype circuit and test the program to the test schedule given in Table 7.5. Refer forward to Chapter 12 if necessary.

3. Modify the program as BIN4 and confirm that its operation is essentially the same.

4. Modify the program to scan the output, that is, move one lit LED up and down the display, repeating indefinitely. Use the rotate instructions, and subtraction from 1 and 80h to check if the test is correct.
Chapter 8
PIC 16F84 Architecture

8.1 Block Diagram

A somewhat simplified internal architecture (Fig. 8.1) 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 as block arrows. The 8-bit data paths are shown in an alternative style as single arrows in this diagram. The timing and control block has control connections to all other blocks, which set up the processor operation, 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 PR0, PR1, PR2, and PR3. Some registers (EXCLKS), and clock sources (MDT and INTCNT) are not shown. There are also a number of special purpose registers (SFRs) which can be used as data registers, counters, and so on. The file registers are numbered 00–4F, but are usually given suitable labels in the program source code. File registers also give access to a block of EEPROM, a non-volatile data memory.

8.2 Program Execution

8.3 Register Set

An overview of programming the PIC microcontroller has been provided in Chapter 7; we can now look at the PIC's internal hardware arrangement in more detail. We will use the 16F84 as a reference, since it has all the essential elements, without some of the more advanced features such as analogue inputs and serial ports found on larger chips such as the 16F877, which we will look at later. The other members of the PIC family are based on the same architecture, with elements added, removed or modified according to the combination of features provided in each chip.

The key reference is Fig. 1.1 in the PIC 16F84A Data Sheet, the 'PIC 16F84A Block Diagram.' The data sheet contains all the details of the internal structure discussed in this chapter. Refer back to Chapter 4 for a description of the function of elements such as registers, ALU, multipliter, decoder, program counter and memory.
8.1.1 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. Typically, relatively low frequencies are used (<1 MHz) with an RC clock. For more precise timing, a crystal oscillator is used (see Fig. 8.1), 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 maximum 20 μs instruction execution period, and a maximum instruction execution rate of 50 MIPs (millions of instructions per second).

This timing and control circuit also contains start up timers which means that the reset input MCLR can simply be connected to Vdd, the positive supply, to enable the processor. In earlier processors, an external reset circuit was often needed to ensure a smooth start-up. An external reset button or control signal can still be connected to MCLR if required. The controller program can then be restarted by pulsing the reset input low.
3.2 Program Execution

The program consists of a block of 16-bit codes which contain both operation code and operand in a fixed-length instruction. This machine code program is normally created using a host PC, and downloaded, as outlined in Chapter 7. We are not too concerned here 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 the I/O port pin, RB7, and written to the program memory while a programming voltage applied to !MCLR. RB6 acts as a serial data clock.

3.2.1 Program Memory

Since the program memory is flash ROM, an existing program can be replaced by simply overwriting with a new program. Up to 1024 (1k) instructions can be stored in program memory.

The program counter (PCL) holds the current address, and is reset to 0000 when the chip is powered up or reset.

The user program must therefore start at address 0000, but the first instruction can be GOTO the start of the program at some other labelled address. This is essential when using interrupts, as we shall see later, because the interrupt service routine (or GOTO ISR) must be placed at address 0004. The address is fed to the program memory via a 13-bit address bus. As the 16F84 memory only contains 1k locations, the actual address required is only 10 bits, but a standard 13-bit address is used to maintain compatibility with other PIC processors with more program memory, and other future products.

3.2.2 Instruction Execution

This section contains the instruction register, instruction decoder, and timing and control logic. The 16-bit instructions stored in program memory are copied to the instruction register for decoding. The instruction decoder logic converts the instruction input into settings for all the internal control lines. The 16-bit instruction contains both the operation code and operand. The instruction decoder will only use the operation code bits, while the operand provides a literal, file register address or program address when the instruction is executed.

For example, the instruction MOVLW D16 (Move a Literal into W) will set up the contents of W to be the literal operand D16. If the instruction is MOVWF, the contents of W will be moved to the specified file register. The operand will be the address of the file register (00–4F) required. If we look at the 'move' instructions for W, we can see that the contents of W can be moved to any of the 16F84's 40 file registers. Each file register has a unique address in the 00–4F range.
PIC 16F84 Architecture

Instruction codes quoted in the Data Sheet, Table 7-2, we can see the difference in the code structure:

- **MOVLW k**: 
  ```
  MOVLW k = 11 00xx kkkk kkkk
  ```

- **MOVWF f**: 
  ```
  MOVWF f = 00 0000 1fff ffff
  ```

- **MOVF f,d**: 
  ```
  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) and 'f' indicates the file register address bits. Only 7 bits are used for the register address, allowing a maximum of 2^7 = 128 registers to be addressed. In fact, the 16F84 has only 80 registers in all, but 7 bits are still needed to address these.

In the MOVW instruction the operation code is (0000), and the file register address is needed only to identify the data source register. However, there is an 'f' which 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.

**8.2.3 Data Processing**

The ALU can add, subtract or carry out logical operations on single data bytes or pairs of numbers. 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 the temporary store. Final results are stored back in the file register.

**8.2.4 Jump Instructions**

If a GOTO instruction is decoded, 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 unconditional branching, any file register bit can be used by the 'Bit Test and Skip' instruction, which is then followed by a GOTO or CALL instruction.

If a CALL instruction is decoded, the destination address is loaded into the PC in the same way as for the GOTO, but in addition, the next address to the CALL is stored onto the stack (the return address). The subroutine is then executed until a RETURN instruction is encountered. The return address is then pulled from the stack and placed 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 typically located in main memory, but in the PIC this is not the case – stack operation is entirely automatic.

**8.3 Register Set**

All the file registers are 8-bit wide. They are divided into two main blocks – SFRs, which are reserved for specific purposes, and GPRs which can be used for temporary storage of any data byte. The file register set is shown in Fig. 8.2 in numerical order.
## Register Set

### Address Page 0 Page 1 Address

<table>
<thead>
<tr>
<th>Address</th>
<th>Page 0</th>
<th>Page 1</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>IND0</td>
<td></td>
<td>80</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>01</td>
<td>81</td>
</tr>
<tr>
<td>2</td>
<td>TMR0</td>
<td>OPTION</td>
<td>82</td>
</tr>
<tr>
<td>3</td>
<td>PCL</td>
<td></td>
<td>83</td>
</tr>
<tr>
<td>4</td>
<td>STATUS</td>
<td></td>
<td>84</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>05</td>
<td>85</td>
</tr>
<tr>
<td>6</td>
<td>PORTA</td>
<td>TRISA</td>
<td>86</td>
</tr>
<tr>
<td>7</td>
<td>TRISB</td>
<td></td>
<td>87</td>
</tr>
<tr>
<td>8</td>
<td>PORTB</td>
<td></td>
<td>88</td>
</tr>
<tr>
<td>9</td>
<td>EEDATA</td>
<td>EECON1</td>
<td>89</td>
</tr>
<tr>
<td>10</td>
<td>EEADR</td>
<td></td>
<td>90</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>0A</td>
<td>91</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>0B</td>
<td>92</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>0C</td>
<td>93</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>0D</td>
<td>94</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>0E</td>
<td>95</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>0F</td>
<td>96</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>10</td>
<td>97</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>11</td>
<td>98</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>12</td>
<td>99</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>13</td>
<td>9A</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>14</td>
<td>9B</td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>15</td>
<td>9C</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>16</td>
<td>9D</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>17</td>
<td>9E</td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>18</td>
<td>9F</td>
</tr>
<tr>
<td>26</td>
<td></td>
<td>19</td>
<td>A0</td>
</tr>
<tr>
<td>27</td>
<td></td>
<td>1A</td>
<td>A1</td>
</tr>
<tr>
<td>28</td>
<td></td>
<td>1B</td>
<td>A2</td>
</tr>
<tr>
<td>29</td>
<td></td>
<td>1C</td>
<td>A3</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td>1D</td>
<td>A4</td>
</tr>
<tr>
<td>31</td>
<td></td>
<td>1E</td>
<td>A5</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>1F</td>
<td>A6</td>
</tr>
<tr>
<td>33</td>
<td></td>
<td>20</td>
<td>A7</td>
</tr>
<tr>
<td>34</td>
<td></td>
<td>21</td>
<td>A8</td>
</tr>
<tr>
<td>35</td>
<td></td>
<td>22</td>
<td>A9</td>
</tr>
<tr>
<td>36</td>
<td></td>
<td>23</td>
<td>AA</td>
</tr>
<tr>
<td>37</td>
<td></td>
<td>24</td>
<td>AB</td>
</tr>
<tr>
<td>38</td>
<td></td>
<td>25</td>
<td>AC</td>
</tr>
<tr>
<td>39</td>
<td></td>
<td>26</td>
<td>AD</td>
</tr>
<tr>
<td>40</td>
<td></td>
<td>27</td>
<td>AE</td>
</tr>
<tr>
<td>41</td>
<td></td>
<td>28</td>
<td>AF</td>
</tr>
<tr>
<td>42</td>
<td></td>
<td>29</td>
<td>B0</td>
</tr>
<tr>
<td>43</td>
<td></td>
<td>2A</td>
<td>B1</td>
</tr>
<tr>
<td>44</td>
<td></td>
<td>2B</td>
<td>B2</td>
</tr>
<tr>
<td>45</td>
<td></td>
<td>2C</td>
<td>B3</td>
</tr>
<tr>
<td>46</td>
<td></td>
<td>2D</td>
<td>B4</td>
</tr>
<tr>
<td>47</td>
<td></td>
<td>2E</td>
<td>B5</td>
</tr>
<tr>
<td>48</td>
<td></td>
<td>2F</td>
<td>B6</td>
</tr>
<tr>
<td>49</td>
<td></td>
<td>30</td>
<td>B7</td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>31</td>
<td>B8</td>
</tr>
<tr>
<td>51</td>
<td></td>
<td>32</td>
<td>B9</td>
</tr>
<tr>
<td>52</td>
<td></td>
<td>33</td>
<td>BA</td>
</tr>
<tr>
<td>53</td>
<td></td>
<td>34</td>
<td>BB</td>
</tr>
<tr>
<td>54</td>
<td></td>
<td>35</td>
<td>BC</td>
</tr>
<tr>
<td>55</td>
<td></td>
<td>36</td>
<td>BD</td>
</tr>
<tr>
<td>56</td>
<td></td>
<td>37</td>
<td>BE</td>
</tr>
<tr>
<td>57</td>
<td></td>
<td>38</td>
<td>BF</td>
</tr>
<tr>
<td>58</td>
<td></td>
<td>39</td>
<td>C0</td>
</tr>
<tr>
<td>59</td>
<td></td>
<td>3A</td>
<td>C1</td>
</tr>
<tr>
<td>60</td>
<td></td>
<td>3B</td>
<td>C2</td>
</tr>
<tr>
<td>61</td>
<td></td>
<td>3C</td>
<td>C3</td>
</tr>
<tr>
<td>62</td>
<td></td>
<td>3D</td>
<td>C4</td>
</tr>
<tr>
<td>63</td>
<td></td>
<td>3E</td>
<td>C5</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td>3F</td>
<td>C6</td>
</tr>
<tr>
<td>65</td>
<td></td>
<td>40</td>
<td>C7</td>
</tr>
<tr>
<td>66</td>
<td></td>
<td>41</td>
<td>C8</td>
</tr>
<tr>
<td>67</td>
<td></td>
<td>42</td>
<td>C9</td>
</tr>
<tr>
<td>68</td>
<td></td>
<td>43</td>
<td>CA</td>
</tr>
<tr>
<td>69</td>
<td></td>
<td>44</td>
<td>CB</td>
</tr>
<tr>
<td>70</td>
<td></td>
<td>45</td>
<td>CC</td>
</tr>
<tr>
<td>71</td>
<td></td>
<td>46</td>
<td>CD</td>
</tr>
<tr>
<td>72</td>
<td></td>
<td>47</td>
<td>CE</td>
</tr>
<tr>
<td>73</td>
<td></td>
<td>48</td>
<td>CF</td>
</tr>
<tr>
<td>74</td>
<td></td>
<td>49</td>
<td>D0</td>
</tr>
<tr>
<td>75</td>
<td></td>
<td>4A</td>
<td>D1</td>
</tr>
<tr>
<td>76</td>
<td></td>
<td>4B</td>
<td>D2</td>
</tr>
<tr>
<td>77</td>
<td></td>
<td>4C</td>
<td>D3</td>
</tr>
<tr>
<td>78</td>
<td></td>
<td>4D</td>
<td>D4</td>
</tr>
<tr>
<td>79</td>
<td></td>
<td>4E</td>
<td>D5</td>
</tr>
<tr>
<td>80</td>
<td></td>
<td>4F</td>
<td>D6</td>
</tr>
</tbody>
</table>

### Special Function Registers

The operation of the SFRs is summarised below, with the emphasis on those which are used most frequently. The functions of all the registers are detailed in the chip data sheet. The shaded registers in Fig. 8.2 either do not exist, or are repeated at addresses 80–CF (Page 1).

#### File Register Numbers

- **PCL**: Program Counter Low Byte
- **PORTA**: Port A Data Register
- **PORTB**: Port B Data Register
- **PORTC**: Port C Data Register
- **PORTD**: Port D Data Register
- **PORTE**: Port E Data Register
- **PORTF**: Port F Data Register
- **PCLATH**: Program Counter High Byte
- **STATUS**: Status Register
- **OPTION**: Option Register
- **TRISA**: TRIS A Data Direction Register
- **TRISB**: TRIS B Data Direction Register
- **TRISC**: TRIS C Data Direction Register
- **TRISD**: TRIS D Data Direction Register
- **TRISE**: TRIS E Data Direction Register
- **TRISF**: TRIS F Data Direction Register

### Figure 8.3 PIC 16F84 File Register Set

The registers in Page 0 (file addresses 00–4F) can be directly addressed, and it is recommended that the register labels given in Fig. 8.2 which match the data sheet, are used in the register tables. These labels are also used by default in MPLAB. Standard header files can be included in your programs which define all the register names.

Special instructions are available to access the Page 1 registers. We have already used the instruction TRIS to access the data direction registers TRISA and TRISB. In a similar way, we will use the instruction OPTION to access the option register; this will be used later to set up the hardware timer, TMR0. Alternatively, a register bank select bit in the status register can be used to access Page 1 file registers; this method is recommended for more advanced programming (see Section 9.4.2). Note that Page 1 means the same as Bank 1.

### 8.3.1 Special Function Registers

The operation of the SFRs is summarised below, with the emphasis on those which are used most frequently. The functions of all the registers are detailed in the chip data sheet. The shaded registers in Fig. 8.2 either do not exist, or are repeated at addresses 80–CF (Page 1).

#### Program Counter Low Byte
- **PCL**: Program Counter Low Byte

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 2 bits (00–03) 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 entirely for a jump.

PORTA Port A Data Register

Port A has 140 bits. Before use, the data direction for each pin must be set up by loading the TRIS register with a data direction code (see below). A bit 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 16F84, RA0 can alternatively be used as an input to the counter timer registers (TMRO) for counting applications. The use of the hardware timer will be covered in Chapter 9. The PORTA register bit allocation is shown in Table 8.1. In other PIC chips, most port pins will have at least two uses, depending on the register settings.

Port A has 5 I/O bits, RA0–RA4. Before use, the data direction for each pin must be set up by loading the TRIS register with a data direction code. If a bit is set to output, data moved to this register appears at the output pin 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 16F84, RA0 can alternatively be used as an input to the counter timer register (TMRO) for counting applications. The use of the hardware timer will be covered in Chapter 9. The PORTA register bit allocation is shown in Table 8.1. In other PIC chips, most port pins will have at least two uses, depending on the register settings.

All registers are read and written as 8-bit words, so we sometimes need to know what will happen to unused bits. When the Port A data register is read within a program (MOVF), the 3 unused bits will be seen as '0'. When writing to the port, the high 3 bits are simply ignored.

When used as outputs, the port lines are able to provide up to 20 mA of current (except RA4), which is enough to drive our LEDs in the demonstration circuits. An equivalent circuit for each port pin is given in the data sheet, Section 5.

Table 8.1 PIC 16F84 port bit locations

<table>
<thead>
<tr>
<th>Register bit</th>
<th>Chip pin label</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RA0</td>
<td>Input or Output</td>
</tr>
<tr>
<td>1</td>
<td>RA1</td>
<td>Input or Output</td>
</tr>
<tr>
<td>2</td>
<td>RA2</td>
<td>Input or Output</td>
</tr>
<tr>
<td>3</td>
<td>RA3</td>
<td>Input or Output</td>
</tr>
<tr>
<td>4</td>
<td>RA4/T0CKI</td>
<td>Input or Output or Input to TMR0</td>
</tr>
<tr>
<td>5</td>
<td>–</td>
<td>None</td>
</tr>
<tr>
<td>6</td>
<td>–</td>
<td>None</td>
</tr>
<tr>
<td>7</td>
<td>–</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 8.1 PIC 16F84 port bit locations

<table>
<thead>
<tr>
<th>Register bit</th>
<th>Chip pin label</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RB0/INT</td>
<td>Output or Input or Interrupt Input</td>
</tr>
<tr>
<td>1</td>
<td>RB1</td>
<td>Output or Input</td>
</tr>
<tr>
<td>2</td>
<td>RB2</td>
<td>Output or Input</td>
</tr>
<tr>
<td>3</td>
<td>RB3</td>
<td>Output or Input</td>
</tr>
<tr>
<td>4</td>
<td>RB4</td>
<td>Output or Input or Interrupt on change</td>
</tr>
<tr>
<td>5</td>
<td>RB5</td>
<td>Output or Input or Interrupt on change</td>
</tr>
<tr>
<td>6</td>
<td>RB6</td>
<td>Output or Input or Interrupt on change</td>
</tr>
<tr>
<td>7</td>
<td>RB7</td>
<td>Output or Input or Interrupt on change</td>
</tr>
</tbody>
</table>
and for all outputs is 00000000 (00h). Any combination of inputs and outputs can be set by loading the TRIS register with the required binary code.

When the chip is powered up, these bits default to ‘1’; it is not necessary to initialize input pins to output. This makes sense because if the pin is accidentally turned on it is unlikely that a high state is required. If the pin is eventually turned off, it is desirable to set it to a high state, so the short-circuit current is likely to damage the output circuit if set on an input, no damage would be done.

The data direction register TRISB is loaded by placing the required code in W and then using the instruction TRIS 06 or TRIS 07 for Port A and Port B, respectively. Alternatively all file registers with addresses 80–CF can be addressed directly, using the page selection bits in the option register, and this may be seen 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, as described above, and all bits default to input on power up. The PORTB register bit allocation is shown in Table 8.1.

Bit 0 of Port B has an alternate 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 (ISR) upon completion of the current instruction (see Section 9.3). 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 = 06

As for 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 (must be initialized). The program instruction TRIS 06 moves the data direction code from W to TRISB register.

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, which indicates if the result of any operation is zero. The zero flag bit is set to ‘1’ if it is used by the Decrement/Increment and Skip if Zero instructions, and can be tested by the Bit Test and Skip instructions, to implement conditional branches of the program flow. The status register bit functions are shown in Table 8.2. We can leave consideration of the rest of these for the moment; the data sheet and more advanced programming references will provide more information.

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, being an 8-bit register, can count up to 255 pulses. For external inputs, the pulses are applied at pin RA4. When used as timer, the internal clock is used to supply the pulses. If the processor clock frequency is known, the time taken to reach a given count can be calculated. When the counter rolls over from FF
PIC 16F84 Architecture

**Table 8.2** STATUS register bit functions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Label</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>C</td>
<td>Carry Flag</td>
<td>Set if register operation causes a carry out of bit 8 of the result (8-bit operations)</td>
</tr>
<tr>
<td>1</td>
<td>DC</td>
<td>Digit carry flag</td>
<td>Set if register operation causes a carry out of bit 3 of the result (4-bit operations)</td>
</tr>
<tr>
<td>2</td>
<td>Z</td>
<td>Zero Flag</td>
<td>Set if the result of a register operation is zero</td>
</tr>
<tr>
<td>3</td>
<td>PD</td>
<td>Power down</td>
<td>Cleared when the processor is in Sleep mode</td>
</tr>
<tr>
<td>4</td>
<td>TO</td>
<td>Time out</td>
<td>Cleared when Watchdog Timer (WDT) times out</td>
</tr>
<tr>
<td>5</td>
<td>RP0</td>
<td>Register bank select</td>
<td>RP0 selects File Registers 00–7F or 80–FF</td>
</tr>
<tr>
<td>6</td>
<td>RP1</td>
<td>Select bits</td>
<td>RP1 not used</td>
</tr>
<tr>
<td>7</td>
<td>IRP</td>
<td>IRP not used</td>
<td>IRP not used</td>
</tr>
</tbody>
</table>

**OPTION Register**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Label</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PS0</td>
<td>Prescaler rate select bit 0</td>
<td>These 3 bits form a 3-bit code to select one of 8 prescale values for the counter/timer TMR0 or WDT</td>
</tr>
<tr>
<td>1</td>
<td>PS1</td>
<td>Prescaler rate select bit 1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>PS2</td>
<td>Prescaler rate select bit 2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>PSA</td>
<td>Prescaler assignment</td>
<td>Assigns prescaler to WDE or TMR0</td>
</tr>
<tr>
<td>4</td>
<td>ISC</td>
<td>Interrupt edge select</td>
<td>Select rising or falling edge trigger for T0CKI input at RA4</td>
</tr>
<tr>
<td>5</td>
<td>TCS</td>
<td>Timer counter select</td>
<td>Select internal clock input to RA4 or internal clock</td>
</tr>
<tr>
<td>6</td>
<td>ISE</td>
<td>Interrupt edge select</td>
<td>Select rising or falling edge trigger for INT0 external input</td>
</tr>
<tr>
<td>7</td>
<td>BPU</td>
<td>Port B pull-up enable</td>
<td>Enable pull-ups on Port B pins so input data defaults to '1'</td>
</tr>
</tbody>
</table>
If a crystal clock is in use, the timing will be very accurate. Prescaling can be selected to increase the accuracy by a factor of 2, 4, 8, 16, 32, 64, 128 or 256. As the case with the TRISA and TRISB registers, the OPTION register has to be accessed using a special instruction, namely 'OPTION'. The alternative method, which is recommended by the manufacturers, uses bank selection. There is more on using the timer in Chapter 8.

### Interrupt Control Register

The Interrupt Control Register (INTCON) bit functions are given in Table 8.4. An interrupt is a signal which causes the current program execution to be suspended, and an ISR to be carried out. An interrupt can be generated by an external device, via port B, or from the timer. However, in all cases, the ISR must start at address zero, which is also the program start address. 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 required during the program initialization by writing a suitable code to the INTCON register. Program 9.2 demonstrates the use of interrupts.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Label</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>RBIF</td>
<td>Port B change interrupt flag</td>
<td>Set when any one of RB4–RB7 changes state</td>
</tr>
<tr>
<td>1</td>
<td>INTF</td>
<td>RB0 pin interrupt flag</td>
<td>Set when RB0 detects interrupt input</td>
</tr>
<tr>
<td>2</td>
<td>T0IF</td>
<td>Timer overflow interrupt flag</td>
<td>Set when timer TMR0 rolls over from FF to 00</td>
</tr>
<tr>
<td>3</td>
<td>RBIE</td>
<td>Port B change interrupt enable</td>
<td>Set to enable Port B change interrupt</td>
</tr>
<tr>
<td>4</td>
<td>INTE</td>
<td>RB0 pin interrupt enable</td>
<td>Set to enable RB0 interrupt</td>
</tr>
<tr>
<td>5</td>
<td>T0IE</td>
<td>Timer overflow interrupt enable</td>
<td>Set to enable timer overflow interrupt</td>
</tr>
<tr>
<td>6</td>
<td>EEIE</td>
<td>Data EEPROM write interrupt enable</td>
<td>Set to enable interrupt on completion of write operation of non-volatile data memory</td>
</tr>
<tr>
<td>7</td>
<td>GIE</td>
<td>Global interrupt enable</td>
<td>Enable all interrupts which have been attached</td>
</tr>
</tbody>
</table>

### Other SFRs

Registers EEDATA, EEPGD, EECON1 and EECON2 are used to access the non-volatile ROM data area (see Table 8.5). 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 EECON register. It can be used with INC, which gives indirect access to the file register selected by FSR. This is useful...
8.3.2 General Purpose Registers (GPR1–GPR6)—

The GPRs are numbered 0C–4F. They are also referred to as SRAM registers, because they can be used as a small block of static RAM for storing blocks of data, such as a data table of values read in at a port at intervals. We have already seen an example of using the GPR1 (address 0C) as a counter register in a delay loop. The register was labelled ‘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 doing an output a certain number of times. We could have used any of the GPRs for this function because they are all operationally identical; however, we do need to declare a different name for each when using more than one.

### Table 8.5 Other PIC 16F84 registers

<table>
<thead>
<tr>
<th>Num</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>INDF</td>
<td>File register memory indirect addressing</td>
</tr>
<tr>
<td>4</td>
<td>FSR</td>
<td>For block access</td>
</tr>
<tr>
<td>10</td>
<td>PCLATH</td>
<td>Program counter high byte</td>
</tr>
<tr>
<td>18</td>
<td>EEDATA</td>
<td>Data EEPROM indirect addressing for block access</td>
</tr>
<tr>
<td>19</td>
<td>EEADR</td>
<td>Data EEPROM indirect addressing for block access</td>
</tr>
<tr>
<td>88</td>
<td>EECON1</td>
<td>Data EEPROM read and write control</td>
</tr>
<tr>
<td>89</td>
<td>EECON2</td>
<td>Data EEPROM read and write control</td>
</tr>
</tbody>
</table>

**Summary**

- The 16F84 internal architecture can be represented as a block diagram showing the main functional blocks, which are: program ROM, execution logic, data processing, file registers and data EEPROM.
- The program memory stores up to 1024 14-bit instructions. The program execution starts at address 0000.
- The 14-bit instruction contains operation code and operands, which can vary in length.
- The ALU processes data from the instruction, registers or W.
- Jump instructions modify the program counter to change the execution sequence.
- The file register set contains SFRs and GPRs.
The most important of the SFRs (with their address/number) are the timer (01), program counter (02), status register (03), Port A (05), Port B (06) and interrupt control (0B).

The GPRs are a block of registers that can be used separately, or in blocks, to store temporary data, act as counters, and so on.

Questions

1. State the function of the following blocks within a PIC microcontroller: program memory, program counter, instruction decoder, ALU, W.
2. Why is it not necessary to initialise a PIC port for input?
3. State the main functions of the ALU, and the three sources of its data input.
4. Why is the stack needed for subroutine execution?
5. State the function of the following PIC file registers: PORTA, TRISA, TMR0, PCLATH, GPRxx.
6. State the function of the register bits: STATUS, 2; INTCON, 1; OPTION, 5.
7. Which port pin gives access to TMR0?
8. What is the default destination of a 'move' operation?

Activities

1. Refer to Table 8.6. Complete the logic 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 is executed. Copy the table and complete the additional columns to the right for each of the remaining four instructions. The first is given as a guide.

<table>
<thead>
<tr>
<th>Instruction word</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>30 00 00 00 00</td>
<td>30 00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>Subtrees</td>
<td>0000 0000 0000 0000 0000</td>
<td>0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>Subtrees code</td>
<td>0000 0000 0000 0000 0000</td>
<td>0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>Program address</td>
<td>0000 0000 0000 0000 0000</td>
<td>0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>File register address</td>
<td>0000 0000 0000 0000 0000</td>
<td>0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>Subtree code reg</td>
<td>0000 0000 0000 0000 0000</td>
<td>0000 0000 0000 0000 0000</td>
<td></td>
</tr>
</tbody>
</table>
Table 8.6 continued

<table>
<thead>
<tr>
<th>Instruction number</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal bus</td>
<td>0000</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data bus (8 bits)</td>
<td>XXXX</td>
<td>XXXX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Working register</td>
<td>0000</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Port B data register</td>
<td>0000</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Port B data direction register</td>
<td>0000</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. Study the PIC 16F84A Data Sheet (see Appendix A), Section 4.2. Study the block diagram of the internal circuit connected to pin RA0. The TRIS of the output form a complementary pair of switches, a P-type and an N-type. The P-FET is on when its gate is low. The N-FET is on when its gate is high. For output, the TRIS latch is loaded with the data direction bit 0, and the data is loaded into the data latch from the data bus.
(a) Draw 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.
(b) Extend the logic table and prove that P and N are both off when the pin is initialised for input.
(c) By referring to Chapter 3 if necessary, describe how a data bit is read onto the data bus when the pin is set for input.
(d) What are the functions of the output FETs in the operation of the I/O pin?
Chapter 9
Further Programming Techniques

9.1 Program Timing

The microcontroller program execution is driven by the clock signal generated by an internal oscillator whose frequency is controlled by either an external RC or crystal (XT) network. This signal is divided into four internal clocks (Q1–Q4) which run at a quarter of the oscillator frequency (Fosc). These provide four sequence pulses during each cycle to trigger the processor operations. These include fetching the instruction code from the program memory, and executing it in the instruction register. The instructions are then used by the decoder to set up the control lines to carry out the required process. The four clocks are used to operate the data gates and latches within the MCU to complete the data movement and processing.

This instruction timing is illustrated in Fig. 9.1. Note that, if the CR clock option is used, an output instruction clock signal at Fosc/4 is available at the CLKOUT pin to operate external circuits synchronously. It can also be used in hardware testing to check if the clock is running, and to measure its frequency.

The result of this clocking scheme is that each instruction takes four clock cycles to execute, unless a jump (GOTO or CALL) occurs. There will take eight clock cycles because the program counter contents have to be replaced, and this takes an extra instruction cycle.
Further Programming Techniques

9.1.1 Pipelining

The instruction fetch and execute cycles can be carried out simultaneously because the data is being transferred on separate data paths (see Fig. 8.1). While one instruction is being executed, the next is being fetched from the program memory into the instruction register. This overlapping of execution stages is called 'pipelining', with the PIC having a two-stage pipeline. The CISC microprocessors such as Pentium use more elaborate pipelining to break the instruction processing into multiple stages, and thereby boost performance.

9.1.2 Execution Time

We can now predict how long a particular sequence will take to execute. A clock rate of 4 MHz is a convenient default value because it is the maximum operating frequency in standard XT mode (see PIC 18F84A data sheet, Table 5-4), and also gives an instruction execution rate of 1 MIP (millions of instructions per second) and an instruction cycle time of 1 \( \mu s \).

A delay loop is shown in Table 9.1. The move instructions take one cycle each, and the DECFSZ instruction is then repeated 254 times. The GOTO takes two cycles, because each time the GOTO is executed, the RETURN is pre-fetched, and then not executed, so a cycle is wasted. On the 255th loop, the register becomes zero and the GOTO is skipped, and the RETURN executed. This also takes two cycles, because of another wasted pre-fetch cycle, but is only executed once per delay sequence. The total loop time can then be calculated. By timing the time taken for each instruction and the loop, we can see, this takes 768 \( \mu s \) at 4 MHz. This figure can be confirmed if the program containing the loop is run in the simulator, using the stopwatch, with the clock frequency set to 4 MHz.

The block execution time for a section of code can be predicted before testing in simulator or hardware. Alternatively, the timing can be checked and modified using the simulator. Incidentally, NOP (No Operation) is useful here. For time critical sequences, NOP may be used to create a delay of one instruction cycle, that is, four clock cycles; it has no other effect. Using this, a delay of 1 ms can be created using the delay loop with the count set to 249 and a NOP in the loop to make the loop execution time 4 \( \mu s \). The total loop time is then (249 \times \( \frac{4}{\mu s} \)) plus a few cycles for the loop initialisation and return.
### 9.2 Hardware Counter/Timer

Accurate event timing and counting is often needed in microcontroller programs. For example, if we have a sensor on a motor shaft which gives 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, the controller program continues with other operations, such as processing the timing information, controlling the outputs and checking the sensor input, while the timer keeps an accurate record of the time elapsed. The motor application in Chapter 13 uses this technique.

#### 9.2.1 Using TMR0

The special file 01h in the 16F84 is called timer zero (TMR0); it is an 8-bit counter/timer register which runs independently. This means it can count inputs or clock pulses concurrently, with the same time as the main program execution. The counter can also be set up to generate an interrupt when it has reached its maximum value, so that the main program does not have to keep checking it to see if a particular count has been reached. A block diagram of TMR0 and its associated hardware and control registers is shown in Fig. 9.2.

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 usually, from the chip oscillator. When it reaches its maximum value (FF), it "rolls over" to 00. This register overflow is indicated by

![Table 9.1 Sequence execution time](image)
Further Programming Techniques

Bit Timer interrupt bit label and function

- **T0IF**: TMR0 overflow interrupt flag
  - 0 = No overflow
  - 1 = Overflow

- **T0IE**: TMR0 overflow interrupt enable
  - 0 = Disable
  - 1 = Enable

- **GIE**: Global interrupt enable
  - 0 = Disable
  - 1 = Enable

Bit Timer control bit label and function

- **PS0**: Prescaler rate select bit 0
  - 0
- **PS1**: Prescaler rate select bit 1
  - 1
- **PS2**: Prescaler rate select bit 2
  - 2
- **PSA**: Prescaler assignment bit 0
  - 3 = Select prescaler for TMR0
  - 1 = Deselect prescaler for TMR0

- **T0SE**: TMR0 source edge select bit 0
  - 4 = Increment on rising edge of RA4
  - 5 = Increment on falling edge of RA4

- **T0CS**: TMR0 clock source select bit 0
  - 6 = Instruction clock
  - 7 = Ext. clock /4

- **T0CKI**: TMR0 clock source select bit 1
  - 8 = Instruction clock
  - 9 = Ext. clock /4

- **PS**: Prescaler value select
  - 0000 11 1 1
  - 0011 00 1 1
  - 0101 01 0 1

- **TMR0**: TMR0 register

- **INTCON**: Interrupt control register

- **OPTION**: Option register

Figure 9.2 Hardware counter/timer setup and operation.

The simplest mode of operation of TMR0 is counting pulses applied to RA4, which has the alternate name T0CKI, Timer Zero Clock Input. These pulses could be input manually from a push button, in which case the microcontroller could be programmed to increment the timer at a fixed rate, or they could be generated by another signal source, such as a sensor coupled to the motor shaft, and one of the PIC outputs controls the sensor. The microcontroller could be programmed to count the pulses at a rate depending on the number of revolutions, and the motor shaft could be designed to move the output through a set angle, as in a robot, for example.
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 2, 001 divide by 4 and so on up to 111 for divide by 256. TMR0 can also be pre-loaded with a value, and the overflow detected when it has been topped up by a set number of pulses.

9.2.3 Timer Mode
The internal clock is selected by setting the OPTION register, bit 5, to 0. To use TMR0 as an accurate hardware timer, a crystal oscillator must be used as the chip clock source. A convenient crystal frequency is 4 MHz, because it is divided by four before it is fed to the input of TMR0, giving a pulse frequency of 1 MHz. The counter would then be clocked every 1 microsecond, and would take 256 pulses to count from one second to the next. Again, by pre-loading with a suitable value, a smaller time interval could be selected, with time out indicated by the timer interrupt flag. For example, by pre-loading with the value 156 (0x9C), the overflow would occur after 100 microseconds. Alternatively, the time period measured can be extended by selecting the prescaler. The maximum timer period would then be 65.536 ms, if the prescale value of 32 is selected. Crystals are also available in frequencies that are more conveniently divisible by 2. For example, a 32.768 kHz crystal frequency will produce a time-out every 1 second, if the prescale value of 32 is selected. In this mode, 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).

9.2.4 TIM1 Timer Program
Program TIM1, which demonstrates the use of the timer, is listed as Program 9.1. It is designed to increment a binary output once per second. The program uses the same demonstration BIN hardware, as the previous programs, with eight LEDs displaying the contents of Port B. An adjustable CR clock is used as the source frequency of 65536 Hz (approximately). This frequency is divided by four, and is then divided by 64 in the prescaler, giving an overall frequency division of 65536 ÷ 256. The timer register is therefore clocked at 256 Hz. The timer register counts from zero to 256 and so overflows every second. The output is then incremented. It will take 256 cycles to complete the 8-bit binary output count.

9.2.5 Timing Problems
Each instruction in the program takes four clock cycles to complete, with jumps taking eight cycles. If the program sequence is studied carefully, extra time is taken in completing the program loop before the timer is restarted. In this application, it will cause only a small error, but in other applications it may be significant. Also notice that the program has to keep checking to see if the time-out flag has been set by the timer overflowing. It is more efficient to allow the processor to carry on with some other process while the timer runs, and allow the time-out condition to interrupt the main program when it has finished.

9.2.6 More Timers
Because they are so useful, some larger PIC chips have more than one timer/clock. The 16F877, for example, has three. Its addition to Timer 0 (TMR0) is Timer 1, a 16-bit counter
Further Programming Techniques

Program 9.1  TIM1 source code

; *******************************************************************
; TIM1.ASM  M. Bates  6/1/99  Ver 1.2
; *******************************************************************

; Minimal program to demonstrate the hardware timer operation.
; The counter/timer register (TMR0) is initially set 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 counter 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 16F84
; Hardware: PIC BIN Demo Hardware
; Clock: CR = 65536 Hz (approx)
; Outputs: RB0 - RB7: LEDs (active high)
; 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

; Initialise 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)
Interrupts

Interrupts are generated by an internal or external asynchronous (not linked to the program timing) event, and the interrupt signal can be received at any time during the execution of the main process. For example, when you hit the keyboard or move the mouse on a PC, 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. This code which is executed as a result of the interrupt is called the 'interrupt service routine' (ISR). When the ISR has finished its task, the process which was interrupted must be resumed as though nothing has happened. This means that any information being processed at the time of the interrupt may have to be stored temporarily, so that it can be reactivated later. The program returns to saved automatically on the stack, or 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.

9.3.1 Interrupt Setup

A block diagram detailing the 16F84 interrupt system is given in Fig. 9.3. The PIC has four possible interrupt sources:

1. RB0 can be selected as an edge-triggered interrupt input by setting INTCON,4 (INTE), with the active edge selected by OPTION,6 (INTEDG).
2. RB7–RB4 can be selected to trigger an interrupt if any of them changes state, by setting INTCON,3 (RBIE).
3. TMR0 overflow interrupt can be selected by setting INTCON,5 (T0IE).
4. Completion of an EEPROM (non-volatile memory) write operation can be used to trigger the interrupt.
If interrupts are required, the interrupt source must be enabled in the INTCON (interrupt control) register. Then, the global interrupt enable bit, which enables all interrupts, must be set (INTCON,7) and finally the specific interrupt bit must be set. Note that, although there are four interrupt sources, they all still call CSEG at location 0004. If more than one interrupt source is to be used, a mechanism for identifying which is active must be included in the application program. There is no hardware interrupt priority system, as is available in more complex processors.
9.3.2 Interrupt Execution

Interrupt execution is also illustrated in Fig. 9.3. Each interrupt routine 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 may contain a GOTO address (add/label) if the ISR is to be placed elsewhere in program memory. If interrupts are to be used, a GOTO must also be used at the reset vector address, 000.

When the interrupt service routine ends, the ISR must be terminated with the instruction RETFIE. This causes the original program address to be pulled from the stack, and program execution resumes at the address following the one which was interrupted. It may be necessary to save other registers as part of the ISR, so that they can be restored after the interrupt. This is called 'context saving'. This is illustrated in INT1 program below by saving and restoring the contents of Port B data register as part of the ISR.

9.3.3 INT1 Interrupt Program

A demonstration program, Program 9.2, illustrates the use of interrupts. The RB0 hardware must be modified to run the program, with the push buttons connected to RB0 and RA4. This is necessary because only Port B pins can be used for external interrupts.

The program outputs the same binary count to LEDs on PortB, bits 1-7, as seen in the BINx programs, to represent its normal activity. This process is then interrupted by RB0 being pulsed manually. The interrupt service routine causes all the outputs to be switched on, and then waits for the button on RA4 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 can be represented by the flowcharts in Fig. 9.4.
Clock: \( \sim 100\text{kHz} \)
Inputs: Push Buttons
Interrupts: RB0 Interrupt enabled
Outputs: RB1 - RB7: LEDs (active high)
Code Protect: Disabled
Subroutines: DELAY
Parameters: None

Register Label Equates:

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

Input Bit Label Equates:

\[ \text{intin} \quad \text{EQU} \quad 0 \quad ; \text{Interrupt input = RB0} \]
\[ \text{resin} \quad \text{EQU} \quad 4 \quad ; \text{Restart input = RA4} \]
\[ \text{INTF} \quad \text{EQU} \quad 1 \quad ; \text{RB0 Interrupt Flag} \]

Set program origin for Power On Reset:

\[ \text{org} \quad 000 \quad ; \text{Program start address} \]
\[ \text{GOTO setup} \quad ; \text{Jump to main program start} \]

Interrupt Service Routine at address 004:

\[ \text{org} \quad 004 \quad ; \text{ISR start address} \]
\[ \text{MOVF} \quad \text{PORTB},\text{W} \quad ; \text{Save current output value} \]
\[ \text{MOVWF} \quad \text{tempb} \quad ; \text{in temporary register} \]
\[ \text{MOVLW} \quad \text{b'11111111} \quad ; \text{Switch LEDs 1-7 on} \]
\[ \text{MOVWF} \quad \text{PORTB} \quad ; \text{at the LEDs} \]
\[ \text{BCF} \quad \text{INTCON},\text{INTF} \quad ; \text{Clear RB0 interrupt flag} \]
\[ \text{RETFIE} \quad ; \text{Return from interrupt} \]
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 occurs 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 which is most likely to be interrupted.

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.

9.3.4 More Interrupts
In larger PIC chips, additional interrupt sources are typically present, such as analog inputs, serial ports and additional timers. These all have to be set up and controlled via additional

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

; Main Program***********************************************
; Initialise 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
```

The interrupt routine is placed at address 004. The instruction ‘GOTO setup’ jumps over it at run time to the initialisation process at the start of the main program. The interrupt and delay routines must be assembled before the main program, because they contain the subroutine start address labels referred to in the main program, so they are entered first in the source code. The last instruction in the ISR must be RETFIE. This instruction pops 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.
Further Programming Techniques

9.4 More Register Operations

special function registers. As an example, the 16F877 has 14 interrupt sources, but still has only one interrupt vector address, 0004, to handle them. The interrupt bits must be checked in software to see which is active before calling the appropriate ISR. Also, the stack can only hold eight return addresses, despite the program memory being 8k. The limit of eight levels of subroutine or interrupt can easily be exceeded if the program is too highly structured, so this must be borne in mind when planning the program implementation.

Figure 9.4 INT1 interrupt program flowcharts. (a) Main sequence; (b) Delay routine; (c) Interrupt service routine.

The functions of the most commonly used registers are described in Chapter 8, and further operations using the PIC registers are outlined in this section.
9.4.1 Data Destination W

The default destination for operations which generate a result is the file register specified in the instruction. For example:

```plaintext
INCF spare
```

increments the register labelled 'spare', with the result being left in the register. The above syntax generates a message when the program is assembled to remind the user that the default destination is being used. This is because the full syntax is:

```plaintext
INCF spare,1
```

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:

```plaintext
MOVF spare,W
```

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

```plaintext
INCF spare,0
```

or

```plaintext
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 MOVWF (Move Working Register) which are by definition register specific. MOVW and ADDWF have the option of specifying the working register as a destination, which makes it possible to store the result in the working register or in directly in memory. This option can be used to reduce both instruction execution time and memory requirements, which in PIC applications may be quite significant, and compensate for the lack of instructions to make direct moves between file registers.

9.4.2 Register Bank Select

The 16F84 file register set (Fig. 8.2) is organised 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 SFRs can be accessed in either bank. Others have special access instructions, namely TRIS to write the Port A and B data direction registers, and OPTION which is used to set up the real time clock counter.

The manufacturer recommends using bank selection to access all these registers, and the instruction set states that the instructions TRIS and OPTION may not be supported by future assemblers. Bank 0 is enabled by default, and bank 1 registers (OPTION, TRISA, TRISB, RECF0, and RECF1) can be selected by clearing bit 5, RP0, in the STATUS register, prior to accessing the corresponding register number. The alternative method is to set Port B to output as follows:

```plaintext
BSF STATUS,5 ; select bank 1
CLRW ; load W with data direction code
MOVWF TRISB ; set Port B as outputs
BCF STATUS,5 ; re-select bank 0
```
Further Programming Techniques

It is a good idea to re-select bank 0 immediately, as this is the most commonly used. However, if further bank 1 access is required, leave this step until later. Once a bank has been selected, it remains accessible until de-selected. Larger PICs ship with more special function registers and provide more data registers than the PIC in this booklet, requiring two bits for bank selection.

An alternative is to use the pseudo-operation BANKSEL as follows:

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

BANKSEL selects the bank that the specified register is in, so, to change banks, any register in the required bank will do. The temperature control program (Program 15.1) uses this technique, and it is recommended as the best option for accessing registers not in bank 0.

Pseudo-operations, or special instructions, are explained in Section 9.8.

9.4.3 File Register Indirect Addressing

File register 04 is the File Select Register (FSR). It is used for indirect or indexed addressing of the other file registers, particularly the GPRs. If the register address (00–3F) is loaded into FSR, the contents of that file register can be read or written through file register 00, the Indirect File Register (INDF). This method can be used for accessing a set of data RAM locations, by reading or writing the data via INDF, and selecting the next file register by incrementing FSR (see Fig. 9.5). This indirect indexed file register addressing is particularly useful for storing a set of data which has been read in at a port, in, for example, a data logging application. An output data table of predefined values, such as seven-segment display codes, can use the program data table method described in Section 9.6.

The demonstration Program 9.3 loads a set of file registers, 20–2F, 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 moved into INDF each time.

9.4.4 EEPARM Memory

Many PIC chips have a block of electrically erasable read only memory (EEPROM) which operates as non-readable read and write memory; the data written to the block is retained when the power is off. This is ideal for example for security applications such as an electronic lock, where the correct combination can be stored and changed as required. Access to EEPROM is illustrated in Fig. 9.6.

The four registers used to access the memory are EEDATA, EEADR, EECON1 and EECON2. The data to be stored is placed in EEDATA, and the address to which it is to be written is placed in EEADR. A complex write sequence is designed to reduce the possibility of an accidental write to EEPROM, whereby valuable data is lost. Reading the EEPROM is more straightforward. The LOCK application program in Appendix B includes examples of the code sequences required to read and write EEPROM. Other devices use a different technique to access the EEPROM; the 8-pin PIC 12C508A devices use serial
More Register Operations

00 INDF
Indirect file register

04 FSR
File select register

20 General purpose registers

2F

Set address, inc/dec, check, and repeat
Write data

File register address

Select file register

Block of GPRs loaded with dummy data

Figure 9.5 Indirect file register addressing.

access via the unused bits of the port register. The individual device data sheet must therefore be studied carefully to use this feature.

9.4.5 Program Counter High Register, PCLATH

The 16F84 has 1k of program memory (000–3FF), requiring a 10-bit address; the 8-bit PCL (program counter low byte) can only select one of 256 addresses. The 1k of program memory is therefore divided into four 256-word blocks (pages), one of which is selected with 2 extra bits in the PCLATH (program counter latch high) register. The PCL provides the address within each page of memory and is fully readable and writable. When a program jump is executed, PCL and PCLATH are modified automatically, but in CALL and GOTO use a full 10-bit operand for jumps, so do not require any special manipulation of the address for jumping across page boundaries. However, if PCL is modified by a direct write under program control, PCLATH bits 0 and 1 may need to be manipulated to access page boundaries successfully.

In other PIC devices, there may be other limitations to program branching operations. For example, CALL instructions in the 12C5XX group are limited to the first 256 locations of the program, even though the overall memory may be up to 1k. Check the data sheet carefully to avoid problems with the limitations.

9.6 Program Counter High Register, PCLATH

The 16F84 has 1k of program memory (000–3FF), requiring a 10-bit address; the 8-bit PCL (program counter low byte) can only select one of 256 addresses. The 1k of program memory is therefore divided into four 256-word blocks (pages), one of which is selected with 2 extra bits in the PCLATH (program counter latch high) register. The PCL provides the address within each page of memory and is fully readable and writable. When a program jump is executed, PCL and PCLATH are modified automatically, but in CALL and GOTO use a full 10-bit operand for jumps, so do not require any special manipulation of the address for jumping across page boundaries. However, if PCL is modified by a direct write under program control, PCLATH bits 0 and 1 may need to be manipulated to access page boundaries successfully.

In other PIC devices, there may be other limitations to program branching operations. For example, CALL instructions in the 12C5XX group are limited to the first 256 locations of the program, even though the overall memory may be up to 1k. Check the data sheet carefully to avoid problems with the limitations.
Further Programming Techniques

Program 9.3
Indexed file register addressing

; Demonstrates indexed indirect addressing by writing a dummy data table to GPRs 20 - 2F

PROCESSOR 16F84 ; 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
MOVWF INDF ; to INDF and GPRxx
INC FSX ; Increment GPR Pointer
BTFSS FSR,4 ; Test for GPR = 30h
GOTO next ; Write next GPR
SLEEP ; Stop when GPR = 30h
END ; of source code

9.5 Special Features
PIC chips have a number of special features which enhance its flexibility and range of applications. Different oscillator types can be used, timers enabled to ensure reliable program start up and recovery, and in-circuit programming and code protections are available.
9.5.1 Oscillator Type

PIC chips can be operated with an external RC network, a crystal oscillator and an externally or internally generated clock signal. Typical oscillator circuits are illustrated in Fig. 9.7.

For applications where precise timing of the program is not important, an inexpensive RC clock circuit (Fig. 9.7(a)) can be used. This requires only a resistor and capacitor connected as shown 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. For example, the outputs from the program BIN1. The clock and output frequencies can be ‘trimmed’ to a required value. On the other hand, the clock signal will not be very accurate or stable.

The crystal is slightly more expensive, but is far more precise than the RC clock. In the XT oscillator circuit (Fig. 9.7(b)) the crystal resonates at a fixed frequency, with an accuracy of around 1 part in 10 million, or 0.0001%. This will allow the hardware timer to measure exact intervals and to generate accurate output signals. The overall execution time of the

![Figure 9.7 PIC clock circuits. (a) RC oscillator; (b) Crystal oscillator.](image-url)
Further Programming Techniques

Program blocks can also be predicted; this can be done by calculation, or, more readily, by use of the stopwatch in MPLAB.

If the PIC chip is part of a larger system, or one with more than one processor, a system clock generated by a master oscillator can be input at CLKIN. One of the clock options must then be selected. The clock type must be selected when programming the chip, to match the target system hardware design. There are three types of clock which can be used: standard (XT), low power (LS), or high speed (HS). XT mode should be used for clock speeds up to 4Mhz, and HS mode up to 20Mhz.

In order to minimise the number of external components required, some PIC chips now have an on-board oscillator option, which provides a 4Mhz clock and 1μs instruction cycle. Because this is not a precise oscillator, it is tested in production and a calibration value is pre-programmed in the first program memory location. This value must then be loaded into the oscillator calibration register OSCCAL. Even so, the accuracy achieved is specified as only about 5% (3.8–4.2MHz).

9.5.2 Power-on Timers

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 circuits connected to it. If the processor is powered up too soon, it may not start correctly. In a conventional microprocessor, an external circuit is typically connected to the CPU reset input, which provides a delay between the power supply being switched on and the processor starting. The PIC has the required power-on timers built in to the chip. The reset input can therefore simply be connected to the positive supply (+5V) for many applications, as is the case in the examples in this book. When the PIC power-on timer is preset with the power-up, it is activated when the supply voltage detected at Vdd rises through about 1.5V. This starts a power-up timer which counts up after 72μs, which in turn triggers an oscillator start-up delay, which delays another 16 clock cycles, to allow the internal clock to stabilise. An internal reset is then generated, and the program starts executing. The power-up timer should normally be initialized when programming the chip, as the resulting delay on start up will normally be insignificant.

9.5.3 Watchdog Timer (WDT)

This is an internal independent timer which, by default, forces the PIC to automatically restart after a fixed period (about 18μs). The idea is to allow the processor to escape from an endless loop or other error condition, without having to be reset manually. This facility would 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 execution and timing of our programs.

The WDT can be disabled by selecting the appropriate configuration setting during program downloading, and the reset signal is then inoperative. If the watchdog is re-energized, the WDT must be reset by writing the instruction CLRWDT. If this instruction is not executed at 4Mhz, the WDT will not be reset. To prevent a program malfunction in the simulator, check that WDT is disabled.

9.5.4 Sleep Mode

The instruction SLEEP causes normal operation to be suspended and the clock oscillator to be 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 used (see Program 9.3) if the program is not required to keep running. If the program execution is allowed to run into unprogrammed locations, there will be a problem because the last instruction code will be the highest. In fact, this is a valid PIC instruction, ADDLW FF, which means add literal ‘FF’ to W, so this instruction will be repeated throughout the unused locations. The program counter will roll over to zero after executing these meaningless instructions up to address 3FF, and the program at 000 will be restarted, so the program will loop by default. It is therefore a sensible precaution to terminate the program with a SLEEP instruction if does not run in continuous loop. If SLEEP is used to stop the program at the end, a power-on reset, external reset or an interrupt can then restart the processor.

9.5.5 In-Circuit Programming and Debugging

In-circuit programming allows the chip to be programmed without being removed from the circuit, which avoids possible mechanical (broken/bent legs) and electrical (static) damage. The programming module is connected to the serial port of the host PC and to the chip via two port pins (RB6 and RB7 in the 16XXX) via a suitable connector (Fig. 9.8). The program can then be downloaded in serial form. Note, however, that the circuit must be designed so that the normal operational connections to the port pins do not interfere with downloading the program. When programming is complete, the connector can be removed, the board set to run, and the port pins used for their normal function.

PIC chips are now designed to allow this programming link to be used in a cheap but effective debugging system. PICs supported by the MPLAB simulator and associated development system can be programmed and debugged interactively. The same MPLAB simulator can be used to test the program as it runs in the actual chip, rather than in the purely software model, with the real hardware acting as inputs and outputs. This allows the hardware to be verified, and timing critical operations to be tested easily and efficiently. All the usual techniques are available, single stepping, breakpoints, register logging, and so on. The finished program can then be tested at full speed in the actual hardware, with any bugs removed. Previously, an expensive in-circuit emulator would be necessary for this type of testing, an invaluable tool for PIC program development.

By incorporating the programming interface into the target hardware, it is also possible for microcontrollers to be reprogrammed remotely after final installation. If a suitable

---

**Figure 9.8**

Serial programming and ICD connections.
Further Programming Techniques

A communication link is available, a new control program can be downloaded while the target processor remains at its remote site. This is a great advantage in, for example, distributed sensing and monitoring applications, where a site visit would be expensive or time-consuming. However, the new program would need to be fully tested on an identical local system before being downloaded to the target system.

9.5.6 Code Protection

In commercial applications, the PIC program designer does not want the software supplied with a product to be copied by a market competitor. The “Code Protect” fuse, selected during programming, is designed to prevent unauthorized copying. The chip can also be given a unique identification code during programming, if required. For our purposes, the code protection should not be enabled, as the program cannot then be read back for verification.

9.5.7 Configuration Word

The oscillator selection bits (2), enabling the timer, power-up timer, and code protection are all selected by setting the bits of a configuration word, located at a special address which is only accessible when a chip is being programmed. These bits can be set from the programming dialogue in MPLAB, as described in Chapter 7. Alternatively, the configuration options can be set by including an assembler directive in the source code.

9.6 Program Data Table

A program may be required to output a set of pre-defined data bytes, for example, the codes to light up a seven-segment display with the correct pattern for each display digit, as in Program 12.2. This data set is written into the program as a table within a subroutine and the data set accessed using CALL and RETLW. To fetch the table value required, the position in the table is placed in W, then the first item, 0, the second and so on. For this purpose, the subroutine ADDWF PCL is used to add the table pointer value to the program counter register so that the seven-segment display is illuminated. The program counter is then used to return the table value in W, and it can then be checked for the required table.

Program 9.4, TAB1, shows how such a table may be used to generate a sequence at the LEDs, using the seven-segment display. The data set is written into the program as a table within a subroutine, and the subroutine ADDWF PCL is used to add the table pointer value to the program counter so that the seven-segment display is illuminated. The program counter is then used to return the table value in W, and it can then be checked for the required table.

For each output, the pointer value (0–8) is placed in W, and the table subroutine is called. The first instruction ADDWF PCL adds the pointer value to the program counter, so that the code is added to the number 1 to give the correct pattern. The table subroutine is then called, and the table value is added to the program counter, so that the next code is added to the number 1. This process continues until the ninth code, 1FF, is returned to the main output loop for display. After that, the test of
Program 9.4 TAB1 program

Program Data Table

149

Output binary sequence gives a demonstration of a bar graph display, using a program data table...

Processor: PIC 16F84

Hardware: PIC Demo System

Clock: CR ≈ 10kHz (Cycle time ≈ 0.7s)

Inputs: none

Outputs: LEDs (active high)

WDTimer: Disable
PUTimer: Enable
Code Protect: Disable
Interrupts: Disabled
Subroutines: 'delay' (one argument)
'table' (argument 'point')

Register Label Equates....................................
PCL EQU 02 ; Program Counter Low Register
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 program

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

continued...
Further Programming Techniques

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

; Start MOVLW b'00000000' ; Set Port B Data Direction Code
TRIS PORTB ; and load into TRISB

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

NEWBAR CLRF point ; Reset pointer to start of table

NEXTON MOVLW 009 ; Check if all outputs 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 table value

GOTO nexton ; and repeat...

; End of main loop ..................................

END ; Terminate source code


The pointer being equal to 9 succeeds, the jump back to 'newbar' 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.

9.7 Assembler Directives

Assembler directives are commands inserted in PIC source code which 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 after a good knowledge of the programming language has been acquired, so we will refer to a small number of selected examples at this stage. The use of some of these is illustrated in Program 9.5, ASD1. The assembler directives are placed in the second column, with the instruction mnemonics. We have already met some of the most commonly used directives, but END is the only one which is essential, all the others are simply available to make the programming process more efficient. For further information refer to the documentation and help files supplied with your current assembler system.

9.7.1 Control Directives

Assembler directives are commands inserted in PIC source code which 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 after a good knowledge of the programming language has been acquired, so we will refer to a small number of selected examples at this stage. The use of some of these is illustrated in Program 9.5, ASD1. The assembler directives are placed in the second column, with the instruction mnemonics. We have already met some of the most commonly used directives, but END is the only one which is essential, all the others are simply available to make the program more efficient. For further information refer to the documentation and help files supplied with your current assembler system.

9.7.1 Control Directives

Processor

Specify the PIC processor for which the program has been designed, and allows the assembly to check that the syntax is correct for that processor. The symbolizer also uses this to automatically select the right processor. The processor can be specified in the assembler command line, if so, this supersedes the source code directive.
**Assembler Directives**

Program 9.5

ASD1 assembler directives program

;************************************************************
; ASD1.ASM M. Bates 17/12/03 Ver 1.1
;************************************************************
; Assembler directives, a macro and a pseud-operation are
; illustrated in this counting program ...
; ***********************************************************

; Directive sets processor type:
PROCESSOR 16F84

; Set configuration fuses:
__CONFIG B'1111111111110011'

; Code protection off, PuT on, WDT off, RC clock

; SFR equates are inserted from disk file:
INCLUDE "C:\PIC\REG84.EQU"

; Constant values can be predefined by directive:
CONSTANT maxdel = 0xFF, dircb = b'00000000'

; Define DELAY macro *******************************************
DELAY MACRO
    MOVLW maxdel ; Delay count literal
    MOVWF timer ; loaded into spare register
    down DECF timer ; Decrement spare register
    BNZ down ; Pseudo-Operation:
        Branch If Not Zero
ENDM

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

; Initialise Port B (Port A defaults to inputs)
MOVLW dircb ; Port B Data Direction Code
TRIS PORTB ; Load the DDR code into F86

; Start main loop .....................................
CLRF PORTB ; Clear Port B Data & restart
again INCF PORTB ; Increment count at Port B
DELAY ; Insert DELAY macro
GOTO again ; Repeat main loop always
END ; Terminate source code

__Config

The configuration directives allows the configuration bits to be specified in the source code, so that they do not have to be set up each time when downloading. This is obviously useful if the program has to be downloaded several times before completion of debugging. The significance of each bit is shown in the data sheet, Section 6.1. A 16-bit word is loaded into the configuration register by the directive. Bits 0 and 1 set the clock type (11 = RC, 01 = XT), bit 2 disables the
Table 9.1 Assembly Directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Org</td>
<td>Sets the code 'origin', meaning the address which will be allocated to the first instruction following this directive. We have already seen (Program 9.2) how it is necessary to set the origin of the interrupt service routine as 000. The default origin is 000, the first program memory location, so if not specified, the program will be placed here. This is the reset address where the processor always starts on power up or reset. If using interrupts, an unconditional jump 'GOTO addlab' should be used as the first instruction at the reset address 000. This will jump over the ISR (or a jump to it) placed at address 004. The main program can then be placed at a higher address using the ORG directive.</td>
</tr>
<tr>
<td>End</td>
<td>Informs the assembler that the end of the source code has been reached. This is the one directive that must be present.</td>
</tr>
<tr>
<td>9.7.2 Conditional Directives</td>
<td>These directives allow selective assembly of source code blocks. That is, sections of code can be omitted during assembly, or repeated, by use of high-level language type statements such as IF...ELSE...ENDIF. Assembler 'variables' are used to define the conditions for assembly.</td>
</tr>
<tr>
<td>9.7.3 Listing Directives</td>
<td>This directive has a number of options which allow the format and content of the listing file to be modified, e.g. number of lines and columns per page, error levels reported, processor type and so on.</td>
</tr>
<tr>
<td>Page</td>
<td>Forces a page break when printing.</td>
</tr>
<tr>
<td>9.7.4 Data Directives</td>
<td>EQU is probably the second most commonly used directive, because it allows literal and register labels to be defined, and we have already used it many times. It assigns a label to any numerical value (hex, binary, decimal or ASCII), and the assembler then replaces the label with the number. This allows recognizable labels to be used instead of numbers.</td>
</tr>
</tbody>
</table>
This directs the assembler to include a block of source code from a named file on disk. If necessary, the file path must be given. 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, but any program block, subroutine or macro could be included in the same way. This allows separate source code files to be included, and forms the way for the user to create libraries of reusable program modules. In the example ASD1, it is used to include a standard header file (REG84.EQU) which defines labels for all the special function registers in the PIC. Use of this option is recommended when the basics have been mastered; standard header files, which use labelling which is consistent with the symbols used in the register monitoring windows in MPLAB, are supplied with the development system files for all processors.

Data, Zero, Set, Res
Allow program constants and data blocks to be defined and memory allocated for specified purposes.

9.7.5 Macro Directives
Macro... Endm
A macro is a block of source code which is inserted into the program when its name is used as an instruction. In ASD1, for example, DELAY is the name of the macro, and its insertion in the main program can be seen in the list file. Thus using a macro is equivalent to creating a new instruction from standard instructions, or an automated copy and paste operation. The directive MACRO defines the start of the block (with a label), ENDM terminates it. It effectively allows you to create your own instruction mnemonics (see also LOCAL and EXITM).

9.8 Special Instructions
Special instructions are essentially macros which are pre-defined in the assembler. A typical example is shown in the program ASD1: BRNZ down, which stands for ‘Branch if Not Zero to label’. It is replaced by the assembler with the instruction sequence Bit Test and Skip and GOTO:

```
BNZ down
```

These two instructions are inserted into the program in place of the special instruction. The zero flag (Z) 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 test not zero, the GOTO is executed, and the program jumps to the address label specified. Special instructions are designed to simplify operation using the carry or zero flag, and are equivalent to conditional branch instructions in complex instruction set processors. This type of instruction is included in the main instruction set of the more powerful 18XXX series of PICs.
9.9 Numerical Types

Literal values given in PIC source code can be written using different number systems. The default is hexadecimal, so 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. Then 8-bit literals will be written as three digits, with the first always zero (000–0FF).

The numerical types supported by the MPASM assembler are:

- hexadecimal
- decimal
- binary
- octal
- ASCII

To specify a type, the initial letter of the type can be used with quotes, such as:

```
H'3F'
D'47'
B'10010011'
A'K'
```

Binary is useful for specifying register values which are bit-oriented, as in the case for many SFRs; the state of each bit can be clearly seen. In particular, we have used binary to define port data direction codes in our demonstration programs.

If an ASCII character is specified, the corresponding 8-bit code in the range 00–7F will be loaded representing the code for each character in the set (see Table 9.2). This option is used

```
<table>
<thead>
<tr>
<th>ASCII character set</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```

Table 9.2: ASCII character set
in sending data to alphanumeric liquid crystal displays, for example. The character itself may
then be used in the program, and the assembler does the code conversion:

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

Note that the A for ASCII can be left out, and the character still be correctly recognised by the
assembler.

Summary

• Each PIC instruction takes four clock periods to execute (instruction cycle time). Jumps take
two instruction cycles. Block execution times can therefore be calculated.

• The hardware counter/timers can be used to count inputs or time intervals. Programmable
prescalers extend the range of the counter. Timer overflow sets the time-out flag, which can
be used to trigger an interrupt.

• Interrupts allow an internal or external event to change the program sequence, and force the
execution of an ISR. There are multiple interrupt sources, but no interrupt priority.

• Register and memory bank selection are sometimes necessary; EEPROM is available for
non-volatile storage.

• The clock signal which drives the chip can be obtained from an RC or crystal circuit, or
master system clock. Power-on timers, watchdog timer, sleep mode, in-circuit programming
and code protection are available.

• Program data tables can be operated using CALL and RETLW, with an incrementing PCL
offset.

• Assembler directives are instructions to the assembler which are not converted into machine
code.

• Macros are user-defined instructions. Special instructions are pre-defined macros.

• Numerical types hex, decimal, binary, octal and ASCII character codes can be used in the
source code.

Questions

1. State the number of clock cycles in a PIC instruction cycle and the number of instruction
cycles taken to execute the instructions (a) CLRW (b) RETURN.

2. If the PIC clock input is 100 kHz, what is the instruction cycle time?

3. Calculate the pre-load 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.
Further Programming Techniques

4. List the bits in the SFRs which have to be initialised to enable an RB7:RB3 interrupt.

5. Sketch the circuit for an RC and crystal clock, showing typical component values and chip connections. State one advantage of each type.

6. State the assembler directive that must be used in all PIC programs.

7. Explain the difference between a subroutine and a macro.

Answers

2. 40µs

3. 6

4. TRISB, 3, 4, 5, 6 and INTCON bits 0, 3, 7

Activities

1. Calculate the time taken to execute one complete cycle of the output obtained from TAB1 with a clock rate of 10kHz.

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 10–100kHz. The input period should be stored in a GPR called ‘period’ as a value where 0A = 10µs and 64A = 100µs, with a resolution of 1µs per bit. The clock uses a 4MHz crystal. Estimate the accuracy of the frequency measurement at each end of the range.
Part C
Applications

10 Application Design
11 Program Debugging
12 Prototype Hardware
13 Motor Applications
This chapter will take you through the complete process of application design and development, based on a simple motor drive system. At each stage, relevant design techniques will be explained and a suitable implementation developed, step by step.

Before designing hardware or writing a program, we have to describe as clearly as possible what an application is required to do. That means a specification is needed which defines the user’s requirement. There are national and international standards which should be observed when designing commercial products to aid clear communication between the engineer, management and client. Here we will simply establish some basic, ‘common sense’ rules.

Once the specification has been written, a useful starting point for hardware design is a block diagram. We have already seen numerous examples in previous chapters. This should represent the main parts of a system and the informations 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 PCB. In a similar way, software can be designed using techniques which 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. The program outline is entered directly into the source code text editor as a set of general statements which describe end-user steps. Control, which would normally be defined at functions and procedures in a high level language, and statements such as ‘if’, ‘while’, ‘for’ in a low level language, are then added under each heading. Each block of pseudocode clearly describes one action which is to be controlled. The pseudocode can be described at level 1, 2, 3 and so on, as more detail is added.

At this stage, we will concentrate on flowcharts, because they are suitable for simple real-time applications, as they graphical nature makes them a good learning tool. The first step in the software design process is to establish a suitable algorithm for the program, that is, a processing
method which will achieve the specification using the features of an available programming language. This obviously requires some knowledge of the range of languages which might be suitable, and experience in the selected language. Formal software design techniques cannot be properly applied until the software developer is fairly 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.

In this chapter, a simple demonstration application will be used to illustrate the development process. Real software products will generally, of course, be far more complex 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 are required to select the most appropriate 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.

10.1 Design Requirements

A system is required to provide a PWM drive signal for a small DC motor. Under PWM control, the speed of the motor is determined by the average level of the signal, which in turn is dependent on the ratio of the on (mark) to off (space) time. This method provides an efficient method of using a single digital output to control output power from a motor. Variable speed output is achieved by varying the Mark–Space ratio. The basic drive waveform is shown in Fig. 10.1. 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 (1–100kHz). 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! However, a dedicated PWM interface is

---

**Figure 10.1**  Pulse width modulated signal.
needed to achieve this. We will go for low-frequency operation just to demonstrate the principles involved. The hardware is also simplified. For example, instead of a single FET drive transistor as in Chapter 6, a full bridge drive IC would probably be used to provide bidirectional motor control.

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 at rest 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 stay at the maximum and minimum values in particular 0% must not roll over to 100%, causing a zero to maximum speed transition in a single step. The inputs and outputs must be TTL compatible for interfacing purposes, allowing PWM control from another master controller for multiple motor control. A programmed device will be used so that it is possible to modify the control algorithm to suit different motors and to enable future enhancement of the controller options and performance. A logic table (Table 10.1) specifies the operations required.

In addition, a performance specification should be provided to quantify the performance criteria as far as possible:

---

**PERFORMANCE SPECIFICATION**

***************************************************************

<table>
<thead>
<tr>
<th>Project: MOT1</th>
<th>Variable Speed Controller for Small DC Motor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Maximum load: 5V 1A (2.5W)</td>
<td></td>
</tr>
<tr>
<td>2. Manual or remotely controlled variable speed:</td>
<td></td>
</tr>
<tr>
<td>2.1 Start: at 50% MSR</td>
<td></td>
</tr>
<tr>
<td>2.2 Range: 0% to 100%</td>
<td></td>
</tr>
<tr>
<td>2.3 Steps: 1%, Max &gt; 99%</td>
<td></td>
</tr>
<tr>
<td>2.4 Step Resolution: &lt; 1%</td>
<td></td>
</tr>
<tr>
<td>2.5 Control:</td>
<td></td>
</tr>
<tr>
<td>2.5.1 Increment, Decrement</td>
<td></td>
</tr>
<tr>
<td>2.5.2 Hold MSR when inputs inactive</td>
<td></td>
</tr>
</tbody>
</table>

***************************************************************

---

**Table 10.1 MOT1 application control logic**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Disabled</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Disabled, set speed to 50%</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>Run with PUGA 50% or at current speed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Increment MSR (hold at max)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Increment MSR (hold at min)</td>
</tr>
</tbody>
</table>

---


10.2 Block Diagram

In the block diagram, the system inputs and outputs must be identified, and if necessary, a provisional arrangement of sub-systems worked out (see Fig. 10.2). The directions and type of information flow between the blocks should be identified clearly, using directed lines (arrows). Block diagrams can be used to illustrate the transitions of a signal, if necessary. Parallel data paths should be shown as broad arrows, or with suitable signal labelling.

Figure 10.2
MOT1 system block diagram.

The block diagram can be drawn using the drawing tools in Word, other standard wordprocessor or DTP packages, since it needs only basic shapes, arrows and text boxes. Experiment with your usual package.

In Word, the drawing toolbar may need to be enabled 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.

The main elements can be drawn using text boxes, and the same object used for labelling with '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 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 re-positioned on the page if necessary.

10.3 Hardware Design

Unlike the program in 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 is a significant consideration in the selection of a microprocessor or microcontroller, as is the number and type of inputs and outputs, data storage and interfacing.

The design requirements of MOT1 could be satisfied using relatively complex controller systems, based on a conventional CISC processor, such as the 8080, and additional features.
Hardware Design

More inputs and outputs could allow control of several motors simultaneously, a standard serial interface to a host computer system would be available, and the larger memory could accommodate a more complex program, or a program written in a high level language such as 'C'. In the simple PWM example proposed here, however, the requirement is of minimal complexity with no special interfacing specified.

Alternatively, a purely hardware solution could be produced based around, for example, the 555 timer. However, this would not provide the push button digital control required at the PWM output if controlled by an analog input. Therefore a small microcontroller solution appears the most suitable.

A circuit derived from the block diagram is shown in Fig. 10.3. The motor is controlled by an FET, which acts as a current switch operated by the PIC TTL level output. The motor forms an inductive load, so a diode is connected to protect the FET from any back emf from the motor. The input control uses simple active low push buttons, with connections for remote system control.

![Circuit Diagram](image_url)

**Figure 10.3** MOT1 circuit diagram.

The microcontroller only needs four I/O pins, so a 6 I/O 12XXX series device could be considered. However, an external reset is required, so our reference device 16F84 will be used. The controller I/O allocation can thus be specified as shown in Table 10.2.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Type</th>
<th>Pin</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>System</td>
<td>CLKIN</td>
<td>RC clock</td>
<td>∼100 kHz</td>
</tr>
<tr>
<td>Reset</td>
<td>System</td>
<td>!MCLR</td>
<td>Active low</td>
<td>Restart at default speed</td>
</tr>
<tr>
<td>PWM</td>
<td>Output</td>
<td>RA0</td>
<td>Pulse</td>
<td>FET drive</td>
</tr>
<tr>
<td>Run</td>
<td>Input</td>
<td>RA4</td>
<td>Active low</td>
<td>Enable motor</td>
</tr>
<tr>
<td>Up</td>
<td>Input</td>
<td>RA2</td>
<td>Active low</td>
<td>Increase speed</td>
</tr>
<tr>
<td>Down</td>
<td>Input</td>
<td>RA3</td>
<td>Active low</td>
<td>Decrease speed</td>
</tr>
</tbody>
</table>

**Table 10.2** MOT1 I/O allocation using PIC 16F84
The PIC provides motor speed control with a PWM output at RA0. The \textit{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 (\textit{UP}) is low, the MSR at RB0 should increase, and the motor speed up. When RA3 (\textit{DOWN}) is low, the MSR should be reduced, slowing the motor down. \textit{MCLR} (Master Clear) is the reset input to the PIC, which will restart the program when pulsed low, and hence return it to the default value of 50\% MSR.

We can now start work on the software using a flowchart to outline the program. A few simple symbols and rules will be used to help devise a working assembly code program. These are explained below.

**10.4 Software Design**

Computer and controller programs in general incorporate these main types of operation:

1. **Sequence (no jump)**: A sequence of instructions is executed without branching. The program counter is not modified.

2. **Selection (conditional jump)**: A condition is tested for true or false and a jump is made or not, depending on the result. In high level languages, these operations may be combined to provide a multiple choice branch depending on the value of the variable.

3. **Iteration (repeating loop)**: A conditional branch is used to jump back and repeat a sequence while a condition is true, or until a condition is met, or endlessly (this is the last option at the end of the program).

A program consists of a sequence of instructions in a low level language (LLL) such as PIC assembler, or statements in a high level language (HLL) such as BASIC, Pascal or 'C'. These instructions are executed in the order that they appear in the source code, unless there is an instruction or statement which causes a jump or branch. Usually jumps are 'conditional', which means that some input or variable condition is tested and the jump made or not, depending on the result. In PIC assembler, \textit{Bit Test and Skip if Set/Clear} and \textit{Decrement/Increment File Register and Skip if Zero} provide conditional branching when used with a \textit{GOTO label} or a \textit{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 which is decremented within the loop reaches zero. In high level languages, conditional operations are created using the \textit{IF} (a condition is true) \textit{THEN} (do a sequence) and \textit{DO} (a sequence) \textit{WHILE} (a condition is true) syntax.

**10.4.1 MOT1 Outline Flowchart**

Flowcharts illustrate the program sequence, selections and iterations in a pictorial way, using simple sets of symbols. Some basic recommendations for laying out flowcharts will be made here which will help to create consistently drawn and useful flowcharts. An outline flowchart for the motor speed control program \textit{MOT1} is shown in Fig. 10.4.

The outline flowchart shows a sequence where the inputs (\textit{RUN}, \textit{UP}, \textit{DOWN}) are checked and the delay count modified if either of the speed control inputs are active. The output is then set high and low for that cycle, using the calculated delays to give the required speed.
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.

In common with most so-called ‘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 omitting 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 crossovers (see below).

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.

10.4.2 MOT1 Detail Flowchart

The outline flowchart given in Fig. 10.4 may show enough information for an experienced programmer. If more detail is needed, boxes in the main program can be elaborated until there are enough process boxes or subroutines. Sub programs are not a part of this book, but for detailed examination of an assembly program it is necessary to break the sequence into assembly code. A detailed flowchart is shown in Fig. 10.5.

When the interrupt occurs, a set of conditional jumps is required to specify the motor check the ‘up’ and ‘down’ inputs, and test for the minimum and maximum values of the value of ‘Count’. (H and L). Two different forms of the decisions box have been used in the example, both of which may be seen in other references. The diamond-shaped decisions symbol is used...
Define motor output up, down, run inputs

\[ \text{count} = 80h \]

Run = 0? No

Inc and test \[ \text{count} = 0? \]

No

Decrement \[ \text{count} \]

No

Set motor on

Delay for \( \text{count} \)

Set motor off

Delay for \( (256 - \text{count}) \)
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 into one. In either case, the decision box should contain a question with its outputs representing a 'Yes' or 'No' result of the test.

10.4.3 Program Structure

In the previous example program BIN4, a delay subroutine is used. Recall that this is a process defined as a compact block of code which can be executed from any point. It is indicated in the main program flowchart (Fig. 7.3a) with the subroutine box with double sides. The delay routine sequence is then detailed in Fig. 7.3b. It starts with a terminal symbol which contains the subroutine start label used in the program source code, and ends with 'Return'. All subroutines which are nested with the CALL instruction must be terminated with a RETURN instruction. The values of the parameters must be passed to the subroutine. When the subroutine is executed, the values of the parameters are transferred into the variables used in the subroutine. After the subroutine is finished executing, the return address is transferred back into the program counter. Failure to observe this rule will result in a stack error.

'Decrement and Skip if Zero' is used to create the standard software delay loop. The two delays required in BIN4 could be written separately, but the function is the same, with only the delay count differing, using the same delay block twice is more code efficient. Sometimes separate blocks might be used if the timing is critical. Alternatively, a macro could be defined for the delay which the assembler would insert twice (see Chapter 9) which would effectively create a 'delay' special instruction.

A subroutine will often use a value set up in the calling routine. In BIN4, the delay time is placed in W for use by the delay routine; this is an example of 'parameter passing'. Other blocks in the main program can also be created as subroutines, but they are most useful when the routine is to be used more than once. The subroutine can then be copied for use in another program or saved as a separate file for inclusion in new programs.

Application MOT1 does not require the use of subroutines.

10.4.4 Flowchart Symbols

A minimal set of flowchart symbols is shown in Fig. 10.6, most of which have already been used. For commercial applications, the relevant standards should be applied, and would override any recommendations made here.

**Terminals**

These symbols are used to start or end the main program or a subroutine (see Fig. 10.6a). The program name or routine start label used in the source code should also be used 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 (MOT1) as the start symbol of the main program, and the subroutine start address label in subroutine start symbols.

**Procedures**

The process box is a general purpose symbol which represents a sequence of instructions, possible including loops inside it (see Fig. 10.6b). The top level flowchart of a complex program can be simple, with a list of detail encapsulated in each box. A subroutine is a process...
which will be implemented in the source code as a separate block, and which may be used more than once within a program. It should 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.

Input/Output
This represents processes whose main function is input or output using a port data register in the microcontroller or microprocessor system (see Fig. 10.6(c)). It is a statement in the form which describes the general effect of the I/O operation, for example, ‘Switch Motor On’ rather than ‘Set RA0’. That will make the flowchart easier to understand.

Decisions
The decision symbol contains a description of the situation as a question. There will be two alternate exit paths, for the answers ‘yes’ or ‘no’ (see Fig. 10.6(d)). Only the arrow bearing back or forward needs to be labelled ‘yes’ or ‘no’; the default option, which continues the program flow down the centre of the chart, need not be labelled. In PIC assembly language, this symbol would refer to the ‘Test and Skip’ instructions. In the MOT1 detailed flowchart, an enlarged
10.4.5 Flowchart Structure

In order to obtain good program structure, there should be a single entry and exit point to and from all process blocks, as illustrated in the complete flowchart. Loops should therefore span the main flow between symbols, and not connect into the side of a process symbol, as is sometimes seen. Terminal conditions have a single entry or exit point. Decisions in assembly programs only have two outcomes; branches set, giving two exits. Loops back should always occur to the left of the main flow, and loops forward to the right of the main flow, if possible. For the main flow down the page, the arrowheads may be omitted or formal flow is clearly implied. Connections between pages are sometimes used in flowcharts, shown by a circular labeled symbol. Connections between pages are sometimes used in flowcharts, shown by a circular labeled symbol. It is recommended that such connections be avoided; it should be possible to represent a well-structured program on a set of separate flowcharts, each of which should fit on one page. An outline flowchart should be devised for the overall sequence, and then each process detailed with a separate flowchart, so that each process can be ideally 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 ideally 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 into source code statements. A well-structured outline flowchart is more useful than an overly complex single page. The depth of subroutines can be increased to any required depth, depending on the stack size of the system. Smaller PICs tend to have an 8-level deep hardware stack, which means that only eight levels of subroutine are allowed.

10.4.6 Structure Chart

The structure chart is another method for representing complex programs (see Fig. 10.7). Each program block is placed in a hierarchical diagram, which relates to the rest of the program. The structure chart shows the relationship between different parts of the program, allowing for easy debugging and modification. Subroutines can be 'nested' to any required depth, depending on the stack size of the system. Smaller PICs tend to have an 8-level deep hardware stack, which means that only eight levels of subroutine are allowed.

![Figure 10.7] Structure chart
Application Design

This technique is most commonly used in data processing in business applications running on larger computer systems.

The program shown in the structure diagram has four levels. The main program calls subroutines to initialise, process inputs and process outputs. The input processing routine in turn calls Sub1 and Sub2. The output processing only requires Sub3, but Sub2 calls Sub4 and Sub5 at the lowest level. At this level, three stack locations will be used up.

10.4.7 Pseudocode

Pseudocode is a text form of the program design. The main operations are written as descriptive statements which are arranged as functional blocks. The structure and sequence are represented by suitable indentation of the blocks, as is the convention for HLLs. A pseudocode version of MOT1 is shown in Fig. 10.8.

The pseudocode version of the program is a high level style similar to IF … THEN to describe the selection in the program. It has the advantage that no drawing is required. It can start as a brief outline, and be developed in stages ready to be translated into assembly code. It can be left as the outline code or be translated into assembly code. The pseudocode can be left as the input code as the base of program comments, or replaced, whichever suits the programmer. Although used here to represent an assembler program, pseudocode is probably the more suitable for developing C programs for application on the more powerful PIC microcontroller.

Figure 10.8 MOT1 pseudocode
10.5 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 text and debugging package.

10.5.1 Flowchart Conversion

The program design method should be applied so as to 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 more tricky 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 pre-defined macros) are available which combine 'test', 'skip' and 'goto' instructions to provide equivalent to conventional conditional branching.

The representation of the program with different levels of detail is illustrated in Fig. 10.9.

Figure 10.9(a) shows the process in detail so that each process box converts into only one or two lines of code. This may be necessary when learning the programming system. Later, when the programmer is more familiar with the language and the standard processes which tend to occur, such as simple loops, then a more condensed flowchart may be used, such as Fig. 10.9(b), where the loop is concealed within the 'delay' process. As we have seen above, this process can also be hidden in a separate, reusable, software component, a subroutine. The corresponding source code fragment is shown in Table 10.3.

![Figure 10.9 PIC program branch flowcharts. (a) Detailed Flowchart; (b) Outline Flowchart.](image-url)
Table 10.3: PIC program branch code fragment

<table>
<thead>
<tr>
<th>Branch Program Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSF PortA,0 ; Set Output</td>
</tr>
<tr>
<td>MOVLW 0FF ; Set Count Value</td>
</tr>
<tr>
<td>MOVWF count1 ; Load Count</td>
</tr>
<tr>
<td>back1 DECFSZ count1 ; Dec. Count &amp; Skip if 0</td>
</tr>
<tr>
<td>GOTO back1 ; Jump Back</td>
</tr>
<tr>
<td>BCF PortA,0 ; Reset Output</td>
</tr>
</tbody>
</table>

Another limitation in PIC 16XXXX 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 CISC processors. This problem is overcome to some extent by the availability of the destination register option with the store processing operation. Nevertheless, the advantage of simplicity when learning PIC programming offsets these limitations, especially when learning assembly programming for the first time!

Another limitation in PIC 16XXXX 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 CISC processors. This problem is overcome to some extent by the availability of the destination register option with the store processing operation. Nevertheless, the advantage of simplicity when learning PIC programming offsets these limitations, especially when learning assembly programming for the first time!

We can see from the above examples that the software design techniques should be applied in a way which suits the application, the language and the level of expertise of the programmer.

10.5.2 MOT1 Source Code

The program source code for the MOT1 program is given in Program 10.1. The program produces the PWM output by toggling RA0 with a delay. A register labelled ‘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 back to zero. This roll-under at the low end of the scale is prevented in a similar way.

Using source code editing conventions like this is not obligatory, but consistent layout and presentation improves program modularity and makes it easier to understand. Unfortunately, there is no generally accepted convention for assembler source code presentation.
Program Implementation

Program 10.1

MOT1 source code

; *****************************************************************
; 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 16F84

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

; Initialise .......................................................
```assembly
; Application Design
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

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
```

## 10.6 Source Code Documentation

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. Elements 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 standardised, especially in commercial products. 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 author's name, organisation, date, and a program description is 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 some variation in the syntax required for each PIC chip type. The processor type may be specified in the header block, or in an assembler directive, or by selecting the processor in the MPLAB development system options.

```plaintext
10.6 Source Code Documentation

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. Elements 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 standardised, especially in commercial products. 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 author's name, organisation, date, and a program description is 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 some variation in the syntax required for each PIC chip type. The processor type may be specified in the header block, or in an assembler directive, or by selecting the processor in the MPLAB development system options.
```
Target hardware details such as input and output pin allocation is useful, and the design clock speed needs to be specified in programs where code execution timing is significant. Programmer settings which enable or disable hardware features such as the watchdog timer, power-up timer and code protection should also be listed.

The general layout of the source code should be designed to make the structure clear, with subroutines headed with their own brief functional description. Blank lines should separate the functional program blocks, that is, instructions which 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.

**Summary**

- The application requirements and target performance specification should be clearly stated as the first step in software design.
- A block diagram should be used to outline the hardware, components selected and a circuit designed.
- Programs consist of statements which allow sequence, selection and iteration.
- The software algorithm should be represented with a suitable software design aid, and elaborated until sufficiently detailed to translate into source code.
- Flowcharts should be structured, using separate charts to expand the processes in the higher level chart.
- Flowcharts can be constructed from symbols representing terminals, processors, tests/decisions, and subroutines.
- Other methods for software design are pseudocode and structure charts.
- Source code should be fully commented for future reference, maintenance and modification.

**Questions**

1. Explain how PWM offers an effective way of controlling DC loads from a single digital output.
2. Explain briefly the role of the block diagram and flowchart in creating the final hardware and software design for an application.
3. State the three basic operations that make up a microcontroller program, how they are represented in a flowchart, and give an example of how each is implemented in PIC assembler code.
4. Explain briefly the role of the subroutine in structured programming, and why it is generally desirable to use them.
1. Compare the source code for MOT1 with the flowchart in Fig. 10.5, and the pseudocode in Fig. 10.8, and check that they correspond.

2. (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 high outputs will be used to enable the motor in each direction. Investigate and select a suitable component to provide the interface to a small DC motor.

   (b) Construct a flowchart for a PIC program which will allow the user to turn the motor on and off from a single active low input, but only allow the direction to be changed when the motor is off. Produce a logic table, outline and detail flowcharts, and write the code observing the recommendations for source code documentation.

3. (a) Devise a set of structural flowcharts for making a 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.
Chapter 11
Program Debugging

11.1 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, which analyses the source code line by line, and converts the instructions...
 mnemonics into the corresponding binary codes for loading into the chip program memory, as
PROGNAME.HEX. Only valid statements as defined in the PIC instruction set (see Appendix A)
will be recognised and successfully converted. Assembler directives are also included, but these
are not converted to machine code.

Before starting a project, create a folder to keep the project files in, named, for example,
'Motor'. In MPLAB, the source code is created in an edit window opened by clicking the
'New File' button. Type in the file name and save it immediately to the application folder as
MOT1.ASM or any file name with an ASM extension. At the same time, create a backup
version of the file on a different drive (floppy, network or removable drive).

When the program has been entered and saved, the project menu item 'Quickbuild' (or
equivalent other MPLAB terminology) will assemble a single file. If required, a project can be
created, so that the MPLAB setup can be saved between sessions. 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.

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 errors which have been found. The error messages are saved in a
text file PROGNAME.ERR, and displayed when the assembler is finished.

For demonstration purposes, deliberate errors were introduced into the example program
MOT1.ASM (Program 10.1), and the error file MOT1.ERR generated by the assembler is listed
in Table 11.1. 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 - due to the presence of
the error itself, the assembler may be misled as to the actual error. Consequently, the message
generated is not always 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 initialisation using
page selection is more complicated to use.

Table 11.1
Selected error messages from assembly of MOT1

<table>
<thead>
<tr>
<th>Error Type</th>
<th>File Name</th>
<th>Line No.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warning</td>
<td>MOT1.ASM</td>
<td>24</td>
<td>Found directive in column 1 (PROCESSOR)</td>
</tr>
<tr>
<td>Warning</td>
<td>MOT1.ASM</td>
<td>44</td>
<td>Use of this instruction is not recommended.</td>
</tr>
<tr>
<td>Message</td>
<td>MOT1.ASM</td>
<td>56</td>
<td>Using default destination of 1 (file).</td>
</tr>
<tr>
<td>Warning</td>
<td>MOT1.ASM</td>
<td>58</td>
<td>Found label after column 1 (DEC).</td>
</tr>
<tr>
<td>Error</td>
<td>MOT1.ASM</td>
<td>71</td>
<td>Symbol not previously defined (Timer).</td>
</tr>
<tr>
<td>Error</td>
<td>MOT1.ASM</td>
<td>73</td>
<td>Symbol not previously defined (again1).</td>
</tr>
<tr>
<td>Error</td>
<td>MOT1.ASM</td>
<td>92</td>
<td>Expected (END).</td>
</tr>
<tr>
<td>Error</td>
<td>MOT1.ASM</td>
<td>94</td>
<td>Symbol not previously defined (again2).</td>
</tr>
<tr>
<td>Error</td>
<td>MOT1.ASM</td>
<td>96</td>
<td>Symbol not previously defined (again3).</td>
</tr>
</tbody>
</table>

Halting build on first failure as requested.
BUILD FAILED
11.2 Logical Errors

When all syntax errors have been eliminated, the program will assemble successfully, and the hex file 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 which 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.

11.2.1 Simulation

Conventional microprocessor system hardware varies between each application because they are built from discrete chips. The configuration of the CPU, memory and I/O chips is designed to suit the application. Therefore, the CPU itself can normally be simulated, not the whole system. Some logical errors can only be detected by running the program on the actual hardware, and testing for the correct output in response to the specified inputs. In some systems, it is possible to download the program in SIMM form, so that the computer can be easily corrected. In other systems, it is impossible to download the program in SIMM form, so that the computer can only be corrected by programming an EPROM memory chip and actually fitting it in the hardware. This could mean erasing and reprogramming the EPROM repeatedly, in order to get the program right. This can obviously be quite time-consuming, and can be avoided if possible.

If the program is being tested in the hardware, it may also be difficult to track down exactly what the problem is. The program should be very simple, with only one instruction in each routine. If the system being tested has some debugging capability, and some means to display the data, the program will need to be designed for this purpose. However, if the system has no debugging capability, the program will need to be designed to test the system itself, which may not be as easy as the application program is and running correctly. Alternatively, an in-circuit emulator (ICE) can be used to test hardware testing (see Section 11.6.2), but these are relatively expensive.
**180**

**Program Debugging**

The advantage of the microcontroller is that the design of the chip is fixed, so a full simulation model can be provided for each. This allows the program to be tested for logical errors before downloading. The simulation package (MPSIM) allows windows to show the source code, machine code, registers, simulated input, time checks and so on. The program can then be run, stopped and stepped through one instruction at a time. Source level debugging shows the current execution point on the source code itself. Breakpoints allow the code to be stopped at any point, so that the registers and outputs can be inspected.

The software simulator is therefore a very useful tool for program testing. One of the advantages of the PIC range is that the development system, including the assembler and simulator, has always been provided free by Microchip to encourage the market for its chips.

### 11.2.2 Program Testing

The simulator must model the operation of the selected microcontroller as accurately as possible. The user must be able to provide the inputs which would occur in the actual system, and be able to monitor the effect on relevant registers. The program will need to be started, stopped or 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 where the application is in use.

#### Inputs

The simplest method of simulating inputs is the asynchronous stimulus. Single bit inputs are changed to simulate values that would occur at the required time while the program is executed in single-step mode. The other method is to use a stimulus file, which automates this process, allowing the same test sequence to be input every time the program is run in the simulator. This will save time while debugging more complex applications.

#### Outputs

For monitoring the outputs, various options are available. All the file registers may be displayed simply as hex numbers in a table. The SFRs can be displayed separately, in a choice of formats; hex, binary and decimal. Selected registers can be viewed in a 'watch' window, using the register labels from the source code.

#### Timing

Program timing can be checked using the stopwatch. First, the clock rate must be entered so that the actual timing can be predicted. The time taken for instructions to execute can then be checked. For example, the period of the waveform output by MOT1.

### 11.2.3 Testing in MPLAB 6

To test the program, the source code must be assembled or the project built. Ensure that the correct processor is selected, via the 'Configure', 'Target Device' dialogue. Select the IRAM for the MOT1 project. At the same time, it is advisable to set the chip fuse via 'Configure'.
Logical Errors

181 'Configuration Bits', selecting oscillator = RC, watchdog timer off, power-up timer on and code protect on.

Now select 'Debugger', 'Select Tool' then 'MPLAB SIM'. The debugging tools should then appear in the toolbar:

- **RUN** executes the program.
- **HALT** stops the program with the current execution point indicated.
- **STEP INTO** executes program one instruction at a time.
- **STEP OVER** single-step current routine, but execute subroutines at full speed.
- **RESET** starts again at the top of the program.

The program can now be run and stopped to make sure the simulator is working. When halted, the current execution point is indicated. We now need to set up the simulator so that the relevant information is displayed so that correct program function can be confirmed or logical errors corrected. A typical display for MOT1 is shown in Fig. 11.1.

### 11.2.4 Setting up MPLAB 6

Simply running the program in the simulator does not generally provide enough information to confirm correct operation. Single stepping allows the program to be executed one instruction at a time; the registers can then be checked for the correct results.

![MPLAB 6 debug screen for MOT1](image)
View Outputs
The file registers can be displayed in simple hex by selecting 'View', 'File Registers' but this is of limited use. Instead, select 'View', 'Special Function Registers', Port A is displayed in binary. Also select 'View', 'Watch' and display the labelled Count and Timer registers using the 'Add Symbols' dialogue.

Assign Inputs
We now need to set up the simulator inputs. Select 'Debugger', 'Stimulus' and the 'Pin Stimulus' tab in the Simulator Stimulus dialogue. Add 'Row' and click on the 'Pin' cell and select 'RA4'. Click on the 'Action' cell and select 'Toggle' mode. Type 'Run' in the 'Comment' column. Repeat this for RA2 (Up) and RA3 (Down). Hide the blank columns. When the 'Fire' button is pressed, the input will change over, but only after the next step in the simulation.

Set up Stopwatch
Select 'Debugger', 'Settings' and the 'Clock' tab. Set the oscillator frequency to 100 kHz. Select 'Debugger', 'Stopwatch' to display the stopwatch. Arrange the windows for best visibility.

11.2.5 Testing MOT1
Set all the inputs high by clicking on each fire button and stepping once. Reset the program and single step through the initialisation sequence. Check that the Port A input bits are set high in the SFR window. Step through the initialisation sequence to 'Stop', and check that 'Count' is set to a low value (0). Now clear input RA4 by hitting the Fire button, and step to the start of the delay sequence. Note that single stepping is not helpful, so we will use breakpoints to test the next sequence. Set a breakpoint at the BCF instruction on the line:

```
COUNT := COUNT - 1;  ;
```

Now, the Port A bits should be low in the SFR window. Set a breakpoint at the BSF instruction on the line:

```
COUNT := COUNT + 1;  ;
```

Now, the Port A bits should be high in the SFR window. Reset the program and run to the first breakpoint. Zero the stopwatch and 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 approximately 50% MSR. At the same time, check that RA0 is toggling each time, which indicates that the output PWM signal is present.

Now, clear 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 use breakpoints to test the main loop. Set a breakpoint at the BCF instruction on the line:

```
COUNT := COUNT - 1;  ;
```

Now, the Port A bits should be low in the SFR window. Set a breakpoint at the BSF instruction on the line:

```
COUNT := COUNT + 1;  ;
```

Now, the Port A bits should be high in the SFR window. Reset the program and run to the first breakpoint. Zero the stopwatch and 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 approximately 50% MSR. At the same time, check that RA0 is toggling each time, which indicates that the output PWM signal is present.

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, we can use the 'step over' option to move through the subroutine at full speed. This allows each program block to be tested separately. However, as the delays in MOT1 are not subroutines, we will use 'step into' to test them as well as a 'step over' test. Alternatively, the process can be shortened by changing the count register value to a low value either in the source code or simulator, or commenting out the call to the subroutine.
11.2.6 Stimulus File
Testing the program in the simulator can be automated by use of a stimulus file. This allows
the state of an input or file register to be changed at particular step in the program. The same
test sequence can then be applied each time the simulator is run, making the testing process
easier and saving time in complex applications. In MPLAB, select 'File', 'Stimulus' and the
'Simulate' tab and follow the help guide to create the necessary files.

11.2.7 Tracing
Tracing is another way of checking the program sequence. As each instruction is executed, the
changes in internal registers are logged to a file. In MPLAB, run the program, and select
'View', 'Simulator Trace' to see the trace record up to that point. Scroll down the bottom of
the list to view the most recent events. Note that the 'probe' columns are not used in the software
version 2.00.4. These should be enabled in the emulator settings.

11.3 MPLAB Tools
The most commonly used windows in the simulator are described below:

11.3.1 Edit Window
The Edit window is used for creating and editing the program. When one of the assembly
options is selected, the source code in the current edit window is converted to machine code
(PROGNAME.HEX). If syntax errors are detected, the error file is displayed automatically,
and the number of the incorrect line in the source code is given. When the program is stopped,
the current instruction is indicated in the source code window. Breakpoints can be inserted by
clicking on the line required; they are then indicated in the left margin.

11.3.2 Special Function Register Window
The SFRs are displayed in a number of different formats. It is particularly useful to see the
contents in binary, as many of the SFRs are bit-oriented. The most commonly needed are:

- WREG Working register (not an SFR)
- TMR0 Timer zero hardware counter
- PCL Program counter
- STATUS Flag register, zero flag = bit 2
- PORTA Data direction register for PORTA
- PORTB Data direction register for PORTB

11.3.3 Watch Window
A watch window allows selected registers to be displayed, and only those of interest in a
particular application. In MPLAB, the 'Add SFR' button is used to display the SFRs, and 'Add
label' to display the registers identified by a user label. The numerical format, among other
characteristics, can be changed by right-clicking on the window and selecting 'Properties'.

11.3.4 Edit Window
The Edit window is used for creating and editing the program. When one of the assembly
options is selected, the source code in the current edit window is converted to machine code
(PROGNAME.HEX). If syntax errors are detected, the error file is displayed automatically,
and the number of the incorrect line in the source code is given. When the program is stopped,
the current instruction is indicated in the source code window. Breakpoints can be inserted by
clicking on the line required; they are then indicated in the left margin.

11.3.2 Special Function Register Window
The SFRs are displayed in a number of different formats. It is particularly useful to see the
contents in binary, as many of the SFRs are bit-oriented. The most commonly needed are:

- WREG Working register (not an SFR)
- TMR0 Timer zero hardware counter
- PCL Program counter
- STATUS Flag register, zero flag = bit 2
- PORTA Data direction register for PORTA
- PORTB Data direction register for PORTB
- TRISB Data direction register for PORTB

11.3.3 Watch Window
A watch window allows selected registers to be displayed, and only those of interest in a
particular application. In MPLAB, the 'Add SFR' button is used to display the SFRs, and 'Add
label' to display the registers identified by a user label. The numerical format, among other
characteristics, can be changed by right-clicking on the window and selecting 'Properties'.

MOT1 is now tested. Save the program if it has been corrected, and save the project.
Program Debugging

11.3.4 Simulator Stimulus

To test the MOT1 program, the inputs which enable the motor and change the speed must be simulated. The simplest method is to use the 'Pin Stimulus' option in the simulator stimulus window. Each input is assigned a row in the table, and normally set to 'toggle' mode. When the table is complete, it can be saved as a separate file.

11.3.5 Stopwatch

The stopwatch window allows the simulated chip clock to be set to the frequency that will be used in the actual hardware. For simulating MOT1, the value 100 kHz is entered. The total number of instructions executed and the corresponding elapsed time are then correctly displayed. These can be reset to zero at any time. When testing MOT1, the time taken for the delays and overall output period can be checked. This type of check can also be carried out using a break point.

11.3.6 Program Memory

If the 'Program Memory' is selected from the 'View' menu, the program can be viewed in different forms. The 'Opcode Hex' option shows the raw hex code. 'Machine' displays the disassembled program. This means the simulator has taken the machine code program and converted it back into machine instruction mnemonics, to confirm that it is correct. The hex machine code (Opcode) for each instruction alongside the program memory location (Address) at which it is stored. Selecting the 'Symbolic' display restores the original labels used. When the simulation is run, the execution point is also shown in this window.

11.4 Test Schedule

Using the simulator, the program functions can be tested against the specification. The specification outlines should be converted into a test procedure which will test all its functions, and all possible unexpected input sequences, especially where these are generated by an operator. A test procedure for MOT1 is suggested in Table 11.4.

The test procedure looks very detailed when written down, but in practice it does no more than test all the features of MOT1. The software product needs to meet the specification only once, in a prototype hardware. Once the software is proved, the hardware in the production units can be tested in conjunction with software which is known to be correct.

11.5.1 Typical Logical Errors

It is difficult to anticipate exactly what kinds of logical errors will arise, as they are usually the result of assumptions, but the following types of errors are typical:

Port Initialisation

If a port does not appear to respond to input operations, check that the initialisation is correct.
## Test Schedule

Table 11.2 Simulation test schedule for MOT1

<table>
<thead>
<tr>
<th>Test Action</th>
<th>Required/Performance</th>
<th>Fault/Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Initialise</td>
<td></td>
<td>All inputs inactive</td>
</tr>
<tr>
<td>2. Start</td>
<td></td>
<td>Waiting for run enable</td>
</tr>
<tr>
<td>3. Enable run</td>
<td></td>
<td>Waiting for run enable</td>
</tr>
<tr>
<td>4. Disable run</td>
<td></td>
<td>Waiting for run enable</td>
</tr>
<tr>
<td>5. Select increment</td>
<td></td>
<td>Count increments not 00, 01, 02, 03</td>
</tr>
<tr>
<td>6. Select decrement</td>
<td></td>
<td>Count increments not 00, 01, 02, 03</td>
</tr>
<tr>
<td>7. Restart</td>
<td></td>
<td>Count increments not 00, 01, 02, 03</td>
</tr>
<tr>
<td>8. Program reset</td>
<td></td>
<td>Count increments not 00, 01, 02, 03</td>
</tr>
</tbody>
</table>

Tested: MBates Date: 4/11/03
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.

Bit Test and Skip
Obtaining the correct sequence of operations in the program depends on these instructions. Make sure the skip condition is correct, clear or set, as this is easy to get 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. Note that the stack overflow/underflow warning which indicates that CALL and RETURN are not matched correctly is disabled by default in MPLAB. Stack error detection can be enabled via the 'Debugger', 'Settings', 'Break Options'.

11.4.2 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 cannot be 100% realistic, and its limitations need to be taken into account in testing the real system. The following is an example of the kind of error that might easily be missed, but seriously affects the operation of the application, and could compromise safe operation of the real system if a more powerful motor were used.

The simulation showed that after initialising Port A, with bit zero (RA0) set as output, this data bit goes high by default (all port bits default to logic ‘1’). The motor will therefore come on, even if the enable input has not yet been operated. This is obviously a major flaw, which can be fixed by following the port initialisation instructions with one to clear the motor output bit. This would result in a very short pulse to the motor at the start of the program. If this caused a problem to the motor drive, an alternative fix would be needed. For example, an external circuit which ensured that the motor was not powered up until the controller had been started.

11.5 Hardware Testing
When the program has been fully debugged in the simulator, it can be downloaded to the chip, which is then placed in the target system. However, the target hardware layout and connections should be checked and tested separately before inserting the chip. The board should be carefully inspected for correct assembly: solder bridges between tracks and dry joints are common faults. Connections can be tested out with a continuity tester and checked against the circuit diagram.
Before fitting the chip, it is a good idea to apply power and check the rest of the circuit, but make sure the components connected to the chip output can be safely powered up with an open-circuit input. For example, in the MOT1 circuit, the 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 (Vdd and Vss) of the chip should also be checked, an incorrect connection of the supply is likely to damage the ICs.

If all is well, switch off the power and fit the chip using a suitable tool. Anti-static 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. It will be obvious for more than a few seconds, the chip will probably be destroyed, but you may be able to rescue it if you catch it before the smoke appears.

11.5.1 In-Circuit Program Testing
Connect an oscilloscope to the output. No power up, there should be no output from MOT1. When the ‘Run’ button is pressed, the default output sequence would 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 operates at the minimum and maximum values, and does not ‘roll over’ from zero to full speed in one step. Note that the program algorithms do not give an MSR of 100% or 0%, but steps 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 which would need to be considered are input switch bounce, variation in component performance (check specifications), dynamic operation of motors, minimum current required to make the motor run, and so on. More complex applications are likely to have more potentially incorrect input conditions and component-related problems, but the test sequence should ideally anticipate all possible fault modes (not easy!). If the circuit is to be used in a commercial application, then a formal test schedule would be needed, and the performance certified as correct to the product specification.

A basic test schedule for the MOT1 program running in a PIC 16F84 is given in Table 11.3. 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.

11.5.2 In-Circuit Emulator
The emulator allows the software to be tested without the target hardware. However, the execution speed may be slower than the real hardware, and input conditions due to real components such as control signals from other devices may not be simulated. This limits the accuracy of the test.

On the other hand, an IC allows the hardware to be tested without the microcontroller (or microprocessor). It allows a host computer with a special simulation tool attached to replace the
### Program Debugging

#### Table 11.3  Test schedule for MOT1 application board

<table>
<thead>
<tr>
<th>Batch No.</th>
<th>Specification Number</th>
<th>Test</th>
<th>Required Result</th>
<th>TC</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Inspected</td>
<td>No faults observed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Power Up</td>
<td>Motor off, Output low</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>!RUN operated</td>
<td>Motor on, 50% MSR, Frequency ∼30 Hz</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>!UP operated</td>
<td>MSR increases &gt;99%, Step Resolution &lt;1%</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>!DOWN released</td>
<td>MSR decreases &lt;1%, Step Resolution &lt;1%</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>!MCLR pulsed</td>
<td>Speed reset 50% MSR, Restarts program</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>!RUN released</td>
<td>Motor Off, Output low</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tested by: [Name]
Signed: [Name]
Date: [Date]

Target processor (see Fig. 11.2). A header connector with the same pin out as the processor is connected to its socket on the board. The pod provides the specific processor emulation hardware, which is able to represent that particular processor at full speed. The program can then be tested with the actual hardware, giving more realistic results.

![In-circuit emulator set-up](image-url)
Professional development systems use this technique as it allows the complete application hardware system to be tested as it interacts with a virtual processor. This type of equipment is usually regarded essential in the commercial development environment, but it is relatively expensive. Individuals, educational users and small companies, with limited development budgets, will find the low-cost IDE (see Section 7.7.2) option a very useful alternative.

Summary

- There are two main types of error that can occur in source code, syntax and logical errors.
- Syntax errors are invalid statements which are detected by the assembler; error messages are generated to assist debugging.
- Logical errors are mistakes in the program design or implementation, which can be detected by using the simulator to test the program operation.
- MPLAB is an IDE featuring an editor, assembler, simulator, emulator support and downloading software.
- A typical MPLAB simulation setup would use the source code, program memory, SFR, watch, trace, stopwatch and stimulus windows.
- Logical errors can be identified using step into, step over and breakpoints functions with register monitoring.
- The ICE allows the software to be tested at full speed on the target hardware.

Questions

1. Explain briefly the difference between syntax and logical program errors, and how they are detected.
2. Explain the difference between ‘simulation’ and ‘emulation’ in the PIC system. Why does simulation need some extra hardware, and simulation does not?
3. How are the following used in program debugging: single stepping, breakpoint, pin stimulus, watch window?
4. 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.

Activities

1. (a) In MPLAB, open a source file edit window, enter the source code for MSW1 and assemble it. 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.
In MPLAB, create a project MOT1, assign MOT1.ASM to the project and test the program using the setup suggested in Section 11.4. Use the test schedule in Table 11.2 to check the program operation and confirm that it meets the specification.

Modify the application by eliminating the inputs and generating a PWM output whose MSR increments automatically, once per output cycle. Allow the incremented count register to roll over from 100% MSR to restart at 0% MSR. Devise a test schedule to confirm correct operation. Download to a PIC chip and run the program in the hardware, monitoring the output with an oscilloscope. What output should you see? Predict the time taken (to the nearest half second) for a complete cycle from 0 to 100% MSR.

Answer: 8.5 s
We now come to the stage where we need to look at the techniques available for designing and building our PIC circuits. Circuit design, simulation and layout software has developed to the point where powerful packages are now available at a reasonable cost. Current software allows the circuit to be drawn, tested by simulation, and the circuit circuit (list of components and connections) produced. This is then exported into a PCB design package where the circuit is laid out on screen, this can be plotted onto a masking sheet, or a file generated which can be used to automatically produce a PCB. Currently, a popular choice in the UK is Proteus™ from Labcenter Electronics. This consists of two main parts, ISIS™ for circuit design and schematic capture, and ARES™ for PCB layout. However, the design, layout and construction of PCBs will not be discussed in detail, because it is not within the scope of this book.

12.1 Hardware Design

Traditionally, circuits have been designed as sketches on paper, and a final version produced by a draughtsman. This relied heavily on the experience of the electronics engineer to be able to predict the circuit performance from theoretical knowledge and practical experience. Numerous prototypes would typically be needed to arrive at a working solution. This process has, since the development of increasingly powerful desktop computers, been radically improved. The designer still has to come up with the original ideas, but the proposed circuits can now be quickly sketched on the screen, and a working design produced without a pencil touching paper or any component being inserted on a prototype board. The design cycle is much faster and time is spent refining concepts rather than testing. ECAD has become almost a vitally important tool for the electronics engineer, just as CAD has become for the mechanical engineer. This is because it is possible to design, simulate and produce a PCB within a single software package. The circuit design can be tested for correct function by software simulation.
which incorporates mathematical models for the behaviour of each component, and their interaction. Libraries of microprocessor and microcontroller models, as well as interactive on-screen components, are now included; a simulation be driven in-situ; the application program attached to the simulation input, and the program tested by operating the on-screen inputs, such as switches and displays (LED and LCD) or operate animated output devices such as relays and motors.

As an example, a simple circuit design created in the schematic capture package ISIS is shown in Fig. 12.1. It is an electronic dice board with a push button, seven-segment display and buzzer controlled by a PIC 16F84. It can be programmed to display a random number between one and six when the button is pressed.

![Circuit design for DICE board.](image)

When a suitable program is assigned to the PIC in the simulation, 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 make a sound, the waveform can be displayed on a virtual oscilloscope, and even reproduced from the PC audio output, if the CPU is fast enough.

### 12.2 Hardware Construction

First, we will look briefly at some traditional techniques suitable for building one-off boards and prototypes. A 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. The DIZI board (display and buzzer with interrupt) has a seven-segment display and an audio output for some simple display and sound applications (see Appendix B).

### 12.2.1 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 fibreglass board with a layer of copper on one side. The circuit connections are made by photographically transferring a pattern of conducting tracks and pads for the component connections onto the copper. For complex circuits, such as a PC motherboard, multilayer boards are used to accommodate the large number of parallel data connections.

The layout for a simple PIC circuit is shown in Fig. 12.2. It has a PIC 16F84, push button, seven-segment display, buzzer and associated components and can be programmed to operate as an electronic dice, generating a random number between one and six. The pattern of the copper tracks is shown, as well as the “silk screen” printing which will be applied to the component side of the board to show where to place the components.

![Figure 12.2 - PIC dice board layout](image)

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 photo-sensitive material becomes soluble and is removed by a water-solvent, exposing the copper below. This is then developed (etched) in an acid bath, leaving behind the copper layer where it was protected by the mask during exposure. The components are then fitted to the silk screen side of the board, and the leads and pins soldered to the pads. Once the layout has been designed, it can be used for batch production of the application hardware. Specialist companies are often used to manufacture the boards direct from the file output of the PCB design software. The final PCB-based product is shown in Fig. 12.3.

Even with the current 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.
12.2 Breadboard

A common method of constructing prototype circuits uses breadboard (plugboards) to wire up prototype circuits. The connecting wires are pushed into an array of interconnected sockets, allowing circuits to be quickly built and modified. A typical circuit, which will be discussed later, is illustrated in Fig. 12.9. The circuit in the breadboard is shown in Figs. 12.9c and d. The breadboard is a grid of interconnected sockets, allowing for easy modification.

The breadboard plugs into a center line of the board, where the ICs are inserted. Typically, four contacts connect IC pins. At each side of the board, there are rows of contacts for power supplies. Some types of breadboard can be supplied in blocks that plug together to accommodate larger circuits, or are mounted on a base with built-in power supplies.

Breadboard circuits can be built quickly, with no special tools required. However, the connections are relatively unreliable, so bad connections are likely in more complicated circuits. Therefore, method of producing prototype circuits with more reliable soldered connections is useful.

12.2.3 Stripboard

Stripboard is a perforated method which eliminates the need for special tools or chemical processing. The components are punched out, and the traces are placed on the board in strips. The components are inserted into the holes and soldered to the traces using wire links placed on the component side and soldered to the tracks on the copper side. The tracks must then be cut when the same chip is used for separate connections in the circuit. The components are generally placed across the tracks, so that each pin connects with a separate contact.
Hardware Construction

Figure 12.4 Breadboard layout.

<table>
<thead>
<tr>
<th>Function</th>
<th>Label</th>
<th>Pin</th>
<th>Pin</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O Port A bit 2</td>
<td>RA2</td>
<td>18</td>
<td>RA1</td>
<td>Port A bit 1</td>
</tr>
<tr>
<td>I/O Port A bit 3</td>
<td>RA3</td>
<td>17</td>
<td>RA0</td>
<td>Port A bit 0</td>
</tr>
<tr>
<td>I/O RA4 or timer input</td>
<td>/T0CKI</td>
<td>16</td>
<td>OSC1/CLKIN</td>
<td>Crystal or RC oscillator</td>
</tr>
<tr>
<td>In Master clear (Reset)</td>
<td>!MCLR</td>
<td>15</td>
<td>OSC2/CLKOUT</td>
<td>Crystal circuit (if used)</td>
</tr>
<tr>
<td>In Supply</td>
<td>VSS</td>
<td>14</td>
<td>Supply</td>
<td>+5 V</td>
</tr>
<tr>
<td>I/O Port B bit 0</td>
<td>RB0/INT</td>
<td>13</td>
<td>RB7</td>
<td>Port B bit 7 (interrupt)</td>
</tr>
<tr>
<td>I/O Port B bit 1</td>
<td>RB1</td>
<td>12</td>
<td>RB6</td>
<td>Port B bit 6 (interrupt)</td>
</tr>
<tr>
<td>I/O Port B bit 2</td>
<td>RB2</td>
<td>11</td>
<td>RB5</td>
<td>Port B bit 5 (interrupt)</td>
</tr>
<tr>
<td>I/O Port B bit 3</td>
<td>RB3</td>
<td>10</td>
<td>RB4</td>
<td>Port B bit 4 (interrupt)</td>
</tr>
<tr>
<td>5V</td>
<td>DD</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 12.5 16F84 pinout.

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 drafting pen can be used to draft the layout, if necessary, but a reasonably experienced constructor can build the circuit directly onto the board, with maybe some additional wastage of board area. It is also possible to work out the layout using a simple drawing package, or the drawing tools in a wordprocessor.
12.6 Prototype Hardware

Figure 12.6 Stripboard connections.

Figure 12.6 shows how the simple PIC circuit can be laid out for construction on stripboard using general purpose drawing tools, such as those provided in 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 at 0.1; 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 the cursor movement from disrupting the drawing, and the whole diagram can be re-positioned on the page if required.

12.3 Demo Board

A circuit will be now designed, and a set of programs written, to illustrate the hardware design process and programming principles discussed in previous chapters. The DIZI board will allow the user to experiment with the various features of the PIC hardware and programming techniques within a single hardware module.

12.3.1 Hardware Specification

The microcontroller demonstration board will be suitable for demonstrating a wide range of processes incorporating display, audio, counting, timing and interrupt operations. The board will support a single-digit seven-segment display, a single low-power audio transducer, and directly connected four-bit parallel input. Two input push buttons should be available; one to simulate input events to be counted, the other to simulate an external interrupt input. Timed events should be measured or generated with an accuracy better than 1%. The board 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.

### 12.3.2 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 power-buzzed device looked to be a feasible choice, since its power consumption is low. Although the device is specified to operate at a fixed frequency, it was found to be satisfactory in its output response. A miniature dip switch bank will be used for 4-bit input, and miniature push buttons used, to conserve space.

Fourteen I/O pins are required, the PIC 16F84A has only thirteen, so a chip with more I/O could be considered. However, the audio output and interrupt input could use the same I/O pin, because the high impedance of the buzzers 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 are needed per display segment to control the maximum load on the output when all the segments are on.

The I/O allocation for the project is therefore as follows:

**Seven-Segment display Outputs**
- RA0–RA3
- RA4
- RB0 (audio transducer)
- RB1–RB7

**4-Bit switch bank Inputs**
- RA0–RA3

**Push button Input**
- RA4

**Push button interrupt Input**
- RA0 (dual function)

**Audio transducer Output**
- RA0 (dual function)

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 and 5.5 V, so the circuit will be powered from 3 × 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 Fig. 12.3. The inputs and outputs are given the labels which will be assigned in the application programs.

### 12.3.3 Implementation

A circuit for the DIZI board is shown in Fig. 12.8. The PIC 16LF84A drives an active high (common-cathode) low current seven-segment LED display at Port B, RB1–RB7, via a block of 270 R current-limiting resistors. RB0 drives an audio sounder when set as an output, but can also be used to detect the interrupt push button whenever it is 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 output drive is turned off if the push button is pressed, and turned back on when the button is released. This alone prevents the operation of the sounder, but it is doubtful if a resistor would allow the all output pins to be monotonically used. In practice, the only output which is currently used is the audio output. The push button is connected to RA4, which can be used as an external pulse input to the counter/timer register RTCC. It is not used as active low inputs with 10k pull-up resistors, as does the interrupt push button.
Prototype Hardware

RA4 to RB1
RA3 to RA0
RB0
OSC1/2
XTAL 4 MHz

INPUT
INTER
PIC 16F84
PORT A
PORT B
BUZZER
Power on indicator

Seven-segment display

Figure 12.7 Block diagram of DIZI demonstration board.

A breadboard version of the circuit is shown in Fig. 12.9. A stripboard layout for the DIZI board is shown in Fig. 12.10. The detail of the component pin connections has been omitted due to the reduced scale of the illustrations, but this information can be obtained from the component pin out data, when selecting particular components.

The finished stripboard circuit is shown in Fig. 12.11.
Figure 12.9 DIZI breadboard prototype circuit.

Figure 12.10 Stripboard layout for DIZI board.
12.4 Demo Board Applications

A set of programs to run on this hardware is listed below. Selected applications will be
developed and seeded (*) and the reader is invited to investigate the others using the techniques
covered thus far.

**Display**
- FLASH: Flash all segments
- STP1: Step through segments
- MSG1: Message display
- SEC1: One-second clock
- REACT1: Reaction timer

**Sound**
- BUZZ1: Output single tone
- SWEEP1: Sweep tone frequency
- TONE1: Switch tone on/off
- SEL1: Select tone on switches

---

Figure 12.11  DIZ displayed circuit
Demo Board Applications

- GEN1 Audio frequency generator
- MET1 Metronome
- GIT1 Guitar tuner
- SCALE1 Musical scale
- BELL1 Doorbell tune
- 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
  - STORE1 Store a display sequence in EEPROM
  - STORE2 Store a tone sequence in EEPROM
  - LOCK1 Store a code and buzz if matched

12.4.1 Program BUZZ1

A flowchart for the program BUZZ1 is shown in Fig. 12.12. 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/\mu s instruction cycle time, we have seen that the loop itself will take:

\[
255 \times 3 \times 1 = 765 \text{ \mu s}
\]

which will give a frequency of approximately:

\[
\frac{1000000}{765} \times 2 = 650 \text{ Hz}
\]
The frequency is not critical, so we will ignore the additional loop instructions, because they will only make a small difference. The result is well within the audio range, and therefore is suitable. It can be adjusted by simply reducing the count value in the delay loop; 650 Hz is the minimum frequency available. A more precise calculation of the delay loop period 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 (see Program 12.1).

Program 12.1 BUZZ1 source code

; BUZZ1.ASM M. Bates 6/4/99
; ********************************************************
; 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 16F84 ; 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 ***************************************
;
; Initialise (Default = Input) ..........................
MOVLW b'00000000' ; Define Port B outputs
TRIS PORTB ; and set bit direction
GOTO check
;
; Delay Subroutine .....................................
delay MOVLW 0FF ; Standard Routine
MOVWF Count
down DECFSZ Count
GOTO down
RETURN
;
This program will generate a ‘random’ number at the display between 1 and 6 when the input button is 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. A table is required to work out the display digit codes.

First, the allocation of the segments to the pins on the display chip must be established. The segments of the display are labelled from a–g, as shown in Fig. 12.13. They must be lit in the appropriate combinations to give the required digit display; for instance, segments 'b' and 'c' must be lit for the digit '1' to be displayed. A table is useful here to work out the codes required for output to the display (Table 12.1).

[Diagram of a digital circuit with labels for inputs and outputs]

Figure 12.13 Block diagram for DICE1 system.

This display is ‘active high’ in operation. That means ‘1’ at the pin will light or segment. This arrangement is also described as ‘common cathode’, as all the LEDs 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, Fig. 12.14, uses a spare register as a counter which is continuously incremented from 6 to 0. When the button is pressed, the current number is...
Table 12.1. DICE1 display encoding table

<table>
<thead>
<tr>
<th>Displayed digit</th>
<th>RB7</th>
<th>RB6</th>
<th>RB5</th>
<th>RB4</th>
<th>RB3</th>
<th>RB2</th>
<th>RB1</th>
<th>RB0</th>
<th>Hex</th>
<th>(RB0 = 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0C</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>B6</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>9E</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>CC</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>DA</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>FA</td>
<td></td>
</tr>
</tbody>
</table>

DICE1 Setup display outputs

Set count register to 6
Input button on?
Call TABLE
Output code to display
Decrement count and check = 0?

(a) Table
Return with Seven-segment code

(b) Yes
No
reload
start
nexnum
Add count register to program counter

Figure 12.14. DICE1 program flowcharts.
used to select from the table of codes using the method described in Program 9.4. 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 labelled with letters (A, B, C, etc.) to show the path of execution of the loop. The table subroutine is also named 'table' to match the same subroutine start label (see Program 12.2).

Program 12.2 DICE1 source code

; DICE1.ASM M. Bates 6/4/99
; ************************************************************
; Displays pseudo-random numbers between 1 and 6 when a push button is operated.
; Hardware: PIC 16F84 DIZI Demo Board
; Clock: XTAL 4MHz
; Inputs: RA4 : Roll (Active Low)
; Outputs: RB1-RB7 : 7-Segment LEDs (AH)
; MCLR: Enabled
; PIC Configuration Settings:
; WDTimer: Disable
; PUTimer: Enable
; Interrupts: Disable
; Code Protect: Disable
; Set Processor Options........................................
; PROCESSOR 16F84 ; 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 ************************************************
; Initialise (Default = Input)
MOVLW b'00000001' ; Define RB1-7 outputs
TRIS PORTB ; and set bit direction
GOTO reload ; Jump to main program

continued...
### 12.4.3 Program SCALE1

This program will output a musical scale of eight tones. The frequencies for a musical scale from middle C upwards are:

- 262 Hz
- 294 Hz
- 330 Hz
- 349 Hz
- 392 Hz
- 440 Hz
- 494 Hz
- 523 Hz

These can be translated into a table of delay counts which gives the required tone period, since:

\[
\text{Period, } T = \frac{1}{f} \text{(s)}
\]

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 seen in Section 9.1.2 how to calculate the delay time for a loop. Using a formula for the counter 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 elapsed 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 (see Program 12.3).

Instead of a flowchart, the SCALE1 program source code listing has been annotated with arrows to show the execution sequence. This informal method of analysis can be used to check the program logic prior to simulation. The eight tone frequencies are determined 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 a half 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 incremented 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 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. The program is terminated with the SLEEP instruction to stop program execution running into the unused locations following the program.

Program 12.3 SCALE1 source code

```
; SCALE1.ASM M.Bates 6/4/99
; ****************************************************
; Outputs a scale of 8 tones, 255 cycles per tone,
; tone duration of between a half and one second.
; Hardware: PIC 16F84
; XTAL 4MHz, !MCLR to start
; Audio Output: RB0
; Assign Registers **********************************
PCL EQU 02 ; Program Counter
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)

; Initialise Registers
MOVLW B'11111110' ; RB0 set...
TRIS PORTB ; as output
MOVLW 08 ; Set initial value of...
MOVWF TonNum ; Tone Number
GOTO start ; Jump to main program

; Main Loop
count:...
```
```
; Tone Period Table (HalfT)

getdel ADDWF PCL
\[\text{NOP}\]
RETLW D'156'
RETLW D'139'
RETLW D'124'
RETLW D'117'
RETLW D'104'
RETLW D'92'
RETLW D'82'
RETLW D'77'

; Delay for half tone cycle

delay MOVF HalfT,W
\[\text{MOVWF Timer}\]
again DECFSZ Timer
......
\[\text{\uparrow GOTO again}\]
\[\text{\uparrow RETURN}\]
\[\text{\uparrow ......... ....}\]
\[\text{\uparrow ...... ....}\]
\[\text{\uparrow ...... ....}\]
\[\text{\uparrow ...... ....}\]

; Output 255 cycles of tone

note MOVLW D'255'
\[\text{MOVWF Count}\]
cycle BSF PORTB,0
\[\text{CALL delay}\]
\[\text{\uparrow NOP}\]
\[\text{\uparrow NOP}\]
\[\text{\uparrow NOP}\]
BCF PORTB,0
\[\text{CALL delay}\]
\[\text{\uparrow DECFSZ Count}\]
......
\[\text{\uparrow GOTO cycle}\]
\[\text{\uparrow RETURN}\]
\[\text{\uparrow ......... ....}\]
\[\text{\uparrow ...... ....}\]
\[\text{\uparrow ...... ....}\]
\[\text{\uparrow ...... ....}\]

; Main Loop Outputs 8 Tones

start MOVF TonNum,W
\[\text{CALL getdel}\]
\[\text{CALL getdel}\]
\[\text{MOVWF HalfT}\]
\[\text{CALL note}\]
\[\text{\uparrow DECFSZ TonNum}\]
......
\[\text{\uparrow GOTO start}\]
\[\text{\uparrow SLEEP}\]
\[\text{\uparrow ......... ....}\]
\[\text{\uparrow END \; of source code}\]
```
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 light up the segments in the required pattern for each hex digit display:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E

Note that numbers B and D are displayed in lower case on a seven-segment display so that they can be distinguished from 8 and 0 respectively. Use the switch input number to select one of the 16 codes from a seven-segment table.

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 'HI tHErE'. Output a character code table with delay. The number of characters must be set in a counter, or a termination character used.

SEC1 One-Second Timer

An output is displayed which increments exactly once per second, from 0 to 9 and then repeats. A table of display codes is required as in the 'Hex Converter'. A one-second time delay can be achieved using the hardware timer (see Chapter 9) 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 sound, 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, is then displayed as a number 0–9, giving a maximum reaction time of 900 ms.

GEN1 AF Generator

An audio frequency generator outputs frequencies in the range 20 Hz–20 kHz. The sounder output is toggled with a delay between operations determined by the frequency required, as in the BUZZ program. For example, for a frequency of 3 kHz, a delay of 1 ms is required, which is 1/3600th of a second, 1000 instruction cycles at a cycle time of 1 µs. The information on program timing must be studied in Chapter 10. 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.
Prototype Hardware

**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 tune 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 other musical instruments using the input button, or selecting the tune 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.

Summary

- Methods of circuit construction available include PCB, breadboard and stripboard.
- Design software is available to draw the application circuit, test it by interactive simulation and create a board layout, but it requires time to learn to use the software, and etching equipment is needed to produce a PCB.
- Breadboard is re-usable and allows circuits to be prototyped quickly and easily, but is unreliable for complex circuits.
- Stripboard is more reliable but not re-usable. Connections can be laid out on paper, using computer drawing tools or directly onto the board.
- The DIZI demonstration board has a seven-segment display and audio output, with push button input and interrupts, and a 4-bit switched input, and can be used for demonstrating a range of simple applications.

Questions

1. For constructing a circuit, state one advantage and one disadvantage of (a) breadboard (b) stripboard (c) PCB design software.

2. State the maximum rated current output for port pins on the PIC 16F84A. How do these ratings simplify the interfacing to this chip compared with standard digital outputs?
3. Explain why a common cathode seven-segment display operates as an active high display.
State an input code for (a) all segments off and (b) all segments on for this type of display.

4. Outline an algorithm for generating a fixed frequency output of approximately 1 kHz from
the DIZI board using the hardware timer.

5. Describe a method of generating a set of codes to drive a seven-segment display of
alphanumeric characters at a PIC parallel output.

Activities

1. Confirm by calculation that the values used in the program data table in SCALE1.ASM
will give the required delays.

2. Devise a breadboard layout for the BIN circuit in Fig. 6.3. Build the circuit and test the
BIN program.

3. Devise a breadboard layout for the MOT1 circuit in Fig. 10.3, using a VN66 FET and 2 V
small DC motor. Build the circuit and test MOT1.

4. Build the DIZI circuit on breadboard, stripboard or PCB and test the programs BUZZ1,
DICE1 and SCALE1.

5. Design and implement any of the programs outlined for the DIZI hardware:
HEX1, MESS1, REACT1, GEN1, MET1, BELL1, GIT1.

6. (a) Investigate how an input from a numeric keypad can be detected. The typical keypad,
drawn in Fig. 12.15, 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 be scanned in rows and columns. A key
press is detected as a connection between a row and a column. The pull-up resistors ensure
Prototype Hardware

that all lines default to logic '1'. If a '0' is applied to one of the column terminals (2, 3, 4), and a key is pressed, the '0' can be detected at the row terminal (5, 6, 7, 8). 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 (5, 6, 7 or 8) as inputs. Draw a flowchart to represent the process required.

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. Refer to Chapter 15, the temperature controller application, and Appendix B, the lock application, for further information. Produce structured flowcharts for a lock program.

(b) 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 or unlocked). Research the design for the interface to a solenoid-operated door lock.

(b) 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 or unlocked). Research the design for the interface to a solenoid-operated door lock.
This chapter will focus on a PIC application in which program timing is critical. This kind of application is more demanding because the controlled device (motor) has its own dynamic characteristics which must be taken into account in the program design. The hardware design will be given as an example, so that we can concentrate on the software development for this application.

13.1 Motor Control Methods

There are two main types of control system, open loop and closed loop. An open loop system is essentially manually controlled. For example, a car requires the driver to monitor the direction of travel and correct the steering to follow the road. A closed loop system uses sensors to monitor the system outputs and control the process automatically, once the initial operating conditions have been set. A central heating system is a simple example; the thermostat monitors the temperature and switches the boiler on and off accordingly.

A small, inexpensive, DC motor will be used to demonstrate the use of the PIC microcontroller in a 'real-time' control application, allowing open and closed loop operations to be investigated. Real-time systems are those where timing is critical, and where the dynamic response to control inputs and feedback from sensors is critical. Robot arm positioning is a good example of a closed loop motor control application.

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 does not start until there is a reasonably large current due to inertia, friction and its electromagnetic characteristics. This makes it position non-linear, which means that the speed is not directly proportional to the current or voltage supplied. In addition, the speed cannot be accurately predicted for any given current, because the load on the shaft will affect it. 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 variables, 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, which produces a voltage which 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. It is now more common to use a digital control method, and the microcontroller can be used as the basis of a programmable system in which the control algorithms can be designed to closely match the motor and load requirements. The dynamic (time) response can then also 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; a digital control system is used only where the control is discrete (e.g. the steps only). In digital control, the current is switched on/off by a transistor or similar device, and the switching time can be adjusted. However, the control interface can be simplified if PWM is used, as described in Chapter 10. The 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.

Digital feedback can be obtained from a sensor which detects the shaft rotation, but may also detect the current or speed of the motor. The sensor can be an optical sensor, a magnetic sensor, or an encoder. The signal from the sensor is fed directly to a microcontroller to determine the current or speed, and the output is used to control the speed and/or position of the motor.

### 13.2 Motor Application Board

The block diagram of a motor application (MOTA) board is shown in Fig. 13.1, and a circuit diagram in Fig. 13.2. A stripboard layout of the circuit is shown in Fig. 13.3, and a finished stripboard circuit is shown in Fig. 13.4.
Figure 13.2 Circuit diagram of MOTA board.
A variety of motor control operations can be demonstrated using this hardware:

- MOTOR ON/OFF
- MOTOR FORWARD/REVERSE
- OPEN/CLOSED LOOP POSITION CONTROL
- OPEN/CLOSED LOOP SPEED CONTROL

The MOTA circuit is designed to demonstrate position and speed control with a PIC 16F84(A) microcontroller. Command inputs can be received from an 8-bit switch bank, a remote 8-bit master controller, two push buttons, or an analogue input. The motor can be turned in either direction via bidirectional drive outputs, with LED motor direction indicators. The drive can provide position and speed control, and can be pulse-width-modulated to control the speed. The shaft speed and position are monitored by a slotted opto-sensor and disk with one slot, feeding back one pulse per revolution to the controller. The PIC is crystal clocked at 4MHz, for precise feedback measurements.
Motor Drive

The small, inexpensive 2 V permanent magnet motor is connected in a passive FET bridge with R1 and R2 as load resistors. The motor current direction, forward or reverse, is controlled by switching on one of two VN66 FETs from RA0 or RA1. A dummy load can be switched in series with the motor to allow testing of closed loop control. Light emitting diodes indicate the motor direction. The use of a low-quality motor will emphasise the problems which will arise from imperfections in the motor itself.

Opto-sensor

This consists of an LED and photo-detector mounted either side of the slot in a plastic housing. The perforated disk attached to the motor shaft allows the light to pass through the holes and digital pulses are output from the sensor via a built-in amplifier, which allows the motor speed or position to be monitored by the controller. The sensor input is connected to the T0CKI (RA4) input of the PIC so that the shaft revolutions can be counted in the counter/timer register. Alternatively, the slot pulse interval can be measured at RA4 using the timer. The pulse may also be used to trigger an interrupt at RB0 by setting the pulse interrupt select switch accordingly. This then means that only seven bits can be read from the binary input.

Switched Inputs

The control program can allow the push buttons connected to RA2 and RA3 to stop, start or change speed or direction. The binary input switches could then be used to monitor the speed or position. Alternatively, a remotely generated digital control code can be applied to the digital input connector pins from a master controller, which could be operating a number of motors in a system. 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, but a PIC with a dedicated serial port would be better for this (see Chapter 16).
13.3 Control Methods

The PIC 16F84 is clocked at 4 MHz to give an instruction cycle time of 1 µs. No manual reset is required, but the power on reset should be enabled during programming to ensure a reliable start.

13.3.1 Open Loop Control

Open loop control of a DC motor has been described in Chapter 10 and a program developed which allows the speed to be controlled manually. In the MOTA circuit (Fig. 13.2), the motor can be driven in either direction by setting RA0 or RA1 high, with both set low to stop the motor off. Either can be pulsed for speed control, but they should not be high together; this would switch on both transistors, resulting in no current through the motor, and a lot of wasted power dissipated in the load resistors.

Open loop speed control can be implemented in various ways: programmed, push button manual input, manual/remote binary/analogue input. Sequence control can be incorporated in the program, for instance, the speed can be ramped up, hold constant and then ramped down on a timer. Another way is to control the speed by programming a PWM program with either direction or increment and decrement the speed in one direction by modifying the delay in the PWM program. The speed could be set at the binary inputs to Port B, either manually or at the DIP switches, or with an 8-bit digital input code supplied from a master controller, and analogue control is possible from a manual input (RV1) or from a remote voltage source.

13.3.2 Closed Loop Control

The PIC motor control board has a slotted wheel and opto-sensor to monitor the rotation of the motor. A wheel with only one slot is used for the applications developed here to simplify the calculation of speed. One pulse per revolution will allow calculations to be performed much more easily. The sensor is connected to RA4 of the PIC. As we know, the 16F84 contains an 8-bit counter/timer register, TMR0, which can be clocked from RA4 or the system clock, which we can use to measure the pulse interval, frequency or period.
Closed loop position control involves counting the slots 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, and the pulses counted, and the motor turned off when a set number of pulses have been counted. However, the motor will probably overshoot the required position due to inertia of the rotor. A simple solution would be to keep counting the slots and turn the motor back by the requisite number of slots. This might have to be repeated several times.

With only one slot, the position can only be determined to the nearest whole revolution. This may be unacceptable if a gearbox is fitted 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. In addition, a slotted wheel with more slots can be used, and a proportionate increase in accuracy obtained. With 100 slots, for example, and the gearbox, the accuracy will be 360/5000 = 0.072.

### 13.4 Position Control

A simple program, which moves the motor by a number of revolutions set on the DIP switches, is outlined in Fig. 13.5.

The hardware timer/counter (TMR0) is used to count the pulses from the sensor. The timer flag (T0IF) is set when the counter rolls over from 255 to 000, so the binary number at the switches has to be complemented (subtracted from 256) to set the count to a value whereby it reaches 256 after the correct number of pulses. The option register has to be initialised to select the T0CKI (RA4) pin for input, and to allocate the prescaler to the watchdog timer, so that the count is not prescaled. The processor is put to sleep after the move has been completed, so it would have to be reset to repeat the operation (see Program 13.1). As mentioned above, the position control achieved with the program may not be accurate, because, firstly, it can only count to the nearest whole revolution, and secondly, there is no provision for preventing overshoot. One way of achieving better accuracy is for the current...
Basic position control program runs motor for number of revs set on binary switch input. Uses TMR0 to count revs, but does not correct for overshoot.

Hardware: PIC 16F84 Motor Board

Inputs: RB0-RB7: DIP Switches (High)
T0CKI: Shaft Sensor (Low)

Outputs: RA0: Motor (High)

Set Processor Options

```
PROCESSOR 16F84 ; Declare PIC device
```

Register Label Equates

```
PORTA EQU 05 ; Port A
PORTB EQU 06 ; Port B
TMR0 EQU 01 ; Counter/Timer
INTCON EQU 0B ; Interrupt Control
```

Register Bit Label Equates

```
T0IF EQU 2 ; Timer Overflow Flag = INTCON,2
motor EQU 0 ; Motor Output = RA0
```

Start Program

```
; Initialise .................. Port B defaults to input
MOVLW b'11111100' ; Port A direction code
TRIS PORTA ; Set the bit direction
MOVLW b'00100000' ; Code for Option Register
OPTION ; Select T0CKI timer input
BCF INTCON,T0IF ; Clear Timer Overflow Flag

; Main Loop ......................
MOVF PORTB,W ; Read switches
MOVWF TMR0 ; into counter
COMF TMR0 ; and complement
BSF PORTA,motor ; Motor ON
test BTFSS INTCON,T0IF ; Test Overflow Flag
GOTO test ; and wait until set
BCF PORTA,motor ; Motor OFF
SLEEP ; Suspend processor
END ; Terminate source code
```
position to be continuously compared with the required position, and the motor driven 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 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. Because the position control of a DC motor is not very practical without a gearbox, we will look at speed control in more detail. The speed can also be measured by using an oscilloscope to display the pulse period from the sensor, making it easier to monitor the motor behaviour.

13.5 Closed Loop Speed Control

A typical application may require the motor board to operate as a slave unit under digital control. A master controller would supply an 8-bit code to set the speed of the motor, with the local controller required to maintain it to a specified degree of precision. The MOTA board allows for such an input, it can also be simulated for test purposes using the switch bank.

Suppose that the motor is to be controlled to a speed of exactly 50 revs per second (rps), which is about 40% of nominal full speed. This will produce 50 pulses per second (pps) at RA4 with a single slot in the wheel. The speed can be measured in one of two main ways:

1. counting sensor pulses over a measured time period,
2. measuring the period between sensor pulses.

13.5.1 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. Thus, if 100 slots are counted, the accuracy will be ±1%. If a set of 100 pulses is counted in 1 s, then an accuracy of ±1% is possible. At 50 pps this is 0.02% or 1/50 ±20. Therefore, for 2% accuracy, each count would need ±1 slot, and 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 could however be used if there were more slots in the disk or the motor were running at high speed.

13.5.2 Measuring Pulse Period

At 50 pps, the pulse period will be 1/50 s = 20 ms. This can be measured by comparing with a 20 ms timer, which can be set up using the TH0 timer counter/timer (see Chapter 9). The PIC instruction cycle time is 4 μs with a 4 MHz clock. The counter can therefore be clocked at a maximum 1 MHz (one per instruction cycle). 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. A possible prescale factor is calculated as follows:

Now

20 ms = 20000 μs

= 20000 instructions cycles
The timer can count 256 cycles, therefore the prescale multiplier required
\[ \frac{20000}{256} = 78.125 \]

The nearest available prescale multiplier = 128

Using this prescale value, the number of counts for 20 ms
\[ \frac{20000}{128} = 156.25 \]

\[ = 156 \text{ (nearest whole number)} \]

The actual motor period will then be
\[ 156 \times 128 \times 1 \mu s = 19968 \mu s \]

This count is accurate to within 1\% of 20 ms. Remember, this value must be complemented before loading into the timer register, because it counts up to zero at rollover.

The binary switch can then be used to input the timer value, to provide a variable, but accurate speed control. The longest period that measurable will be
\[ \frac{256 \times 128}{1} \mu s = 32768 \mu s \]

Motor period
\[ = 0.032768 \text{ s/rev.} \]

Motor speed
\[ = \frac{1}{0.032768} \text{ rev/s} \]

Therefore, this method should provide control from the motor maximum speed of about 125 rps down to about 30 rps. Lower speeds may be obtained by increasing the prescale factor in the timer, to give a longer time interval. This could be done ‘on the fly’ by adding code to allow the user feature to select the operating speed range.

Alternative motor control algorithms will now be evaluated to illustrate the method whereby the most appropriate implementation is selected where more than one algorithm is possible.

13.5.3 Synchronous Motor Control

Current signals which could provide closed loop control are illustrated in Fig. 13.4. The motor is driven with a PWM pulse whose length is controlled by a simple software loop. The shaft speed is measured using hardware timer, as outlined above. Initially, the inactive mark period will be set to the minimum value.
Closed Loop Speed Control

At the start of the control cycle, the motor would be switched on and the timer TMR0 cleared and started. It will be then incremented once per 128 instruction cycles. The sensor and time out flag can be continuously checked to see whether the counter has finished if the next pulse has arrived. If the timer times out before the rising edge of the next pulse, the motor is not going fast enough, so the speed must be increased by incrementing the ON time period of the motor. If the pulse arrives fast, the motor is going too fast, so the speed must be reduced by decrementing the ON period. If the ON time starts at zero, it will be increased until the motor starts. The speed should then stabilize at the value corresponding to the switch input.

Using this method of control raises a few problems. Note that the drive signal and sensor signal are locked together, with the same period. This means that the drive frequency will be 50 Hz, which is too low. A higher frequency also prevents commutator switching in the motor from interfering with the drive switching. In addition, with this algorithm, the drive will be on for the same part of each revolution, potentially causing uneven wear on the commutator. We will therefore consider an alternative algorithm.

13.5.4 Asynchronous Motor Control

We would prefer the PWM drive to operate at a higher frequency, and asynchronously (not locked to the sensor cycle). The asynchronous algorithm utilizes the motor using the software delay loop to generate a pulse-width-modulated drive signal to the motor, but checking the timer and sensor each time around the motor delay loop. The MSR is adjusted to control the speed, using the input code and its complement to set the delays for the mark and space.

The process is illustrated in the timing diagram (Fig. 13.7). The timing cycle starts at the falling edge of the negative going sensor pulse, where the timer is started. The program waits for the rising edge of the sensor pulse, then starts checking if the next pulse has arrived, or if the timer has timed out, for just one motor cycle. If the timer times out before the pulse arrives, the drive speed must be increased for the next timing cycle. On the other hand, if the drive signal 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. When the speed is correct, the speed adjustment should alternate between...
Motor Applications

One revolution +5V

0V Slot pulse

Time On delay

Off delay

20 ms

Reload timer

Slot flag set

Timer flag reset

Time out:

Increment speed, Set done flag, Reload timer, Wait for next pulse

On

Off

Slot found

Reload timer

Clear done flag

Set slot flag

Slot found before time out

Set slot flag

Decrement speed

Reload timer

immediately

Timer runs

Time out:

Reload timer

Set slot flag

Figure 13.7 Timing diagram for high-frequency motor speed control.

incrementing and decrementing. This ideal performance may well be affected by imperfections in the motor. In practice, it has also been found that there was significant 'hunting' (variation) in the speed because of the relatively low sampling rate.

Figure 13.8(a) shows the top level flow chart for the program. 'Speed' is a GPR which 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 256, which means the frequency will remain
constant. The 'Reload Timer' (RELTIM) routine restarts the timer (Fig. 13.8c). TMR0 starts counting the internal instruction clock pulses as soon as it is reloaded. It is loaded with the complement of the input switch value because time-out is detected when the TMR0 register rolls over from FF to 00. Therefore, the time-out is measured as the count from the loaded value up to 00.

The most complex part of the program is the TESTEM routine, which checks the inputs and modifies the value of 'Speed' accordingly (Fig. 13.8b). The requirements are:

1. To start the motor from rest, the MSR must be increased upwards when no slot is detected, so that the motor will eventually start. A fairly high MSR is required to get the motor moving initially.
2. When a slot is detected before a timeout, the timer must be restarted immediately.
3. When the timeout occurs first, the speed must be incremented and the timer must be restarted, to get the motor started naturally. If a slot then arrives, the timer must be restarted again, so that it can accurately measure the time between the slot edges.
4. When the sensor is detected as low in the check cycle, it means that a falling edge has arrived, and the timer must be restarted. The program must then wait for the sensor to go high again before it starts looking for the next slot.
5. The speed must be stopped from rolling over from FF (maximum) to 00 (minimum), so the speed is checked after incrementing and decremented again if it is found to be equal to FF.

Figure 13.8 Flowcharts for closed loop motor speed control. (a) Main loop.
Figure 13.8 Flowcharts for closed loop motor speed control. (b) Input testing; (c) Timer reload.
To achieve these requirements, flags have been defined in a GPR to record the fact that the falling edge has been detected and acted upon (flag 'slot'), and another to record the fact that the timer has been reset, to make the program wait for the next slot to restart the timer.

13.5.5 Program Simulation

The source code for the closed loop speed control program is shown as Program 13.2. In order to avoid the need for an input stimulus file, this simulation version of CLS1 loads the timer with the literal value '156' in the subroutine RELTIM, rather than reading Port B switches. For running in the hardware, the comment delimiters in the usual input code to Port B must be removed, and the literal load operation commented out. The RA4 (sensor) input may be simulated using the asynchronous input window in MPLAB. The results shown in Table 13.1 should be obtained.

Program 13.2

Closed loop motor speed control source code

; ************************************************************
; ; CLS1.ASM M. Bates 4/4/99
; ************************************************************
;
; Closed Loop DC Motor Speed Control using Pulse
; Width Modulation (software loop) to control speed
; and hardware timer to set reference time interval
;
; Hardware: PIC 16F84 Motor Board
; Clock: XTAL 4MHz
; Inputs: RB0-RB7 : DIP Switches (High)
; RA4 : Shaft Sensor (Low)
; Outputs: RA0 : Motor (High)
; Configuration Settings:
; WDTimer: Disable
; PUTimer: Enable
; Interrupts: Disable
; Code Protect: Disable
;
; Set Processor Options....................................
; PROCESSOR 16F84 ; Declare PIC device
; Register Label Equates...................................
; PORTA EQU 05 ; Port A
; PORTB EQU 06 ; Port B
; TMR0 EQU 01 ; Counter/Timer
; INTCON EQU 0B ; Interrupt Control
; Speed EQU 0C ; Counter Pre-load Value
; Count EQU 0D ; Delay Counter
; Flags EQU 0E ; User Flags

continued...
Motor Applications

Register Bit Label Equations

`timeout EQU 2 ; Time Out Flag = TMR0,2`
`motor EQU 0 ; Motor Output = RA0`
`sensor EQU 4 ; Shaft Opto-Sensor = RA4`
`slot EQU 0 ; Slot Found Flag`
`done EQU 1 ; Time Out Done Flag`

Start Program

```
; Initialise Port A defaults to input
MOVLW b'11111100' ; Port A bit direction code
TRIS PORTA ; Set the bit direction
MOVLW b'00000110' ; Code for option Register
OPTION ; Sets prescale 1:128
MOVLW 080 ; Initial value for
MOVWF Speed ; count pre-load value
MOVWF Count ; and counter itself
GOTO start ; Jump to main program
```

```
; RELTIM Routine
; Reloads TMR0 timer/counter register with complement of
; switch input (or dummy value for simulation mode)
reltim MOVLW d'156' ; Dummy value for timer (sim)
MOVF PORTB,W ; Input Switches (runtime)
MOVWF TMR0 ; Load Timer with input
COMF TMR0 ; Complement value
BCF INTCON,timout ; Reset 'TimeOut' Flag
RETURN
```

```
; TESTEM Routine
; Increases speed if timeout detected or
; decreases speed if slot is detected...
testem BTFSS INTCON,timout ; Time Out?
GOTO tessen ; NO: Skip Speed Increment
BSF Flags,done ; Set Time Out Done Flag
INCFSZ Speed ; Test for maximum speed
GOTO reload ; NO: jump to timer reload
DECF Speed ; Decrement again
GOTO reload ; & jump to timer reload

tessen BTFSS PORTA,sensor ; Slot Present?
GOTO teslot ; YES: jump to test slot
BCF Flags,slot ; Reset 'Slot' Flag
GOTO datcon ; & continue Count loop
```

```
teslot BTFSC Flags,slot ; 'Slot' Flag Set?
GOTO datcon ; YES: Skip speed decrement
BTFSC Flags,done ; 'Done' Flag Set?
GOTO clrdone ; YES: Skip speed decrement
DECFSZ Speed ; Test for minimum speed
GOTO clrdone ; NO: continue loop
INCF Speed ; YES: increment again
```

```
; Do nothing

; RELTIM Routine
; Reloads TMR0 timer/counter register with complement of
; switch input (or dummy value for simulation mode)
reltim MOVLW d'156' ; Dummy value for timer (sim)
MOVF PORTB,W ; Input Switches (runtime)
MOVWF TMR0 ; Load Timer with input
COMF TMR0 ; Complement value
BCF INTCON,timout ; Reset 'TimeOut' Flag
RETURN
```

```
; TESTEM Routine
; Increases speed if timeout detected or
; decreases speed if slot is detected...
testem BTFSS INTCON,timout ; Time Out?
GOTO tessen ; NO: Skip Speed Increment
BSF Flags,done ; Set Time Out Done Flag
INCFSZ Speed ; Test for maximum speed
GOTO reload ; NO: jump to timer reload
DECF Speed ; Decrement again
GOTO reload ; & jump to timer reload

tessen BTFSS PORTA,sensor ; Slot Present?
GOTO teslot ; YES: jump to test slot
BCF Flags,slot ; Reset 'Slot' Flag
GOTO datcon ; & continue Count loop
```

```
teslot BTFSC Flags,slot ; 'Slot' Flag Set?
GOTO datcon ; YES: Skip speed decrement
BTFSC Flags,done ; 'Done' Flag Set?
GOTO clrdone ; YES: Skip speed decrement
DECFSZ Speed ; Test for minimum speed
GOTO clrdone ; NO: continue loop
INCF Speed ; YES: increment again
```
Closed Loop Speed Control

; Clear 'Done' Flag
clrdone BCF Flags,done

; Set 'Slot' Flag
setslot BSF Flags,slot

; Reload timer
reload CALL reltim

; Decrement & Test Count
datcon DECFSZ Count

; GOTO if Count not zero
GOTO testem

; End motor cycle
RETURN

; Main Loop
start CALL reltim

; Motor ON
again BSF PORTA,motor

; Put ON delay value
MOVF Speed,W
MOVWF Count

; Insert Delay Code
CALL testem

; Motor OFF
BCF PORTA,motor

; Put OFF delay value
MOVF Speed,W
MOVF Count

; Convert to OFF value
COMF Count

; Insert Delay Code
CALL testem

; Insert Delay Code
GOTO again

; Terminate source code
END

Table 13.1
Test table for motor control program CLS1

<table>
<thead>
<tr>
<th>Sensor</th>
<th>flags</th>
<th>Done</th>
<th>Slot</th>
<th>Speed</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timeout</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Motor starting up</td>
</tr>
<tr>
<td>Restart</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Motor stopped</td>
</tr>
<tr>
<td>Enable</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8.5</td>
</tr>
<tr>
<td>Disable</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>8.5</td>
</tr>
<tr>
<td>Slot 1 start and enable</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Slot 2 start and enable</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Slot 3 start and enable</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Increment</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Decrement</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Slot 1 start and enable</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Slot 2 start and enable</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Slot 3 start and enable</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

...continued...
Table 13.1 continued

<table>
<thead>
<tr>
<th>State</th>
<th>Sensor Input</th>
<th>Sensor Flag</th>
<th>Drive Flag</th>
<th>Drive Register</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slot end</td>
<td>1 0 0 0</td>
<td>XX</td>
<td></td>
<td></td>
<td>Repeat and up to</td>
</tr>
<tr>
<td>Slot start and reload</td>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
<td>Set speed</td>
</tr>
<tr>
<td>Slot end</td>
<td>1 0 0 0</td>
<td>XX</td>
<td></td>
<td></td>
<td>Inc. speed</td>
</tr>
<tr>
<td>Slot start and reload</td>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
<td>Set speed</td>
</tr>
<tr>
<td>Slot end</td>
<td>1 0 0 0</td>
<td>XX</td>
<td></td>
<td></td>
<td>Dec. speed</td>
</tr>
<tr>
<td>Slot start and reload</td>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
<td>Dec. speed</td>
</tr>
<tr>
<td>Slot end</td>
<td>1 0 0 0</td>
<td>XX</td>
<td></td>
<td></td>
<td>XX</td>
</tr>
<tr>
<td>Slot start and reload</td>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
<td>XX</td>
</tr>
<tr>
<td>Slot end</td>
<td>1 0 0 0</td>
<td>XX</td>
<td></td>
<td></td>
<td>XX</td>
</tr>
<tr>
<td>Slot start and reload</td>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
<td>XX</td>
</tr>
<tr>
<td>Slot end</td>
<td>1 0 0 0</td>
<td>XX</td>
<td></td>
<td></td>
<td>XX</td>
</tr>
</tbody>
</table>

13.5.6 Hardware Testing
The correct function of the closed loop control program can be tested in the target system by setting the binary input to 356 and checking the actual speed of the motor by measuring the period of the sensor pulse on an oscilloscope; it should be 20 ms. The binary input can then be varied, and the period should vary in proportion, within limits stated above. The transient and start up response can be examined by stalling the motor, and studying the motor response as it reaches the target speed. A dummy load is also provided in series with the motor, which is switched in, the added series resistance will reduce the motor to the current, hence its speed. If the closed loop control is working, the drive should compensate and maintain the speed of the motor by increasing the drive MSR.

13.5.7 Evaluation of Algorithm
Ideally, the PWM speed control should operate at a frequency above about 15 kHz; the program CLM operates at only about 300 Hz, because of the time required to sample the sensor status and sensor input, and to complete the software loop for one drive cycle, which uses 6 of the 16 bit
The frequency could be increased by reducing the total loop count to less than 256, but this will reduce the resolution of the control. Some compromise value could be arrived at by calculation and experimentation, based on the required resolution.

The MOTA hardware provides for alternative implementations of closed loop control. RB0 can be optionally connected to the output from the motor shaft sensor, so that an RB0 interrupt can be used to signal the arrival of a sensor pulse. Most PIC chips have more than one timer, which would be useful in this application. The PWM signal could be generated using software timers, as well as the target sensor period. Some PICs have dedicated PWM outputs, which would simplify the software, but would require the PWM interface to be correctly initialized.

13.6 Commercial Application

Attempting closed loop dynamic control using the PIC 16F84, which has only one timer and one interrupt, illustrates its limitations, and why there is a range of more powerful processors available in the PIC family. A commercial design for a similar system uses a PIC 17C42 running at 16 MHz, with a dedicated PWM output. It is designed for use in printers and plotters for positioning the print or scan head quickly and precisely. A block diagram is shown in Fig. 13.6.

The motor is driven from a dedicated driver chip (LMD 18201) which requires only a single PWM input. A 50% PWM gives zero output which corresponds to the motor being stationary. The motor can then be driven in either direction by varying the MSR above and below 50%. The driver circuit incorporates a full ‘H-bridge’ driver chip, which can supply motor current in either direction with minimal power consumption. The motor speed and position are monitored by a shaft encoder which produces two pulse trains. The relative position of the pulses indicates the direction of rotation of the motor. These are fed to a logic circuit which produces separate count up and count down pulses which are counted by 16-bit counters in the PIC to allow the current position of the motor shaft to be calculated. The control software is also far more complex than our demonstration system, with nearly 2000 instructions.

The PIC 17C42 has a serial port which allows commands to be sent via a single wire, and for the PIC to return information about the actual position, speed, and so on, so that,

![Block diagram of servo control unit.](image-url)
A small DC motor under PIC control can be used to demonstrate a range of real-time processes. The MOTA demonstration hardware allows motor control via a passive load bidirectional FET bridge driver, with analogue and digital inputs.

Open loop control can be implemented relatively simply using pulse counting for position control and PWM for speed control.

Closed loop control requires more complex algorithms which use pulse feedback to continuously modify the drive output for position and speed control.

A practical servo typically uses additional hardware, a more powerful microcontroller and complex software for better performance.

1. Outline an open loop method of controlling the speed of a small DC motor, using a microcontroller. Identify the main hardware components required.

2. Explain how a slotted wheel can be used to provide speed and positions feedback from a shaft to a microcontroller.
3. Explain why closed loop control is necessary for accurate speed control of a DC motor.

4. Calculate the positional accuracy in degrees of the output shaft of robot arm drive with a 90:1 gearbox, if a shaft encoder with 200 steps per revolution is attached to the motor shaft. (Answer: 0.2°)

5. Describe two alternative methods of measuring the speed of a motor shaft using an opto-slot detector with a crystal clocked microcontroller.

Activities

1. Construct the MOTA board using a method of your choice and devise a test schedule to confirm the correct operation of the hardware prior to fitting the PIC chip.

2. Devise a program to rotate the MOTA board output by exactly 100 revs from its start position. Evaluate the performance of the program in terms of speed of response, accuracy and reliability. What are the characteristics of the motor which affect the performance?

3. Investigate the performance of the program CLS1 in terms of reliability, response time, range of control (maximum and minimum speed). 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).

4. Modify CLS1 to read the input push buttons on the MOTA board to increase or decrease the set speed.

5. Modify the program for the MOTA board to use the timer interrupt to signal time out. Compare the performance of this alternative implementation with the program CLS1.

6. Modify the program for the MOTA board to use the RB0 interrupt to monitor the feedback from the motor. Compare the performance of this alternative implementation with the program CLS1.

7. Research and download the data sheet for a PIC chip which has analogue inputs, and redesign the MOTA board and control program so that the motor speed can be accurately controlled from an analogue input in the range 0–5 V.

8. Redesign the MOTA board circuit to use a full bridge motor driver instead of the dual FET and passive bridge drive circuit.
This Page is Intentionally Left Blank
Part D
More Controllers

14 More PIC Microcontrollers
15 More PIC Applications and Devices
16 More Control Systems
Chapter 14
More PIC Microcontrollers

14.1 Common Features of PIC Microcontrollers

All PIC microcontrollers use the same basic architecture and instruction set to provide a parallel progression path from simple programs developed for the 12-series chip through to the most complex applications for the 18-series device. Of course, the common architecture is used in the non-flash memory PICs as well. The architectural features may be compared by studying the

The PIC 16F84 has been used as a reference device so far because its architecture and operation are relatively simple compared with other PIC microcontrollers. The range of flash memory PICs has now expanded such that alternative devices are now available which have more features at a lower unit cost. This chapter will review these features so that the most suitable device for a given application may be selected. Specifically, this means selecting the PIC chip which has the required number and type of inputs and outputs, program memory capacity sufficient for the application and so on. All of this is, of course, price!

We will continue to concentrate on PICs which have flash program memory, since these are the best devices for learning about application development, prototyping and producing one-off or low volume products. For larger production runs, OTP or mask programmed PICs are available. The OTP devices can be programmed by the user or supplied pre-programmed if the quantity justifies it, and the application program will not require any further modification. The masked ROM device has the program built in during production, and would only be used for high volume, mature products.

The main groups of PIC flash devices are shown in Table 14.1. They are divided into three groups: with a different prefix number: the 12XXXX series are 8-pin miniature PICs, the 16XXXX group might be described as the standard series and the 18XXXX devices as the high performance group. Their features are summarised in Table 14.1.

Full details of all these devices and the rest of the PIC range are provided at www.microchip.com, from where the individual data sheets can be downloaded as PDF files.

14.2 Selecting a PIC

14.3 Advanced PIC Features

14.4 Serial Communications
### Table 14.1: PIC Flash microcontrollers

#### 12FXXX
- Low cost and small size
- 6-in packages
- 6/12-pin
- 20k x 14/12-bit instructions
- 14 word program memory
- 20MHz clock
- 8-bit and 16-bit timers
- Up to 4 analogue inputs
- In-circuit programming and debugging

#### 16FXXX
- Mid-range cost and performance
- 16-40-pin packages
- 12-33 I/O pins
- 35 x 14-bit instructions
- 1-8 k word program memory
- 20MHz clock
- 4/8 MHz internal oscillator
- 2 x 8-bit and 1 x 16-bit timers
- Up to 8 analogue inputs
- Serial communication ports, parallel slave port
- 1/2 pulse width modulation outputs, capture & compare inputs
- In-circuit programming and debugging

#### 18FXXX
- High performance
- 18-80-pin packages
- 13-68 I/O pins
- 58 x 16-bit instructions
- 2-64 k word program memory
- 40 MHz clock
- 8/10 MHz internal oscillator
- 2-64 k program memory
- 2 to 6 x 8-bit and 3 to 6 x 16-bit timers
- Up to 16 analogue inputs
- Serial communication ports, parallel slave port
- Up to 14 pulse width modulation outputs, capture and compare inputs
- CAN communication interface
- In-circuit programming and debugging

*Selected devices in the range*
The common features of the PIC architecture are:

- Harvard architecture
- RISC instruction set
- Dual-port RAM 1K x 8-bit
- RAM block including SRAM
- EEPROM non-volatile data memory
- single working register
- dedicated, non-writable stack
- power-up and watchdog timers
- multiple interrupt sources
- interrupt on power-up
- sleep mode
- serial in-circuit programming

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. The instruction fetch operation can therefore be carried out at the same time as the results from the previous operation are stored. As a result, the program executes faster at the same clock speed by carrying out these processes concurrently. The overlapping of instruction fetch and execution stages is described as pipelining.

RISC Instruction Set

The PIC has a small number of instructions compared with a conventional CISC processor. This has two main benefits – the instructions are easier to learn and the code executes faster, because the instruction decoding hardware is less complicated. The down side is that more complex operations may have to be constructed from simpler ones, ending up taking longer to execute. Overall, the RISC performance is frequently better because, in a typical application, these complex instructions are not needed very often.

Flash Program Memory

Flash ROM is a great advance in memory technology which has developed over the last ten years. Although not non-volatile, memory is essential in embedded systems to store the control program. The EPROM was used, but the memory had to be removed from the system for erasing under ultraviolet light. Batteries-backed RAM was an alternative, but, of course, batteries last only a limited time, and the program can be lost. Flash ROM can be re-programmed many times, and serial in-circuit programming (ISP) can be used to program the chip without the inconvenience and possible damage caused by removing it from the circuit.
RAM and SFRs

The individual bits in the SFRs need to be read and written when initialising the chip or during program operation. Because they are located in the same RAM block as the 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.

EEPROM Data Memory

This is very 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 must be 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.

Working Register

Conventional processors tend to have a block of registers for storing current data. The Motorola 68000 processor, for example, has eight data registers. 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 loading a register with a literal takes two instructions, as it has to be loaded into W first, then into the register.

Stack

The stack size determines the number of subroutine or interrupt levels which can be used in the application program. The 12-series chips have only a 2-level stack, the 16-series 8 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 simpler program structure against the increased number of levels.

System Timers

There is an array of features incorporated into the PIC microcontroller to ensure a smooth start-up of the application program when power is applied or a reset generated. A power-on reset is generated internally when the supply voltage has reached the required level. A power-on reset provides a delay to allow the power supplies to stabilise, and an oscillator start-up timer provides a further delay to ensure that the clock is stable before program execution begins. The watchdog timer is another standard feature which allows the chip to reset itself automatically if the program execution fails to follow the normal sequence, thereby improving overall reliability. Brown-out protection allows the chip to reset in an orderly fashion if the power supply fails for a short period of time.

Interrupts

An interrupt is an internally or externally generated signal which forces the processor to suspend the current operation and execute an interrupt service routine. The ISR then has a higher priority
Common Features of PIC Microcontrollers

than the background process. PIC chips provide a variety of interrupt sources, for example, a change on a selected input, or a hardware timer time-out. There is an interrupt priority system available in the more advanced 18-series PICs that allows the chip to be set up to ignore an interrupt source if a more important one is already active. In conventional microprocessors, such as the Motorola 68000, multiple interrupt vectors are available; that is, a different ISR address can be specified for up to eight interrupt sources. This means that a different ISR for each interrupt source can be specified. In the PIC, all interrupts have to be serviced via the single interrupt vector, at address 004 in program memory. Therefore, to determine between the sources 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.

Hardware Timers

The number of hardware timers generally increases with the chip complexity. They are either 8-bit or 16-bit counters, with prescalers or post-scalers, which divide down the input or output of the counter to extend its range. If we take the motor program in Chapter 13 as an example, we can see how additional hardware timers would have been useful in the 20 ms interval and the sensor pulse monitoring could have used RB0 interrupt. This would have simplified the control program significantly. Therefore, a device with two hardware timers would have been a better choice for this application.

Sleep Mode

This is a very useful feature for power saving and also terminating programs which do not keep running. The processor shuts down when the instruction SLEEP is encountered, with current consumed dropping to around 1/20μA. The device can then be woken up via an external signal when required, for example in battery-powered applications to obtain. SLEEP can also be used to terminate a program so that program execution does not continue into unprogrammed locations. These locations default to all 1’s, which generally corresponds to a valid instruction code (ADDLW in 14-bit code, NOP in 16-bit code). If a program is not terminated with a SLEEP or GOTO instruction, the program counter will roll over to zero, and the program will start unexpectedly.

In-Circuit Programming

The PIC microcontrollers use a common program downloading system, which consists of loading the program in serial form via one of the data pins when the chip is in programming mode. The chip can be placed in a programming unit for downloading the application code, and then transferred to the application board. Alternatively, the chip can be programmed in circuit, if the application hardware is designed to allow this. This ensures that the chip is properly set at all times, reducing the risk of damage, and can be programmed after the circuit has been manufactured, and reprogrammed at any time, via an on-board controller. This controller can be seen on the H8S/1 temperature controller board circuit diagram and hardware in Chapter 15.
14.2 Selecting a PIC

Each type of PIC microcontroller provides a different combination of features, so that the most suitable can be selected for any given application. At the time of writing more than 140 are available and increasing all the time. Some of the main selection criteria are:

- number of I/O pins available
- program memory size
- program memory type (ROM, EEPROM, Flash)
- EEPROM size
- flash size (1 kbit or 10 kbit)
- analog inputs (8 bit or 10 bit)
- serial communication interfaces (USART, SPI, I²C, CAN)
- internal oscillator
- in-circuit debugging
- 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. Simply switches will require single digital input, study a four-bit half-register switch. A temperature sensor may require an 8-bit A/D. When specifying the PIC, the developer must be aware that some kinds of information display, such as digital tubes (LEDs), are not included in the number of input/output pins. Similar considerations will apply 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 using MPLAB, 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 modified or a different chip used. MPLAB can also be used for provisional chip selection, since a specific device must be selected before the program is assembled.

We are assuming that flash memory PICs will be used, although for experimental purposes, Flash memory PICs can also be used for a 16-bit PIC with on-chip flash memory. The flash memory is selected on the basis of minimum cost for a given batch size. If overall size of the finished board is important, a surface mount implementation may be needed.

When the design parameters such as I/O requirements, program memory size and so on have been established, the most suitable device can be selected using the search facilities on the manufacturer's website. Summary information for selected PIC flash microcontrollers is provided in Table 14.2, as a guide to the features available.

14.2.1 I/O Pins

The I/O pins are the most important criterion for an embedded controller. The number and type of I/O pins and outputs required should be clearly defined at an early stage in circuit design. The grouping of these pins should also be important, as they are generally arranged as 8-bit ports, with smaller chips having partial port implementations (e.g., Port A is the 18F84 is 5 bits). If analog inputs are required, these must be deduced from the digital I/O total, since they share the same pins with digital I/O. The table shows that the number of I/O pins ranges from...
| PIC | flash | comp. Freq. | comp. Freq. | temp. range | input | output | clock | pins | inputs | outputs | multiplexed | cost | MHz | cost | MHz | cost | MHz | MHz | MHz | MHz | MHz | MHz |
|-----|------|------------|------------|-------------|--------|--------|--------|------|--------|---------|-------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 16F8720 | 4 | 16 | 64 | 128 | 4 x 10 k | 2 x | 22 | 4 | ✓ | - | - | 1.10 |
| 16F8721 | 4 | 16 | 64 | 128 | 4 x 10 k | 2 x | 22 | 4 | ✓ | - | - | 1.10 |
| 16F8620 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8520 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8420 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8320 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8220 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8120 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8020 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8920 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8820 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8721 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8621 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8521 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8421 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8321 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8221 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8121 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8021 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8921 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8821 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8722 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8622 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8522 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8422 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8322 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8222 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8122 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8022 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8922 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8822 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8723 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8623 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8523 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8423 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8323 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8223 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8123 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |
| 16F8023 | 16 | 16 | 128 | 10 | 0 | 22 | 4 | - | ✓ | - | - | 1.10 |

Table 14.2 PIC flash microcontroller features
More PIC Microcontrollers

6 in the 8-pin chips to 68 in the 80-pin device. Most I/O pins have more than one function, one of which can be selected during initialization, setting up the relevant control register. If no setup is performed for a particular pin, it will typically default to a digital input. Pins can be re-configured within the program sequence to have a different function at different times. This reconfiguration can be performed either by the developer or the user, depending on the pin functionality. The designer must ensure that the two functions do not interfere with each other, in terms of both the hardware and software.

14.2.2 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 predict the requirements fairly early in the development. The total program statement size can include several machine code instructions. In this case, an 18-series device is likely to be the best choice, as the memory available is larger. Microchip supply a C compiler for the 18XXX chips, and third party compilers are also available. For prototyping and small production volumes, flash memory will be preferred. For intermediate volumes, where the design is proven, OTP memory will reduce costs per unit. For high volume production, a contract for mask programmed devices supplied by the manufacturer can be considered.

14.2.3 Data Memory

The file register RAM block tends to increase in size with the program memory size and chip complexity. The number of variables and temporary data storage blocks required should be totalled 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 included. As can be seen in the table, RAM ranges from 64 to 3840 bytes in the flash PICs.

14.3 Advanced PIC Features

14.3.1 Timers and CCP

The timer capacity of the flash PICs ranges from only one 8-bit timer in the 16F84 to five in some 18-series chips (2 × 8-bit plus 3 × 16-bit). Hardware timers should be used for timing and counting operations, because the processor cannot execute other tasks while the timer process is running.

CCP stands for Capture/Compare/PWM. Capture mode provides input interval measurement (Fig. 14.1). The value in a timer register is captured (stored) when an input changes; the time between the timer start and input change can thus be measured. In the motor application, for example, a timer can control the rate of rotation of a small DC motor. The output from the counter is fed into the PWM module, which provides a varying duty cycle for the motor drives. If the duty cycle is increased, the speed of the motor will also increase.

![Figure 14.1 Timer capture operation](image-url)

Figure 14.1 Timer capture operation.
Compare mode provides output interval generation (Fig. 14.2). 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 time-out event. This is a convenient way to generate a timed interval, so that, for example, an output can be switched on for a set time after an input has changed.

In PWM mode, preset values are loaded into two registers representing the mark and space period of the PWM output required (Fig. 14.3). 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 and the output from the flip-flop toggles after each mark and space interval to generate a PWM output.

**14.3.2 Analogue Inputs**

Many PIC chips incorporate analogue inputs so that they can be used in control systems with input sensors which produce a voltage, current or resistance change in response to an 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 notified to signal this event.

Figure 14.2  Timer compare operation.

Figure 14.3  Pulse width modulation operation.
environmental variation. The temperature controller described in Chapter 15, for example, is designed to accept inputs from temperature sensors which give an output change of 10 mV per °C. The PIC then operates outputs which control the temperature in the target system. Most PICs with this feature provide 10-bit conversion. This means that the input voltage range must be limited to a maximum of 5.0 V and a minimum of 0.0 V, or a maximum of 2.56 V and a minimum of 0.0 V. This is good enough for all but the most demanding applications. If such resolution is not required, an 8-bit result can be obtained by ignoring the two extra bits. Multiple analogue inputs are usually available; the PIC 16F877 in the temperature controller has eight, although only four are used. Code for performing the analogue input conversion is given in Program 15.1. Refer to the PIC 16F87X data sheet for details of analogue input operation.

The analogue to digital conversion process is illustrated in Fig. 14.4. The port containing the ADC inputs can be set up with a combination of analogue and digital inputs, or all analogue. The port pins are then connected to a resistor divider network, and the output is then measured in an ADC result register. The minimum and maximum voltage levels to be converted can be set externally, or the internal supply voltages used. An external voltage reference of 2.5 V gives a convenient 0.01 V per bit conversion for an 8-bit result. A clock divider must be set up to allow the minimum specified conversion time (about 30 μs); for example, if the clock clock is 20 MHz, divide by 64 must be selected, and at 4 MHz, divide by 8. The GIDONE 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.

![Analogue to digital conversion block diagram](image-url)
14.3.3 Parallel Slave Port

The microcontroller may need to operate as part of a larger system, exchanging data with a master controller, or other devices in the system, in the same way that a peripheral interface device operates as a slave to the microprocessor in a conventional system. An 8-bit port (Port D in the 16F877) can therefore act as a parallel slave port, allowing 8-bit data exchange with a master controller, which may be a conventional processor, another computer or another microcontroller (Fig. 14.5). Three bits of another port provide a control signal interface, acting as active low read, write and chip select inputs to the slave PIC. In addition, the slave may need to output an interrupt signal, which must be generated in software at an convenient output pin.

14.3.4 Internal Oscillator

To save on external components, some PICs offer an internal oscillator option. It provides a fixed 4 MHz or 8 MHz clock based on an internal RC oscillator. It is therefore not very accurate, but its frequency can be trimmed using a calibration value pre-programmed into the chip by the manufacturer. Even so, the frequency is only accurate to about 5%, so an external crystal clock will still be needed to obtain accurate timing operations.

14.3.5 In-Circuit Debugging

By using an ICD hardware module instead of the usual programming interface, the serial programming system can double up as a debugging tool. When the application program has been downloaded, MPLAB can be used to monitor the program execution within the chip, without the need for emulating the hardware. The program can be single stepped, breakpoints set and registers monitored, using the same techniques as with the software simulation, it can then be checked for correct operation at full speed in the target hardware. When the program is fully debugged, it can be loaded into the PIC and run in the target hardware. This technique is much cheaper than the alternative of a circuit simulation.

Figure 14.5  Parallel data port connections.
The ICD system configuration is shown in Fig. 14.6(a). The ICD module acts as a bridge between
the host PC running MPLAB IDE and the application board with the PIC chip fitted. There are
two ways to connect to the chip, using a header and using an on-board ICP/D connector.
If no provision has been made on the application board for in-circuit programming and
debugging connections, an ICD header, which has the same pinout as the chip itself, can
be fitted to the PIC socket. The header connection is shown with its demo-board in Fig. 14.6(b).
The PIC is then fitted into a corresponding socket on the header, which carries the ICP/D
connections to Port B. When debugging is complete, the chip can be switched to run mode
and plugged directly into the board.

A better option is to provide the ICP/D connections on the board, as this only requires a
suitable 6-way connector, a resistor to protect the MCLR input. The connections required are
drawn in Fig. 14.7. Vpp is the programming voltage applied to the target chip by the ICD
module, and the program is downloaded via PGD. When switched to ICD mode, the same
connections can be used to control the processor from MPLAB, and register information to be
relayed back to the host to assist with debugging. The ICD system is shown connected to the
temperature controller board in Fig. 15.6.

---

**Figure 14.6** In-circuit debugging system via header connection. (a) Block diagram; (b) ICD header
with demo board.
Clock speed is the main factor in the performance of any microprocessor system, and is critical in some applications. For example, in the motor control example, 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 ns or 100 ns (nanoseconds) and an execution rate of 5 or 10 MIPS. 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 for 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 consumption and its heating effect.

Power consumption is generally proportional to clock speed in CMOS devices, since most of the power is consumed when the component transistors switch on and off. This is illustrated by the current consumption curve for the 16F84A found in the data sheet under electrical characteristics, summarised in Fig. 14.8. Note that for 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, especially if operating in a small IC package, may necessitate additional cooling measures to keep a chip within its temperature limits. A heat sink or even a fan, as found on the processor in a typical PC motherboard design, can be fitted.
14.3.7 Packaging

The standard package for integrated circuits is the plastic dual in-line (PDIP) chip, which has two lateral rows of pins spaced at 0.1" intervals (Fig. 14.9(a)). 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 chips. The plastic leaded chip carrier (PLCC) package has the pins arranged around the 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 surface area of the integrated circuit may occupy only a small central portion of the DIP package, so miniaturized packages are usually designed. Surface-mount components are now increasingly used in commercial products, as they become smaller, consume less space, and produce more efficient microcircuits. The pins of the ICs are not fitted through holes on the board, but mounted onto the surface on flat pads. Surface-mount boards require very precise manufacturing techniques, generally being produced on automatic assembly machines.

The small outline integrated circuit (SOIC, Fig. 14.9(b)) is a surface-mount DIL package with a pin pitch of 0.05". The smaller shrink small outline package (SSOP, Fig. 14.9(c)) package has a pin pitch of 0.026". Quad flat pack (QFP) is a square surface-mount package for larger chips, such as the 44-pin PIC 16F877, with pins on four sides.

![Figure 14.9](image-url)
14.3 Price

Price is determined by the complexity of the chip, but also by the volume of production. As the microcontroller board is continuously updated, each design can be superseded by a chip with better features, as well as having less features, before slipping into obsolescence.

For example, at the current time the guide price quoted for the 16F84 is $4.39, while the pin-compatible replacement, the 16F818, which has analogue inputs and other extra features, is only $1.71. Therefore, while the old chip has been used as an example, because it is less complicated, the reader should consider using the more recent chip in his or her own design, even if the new features are not used in the full. It does mean, however, that the data sheet may be more difficult to understand, because it has to cover all the additional features.

The relative cost for each chip shown in Table 14.2 is based on the ‘budgetary price’ quoted by the manufacturer at the time of writing. The figures allow relative costs to be compared, while the actual price will obviously increase in time.

14.4 Serial Communications

Serial communications ports allow the PIC to communicate with other devices, or exchange data with a master controller, via a single connection. There are several protocols available:

- USART (universal synchronous/asynchronous receiver/transmitter)
- SPI (serial peripheral interface)
- I²C (inter-integrated circuit)
- CAN (controller area network)

14.4.1 USART

RS232 is a long established asynchronous communication protocol used at the serial (COM) port of the PC. It is low speed, but is easy to understand and has been used as the standard serial communication method between computers, terminals and other systems. It is used to download programs to the 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. Serial data is sent and received as individual bytes using a pair of shift registers (Fig. 14.10(a)). After each bit of the data 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 bit is present. If it does not then take the next sample after a specified interval. The ‘head start’ sets this time interval, at 15000 bits per second, the time interval is about 50μs. The sender and receiver must be set up to operate at the same baud rate; there are a set of standard baud rates of between 300 and 57600 bits per second. The baud rate is determined in Fig. 14.10(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 byte, and is re-triggered for each byte.

When the USART is operated in asynchronous mode (Fig. 14.11), there is a separate data path for send (TX) and receive (RX), which are symmetrical in operation. One byte of data is transmitted at a time down a serial line, with start, stop and optional error check bits. In synchronous mode, the TX pin is used instead to carry a clock (CK) signal, which is sent with
More PIC Microcontrollers

Serial data line
Parallel load
Parallel readSend
Shift out
Idle
Start bit
Stop bit

Data Sample data line

Figure 14.10 Serial data transfer. (a) Shift register operation; (b) Serial data signal.

Host Transmit data, TXD Receive data, RXD
Peripheral RXD TXD

Figure 14.11 USART asynchronous mode connections.

the data to control the receiver, making the process more reliable. In this mode, each device can still send and receive, but only one at a time.

The RS232 type signal data signal produced by the PIC USART is output at TTL levels. Other terminals, such as the PC, will produce a signal which is transmitted at a voltage up to around \( +24 \) 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 also shift the level as required.

14.4.2 SPI

The SPI system uses three pins:

- serial data out (SDO)
- serial data in (SDI)
- serial clock (SCK)

It is a single master, multi-slave system using hardware slave selection (Fig. 14.12). To communicate, all slave outputs, \( M \) it must select a slave by pulling the slave select data line (SSD) high. The slave latch is then set to the desired register. 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.
Serial Communications

Master

Serial data out, SDO
Serial data in, SDI
Serial clock, SCK
Slave select

Slave 1
- SDI
- SDO
- SCK
- !SS

Slave 2
- SDI
- SDO
- SCK
- !SS

7 6 5 4 3 2 1 0

(a)

Figure 14.12 Serial peripheral interface (SPI). (a) SPI connections; (b) SPI signals.

14.4.3 I2C

The I2C system uses two pins:
- serial data (SDA)
- serial clock (SCL)

This system also uses synchronous master/slave communication, but with a software
addressing system (Fig. 14.13). As in a network, the destination address for transmitted data in
a multi-slave system is transmitted on the same line (SDA) prior to the data. A seven or ten-bit
address can be used (up to 1023 slaves), which must be programmed into an address register in
each device. The clock can operate up to 1 MHz. This system is suitable for communication
between separate microcontroller boards, since no slave selection hardware connections are
needed. 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 operations on the serial
data and clock lines.

14.4.4 CAN

The CAN system uses two pins:
- CANTX (Transmit)
- CANRX (Receive)

The CAN system (Fig. 14.14) is designed for transmitting signals in electrically noisy
environments, such as motor vehicle control, using differential current drivers. It is only
available in selected high-performance PIC 18-series devices. A more complete description and
More PIC Microcontrollers

Master
Slave1
Slave2
+5V
SDA
SCL
76543 210
Acknowledgment
Address/data bits
Start

Figure 14.13
Inter IC bus (I2C). (a) I2C connections; (b) I2C signals.

Electronic control unit  ECU1
TX  RX
ECU2
TX RX
CAN_H
CAN_L
Actuators
Sensors

Figure 14.14
CAN bus system.

Information for setting up these serial communication interfaces is provided in the PIC 16F87X, and other PIC data sheets.

Summary
• The PIC family of microcontrollers share a common core architecture and instruction set. The 18XXXX group of high-power MCUs have an enhanced architecture and instruction set.
• All PICs are designed with separate program and data busses, a reduced instruction set, and two-stage pipelining to improve performance.
• To select a device for a given application, the number of I/O pins, the program memory type and size, and special I/O requirements, such as analogue input and PWM output, must be established.
Serial Communications

- In-circuit programming is a common feature, with in-circuit debugging in selected devices.
- A range of serial bus communication interfaces is available in selected chips, as well as a parallel bus for operation within a master/slave system.

Questions

1. Summarise the differences between the PIC 12-, 16- and 18-series of microcontrollers.
2. Describe the features in the PIC chip which help to ensure that the program starts reliably.
4. From the table of PIC flash microcontrollers, name (a) a device which has eight analogue inputs in a package with fewer than 18 pins, and (b) another device which could control 10 PWM motor outputs, runs at 40 MHz and can be programmed in C.
5. What is the minimum time required to read four analogue inputs?
6. State two factors that limit the clock speed in a microcontroller.
7. Describe the essential difference between SPI and I²C addressing. Which has more complex hardware requirements?

Activities

1. Download the data sheet and print the summary page for the PIC 12F675, 16F818 and 18F8720. Summarise the features and suggest a typical application, for each device.
2. A robot has five motors which are PWM driven with an analogue position sensor on each axis. Select the cheapest PIC chip from Table 14.2 which could be used as a controller for the entire positioning system. Download the data sheet and draw a block diagram for the system, identifying the pins which should be connected to the motors and sensors.
3. Sketch a block diagram for an alternative implementation of the robot controller in Activity 2 using a separate controller for each axis connected to a master SPI controller. Select the cheapest suitable chip for the slave controllers. If the pot on one axis rotates through 300 degrees, calculate the smallest movement which can be detected by the controller using a 10-bit converter. Assume the sensor pot is attached directly to the robot axis. Suggest an advantage of the master–slave system over the single controller solution proposed in Activity 2.

Answers

Question 5. \(4 \times 20 = 80\) µs

Activity 3. \(300/1024 \approx 0.3\)
Chapter 15

More PIC Applications and Devices

15.1 16F877 Application

The 16F877 is at the top of the range in this group and will therefore be used to illustrate the range of features available within the 16 series, which have already been described in Chapter 14. Other chips in this group have different combinations of these features; the intention is to help the reader to make the best choice of chip for any given application.

The temperature controller described here uses most of the available I/O provided, including analog inputs. The 8k memory should be sufficient for most applications which might be developed for this hardware. A demonstration program is provided which will exercise the hardware for test purposes, but it will be left to the reader to develop a fully functional application.

15.1.1 Temperature Controller System

A temperature controller is required to control a system such as a greenhouse where the temperature must be kept within set limits (0–50°C) by a heating and ventilation system. The unit will be programmed to accept a maximum and minimum temperature, or a set temperature and operating range. The system operates on the average temperature reading from
A temperature control system is maintained by a switched heater, switched vent and a fan which can be speed-controlled. The system should operate as specified in Table 15.1. 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. It needs to be connected to a PWM output on the controller if the rate of forced heating and cooling is to be varied. A demonstration system was constructed, where the heater was represented by a pair of filament lamps and the fan by a 5 V CPU fan. The temperature sensors used were standard LM35 devices which keep a built-in amplifier which outputs 10 mV per °C. Figure 15.2 shows the interfacing requirements for the application. The temperature sensor readings can be averaged, or processed with a weighting factor for each, to give a representative value for the measured temperature. A heater is then controlled via a suitable interface. A relay can be used if sufficient current is needed. If proportional control is required, a PWM output would be required. In the hardware design provided here, the heater and vent interfaces are implemented as normally open relays, or that an external power supply can be used. The fan output demonstrates the alternatives to solid state relays, using a general purpose power FET. This would allow proportional control, but the internal circuit must be external at 5 V. For PWM control, the FET output would here to be re-assigned to one of the PWM outputs on the PIC 16F877.
15.1.2 I/O Allocation

The I/O functions provided by the PIC 16F877 are detailed in Table 15.2. These were then mapped against the requirements of the application, and the most convenient grouping decided, giving the I/O allocation in Table 15.3.

Table 15.2  16F877 pin functions

<table>
<thead>
<tr>
<th>Pin label</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA0/AN0</td>
<td>Digital I/O or analogue input 0</td>
</tr>
<tr>
<td>RA1/AN1</td>
<td>Digital I/O or analogue input 1</td>
</tr>
<tr>
<td>RA2/AN2/Vref</td>
<td>Digital I/O or analogue input 2 or positive reference voltage for ADC</td>
</tr>
<tr>
<td>RA3/AN3/Vref+</td>
<td>Digital I/O or analogue input 3 or negative reference voltage for ADC</td>
</tr>
<tr>
<td>RA4/T0CKI</td>
<td>Digital I/O or timer 0</td>
</tr>
<tr>
<td>RA5/AN4/SS</td>
<td>Digital I/O or slave select input (SPI mode)</td>
</tr>
<tr>
<td>RB0/INT</td>
<td>Digital I/O or external interrupt input</td>
</tr>
<tr>
<td>RB1</td>
<td>Digital I/O</td>
</tr>
<tr>
<td>RB2</td>
<td>Digital I/O</td>
</tr>
<tr>
<td>RB3/PGM</td>
<td>Digital I/O or select serial programming mode</td>
</tr>
<tr>
<td>RB4</td>
<td>Digital I/O (interrupt on change)</td>
</tr>
<tr>
<td>RB5</td>
<td>Digital I/O (interrupt on change)</td>
</tr>
</tbody>
</table>

Figure 15.2  Temperature controller interfacing.

Figure 15.3  Temperature controller interfacing.
The interfacing for this application is typical of that required for simple control systems using microcontrollers. Fortunately, this application does not need any analogue signal conditioning on the input side, as the temperature sensors can be connected directly to the PIC. Other references on interfacing will cover the range of design techniques needed for the most common sensors and output devices.

### 3.3.3 Temperature Controller Circuit Description

Figure 15.3 shows the circuit for the temperature controller. Each section of the circuit is described separately.

<table>
<thead>
<tr>
<th>PIN</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RB6/PGC</td>
<td>Digital I/O or in-circuit debugger or serial programming clock input (interrupt on change)</td>
</tr>
<tr>
<td>RB7/PGD</td>
<td>Digital I/O or in-circuit debugger or serial programming data input (interrupt on change)</td>
</tr>
</tbody>
</table>

**Port C (8 bits)**

<table>
<thead>
<tr>
<th>PIN</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC0/T1OSO/T1CKI</td>
<td>Digital I/O or Timer 1 oscillator output or Timer 1 clock input</td>
</tr>
<tr>
<td>RC1/T1OSI/CCP2</td>
<td>Digital I/O or Timer 1 oscillator input or Capture 2 input or Compare 2 output or PWM2 output</td>
</tr>
<tr>
<td>RC2/CCP1</td>
<td>Digital I/O or Capture 1 input or Compare 1 output or PWM1 output</td>
</tr>
<tr>
<td>RC3/SCK/SCL</td>
<td>Digital I/O or Synchronous serial clock input or output in SPI and I²C modes</td>
</tr>
<tr>
<td>RC4/SDI/SDA</td>
<td>Digital I/O or SPI data input or I²C data I/O</td>
</tr>
<tr>
<td>RC5/SDO</td>
<td>Digital I/O or SPI data output</td>
</tr>
<tr>
<td>RC6/TX/CK</td>
<td>Digital I/O or USART asynchronous transmit or USART synchronous clock I/O</td>
</tr>
<tr>
<td>RC7/RX/DT</td>
<td>Digital I/O or USART asynchronous receive or USART synchronous data I/O</td>
</tr>
</tbody>
</table>

**Port D (8 bits)**

<table>
<thead>
<tr>
<th>PIN</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RD0/PSP0</td>
<td>Digital I/O or parallel slave port bit 0</td>
</tr>
<tr>
<td>RD1/PSP1</td>
<td>Digital I/O or parallel slave port bit 1</td>
</tr>
<tr>
<td>RD2/PSP2</td>
<td>Digital I/O or parallel slave port bit 2</td>
</tr>
<tr>
<td>RD3/PSP3</td>
<td>Digital I/O or parallel slave port bit 3</td>
</tr>
<tr>
<td>RD4/PSP4</td>
<td>Digital I/O or parallel slave port bit 4</td>
</tr>
<tr>
<td>RD5/PSP5</td>
<td>Digital I/O or parallel slave port bit 5</td>
</tr>
<tr>
<td>RD6/PSP6</td>
<td>Digital I/O or parallel slave port bit 6</td>
</tr>
<tr>
<td>RD7/PSP7</td>
<td>Digital I/O or parallel slave port bit 7</td>
</tr>
</tbody>
</table>

**Port E (3 bits)**

<table>
<thead>
<tr>
<th>PIN</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE0/RD/AN5</td>
<td>Digital I/O or PSP read select or analogue input 5</td>
</tr>
<tr>
<td>RE1/WR/AN6</td>
<td>Digital I/O or PSP write select or analogue input 6</td>
</tr>
<tr>
<td>RE2/CS/AN7</td>
<td>Digital I/O or PSP chip select or analogue input 7</td>
</tr>
</tbody>
</table>

**Port F (1 bit)**

<table>
<thead>
<tr>
<th>PIN</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RF0/AN0</td>
<td>Digital I/O or FET output or analogue input 1</td>
</tr>
<tr>
<td>RF1/AN1</td>
<td>Digital I/O or FET output or analogue input 2</td>
</tr>
<tr>
<td>RF2/AN2</td>
<td>Digital I/O or FET output or analogue input 3</td>
</tr>
</tbody>
</table>
The four temperature sensors are allocated to four of the eight analogue inputs available on the chip. In the demo system, standard sensors with an output of 10 mV per °C were used (0 °C = 10 mV). This relatively low voltage is acceptable if the sensors are not connected on long leads, which could pick up electrical noise. 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. Alternatively, or in addition, screened leads could be used. The inputs are protected by a low pass RC filter; the input impedance at the ADC is high enough for this to have negligible effect on the input voltage.

The ADC normally operates at 10-bit resolution, giving output values 0–1024. It needs reference voltages to set the maximum and minimum values for the input conversion. These can be provided internally as Vdd and Vss (supply values), but Vdd does not give a convenient conversion factor. Therefore, an external reference value is provided from a 2.7 V zener diode and potential divider, giving Vref+ which is adjusted to 2.048 V. This then gives a conversion factor of 2048/1024 = 2 mV per bit. To simplify the software, and to cover the correct range, only the low eight bits of the ADC result will be used, with a maximum value of 255. At 50 °C, the input will be 500 mV = 250, giving a resolution of 0.2 °C per bit. For test purposes, a set of four on-board pots are provided, so that input voltages in this range can be input 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 DIP switches.

### Table 15.3 Temperature controller I/O allocation

<table>
<thead>
<tr>
<th>Device</th>
<th>Function</th>
<th>16F877 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Temperature</td>
<td>RA0, RA1, RA2, RA5, AN0, AN1, AN2</td>
</tr>
<tr>
<td></td>
<td>0–512 mV/°C</td>
<td>AN4</td>
</tr>
<tr>
<td></td>
<td>Heater</td>
<td>RE0</td>
</tr>
<tr>
<td></td>
<td>Switched output</td>
<td>Digital output</td>
</tr>
<tr>
<td></td>
<td>Vent</td>
<td>RE1</td>
</tr>
<tr>
<td></td>
<td>Switched output</td>
<td>Digital output</td>
</tr>
<tr>
<td></td>
<td>Fan</td>
<td>RE2</td>
</tr>
<tr>
<td></td>
<td>Switched output</td>
<td>Digital output</td>
</tr>
<tr>
<td>4 x 3 keypad</td>
<td>Read column</td>
<td>RD3, RD4, RD5, RD6</td>
</tr>
<tr>
<td></td>
<td>Scan row</td>
<td>RD0, RD1, RD2</td>
</tr>
<tr>
<td>2 x 7-segment display</td>
<td>Segments</td>
<td>RC1–RC7</td>
</tr>
<tr>
<td></td>
<td>Digit select</td>
<td>RB1, RB2</td>
</tr>
<tr>
<td></td>
<td>Buzzer</td>
<td>RB0</td>
</tr>
<tr>
<td></td>
<td>Audio alarm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IC/P/D interface</td>
<td>RB3, RB6, RB7, N/A</td>
</tr>
</tbody>
</table>

### Analogue Inputs

The four temperature sensors are allocated to four of the eight analogue inputs available on the chip. In the demo system, standard sensors with an output of 10 mV per °C were used (0 °C = 10 mV). This relatively low voltage is acceptable if the sensors are not connected on long leads, which could pick up electrical noise. 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. Alternatively, or in addition, screened leads could be used. The inputs are protected by a low pass RC filter; the input impedance at the ADC is high enough for this to have negligible effect on the input voltage.

The ADC normally operates at 10-bit resolution, giving output values 0–1024. It needs reference voltages to set the maximum and minimum values for the input conversion. These can be provided internally as Vdd and Vss (supply values), but Vdd does not give a convenient conversion factor. Therefore, an external reference value is provided from a 2.7 V zener diode and potential divider, giving Vref+ which is adjusted to 2.048 V. This then gives a conversion factor of 2048/1024 = 2 mV per bit. To simplify the software, and to cover the correct range, only the low eight bits of the ADC result will be used, with a maximum value of 255. At 50 °C, the input will be 500 mV = 250, giving a resolution of 0.2 °C per bit. For test purposes, a set of four on-board pots are provided, so that input voltages in this range can be input 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 DIP switches.

### Outputs

Two types of output are provided: relay and FET. The relay gives a switched output that is isolated (electrically separated) from the controller. The external circuit operates with a separate...
Figure 15.3 - 16F877 temperature controller circuit diagram.
supply, or the load (heater in the case) can be powered from a high-voltage supply if necessary. The relay also provides a high off resistance (air gap). The FET interface, on the other hand, is more reliable, as it is solid state. The problem is that the load bus to operate from the same supply as the FET, the 5 V board supply. It also does not provide electrical isolation between the controller and the load, unless an opto-isolator is included between the FET and the MCU. However, the FET can be switched at high frequency, while the relay cannot. The outputs include an on-board LED to indicate their state, in case the state of the outputs cannot be seen or are not connected.

Keypad
The 12-button keypad allows the user to input the required temperature and other operational parameters as required by the application program. The target temperature would typically be input as a 2-digit number. It may also be desirable to input upper and lower limits, alarm levels, and so on. These should be displayed as they are entered, to ensure that the correct figures are stored. The keypad is simply a set of switches connected in a row and column arrangement (see Chapter 12) 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). If a '0' is output on each row in turn, and a button pressed, that '0' will appear at the column output, and can be read into the PIC. The combination of active row and column identifies the key. The demonstration program (Program 15.1) includes a simple keypad scanning routine.

Display
A seven-segment display is used as it is relatively easy to drive, compared with a liquid crystal display, and is self-illuminating. The decoding process has been covered in Chapter 12, where a code to illuminate the segments to display digits 0–9 is looked up in a program data table. 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 in the Q1 and Q2, because they are stretch activated for the eye to perceive as being on at the same time, albeit at reduced brightness.

Other Circuit Elements
A buzzer is fitted to provide an audible alarm output. This can be used to signal system failure, temperature too low for too long and so on. Audible feedback from keystrokes is also desirable. An LED clock is used to give 100 ms increments of time, which is sufficient. 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 for via the ICD connector. The ICD module must be connected between the host PC and the application board. MPLAB IDE can then be used for testing the program in software initially, and then finally in ICD mode (see Chapter 14).

15.4 Hardware Development
The circuit was developed using Labcenter™ ISIS schematic capture software, which provides interactive components for circuit testing, and integrated software and hardware testing. When the circuit had been tested by interactive simulation, a stripboard implementation was devised (Fig. 15.6).
A demo target system was then constructed, comprising two filament lamps as the heaters, operating from a high current 5 V supply, controlled by the relay output on the application board. A 5 V DC 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 Fig. 15.5. Note that there was a sensor output on the fan which could be used to monitor the actual fan speed if a suitable interface were designed to convert the fan sensor pulse to TTL levels. The vent was not physically implemented at this stage.
More PIC Applications and Devices

Sensor 2 output (0–512 mV)  
Heater +5 V (1A)  
Sensor 1 output (0–512 mV)  
0 V  
Fan sensor output  
Fan control (0 V = On)  
+5 V (100 mA)  
Sensor 3 output (0–512 mV)  
Heater 0 V  
Sensor 4 output (0–512 mV)  

Figure 15.5  Greenhouse simulator wiring.

The photo of the prototype system (Fig. 15.6) shows the simulator at the right of the picture, with the ICD module (enclosed in ABS box), which is connected to the I/O output of the TEMPCON board. 5V power supplies and a host PC would complete the system.

When final hardware testing was completed, an application board was created using Labcenter ARES™ PCB layout software, shown in Fig. 15.7. This incorporated an on-board +5V supply for operation from a mains adapter.

15.1.5 Temperature Controller Test Program

Program 15.1 was written to establish the hardware and to help the reader start in developing applications for the TEMPCON hardware.

The program will read the analogue inputs and display the raw data on the displays. An apparently random pattern results, which changes if the analogue inputs (test pots) are varied, indicating that the hardware input and display interfaces are working. 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. Key 9 will enable the buzzer test, while '=' and '=' will operate the heater, fan and vent, respectively. A full header has been included with as much information as possible, details of target system, program description, register initialization, port allocation and so on. The ports and analogue control registers have been included using both selections, as recommended.
Figure 15.6 TEMPCON system. (a) Stripboard version of temperature controller board; (b) Temperature controller system with ICD module and dummy load.
Program 15.1  Test program for TEMPCON Board

Test program for PIC16F877 Controller Board using ICD system

- Press keypad buttons 1-4 to display raw data from input pots
- Press keypad button 5-6 to display raw data from input pots

Circuit description:
- PIC 16F877 flash microcontroller
- 4 analogue inputs from temp sensors (or test input pots)
- Control heater, vent and fan in a target system such as a greenhouse
- Target temperature set up using keypad input
- Displayed on 2-digit multiplexed LED display.
16F877 Application

Buttons:
- 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

PROGRAMMING OPTIONS
- When programming PIC 16F877 in ICD mode, select:
  - XT clock mode (4MHz, 1us per instruction)
  - Power-up timer enabled
  - Watchdog timer disabled
  - Code Protection off

I/O ALLOCATION

INPUTS: Analogue temp sensors AN0 - AN3 (0 - 5V)
Keypad column detect RD0 - RD2 = 0

OUTPUTS: Buzzer RB0 = toggle
Keypad row select RD3 - RD6 = 0
7-segment display Select lo digit RB1 = 1
Select hi digit RB2 = 1
Segments RC1 - RC7 = 0
Relay interfaces Heater RE0 = 1
Vent RE1 = 1
FET interface Fan RE2 = 1

PORT DATA DIRECTION CODES REQUIRED

TRISA = 11111111
TRISB = 11111000
TRISC = 00000000
TRISD = 00000111
TRISE = 00000000
RB3, RB6, RB7 reserved for ICD operation

ADC SETUP

- ADCON0 Bits 7 6 5 4 3 2 1 0 = A/Dclock = f/8
- Bits 5 4 3 Channel Select (AN0 - AN7)
  - Select 0 - Digital
  - Select 1 - Analogue
- Bit 2 Go = 1 / Done = 0
- Bit 0 A/D module enable = 1

ADCON0 = 01xxx001 depending on channel required

ADCON1 Bit 7 6 5 4 3 2 1 0 = Select data in ADRESH/ADRESL
- Bits 3 2 1 0 = RA0-RA5 analogue, RE0-RE2 digital
  - Bits 3 2 1 0 = 0010

RESERVED

- A5, A6, A7 reserved for ICD operation

BYTE DECODE

- A6 = 0x78
  - Set A5 to 0/1
  - Set A6 to 0/1
- A5 = 0
  - Set A6 to 0/1

ADDITIONAL NOTES
- All quantities apply in BINARY/16-HEX.
- All offsets are hexadecimal, 8-bit or 16-bit offset
ASSEMBLER DIRECTIVES

; Configuration file and custom processor:
; list p=16f877
; include file containing register labels:
; include "p16f877.inc"

count EQU 020 ; assign GPR1 for counter

; Start program

nop ; No op. required at 000 for ICD mode

; Initialize control registers

banksel TRISA ; Select DDR register bank 1
movlw b'11111000' ; Setup buzzer and display digits
    movwf TRISB ; select...
    clrf TRISC ; Setup 7-segment driver port as outputs
    movlw b'00000111' ; Setup keyboard port for...
    movwf TRISD ; ..row outputs and column inputs
    clrf TRISE ; Setup relay port as outputs

; Setup ADC

banksel ADCON1 ; Select register bank 1
movlw b'00000010' ; Set A/D mode left justify, 4 channels
    movwf ADCON1 ; and write A/D control word
    banksel ADCON0 ; Select register bank 0
    movlw b'01000001' ; Set A/D frequency Fosc/8, select AN0
    movwf ADCON0 ; and write A/D control word

; Initialize outputs

banksel PORTA ; select port data register bank 0
clrf PORTE ; switch off all outputs
    goto start ; jump over subroutines to main loop

; Subroutine to wait about 0.8 ms

del8 clrf count ; Load time delay of 256 x 3 = 768 us
    again decfsz count ; Decrement and test counter
    goto again ; until zero
    return ;

; Subroutine to get analogue input

getAD movlw 007 ; Load time delay of 7 x 3 = 21 us
    movwf count ; Load counter
    ; Subroutine to get analogue input...

; Main 20ms ADC acquisition timing loop:

delay movlw 0ff ; load time delay of T = 200 ns
    movf count ; delay
    return ;

; Subroutine to get analogue input...

; Main 20ms ADC acquisition timing loop:
down decrement and test counter

goto down

; Get analogue input ..

bsf ADCON0, GO

wait btfsc ADCON0, GO

goto wait

return ; from subroutine with result in

ADRESH

; Subroutines to process keys ...................................

proc1 movlw b'01000001' ; Select analogue channel 1
movwf ADCON0 ; and

call getAD ; and get analogue input

return ; for next key

proc2 movlw b'01001001' ; Select analogue channel 2
movwf ADCON0 ; and

call getAD ; and get analogue input

return ; for next key

proc3 movlw b'01010001' ; Select analogue channel 3
movwf ADCON0 ; and

call getAD ; and get analogue input

return ; for next key

proc4 movlw b'01011001' ; Select analogue channel 4
movwf ADCON0 ; and

call getAD ; and get analogue input

return ; for next key

proc5 movlw b'01000001' ; Select analogue channel 1
movwf ADCON0 ; and

call getAD ; and get analogue input

return ; for next key

proc6 movlw b'01001001' ; Select analogue channel 2
movwf ADCON0 ; and

call getAD ; and get analogue input

return ; for next key

proc7 movlw b'01010001' ; Select analogue channel 3
movwf ADCON0 ; and

call getAD ; and get analogue input

return ; for next key

proc8 movlw b'01011001' ; Select analogue channel 4
movwf ADCON0 ; and

call getAD ; and get analogue input

return ; for next key

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

continued ...
More PIC Applications and Devices

; 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 of keypad
    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 proc5 ; process key 5
    key6 btfsc PORTD,2 ; test key 6
            goto key7 ; next if not pressed
            call proc6 ; process key 6
; scan row C of keypad ..........
    key7 bsf PORTD,4 ; deselect row B
            bcf PORTD,5 ; select row C
            btfsc PORTD,0 ; test key 4
            goto key8 ; next if not pressed
            call proc7 ; process key 4
The routine to read an analogue input is based on the model routine provided in the data sheet, where a delay of about 2μs is included to ensure that the input has had time to settle, in case the input is changing rapidly. The conversion is then 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 ‘left justified’ to place the most significant 8 bits in the ADRESH register for output to the display. This allows the full range of the input (0–2.048 V) to be checked.

In a working program, the analogue input value would be converted into a 2-digit decimal value for the display. Using the conversion scaling calculated above, 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 full ADC range is then being used. This result can then be converted into the corresponding display digit ‘5’ and ‘0’, and so on down to ‘0’. The keyboard-scanning routine uses a simple method to check if each key in each row has been pressed, calling the required action if it has. A more elegant and compact keyboard-scanning method is possible according to 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. Pseudocode for the application software is shown in Program 15.2.

**Program 15.2 Pseudocode for TEMPCON Control software**

**TEMPCON**

*Initialise*
- 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 Left 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 sensor1
- Read sensor2
- Read sensor3
- Read sensor4
- IF sensor out of range
  - replace with previous value
- Calculate AverageTemp
- Display AverageTemp
- MSD = AverageTemp/10
- Get 7-seg code & display MSD
15.1.6 Application Modifications

There are some features of the microcontroller which are not yet utilised in this application, which could enhance it. As mentioned above, the PWM module could be used to control the speed of the fan. In addition, a serial communications port could send the temperature data to a remote computer which could be used to display or operate as a supervisory system. As the PIC16F877 can be programmed or reprogrammed from the same memory space, it could be used as a master controller, and the PIC16F818 could be used as a slave. The PIC16F877 could then control the PIC16F818 and itself via a network to a supervisory system.

When designing the application initially, a top-of-the-range device such as the 16F877 is a good choice, as it has most of the available features. When the design has been finalised, it may turn out that some features are not required, some I/O is unused or the program could be fitted into a smaller memory. The designer can then review the alternative, cheaper or otherwise more suitable devices, and transfer the application to that device as long as the hardware requirements are not excessive. The software reconfiguration should also not be too much of a problem within the same PIC group, since the chips are designed to be interchangeable. This idea is illustrated below, where the temperature controller is redesigned for other PIC chips.

15.2 16F818 Application

The PIC 16F818 is a replacement part for the 16F84. It has a compatible pin-out, 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 can be seen from the pin-out (Fig. 15.8), each pin has multiple functions, other than the two supply pins. Analogue inputs can be selected or RA0-RA4, or external reference voltages. There is a CCP module and a synchronous serial port offering SPI or I2C 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.

Thus, many of the features of the more powerful 16F877 group are now available in the smaller 18-pin package. It is recommended that, when the user is familiar with all the options available on that chip, it can be used as a default choice when developing new PIC applications, if the number of I/O is sufficient.
This chip could be used in the temperature controller if the keyboard were eliminated, and the set temperature input from a pot via one of the analogue inputs (see Fig. 15.9). A fixed control range might be necessary, as there would be no facility for entering maximum and minimum temperatures. The display digit selection can be reconfigured to use only one output. The application then only needs 16 I/O pins. Operational parameters could be transferred via the serial interface if the display were left out (RB1, RB2, and RB4).

15.3 12F675 Application

The 12-series of PIC microchips offers flash memory in 8-pin packages. At the current time there is limited choice, but no doubt the range will be expanded. The pin-out for the 12F675 illustrates the I/O features available (see Fig. 15.10).

The chip 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 featured.

A temperature controller could be implemented using this chip if only two analogue inputs are used (see Fig. 15.11). 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 necessary.
15.4 18F452 Application

The 18F-series microcontrollers are the most powerful in the Flash family, spanning from the 16-bit, 4k memory 18F1220 to the 68 I/O, 64k memory 18F8720 at the current time. The group offers different combinations of the advanced features, and the larger memory size means that 'C' can be used for application programming. The instruction set of 75 16-bit instructions is designed with this in mind.

A small selection of the available 18FXXXX devices are listed in Table 14.2. At the time of writing there are a total of 33 in production, with several others listed as future products. 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.

As can be seen, the 18-series device has some advantages: 40MHz 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 those advantages may or may not translate into extra 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'. 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.

15.4.1 PIC 'C' Programming

For those readers unfamiliar with 'C' programming, a simple example is given in Program 15.3. The program will do the same job as BIN1.ASM assembly language code. The program must be converted to PIC 16-bit machine code using a compiler such as MPLAB C18 Compiler, which is supplied as an add-on to the development system. The compiler recognises ANSI 'C', the standard syntax for microcontrollers (ANSI = American National Standards Institute). The 'C' compiler must be selected in the development mode dialogue when building the application.

The main elements of the program functioning are as follows.

/* comment */
Comments in 'C' source code are enclosed between /* and */; and can be run over several lines.

#include<p18F456.h>
This is a compiler directive which calls up a header file named 'p18F456.h'. This contains pre-defined register labels such as TRISB and PORTB, and the corresponding addresses 06h and 86h.
### Table 15.4 Comparison of the 16F877 and the 18F458

<table>
<thead>
<tr>
<th>Feature</th>
<th>16F877</th>
<th>18F458</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total pins</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>Input/output pins</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>Ports</td>
<td>A, B, C, D, E</td>
<td>A, B, C, D, E</td>
</tr>
<tr>
<td>Clock</td>
<td>20 MHz</td>
<td>40 MHz</td>
</tr>
<tr>
<td>Instruction bus</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Program memory size (bytes)</td>
<td>8k</td>
<td>16k</td>
</tr>
<tr>
<td>Instruction set size</td>
<td>35</td>
<td>75</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>368</td>
<td>1536</td>
</tr>
<tr>
<td>EEPROM (bytes)</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Interrupt sources</td>
<td>14</td>
<td>21</td>
</tr>
<tr>
<td>Timers</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Capture, compare, PWM modules</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Serial communications</td>
<td>MSSP, USART, CAN</td>
<td>MSSP, USART, CAN</td>
</tr>
<tr>
<td>Parallel port</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Analogue inputs</td>
<td>8 x 10 bits</td>
<td>8 x 10 bits</td>
</tr>
<tr>
<td>Resets</td>
<td>POR, BOR, Stack, Programmed</td>
<td>POR, BOR, Stack, Programmed</td>
</tr>
<tr>
<td>Internal serial programming and debugging</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

---

**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 RAM (GPR) locations.

**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) encloses the main program with a matching brace at the end. These are lined up in the same column and the main program aligned as between them, so that they can be matched up correctly.

**counter = 1;**

A value of 1 is initially placed in the variable location (low byte).

**TRISB = 0;**

A value 0 is loaded into the data direction register of Port B to initialise 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 which controls 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.
Program 15.3 A simple PIC C program

```c
#include <p18f458.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 */
}
```

to the maximum binary value and stopping. The value 1 means the condition is 'always true', so the loop is endless, until reset.

PORTB = counter;
The value in counter is copied to Port B data register for display on the LEDs

counter++;
The variable value is incremented each time the loop is executed. This causes the output to be incremented the next time.

Delay10KTCY(100);
This calls a predefined block of code which provides a delay, so that the LED output changes are visible. As a minimum count, the processor instructions cycle time is 0.1 μs, so the delay works out to 0.1 ms (100 x 100 cycles). The overall cycle time will then take 25 ms. This function is insensitive to variations in timing, so handling of hardware errors and associated operations are not critical.

The delay function is an example of a function call, which is one of the biggest advantages of "C" – the collection of standard routines, which are automatically available, means that the programmer does not have to keep 're-inventing the wheel', or even invent it for the first time – it is ready-made.

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. However, 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 while of the main loop, and any functional sub-block, must be enclosed between braces.
15.4.2 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 can probably be more easily translated into 'C' than assembly language. For example, the conditional control operations defined using 'IF...THEN...ELSE' statements will translate directly, whereas in assembly, it has to be implemented by suitable combinations of 'Bit Test and Skip' with 'JMP' or 'CALL'. In addition, the comparison of the 'average temperature' with the set values can be done in one statement in 'C', but needs a subtract or compare prior to a bit test, which is much more complicated. On the other hand, checking bit inputs is not so easy in 'C' as in assembly, 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. However, a full treatment will not be attempted here. We can now see the advantages of using 'C' with the 18-series PIC. The chips themselves have a good range of peripheral interfaces and other features, and can be programmed more easily using the high-level language.

Summary

- The PIC 16F877 has a good range of peripheral interfaces, including analog inputs, serial ports, CCP and PWM, and in-circuit debugging.
- The application designed around the PIC 16F877 operates as a temperature controller, with four sensors, three outputs, a keypad and 2-digit display.
- The temperature controller can be programmed to maintain the temperature in a heating/cooling system within a set range, display these parameters and operate alarms.
- A similar application can be implemented using the PIC 16F818 without the keypad.
- A similar application can be implemented using the PIC 12F675 without the display.
- The PIC 18F458 has comparable I/O features to the 16F877, but can be programmed in 'C', and runs at twice the speed.

Questions

1. (a) What interfacing modifications are recommended for the LM35 temperature sensor if the connections are over 1 m long?
(b) 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.
2. State one advantage of (a) the relay output and (b) the FET output as configured in the temperature controller.

3. Describe briefly how a multiplexed seven-segment LED display works. Study the test program and explain how it ensures that each digit has the same brightness.

4. State two reasons why the PIC 16F818 should be preferred over the 16F84 for most applications.

5. Compare PIC assembler and ANSI C programming, outlining the advantages of each.

---

1. (b) 125

---

1. Devise a more code-efficient keypad-scanning routine than that in Program 15.1 using the rotate instruction, such that the binary value for keys 0–9 are stored in a suitable register. Design keys '*' and '#' the hex values A and B, respectively.

2. Design and implement the fully functional program for the temperature controller based on the pseudocode provided in Program 15.2. The user will enter upper and lower temperature limits, set the controller to run mode where the outputs are operated to maintain the temperature between these limits. The system should tolerate a fault in one sensor, which puts its output outside the normal operating range. Devise a full design and performance specification for the controller.

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 setup required to use the PWM output of the 16F877, and redesign the hardware as necessary.

4. Implement the redesign of the temperature controller for the 16F818 or the 12F675.

5. Study relevant C programming references, 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 inputs more difficult in C++?
In this chapter, we will look at the range of other technologies available with which to build controller systems. There are numerous families of other microcontrollers which compete with the PIC range. Conventional microprocessor systems may be a better solution for larger microsystems, and the programmable logic controller provides a self-contained device which requires little additional interfacing. The PC itself can also be adopted as a controller host, with suitable additional hardware. The object is to help the reader to select the most appropriate solution for any given control problem.

16.1 Other Microcontrollers

There are several other families of microcontroller which are well established in the embedded systems market. This book has concentrated on the PIC because it is arguably the most suitable for learning about the principles of microcontroller application design and implementation. The skills and knowledge gained with the PIC can then be transferred to other types of microcontroller. Many of the alternative types of MCU are designed principally for complex embedded applications such as motor vehicle engine management, mass-produced consumer goods, and telecommunications. Digital signal processors, which are devices which specialise in high-speed processing of analogue signals, are also becoming widely used in audio and communications systems. The main manufacturers at the current time include National Semiconductor, Texas Instruments, Analogic Devices and Micrel. The product range from other selected manufacturers is outlined below.

16.1.1 Intel 8051

First introduced in 1980, the 8051 is the most well-established and widely used microcontroller. As can be seen in the block diagram (Fig. 16.1), the original design was a mid-range device,
with multiple parallel ports, timers and interrupts and a serial port. The 8051 can be used as 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 8031 was a version of the 8051 without internal ROM, the applications program being stored in an external EPROM. Many other microcontroller manufacturers now supply variants of the 8051.

16.1.2 Motorola MC68HC11

This series is based on the architecture and instruction set of the standard Motorola 68000 microprocessor, which is discussed below. The MC68HC11A is a typical member of the family, offering 4k program ROM, 512 bytes of data, 128 bytes of stack, 16-bit multifunction timers, serial and synchronous serial communications, eight A/D channels and 38 I/O pins. It can also operate as microcontroller or microprocessor mode with external memory access via multiplexed address and data buses at the multipurpose port pins. The Motorola range is well established in motor vehicle and similar applications.
More Control Systems

16.1.3 Zilog Z8
The Z80 microprocessor was for many years the standard 8-bit microprocessor for industrial applications, and the Z8 series is based on its architecture and instruction set. The Z8 Encore! flash MCU family currently ranges from a device with 4k program memory and 11 I/O (Z8F0441) to one with 64k memory and 60 I/O (Z8F6423) running at 20 MHz, most with 10-bit analogue inputs and the usual selection of peripheral interfaces.

16.1.4 Atmel AVR
Atmel also offers 8051-compatible devices as well as a further range of chips including the ATM range of mid-range microcontrollers, the miniature mid-range ‘tiny’ series and the power ‘mega’ group. The AT90 series available in flash memory devices range from 16 to 36 working registers, 128 instructions, 35 working registers, 16 I/O pins, on-chip debugging, serial ports, and two 10-bit analogue converters in a 20-pin package. A typical example is the ATtiny2313: an 8-bit MCU, which has 2k flash program memory, 120 instructions, 32 working registers, 18 I/O pins, on-chip debugging, serial ports, and two timer/counter/compare modules in a 20-pin package. The larger instruction and register set may give the Atmel chip an advantage in some applications, and the AVR series is currently a popular choice where this is the case.

16.2 Microprocessor System
The conventional microprocessor system was the forerunner of the microcontroller. That is, the 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 of a conventional system, where the individual CPU, memory and I/O devices are linked together by system address, data and control busses.

The advantage of the conventional microprocessor system is that it can be designed to suit the particular application. The disadvantage is that it is considerably more complicated, and memory chips may be required. Obviously, the system must continue to grow and become larger as the type of application to be used for the target application increases.

16.2.1 M68000 Hardware
The current PC architecture in its architecture is quite complicated, with a hierarchy of busses operating at different speeds. We will therefore look at a simpler example based on another common microprocessor, the Motorola 68000 (M68k), which has been widely used in both home computers and industrial applications for many years. It is the CISC processor that has most often been used in education and training because of its relatively regular architecture.

The block diagram of a typical development and training system is shown in Fig. 16.2. The target board, a typical development board from a series of chips on board (SCOB) which would typically incorporate six chips on board (SCOB). The 68020 was the first chip in the line, which was later replaced by the 68030. The 68030 is a 32-bit processor, which is used in a wide range of applications, such as personal computers, workstations, and embedded systems. This is controlled by the 68040, which is a 64-bit processor, and provides data transfer and part B of the data control and data lines.
Figure 16.2 M68000 microprocessor demonstration system.

Figure 16.3 View of M6800 board.
The program for the 68000 is prepared on a host PC, in a similar way to the PIC programs.

Assembly language source code is written using a text editor and converted to machine code by
an assembler program. Alternatively, the source code can be written in the high level language,
typically standard 'C'. A compiler then converts the source code initially into assembler code,
which is then assembled. The machine code program created is downloaded via the PC serial
port to the serial port of the target board and hence into its RAM block.

A 'terminal emulator' utility is used for downloading, which also allows the target board to
use the PC screen and keyboard as a text interface for its monitor program. Which functions as
a minimal operating system. Simple monitor commands are used to run and debug the program.
In single step or trace mode, the 68000 can display its register contents on the PC screen.
The PC provides the keyboard, screen, disk storage and printer during program development.

Once an application is up and running on the 68000 board, a user interface may no longer be
required, or if it is, a simple keypad and display may be sufficient. At this stage, the program
can be blown into EPROM to run independently and the PC disconnected.

The block diagram of the M68000 target board is shown in Fig. 16.4, so that it can be
compared with the PIC 16F84 internal architecture. Note that in the PIC, the internal architecture
of the processor is clearly illustrated in the manufacturer’s block diagram, whereas in the 68000
systems, it is embedded within the CPU. Therefore, to see all the details of the 68000 system,
both the internal CPU architecture and the board circuit diagram must be studied.

16.2.1 M68000 program

A simple program for the 68000 system is shown in Program 16.1, so that the syntax for a
CISC processor can be compared with the PIC assembly language. The program has a similar
function to the PIC program BIN2, outputting a binary count to LEDs with a delay.

Comments

The comments are delimited with a '/*'

```
/* tim.ini */
/*
This is equivalent to the include directive in the PIC, it incorporates a file 'tim.ini' which
contains standard register labels, Port A and DirecA. Port A is the 8-bit port data register
and DirecA the data direction register (DDR).
*/
```

```
move.b #$ff,DirecA
/* Move the literal FF into the DDR to set all bits as output. The '.b' means this is a byte operation
(16- and 32-bit words can be moved in the 68000). '#' means this is a literal (immediate data
in 68000 speak). '$' indicates a hex number. Note that in the 68000, a '1' in the DDR set that
bit as output – this is the opposite to the PIC.*/
```

```
move d0,d0
/* The word 'again' is an address (line) label, '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.*/
```
Figure 16.4 Block diagram of M68000 board.
Program 16.1 Simple program for 68000 board

```
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
doctor move.w #$0fff,d1 Initialise delay count
delay subq.w #1,d1 Decrement and bne delay Loop until zero
bra again Repeat forever...........
```

Move a 16-bit word (w) into d1 to initialise the delay loop.

delay subq.w #1,d1

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) was 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.

16.2.3 Program Execution

The M68000 system (Fig. 16.4) is controlled by the CPU driven at 8 MHz. The RAM holds
the user data and program, and the EPROM stores the monitor and communications program
which allows the program in RAM to be downloaded, run and debugged.

The system has a 36-bit data bus, and 24-bit addressing giving a maximum address space
of 16 Mb. The RAM is made up of 64k byte blocks. The EPROM is made up of 8192 byte blocks.
The 68000 instructions are of variable length, 2 to 8 bytes long, stored in adjacent locations.

The lowest memory location 0000-0FFF are used by the system control software in the
68000, so the user’s machine code programs is typically stored in RAM at a set of locations.
The program is executed by the CPU fetching each 16-bit code in turn from memory into the CPU via the data bus. The program codes are found in memory by the CPU sending out the addresses in sequence, starting with $1000, from its program counter register. The address is 'decoded' by the external logic, which is contained partly in a PAL (programmable array logic) chip and partly in the memory chips themselves. In this way, each individual memory location (1 byte) can access for reading and writing.

The address decoder chip generates the system chip select (CS) signals from the address lines A21, A22, and A23, which 'enable' the memory or I/O device which is to be accessed. The lower order address lines are fed directly to the memory chips to select an individual location within the memory array. The microprocessor (CPU) controls the data direction for the data transfer through the CPU and the other chips.

The address decoder chip generates the system chip select (CS) signals from the address lines A21, A22, and A23, which 'enable' the memory or I/O device which is to be accessed. The lower order address lines are fed directly to the memory chips to select an individual location within the memory array. The microprocessor (CPU) controls the data direction for the data transfer through the CPU and the other chips.

Note that in the conventional architecture, the program instructions and data transfers between memory, ports and CPU use the same data bus lines. This slows down the system; this is why, in the PIC design, the instruction and data busses are separated (Harvard architecture), allowing faster operation at the same clock speed.

16.2.4 Ports
The PI/T chip connects the 68000 board to the application board. In a 'real' application, this board would be replaced by the hardware required by that particular application, and a keypad and display connected, if required. The DUART serial port allows the program to be downloaded to RAM from the host PC. The MAX 232 chips are line drivers that boost the signal power on the serial connection between the host and the target board.

The ports can request service from the processor by using the interrupt signal (PIRQ, IRQ) so that more important data transfers can be completed quickly. The interrupt priority encoder converts the active interrupt line number into a 3-bit code which identifies the device requiring service to the CPU, which then runs a corresponding interrupt service routine. Here, the conventional CPU has the advantage it provides an interrupt priority system which allows more important operations to take precedence over less urgent ones.

16.2.5 Bus Control
The 68000 has what is called an 'asynchronous bus'. This means that the memory or I/O data transfer is not completed unless the CPU receives a Data Acknowledge (DTACK) signal from the peripheral chip. The 68000 I/O chips (PI/T and DUART) are designed to provide DTACK, but memory chips are not processor specific in their design, so the DTACK for the memory access cycles must be generated by the PAL decoder. If an interrupt occurs within a process time, the PAL can generate an interrupt signal to the CPU, and the bus cycle is aborted, and an interrupt initiated.

The address and data bus lines are available at an expansion connector so that additional devices can be attached or data passed to another processor. The address decoder chip generates a VPA signal which allows external devices to be added to the decoding system.
16.2.6 M68k Application

Figure 16.5 shows a block diagram of the M68000 board used as a supervisory controller, making use of its multiple I/O facilities and the keyboard and display options. The motor control board is designed to operate six motors under closed-loop speed control. Each motor has its own microcontroller, while the 68000 acts as a master controller. It could be used as a robot drive system, for example.

![Block diagram of M68K motor control system.](image)

In a robot controller, the command to move to a particular position, given as a set of three-dimensional coordinates, could be received via the serial port. The 68000 would work out the move required for each motor (quite complicated!) and then select each motor in turn and download data to move the motors through a calculated angle. The individual controllers could ensure that each joint speeds up and slows down gradually to achieve smooth motion. Motor operating sequences could be stored in the RAM block, while the communication and control firmware would be resident in ROM.

16.3 Control Technologies

Microprocessors and microcontrollers fit into a range of technologies which may be used to implement control systems, which includes:
- electromechanical relays
- programmable logic controllers
Control Technologies

16.3.1 Relay Control

The relay is an electromechanical switching device, which can be used to build simple control systems. A small input current through a coil operates, electromagnetically, a set of contacts which can switch a load (motor, fan, pump, etc.) on and off (Fig. 16.6(a)). Thus a low-voltage input signal controls a high-voltage load circuit, and relays can be wired together to form a sequential control system.

A relay circuit for controlling a machine tool is illustrated in Fig. 16.6(b). The system is designed to provide pushbutton operation and to prevent the main motor from starting unless the machine guard is closed and the cutting fluid pump is on. There is also a 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 1 and Relay 2 are normally open and normally closed contacts, respectively.

![Figure 16.6](https://via.placeholder.com/150)

**Figure 16.6** Relay control system. (a) Relay operation; (b) Wiring diagram of relay machine control system.
Another familiar example of a sequence control system is a domestic washing machine. Traditionally, a motorised rotary switch operates multiple contacts in the required sequence to open valves (filling), switch on motors (washing, spinning and pumping) and heaters. Switched sensors (level, temperature) and safety interlocks (door switch) are also connected. Thus, electro-mechanical devices provide a relatively straightforward solution for simple control applications, or ones where the environment is hostile to delicate electronics. However, switches and relays are inherently unreliable because of the moving parts.

16.3.2 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, but with all the electronics and interconnects built-in (Fig. 16.7). It can be programmed to operate 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, solenoids and other loads must be switched directly from a power supply. The same machine tool used in the previous example is now shown under PLC control in Fig. 16.8.

The PLC has inputs labelled 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, labelled Y0 and Y1, according to the input sequence. The inputs are also simply switched contacts, so in the normally open contact of a relay, which opens a load circuit when switched to supply. They are typically connected to switches, solenoids or other loads. If necessary, the PLC output can control external

Figure 16.7 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 consisting of an LED and photodetector. The on/off signal is passed as infra-red light, giving complete electrical isolation between the input and controller internal circuits.

The control program can be written in ladder logic (Fig. 16.9) which allows the control program to be defined as if the PLC contained the relay system shown in Fig. 16.6. It is a graphical programming method using short cut, normal open contacts, inverted contacts and output coils. These are labelled to associate them with a physical input (Xi) or output (Yo). 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 (not shown in this example) reverses the polarity of the external switch. The PLC will respond as if the ladder program were a wiring circuit. Assuming that the rails of the ladder are supply rails, 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 Fig. 16.9, Y1 will come on when the ‘Start’ button is pressed, if the ‘Stop’ button is also on (normally closed) and the ‘Guard’ switch is closed (guard closed). The contact labelled Y1 (Pump) closes as a result, 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 in run mode. If the motor is overloaded, the thermal cut-out opens and the Y1 coil is de-energised. The output as Y1 maintains contact level. If the guard is opened or the stop button is pressed, both the motor and the pump go off.

Ladder programming is a relatively easy method for creating this type of sequential control program, compared with assembler or ‘C’ language. There are other graphical programming methods which are even easier; refer to PLC programming references for more information.
For comparison with other control technologies, Fig. 16.10 shows the 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 pulling resistors. The microcontroller can then be programmed (Program 16.2) to operate the output loads via suitable interfaces which allow its outputs to switch the high-power motors. High-current FETs are useful here, as they can operate with 5 V gates and have no moving parts. The microcontroller has to be programmed in its native language, or 'C', both of which take time to learn, as you know! This is why ladder logic was developed for programming PLCs.

### 16.3.4 PC Host Control

The PC can be used to control external hardware directly via a suitable interfacing card fitted in an expansion slot on the motherboard (Fig. 16.11). The ISA bus has been used for this since the PC architecture was first devised, but the PCI bus, which is more compact, reliable and faster, is now taking over.

The I/O card connects directly into the peripheral bus of the PC (address, data and control lines) and incorporates port chips which allow control devices to exchange data with the PC. The I/O card has 16 digital inputs, 16 digital outputs and eight anaologe inputs, and one anaologe output. The digital I/O operates at TTL levels, the same as the microcontroller, so the interfacing requirements are similar, except that the internal +5 V supply of the PC can be used to provide the input operating voltages. The anaologe inputs required an ADC control sequence equivalent to that required by the anaologe inputs to the microcontroller.

---

**Figure 16.9** Ladder program for machine tool control

**Figure 16.10** Microcontroller

**Figure 16.11** PC Host Control
Program 16.2  PIC machine control program

Program to operate a simple machine tool

Assembler directives ........................................

PROCESSOR 16F84
PortA EQU 05
PortB EQU 06

; Initialization

MVI A, B'11111111B' ; Initialize outputs
; in Motor & Pump

continued
More Control Systems

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

; Switch off

PORTB = 0 ; Motor & Pump

; Check Start button

BTFSC PortA,0 ; & wait is not pressed

; Check for Stop

BTFSC PortA,1 ; & restart if pressed

; Check for Guard in place

BTFSC PortA,2 ; & restart if not safe

; Check for Overload

BTFSC PortA,3 ; & switch off Motor if true

; Motor ON

BSF PortB,0

; Pump ON

BSF PortB,1

; Switch off Motor, keep Pump on

BSF PortB,0

; Switch off Motor, keep Pump on

BTFSC PortA,3

; Motor & Pump

PORTB = 0

; Switch off Motor, keep Pump on

BSF PortB,0

; Switch off Motor, keep Pump on

BSF PortB,0

; Switch off Motor, keep Pump on

PORTB = 0

END

Figure 16.11 I/O card for PC-based controller system.

The PC I/O card could then, for example, control a machine, a process or the temperature control system. The simple machine tool can be operated via a similar interface to the microcontroller (see Fig. 16.12). One of the advantages of using the PC is that the input and display would be provided by the PC keyboard and VDU, so the hardware design required is limited to the interfacing. The PC could be programmed in "C" which allows for user interface and graphical functions. A simple standard 'C' program which will run an I/O card is shown in Program 16.3, which can be compared with the PLC and microcontroller programs above.
Figure 16.12 PC-controlled machine system.

The most significant features are the function calls to perform input and output. These are:

```c
inputnum = inport(inreg);
outport(outreg, xx);
```

For input, the standard function 'inport' needs the address of the input register of the port chip on the I/O card. The 'base' address of the card is the address of the first register, and the other register addresses are calculated by adding an offset to this. The base address is 206h (0x206), and input register has an offset of six locations. The input register address (inreg) is therefore 206 + 6 = 222h in the PC memory map. When the function is called, the input binary code present at the I/O card input pins is stored in integer variable 'inputnum'; it can then be processed or checked. The input number in the program is determined by the state of the input switches, giving results in the range 0–15 (4 bits).

The output register has an offset of 13 (0x0D), so the output register address (outreg) is 22Dh (0x22D). The decimal equivalent number for the required output binary code follows (xx). The number sent by the function will then appear at the output terminals of the I/O card. The output can be 0, 1, 2 or 3 to set the pump on or off under binary control.

The program works by reading the inputs and operating the outputs and displaying messages according to the current state of the switches, using the 'if(condition){block}' syntax to perform
Program 16.3 C machine control program

#include <stdio.h>

main()
{
    int base, inreg, outreg, inputnum, outputnum;
    base = 0x220; /* I/O ports base address */
inreg = base + 6; /* Input register address */
outreg = base + 13; /* Output register address */
inputnum = 0;
outputnum = 0;
clrscr();
printf("nMachine Control Program 
"n***************************
"nEnsure guard is closed before 
"nhitting the start button.....
"n Enter to enable controller... 

/* Control Loop **********************************************/
do { inputnum = inport(inreg); /* Read the input..... */
} while (inputnum != 0);
outport(outreg,3); /* Start the machine.. */
do { inputnum = inport(inreg) ; /* Run ............... */
} while (inputnum == 1);
if (inputnum == 3){
    outport(outreg,0);
    printf("n
nStopped ..... 
")
}
if (inputnum == 5){
    outport(outreg,0);
    printf("n

GUARD OPEN .... close and restart 
")
}
if (inputnum == 9){
    outport(outreg,2);
    printf("n
OVERLOAD! Wait for auto reset.... 
")
}
printf("n
Hit space to end, then restart... 

goto();
/* Control Loop ***************************************************/
do { inputnum = inport(inreg); /* Read the input..... */
} while (inputnum != 0);
outport(outreg,3); /* Start the machine.. */
do { inputnum = inport(inreg) ; /* Run ............... */
} while (inputnum == 1);
if (inputnum == 3){
    outport(outreg,0);
    printf("n
nStopped ..... 
")
}
if (inputnum == 5){
    outport(outreg,0);
    printf("n

GUARD OPEN .... close and restart 
")
}
if (inputnum == 9){
    outport(outreg,2);
    printf("n
OVERLOAD! Wait for auto reset.... 
")
}
printf("n
Hit space to end, then restart... 

goto();
/* Finished ******************************************************/
}
a block of code if the condition in the brackets is true. There are many references to help
you learn 'C' programming, and some of the syntax has been explained in Chapter 15, so the
program will not be analysed any further here.

The program is very basic, and would need development to work with a real system. For
example, it does not deal with switch bounce. However, the PC-based system has some
significant advantages over the other implementations: the keyboard and screen are built in,
allowing the system to be controlled from the keyboard and the status displayed in detail using
text messages, or even graphics; any calculations required will also be easier to write in 'C'.

A more powerful solution is to use a graphical control and instrumentation package such as
National Instruments Labview™, which allows the application software to be created without
the need for text-based programming. All the necessary control and monitoring functions are
put together interactively on screen.

As we have seen, the serial (COM) ports on a PC are used to interface with a great variety
of systems – process controllers, instrumentation, machines and target systems of various kinds,
as well as the PC, programming and debugging systems. Using this interface, the PC can
read and write data to and from the target systems, and can be displayed in graphical format, stored on disk or sent on via a network to a higher level supervisory system. The PLCs, for example, are typically programmed and controlled in this
way. However, USB and network connections will tend to take over this type of function in
future. Networking also allows the PC to be controlled or re-programmed remotely, and to
return information to a supervisory system.

16.3.5 Manufacturing and Process Control

Microprocessors and microcontrollers are central to ongoing development of industrial
technology. The term 'microcontroller' refers to machines that are typically designed for one
production task, but increasingly, flexible manufacturing systems are being introduced; they
can produce a range of similar products in smaller quantities at a viable cost. Motor vehicle
assembly is a well-established example.

A basic flexible manufacturing system (FMS) is illustrated in Fig. 16.13. It consists of a
milling machine, a hydraulic assembly rig and a materials-handling robot. It is designed to
machine and assemble a simple product. The mill produces the casing, the robot assembles it
and a cover is fitted in the hydraulic press.

A block diagram (Fig. 16.14) shows how the subsystems 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 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 PLCs, with the main
PLC controlling the whole system. In the system illustrated, the robot controller has two
microprocessors, because of the complex calculations required for the robot movement. The
main PLC has as a minimum two microprocessors, for programming and monitoring. However, the main system PLC remains connected to its host PC, which then operates as a SCADA system to control and display acquisition and machine interface to the system
in running. It provides a virtual control panel and graphical status display of the system as it
runs, using status bits in the main PLC for its information. The system is stopped and started
from the SCADA Virtual control panel.
16.4 Control System Design

One of the disadvantages of the microcontroller and standard microprocessor board such as
the M68k target board is that the hardware is designed to be multipurpose, that is, it has not
been designed for any specific application. This means that there are likely to be features of
the hardware which are not utilised by a particular application, and the user will be paying for unused facilities. The advantage of a conventional microprocessor is that the hardware can be designed at component level to meet the requirements of the application exactly. For example, the amount of memory, the type and number of I/O ports can be tailored precisely to meet the needs of the application. The decision on which type of hardware to use must therefore be made by balancing the up-front cost of customising against the extra development costs of using a tailor-made design. It takes time for the design engineer to become familiar with any new developments.

The PLC offers an off-the-shelf hardware package, normally requiring no external electronics to interface it. It is therefore commonly used in industrial control systems, which divide into two main categories: process control and manufacturing systems. It is robust, easy to install and easily programmed.

The PC itself can also be used as a controller offering a standard operating system, graphics, disk storage, communications and printing. One of the great advantages of the modular design of the PC is that special interface cards can be fitted for control applications. The PC thus can be turned into an oscilloscope, logic analyser, controller or data logger. It is also a natural platform for running design software, both mechanical and electronic CAD. Then, you can design hardware, run a control application, write the support documentation and sell your product to the customer, all using the same machine! In its role as a development platform for PIC microcontroller designs, it is but one of many uses.

Table 16.1 provides a comparison of the advantages and disadvantages of the different forms of system control outlined above.

We can see that a good working knowledge of all these options is required in order to select the most appropriate technology for any given application. The PIC is one microcontroller among many, and the microcontroller is only one solution to any given problem. Nevertheless, the microcontroller is now meeting an increasing range of needs in current industrial and consumer technology, and the PIC is the best device to start learning with—have fun!
<table>
<thead>
<tr>
<th>Technology</th>
<th>Advantages</th>
<th>Disadvantages</th>
<th>Typical applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relays</td>
<td>Simple to design</td>
<td>Slow</td>
<td>Machine safety interlocks</td>
</tr>
<tr>
<td></td>
<td>No programming needed</td>
<td>Unreliable</td>
<td>Simple process control</td>
</tr>
<tr>
<td></td>
<td>Good electrical isolation</td>
<td>High power consumption</td>
<td>High power systems</td>
</tr>
<tr>
<td>PLC</td>
<td>Minimal interfacing needed</td>
<td>Limited processing functionality</td>
<td>Machine control</td>
</tr>
<tr>
<td></td>
<td>Easy to program</td>
<td>Limited interfacing flexibility</td>
<td>Process systems</td>
</tr>
<tr>
<td></td>
<td>Flexible hardware design</td>
<td>Limited memory</td>
<td>Easy to install</td>
</tr>
<tr>
<td>Microcontroller</td>
<td>Large arrays</td>
<td>Limited memory</td>
<td>Flexible manufacturing systems</td>
</tr>
<tr>
<td>Microprocessor</td>
<td>Easy to design</td>
<td>Limited memory</td>
<td>Hardware design skills needed</td>
</tr>
<tr>
<td>Instrumentation</td>
<td>Expandable</td>
<td>Only suitable for complex applications</td>
<td>Automated reactions</td>
</tr>
<tr>
<td>SCADA</td>
<td>Expandable</td>
<td>Suitable for larger systems</td>
<td>Large computers</td>
</tr>
<tr>
<td>PC Host</td>
<td>Expandable</td>
<td>High cost of operation</td>
<td>Networks and distributed systems</td>
</tr>
<tr>
<td></td>
<td>High physical cost</td>
<td>High cost of operation</td>
<td>Large systems user interface</td>
</tr>
<tr>
<td></td>
<td>Good for simple application</td>
<td>Good for complex applications</td>
<td>SCADA built-in data storage and communications</td>
</tr>
<tr>
<td></td>
<td>Graphical programming and display</td>
<td>Basic interfacing skills only needed</td>
<td>Instrumentation systems host</td>
</tr>
<tr>
<td></td>
<td>Standard operating system</td>
<td>Network and distributed system</td>
<td>Basic interfacing skills only needed</td>
</tr>
</tbody>
</table>
1. Outline the differences between the PIC16F84 and the Intel 8051 microcontrollers.

2. State two advantages and two disadvantages of the conventional processor system over the microcontroller in designing a system to meet a particular specification.

3. Explain briefly the advantages of using a PLC compared to a microprocessor system in control applications.

4. Draw a flowchart for the Program 16.2 to show the control sequence clearly.

5. What are the advantages of using the PC as a controller, compared with the microcontroller?

6. Match up the controller type with the most appropriate programming language or technique:
   (a) Small microcontroller  (a) C
   (b) CISC microprocessor  (b) None
   (c) Relay system          (c) Graphical
   (d) PLC                   (d) Assembly
   (e) PC + I/O card         (e) Ladder logic

Activities

1. Log on to the Atmel website and select a microcontroller from the list of available flash devices which is most similar to the 16F84 and compare its features and block diagram. Identify any advantages that the AVR microcontroller has over the PIC.

2. Obtain a copy of the M68000 summary instruction set, and compare it with the PIC 16XXXX instruction set. Identify three instructions which are not available in the PIC.

3. Devise a circuit to switch a motor on and off using push buttons and a single relay (no safety interlocks needed). Why is this safer than using a simple mains switch?

4. Modify the PLC machine tool controller in Fig. 16.8, and its program, to operate an alarm output if the machine stalls or overloads.

5. 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 derive a flowchart for the control sequence, constructed so that it could be implemented in PIC assembly language.

6. By reference to the temperature controller design in Chapter 15, design the hardware for PIC implementation of the system shown in Fig. 16.10. Select a suitable device according to the I/O and memory requirements, test Program 16.2 in the MPLAB simulator and implement the design using the most readily available construction techniques. Derive a simple system to simulate the machine tool and confirm correct operation in hardware.
PIC16F84A
Data Sheet
18-pin Enhanced FLASH/EEPROM
8-bit Microcontroller
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Revision History</td>
<td>1</td>
</tr>
<tr>
<td>3. Signal Description</td>
<td>1</td>
</tr>
<tr>
<td>4. Package Outline</td>
<td>1</td>
</tr>
<tr>
<td>5. Pin Configuration</td>
<td>1</td>
</tr>
<tr>
<td>6. Electrical Characteristics</td>
<td>1</td>
</tr>
<tr>
<td>7. Functional Description</td>
<td>1</td>
</tr>
<tr>
<td>8. Timing and Timing Parameter</td>
<td>1</td>
</tr>
<tr>
<td>9. General Description</td>
<td>1</td>
</tr>
<tr>
<td>10. Features</td>
<td>1</td>
</tr>
<tr>
<td>11. Ordering Information</td>
<td>1</td>
</tr>
<tr>
<td>12. Application Notes</td>
<td>1</td>
</tr>
<tr>
<td>13. Manufacturer Notes</td>
<td>1</td>
</tr>
<tr>
<td>14. Package Options</td>
<td>1</td>
</tr>
<tr>
<td>15. Revision A</td>
<td>1</td>
</tr>
<tr>
<td>16. Revision B</td>
<td>1</td>
</tr>
<tr>
<td>17. Revision C</td>
<td>1</td>
</tr>
<tr>
<td>Appendix</td>
<td>1</td>
</tr>
</tbody>
</table>

**TO OUR VALUED CUSTOMERS**

Nexperia reserves the right to make changes without further notice. For the most current Data Sheet, please refer to our website at http://www.nexperia.com.

For the latest general information, please refer to the Nexperia website at http://www.nexperia.com.

**Warranty**

Nexperia warrants that this product will be free from defects in material and workmanship for a period of three years from the date of purchase. If any product fails to conform to this warranty, Nexperia’s sole liability shall be, at Nexperia’s option, to repair or replace the product. This warranty is void if the product has been subjected to misuse, neglect, accident, or unauthorized alteration or repair.

**Limited Warranty**

Nexperia warrants that this product will be free from defects in material and workmanship for a period of one year from the date of purchase. If any product fails to conform to this warranty, Nexperia’s sole liability shall be, at Nexperia’s option, to repair or replace the product. This warranty is void if the product has been subjected to misuse, neglect, accident, or unauthorized alteration or repair.

**Disclaimer**

Nexperia assumes no responsibility for any errors that may appear in this document, nor does it make any warranty, expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

**Copyright**

Copyright 2023 Nexperia. All rights reserved.

**Revision History**

Revision 1: Initial release

Revision 2: Update to include latest product information.
## PIC16F84A Pinout Description

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>Port</th>
<th>Pin</th>
<th>I/O Type</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RxD0/RxD1</td>
<td>10</td>
<td>19</td>
<td>SDO</td>
<td>18</td>
<td>RXD0: Digital input, RXD1: Digital input.</td>
</tr>
<tr>
<td>TxD0/TxD1</td>
<td>11</td>
<td>20</td>
<td>SDO</td>
<td>18</td>
<td>TXD0: Digital output, TXD1: Digital output.</td>
</tr>
<tr>
<td>T/R</td>
<td>12</td>
<td>21</td>
<td>SDO</td>
<td>18</td>
<td>Selects TX/RX (Transmit/Receive).</td>
</tr>
<tr>
<td>INT0</td>
<td>13</td>
<td>22</td>
<td>I/O</td>
<td>16</td>
<td>Interrupt on change pin.</td>
</tr>
<tr>
<td>INT1</td>
<td>14</td>
<td>23</td>
<td>I/O</td>
<td>16</td>
<td>Interrupt on change pin.</td>
</tr>
<tr>
<td>INT2</td>
<td>15</td>
<td>24</td>
<td>I/O</td>
<td>16</td>
<td>Interrupt on change pin.</td>
</tr>
<tr>
<td>INT3</td>
<td>16</td>
<td>25</td>
<td>I/O</td>
<td>16</td>
<td>Interrupt on change pin.</td>
</tr>
<tr>
<td>INT4</td>
<td>17</td>
<td>26</td>
<td>I/O</td>
<td>16</td>
<td>Interrupt on change pin.</td>
</tr>
<tr>
<td>INT5</td>
<td>18</td>
<td>27</td>
<td>I/O</td>
<td>16</td>
<td>Interrupt on change pin.</td>
</tr>
<tr>
<td>INT6</td>
<td>19</td>
<td>28</td>
<td>I/O</td>
<td>16</td>
<td>Interrupt on change pin.</td>
</tr>
<tr>
<td>INT7</td>
<td>20</td>
<td>29</td>
<td>I/O</td>
<td>16</td>
<td>Interrupt on change pin.</td>
</tr>
<tr>
<td>T1</td>
<td>21</td>
<td>30</td>
<td>I/O</td>
<td>16</td>
<td>Timer clock input.</td>
</tr>
<tr>
<td>T2</td>
<td>22</td>
<td>31</td>
<td>I/O</td>
<td>16</td>
<td>Timer clock input.</td>
</tr>
<tr>
<td>A0</td>
<td>23</td>
<td>32</td>
<td>I/O</td>
<td>16</td>
<td>Analog input.</td>
</tr>
<tr>
<td>A1</td>
<td>24</td>
<td>33</td>
<td>I/O</td>
<td>16</td>
<td>Analog input.</td>
</tr>
<tr>
<td>A2</td>
<td>25</td>
<td>34</td>
<td>I/O</td>
<td>16</td>
<td>Analog input.</td>
</tr>
<tr>
<td>A3</td>
<td>26</td>
<td>35</td>
<td>I/O</td>
<td>16</td>
<td>Analog input.</td>
</tr>
<tr>
<td>A4</td>
<td>27</td>
<td>36</td>
<td>I/O</td>
<td>16</td>
<td>Analog input.</td>
</tr>
<tr>
<td>A5</td>
<td>28</td>
<td>37</td>
<td>I/O</td>
<td>16</td>
<td>Analog input.</td>
</tr>
<tr>
<td>A6</td>
<td>29</td>
<td>38</td>
<td>I/O</td>
<td>16</td>
<td>Analog input.</td>
</tr>
<tr>
<td>A7</td>
<td>30</td>
<td>39</td>
<td>I/O</td>
<td>16</td>
<td>Analog input.</td>
</tr>
<tr>
<td>A8</td>
<td>31</td>
<td>40</td>
<td>I/O</td>
<td>16</td>
<td>Analog input.</td>
</tr>
<tr>
<td>A9</td>
<td>32</td>
<td>41</td>
<td>I/O</td>
<td>16</td>
<td>Analog input.</td>
</tr>
<tr>
<td>A10</td>
<td>33</td>
<td>42</td>
<td>I/O</td>
<td>16</td>
<td>Analog input.</td>
</tr>
<tr>
<td>A11</td>
<td>34</td>
<td>43</td>
<td>I/O</td>
<td>16</td>
<td>Analog input.</td>
</tr>
<tr>
<td>A12</td>
<td>35</td>
<td>44</td>
<td>I/O</td>
<td>16</td>
<td>Analog input.</td>
</tr>
<tr>
<td>A13</td>
<td>36</td>
<td>45</td>
<td>I/O</td>
<td>16</td>
<td>Analog input.</td>
</tr>
</tbody>
</table>

### Notes:
1. The pullup is a Schottky type and is connected under the recessing switch. |
2. The pullup is a Schottky type and is connected under the recessing switch. |
3. This pullup is Schottky type and is connected under the recessing switch. |
4. This pullup is Schottky type and is connected under the recessing switch. |
2.0 MEMORY ORGANIZATION

2.1 Program Memory Organization

The PIC16F877 has a 2K x 14-bit internal program memory space that implements the 8051 microcontroller memory model. The program memory starts at address 0000h and continues up to 3FFFh. The program memory includes the main program space, which is used for the PIC16F877's application code. Additional space is allocated for special function registers (SFRs), which are used for controlling the device's various peripherals. The SFRs occupy addresses from 0000h to 007Fh. The program memory space is organized in a similar way to the PIC16F84A, with the main program space starting at 0000h and the SFRs starting at 0020h.
2.2 Data Memory Organization

The PIC16F84A is organized as shown in Figure 2.3. Each 12-bit memory location is associated with a 16-bit register. The registers include status registers, program memory registers, and data registers. The program memory is 4 Kbytes and can be accessed directly through the program counter. The data memory is 1 Kbyte and can be accessed indirectly through the data pointer.

Table 2.1: Register File Map

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00H-07H</td>
<td>Port A</td>
</tr>
<tr>
<td>08H-0FH</td>
<td>Port B</td>
</tr>
<tr>
<td>10H-17H</td>
<td>Port C</td>
</tr>
<tr>
<td>18H-1FH</td>
<td>Port D</td>
</tr>
<tr>
<td>20H-27H</td>
<td>Port E</td>
</tr>
<tr>
<td>28H-2FH</td>
<td>Port F</td>
</tr>
<tr>
<td>30H-37H</td>
<td>Port G</td>
</tr>
<tr>
<td>38H-3FH</td>
<td>Port H</td>
</tr>
<tr>
<td>40H-47H</td>
<td>Port I</td>
</tr>
<tr>
<td>48H-4FH</td>
<td>Port J</td>
</tr>
<tr>
<td>50H-57H</td>
<td>Port K</td>
</tr>
<tr>
<td>58H-5FH</td>
<td>Port L</td>
</tr>
<tr>
<td>60H-67H</td>
<td>Port M</td>
</tr>
<tr>
<td>68H-6FH</td>
<td>Port N</td>
</tr>
<tr>
<td>70H-77H</td>
<td>Port O</td>
</tr>
<tr>
<td>78H-7FH</td>
<td>Port P</td>
</tr>
<tr>
<td>80H-87H</td>
<td>Port Q</td>
</tr>
<tr>
<td>88H-8FH</td>
<td>Port R</td>
</tr>
<tr>
<td>90H-97H</td>
<td>Port S</td>
</tr>
<tr>
<td>98H-9FH</td>
<td>Port T</td>
</tr>
<tr>
<td>A0H-A7H</td>
<td>Port U</td>
</tr>
<tr>
<td>A8H-AFH</td>
<td>Port V</td>
</tr>
<tr>
<td>B0H-B7H</td>
<td>Port W</td>
</tr>
<tr>
<td>B8H-BFH</td>
<td>Port X</td>
</tr>
<tr>
<td>C0H-C7H</td>
<td>Port Y</td>
</tr>
<tr>
<td>C8H-CFH</td>
<td>Port Z</td>
</tr>
</tbody>
</table>

Figure 2.3: Register File Map

- The register file map is shown in Figure 2.3. The registers include status registers, program memory registers, and data registers.
- The program memory is 4 Kbytes and can be accessed directly through the program counter.
- The data memory is 1 Kbyte and can be accessed indirectly through the data pointer.
2.3 Special Function Registers

The Special Function Registers (SFRs) are used to control and monitor the microcontroller. These registers are not part of the general purpose memory and are accessed using specific addresses.

### Table 2.1: Special Function Register Summary

<table>
<thead>
<tr>
<th>Register</th>
<th>Name</th>
<th>Bits</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFR1</td>
<td>PORTA</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Port A Data Register</td>
</tr>
<tr>
<td>SFR2</td>
<td>PORTB</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Port B Data Register</td>
</tr>
<tr>
<td>SFR3</td>
<td>PORTC</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Port C Data Register</td>
</tr>
<tr>
<td>SFR4</td>
<td>PORTD</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Port D Data Register</td>
</tr>
<tr>
<td>SFR5</td>
<td>PORTE</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Port E Data Register</td>
</tr>
<tr>
<td>SFR6</td>
<td>PORTF</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Port F Data Register</td>
</tr>
</tbody>
</table>

Note: The above table is a summary of the Special Function Registers. For detailed information, refer to the manufacturer's datasheet.
Appendix A

Table 5.1: STATUS REGISTER (ADDRESS 00H, 01H)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Unimplemented</td>
</tr>
<tr>
<td>6</td>
<td>V</td>
</tr>
<tr>
<td>5</td>
<td>P</td>
</tr>
<tr>
<td>4</td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
</tr>
</tbody>
</table>

Legend:
- V: Negative
- P: Parity
- N: Carry
- C: Overflow
- B: Break
- A: Auxiliary
- 8: 8-bit mode

Note: The N, V, P, and C flags are set whenever the RLC, RLR, RRC, and RR instructions are executed.
### Appendix A

#### Register 2.3.2: OPTION REGISTER

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPTION</td>
<td>OPTION</td>
<td>OPTION</td>
<td>OPTION</td>
<td>OPTION</td>
<td>OPTION</td>
<td>OPTION</td>
<td>OPTION</td>
</tr>
</tbody>
</table>

#### Register 2.3.3: OPTION REGISTER (Continued)

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPTION</td>
<td>OPTION</td>
<td>OPTION</td>
<td>OPTION</td>
<td>OPTION</td>
<td>OPTION</td>
<td>OPTION</td>
<td>OPTION</td>
</tr>
</tbody>
</table>

#### Legend:

- R1: R1 = 0
- R1: R1 = 1
- R1: R1 = 0
- R1: R1 = 1
- R1: R1 = 0
- R1: R1 = 1
### PIC16F84A

**INT1/INT0 Register**

The INT1 and INT0 registers are used to enable or disable interrupts. The bits in these registers can be set or cleared by writing to the appropriate bits in the INTCON register. The INT1/INT0 register is located at address 0x000F.

**Table 1: INT1/INT0 Register**

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

** bits 7-0: INT1/INT0 interrupt enable bits.**

- **Bit 7**: INT1 pin interrupt enable
- **Bit 6**: INT0 pin interrupt enable

** bits 7-0: INT1/INT0 interrupt mask bits.**

- **Bit 7**: INT1 pin interrupt mask
- **Bit 6**: INT0 pin interrupt mask

** bits 7-0: INT1/INT0 interrupt flag bits.**

- **Bit 7**: INT1 pin interrupt flag
- **Bit 6**: INT0 pin interrupt flag
2.4 PIC and PULATh
The program starts by clearing the display and asking for the application details. The user then selects the option to continue and enters the dimensions. The software is then designed to automatically calculate the required lengths and dimensions. This process is repeated until all the required information is entered.

2.5 Indirect Addressing: IDF and PIC Registers
The IDF register is a special register that allows indirect addressing. When the IDF register is set, the register whose address is contained in the IDF register is used for addressing. This is similar to using a pointer.

EXAMPLE 2.1: INDIRECT ADDRESSING
- Register A contains the value 123
- Load the value 123 into the IDF register
- A is copied into the IDF register
- A is held in the IDF register for the value of 123
- A is read from the IDF register now at the value of 123

EXAMPLE 2.2: HOW TO CLEAR RAM USING INDIRECT

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>000</td>
</tr>
<tr>
<td>B</td>
<td>123</td>
</tr>
<tr>
<td>C</td>
<td>456</td>
</tr>
</tbody>
</table>

An indirect flip-flop is used to determine the contents of the memory location. The flip-flop reads the contents of the memory location and then stores it in the register. This process is repeated until all the memory locations have been accessed.
### 3.0 DATA EEPROM MEMORY

The EEPROM data memory is volatile and is written to via the Data EEPROM Register (Address 0x1F00 to 0x1F03) and the Data EEPROM Control Register (Address 0x1F05). It is protected against inadvertent write access by hardware. The data EEPROM memory is organized as 16-bit words. The memory is configured as 256 16-bit words. Each word can be individually programmed to contain any 16-bit value. The EEPROM memory is read by the Data EEPROM Read Register (Address 0x1F06 to 0x1F07).

### REGISTER 3.1: ECON0 REGISTER (Address 0x001F)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET7</td>
<td>Embedded Power-On Reset</td>
<td>0 or 1</td>
</tr>
<tr>
<td>EXT</td>
<td>EXPO Power-On Reset (asserted Format)</td>
<td>0 or 1</td>
</tr>
<tr>
<td>FRES</td>
<td>EXPO Power-On Reset (deasserted Format)</td>
<td>0 or 1</td>
</tr>
<tr>
<td>WDTO</td>
<td>Watchdog Timer Overflow</td>
<td>0 or 1</td>
</tr>
<tr>
<td>CKS0</td>
<td>Clock Source Select</td>
<td>0 or 1</td>
</tr>
<tr>
<td>CKS1</td>
<td>Clock Source Select</td>
<td>0 or 1</td>
</tr>
<tr>
<td>CKS2</td>
<td>Clock Source Select</td>
<td>0 or 1</td>
</tr>
<tr>
<td>CKS3</td>
<td>Clock Source Select</td>
<td>0 or 1</td>
</tr>
<tr>
<td>CKS4</td>
<td>Clock Source Select</td>
<td>0 or 1</td>
</tr>
<tr>
<td>CKS5</td>
<td>Clock Source Select</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>

### SELECTED FEATURES

- **Extended I/O**
- **Programmable I/O**
- **Embedded Power-On Reset**
- **Low Power Modes**
- **Watchdog Timer**

### Additional Information

View the PIC16F84A Data Sheet for more information on the PIC16F84A in the 32-pin PQFP package.
3.1 Reading the EEPROM Data Memory

To read a data memory location, the user must write the appropriate page address to the EEPROM Data Memory register and the EEPROM Data Memory Write Enable. The page address must be page aligned, as shown in Example 3.1. The contents of the page are read into the user-defined register address (u.data) when the EEPROM is cycle completes. The contents of the page are read into the user-defined register address (u.data) when the EEPROM is cycle completes. The contents of the page are read into the user-defined register address (u.data) when the EEPROM is cycle completes.

**Example 3.1: Data EEPROM Read**

```
<table>
<thead>
<tr>
<th>Page Address</th>
<th>Data Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 - 000F</td>
<td>00 - 0F</td>
</tr>
<tr>
<td>0010 - 001F</td>
<td>10 - 1F</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

3.2 Writing to the EEPROM Data Memory

To write to an EEPROM data memory, the user must first write the appropriate page address to the EEPROM Data Memory register and the EEPROM Data Memory Write Enable. The contents of the page are written to the EEPROM Data Memory register (u.data) when the EEPROM is cycle completes. The contents of the page are written to the EEPROM Data Memory register (u.data) when the EEPROM is cycle completes. The contents of the page are written to the EEPROM Data Memory register (u.data) when the EEPROM is cycle completes.

**Example 3.2: Data EEPROM Write**

```
<table>
<thead>
<tr>
<th>Page Address</th>
<th>Data Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 - 000F</td>
<td>00 - 0F</td>
</tr>
<tr>
<td>0010 - 001F</td>
<td>10 - 1F</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

3.3 Writing Verification

Verifying the EEPROM data is performed using a checksum. The checksum is calculated based on the contents of the EEPROM Data Memory register (u.data) when the EEPROM is cycle completes. The checksum is calculated based on the contents of the EEPROM Data Memory register (u.data) when the EEPROM is cycle completes. The checksum is calculated based on the contents of the EEPROM Data Memory register (u.data) when the EEPROM is cycle completes.

**Example 3.3: Write Checksum**

```
<table>
<thead>
<tr>
<th>Page Address</th>
<th>Data Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 - 000F</td>
<td>00 - 0F</td>
</tr>
<tr>
<td>0010 - 001F</td>
<td>10 - 1F</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Table 3.1: Registers/Bits Associated with Data EEPROM**

<table>
<thead>
<tr>
<th>Register/Bit</th>
<th>Description</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u.data</td>
<td>User Defined Data Memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes:
- The u.data register is used to access the data EEPROM.
- This table is not exhaustive and may be expanded as needed.
- The values are read from the EEPROM.
4.8 I/O PORTS

4.1 PORTA and TRISA Registers

PORTA is a 6-bit input/output port. The corresponding data direction register is TRISA. Setting bit 2 in TRISAA (TRISA.0) will output an pull-up resistor on the corresponding port pin. A low level output on a pull-up enabled pin will not drive the pin high. When outputting a logic high, it is recommended that PORTA.2 be tied to VCC through a pull-up resistor.

Note: OR complemented Data Direction bits are complemented automatically.

EXAMPLE 4.1: INITIALIZING PORTS

```c
// Initialize PORTA
void PORTA_Init(void) {
    PORTA = 0b00000000; // Clear PORTA
    TRISA = 0b00001000; // Enable pull-up on PORTA.2
}
```

FIGURE 4.1: BLOCK DIAGRAM OF PORTA

FIGURE 4.2: BLOCK DIAGRAM OF PORTB

Note: Output high impedance modes for PORTA and PORTB.
### PORT FUNCTIONS

<table>
<thead>
<tr>
<th>Name</th>
<th>Bit 1</th>
<th>Bit 0</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2V</td>
<td>00</td>
<td></td>
<td>Timer 2</td>
</tr>
<tr>
<td>T2</td>
<td>01</td>
<td></td>
<td>Timer 2</td>
</tr>
<tr>
<td>T2X</td>
<td>10</td>
<td></td>
<td>Timer 2</td>
</tr>
<tr>
<td>T1</td>
<td>11</td>
<td></td>
<td>Timer 1</td>
</tr>
</tbody>
</table>

Notes: T x or T y means an output pin can be programmed to be Timer x or Timer y active.

### SUMMARY OF REGISTERS ASSOCIATED WITH PORTA

<table>
<thead>
<tr>
<th>Register</th>
<th>Bits</th>
<th>Description</th>
<th>Values on Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCON</td>
<td>0</td>
<td>RCON Register</td>
<td>00000000</td>
</tr>
<tr>
<td>STATUS</td>
<td>1</td>
<td>Status Register</td>
<td>Various</td>
</tr>
</tbody>
</table>

Legend: **1** active. **0** inactive. **x** unimplemented. **N** a reserved register with no functionality.
Appendix A

PORTB and TRISS Registers

4.2 PORTB and TRISS Registers

PORTB is a 6-bit I/O port that has pull-up resistors. Its state can be managed using the PORTB register. The TRISS registers (TRISB0-TRISSB7) control the direction of the PORTB pins.

**EXAMPLE 4.2: INITIATING PORTB**

```plaintext
PORTB = 0x00; // Initialize PORTB as inputs
TRISB = 0x0F; // Set direction of PORTB pins
```

**FIGURE 4.3: BLOCK DIAGRAM OF PORTB INTERFACE**

![PortB Block Diagram](image1)

**FIGURE 4.4: BLOCK DIAGRAM OF TRISS INTERFACE**

![TriSS Block Diagram](image2)
### TABLE A-1: PORT FUNCTIONS

<table>
<thead>
<tr>
<th>Name</th>
<th>Bit</th>
<th>Buffer Type</th>
<th>CS Compatibility Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>RB7</td>
<td>01</td>
<td>TTL</td>
<td>Weak pull-up</td>
</tr>
<tr>
<td>RB6</td>
<td>02</td>
<td>TTL</td>
<td>Weak pull-up</td>
</tr>
<tr>
<td>RB5</td>
<td>03</td>
<td>TTL</td>
<td>Weak pull-up</td>
</tr>
<tr>
<td>RB4</td>
<td>04</td>
<td>TTL</td>
<td>Weak pull-up</td>
</tr>
<tr>
<td>RB3</td>
<td>04</td>
<td>TTL</td>
<td>Weak pull-up</td>
</tr>
<tr>
<td>RB2</td>
<td>03</td>
<td>TTL</td>
<td>Weak pull-up</td>
</tr>
<tr>
<td>RB1</td>
<td>02</td>
<td>TTL</td>
<td>Weak pull-up</td>
</tr>
<tr>
<td>RB0</td>
<td>01</td>
<td>TTL</td>
<td>Weak pull-up</td>
</tr>
</tbody>
</table>

Legend: **T** = TTL, **S** = Schottky

### TABLE A-4: SUMMARY OF REGISTERS ASSOCIATED WITH PORTS

<table>
<thead>
<tr>
<th>Address</th>
<th>Mode</th>
<th>RA0</th>
<th>RA1</th>
<th>RA2</th>
<th>RA3</th>
<th>RA4</th>
<th>RA5</th>
<th>RA6</th>
<th>RA7</th>
<th>RB0</th>
<th>RB1</th>
<th>RB2</th>
<th>RB3</th>
<th>RB4</th>
<th>RB5</th>
<th>RB6</th>
<th>RB7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Legend: * = pull-up, + = pull-down, Shifted bits are not used by PORTS.
5.6 TIMER MODULE

The Timer module includes the following features:

- 2x6 timer/counters
- 8-bit counter registers
- 8-bit prescaler
- End of timer overflow
- Software programmable prescaler
- Interrupt capability

The Timer module is used in a variety of applications, ranging from simple timing functions to complex event management systems.

5.2 Prescaler

The prescaler is programmable for the Timer module. It can be configured to divide the oscillator frequency by a power of 2, ranging from 1 to 256. The prescaler can be used to reduce the frequency output from the Timer module, allowing for more precise timing applications.

---

**Figure 5.1: TIMER BLOCK DIAGRAM**

The Timer block diagram shows the internal architecture of the Timer module, including the prescaler, counter, and other components. The diagram is essential for understanding how the Timer module operates and how it can be integrated into larger systems. The block diagram includes all necessary connections and components to ensure proper operation of the Timer module.
6.0 SPECIAL FEATURES OF THE CPU

What sets the PIC16F84A apart from other microcontrollers is its ability to integrate various features and peripheral functions within a single chip, making it versatile and cost-effective. The PIC16F84A is designed to offer a balance between performance and power consumption, which is crucial in portable and battery-powered applications.

- **Data Flash**: Offers non-volatile memory for storing program code, allowing the device to retain its programming even when power is turned off.
- **Watchdog Timer (WDT)**: Provides a mechanism to ensure the processor remains operational and prevents it from locking up in the event of a software error.
- **CPU**: The core of the CPU allows for efficient processing of data and instructions, ensuring smooth operation of the device.
- **Power Management**: Includes features to manage power consumption effectively, ideal for battery-powered applications.
- **On-Chip Oscillator**: Includes both an internal oscillator and an optional external oscillator, providing flexibility in clock source selection.
- **Flash Programming Interface**: Supports in-system programming (ISP) and in-application programming (IAP), enabling easy updates and reprogramming without external hardware.
- **In-Circuit Serial Programming (ICSP)**: Facilitates easy programming of the device in-circuit, simplifying the development process.
- **Quadrature Encoder** (QEP): Useful for applications requiring precise angular position and speed measurement.
- **Quadrature Delta Encoder** (QDE): Designed for position detection in applications like robotics and automation.
- **Quadrature autofocus encoder** (QFAF): Specialized for autofocus in digital cameras and similar applications.
- **Interrupts**: Supports multiple interrupt sources, allowing the device to respond to various events efficiently.
- **Configuration Bits**: These bits enable the device to be customized for different application needs, offering flexibility in terms of functionality and performance.

### REGISTER 6-1: PIC16F84A CONFIGURATION WORD

<table>
<thead>
<tr>
<th>Bit</th>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DPL</td>
<td>Data Flash Phase Low</td>
</tr>
<tr>
<td>1</td>
<td>DPH</td>
<td>Data Flash Phase High</td>
</tr>
<tr>
<td>2</td>
<td>WDTEN</td>
<td>Watchdog Timer Enable</td>
</tr>
<tr>
<td>3</td>
<td>WDTOV</td>
<td>Watchdog Timer Time Out Value</td>
</tr>
<tr>
<td>4</td>
<td>BWDEN</td>
<td>Power Management Enable</td>
</tr>
<tr>
<td>5</td>
<td>WRT</td>
<td>Write Protection</td>
</tr>
<tr>
<td>6</td>
<td>BOREN</td>
<td>Brownout Reset Enable</td>
</tr>
<tr>
<td>7</td>
<td>WRLLP</td>
<td>Watchdog Timer Low Low Pin</td>
</tr>
<tr>
<td>8</td>
<td>CM1</td>
<td>Configuration Bit 1</td>
</tr>
<tr>
<td>9</td>
<td>CM0</td>
<td>Configuration Bit 0</td>
</tr>
</tbody>
</table>

Additional information on special features is available in the PIC16F84A Data Sheet, which provides detailed specifications and application notes.
6.2 Oscillator Configurations

6.2.1 Oscillator Types

The PIC16F84A can be configured in four different oscillator modes:

- RC
- Crystal
- Ceramic
- Internal

Circuit Diagram:

![Circuit Diagram](image)

6.2.2 Crystal Oscillator/Ceramic

A crystal oscillator or ceramic resonator is connected to the XTAL1 and XTAL2 pins to activate oscillator of Type I.

![Crystal Oscillator/Ceramic](image)

### Table 6-1: Capacitor Selection for Ceramic Resonators

<table>
<thead>
<tr>
<th>Capacitor Value</th>
<th>7pF</th>
<th>8pF</th>
<th>9pF</th>
<th>10pF</th>
<th>11pF</th>
<th>12pF</th>
<th>13pF</th>
<th>14pF</th>
<th>15pF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes:

- The XTAL1 and XTAL2 pins are connected to the crystal oscillator or ceramic resonator.

![Diagram](image)

Note: Do not connect other signals to the XTAL1 and XTAL2 pins. The oscillator should be powered on after the oscillator is established. DO NOT power down the oscillator until the application is in the running state.
### Table 4.3: Capacitor Selection for Crystal Oscillator

<table>
<thead>
<tr>
<th>Model</th>
<th>DC (V)</th>
<th>IC (Q)</th>
<th>RC (µF)</th>
<th>SFR (µS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

**Note:** Higher temperature or pressure may shorten the life of the device, but also increases the output voltage. The output voltage can be increased by increasing the frequency, but also decreases the output voltage. The output voltage can be increased by increasing the resistance, but also decreases the output voltage.

**Figure 4.1: Oscillation Mode**

- **Input:** (Description of input signal)
- **Output:** (Description of output signal)
- **Component:** (Description of component used in oscillation mode)

- **Resistors:** (Description of resistors used in oscillation mode)
- **Capacitors:** (Description of capacitors used in oscillation mode)

- **Requirements:**
  - Input Voltage: ±5V
  - Duty Cycle: 50%
6.3 RESET

See the section on differences between various family members of PIC16F44. For operation in 6.3.1, see the section on differences between various family members of PIC16F44.

FIGURE 6.1

FIGURE 6.2

Note 1: See Table 6.2

TABLE 6.2

<table>
<thead>
<tr>
<th>Condition</th>
<th>Program Counter</th>
<th>STATUS Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset from normal</td>
<td>0000</td>
<td>000</td>
</tr>
<tr>
<td>Reset from normal</td>
<td>0000</td>
<td>000</td>
</tr>
<tr>
<td>Reset from normal</td>
<td>0000</td>
<td>000</td>
</tr>
<tr>
<td>Reset from normal</td>
<td>0000</td>
<td>000</td>
</tr>
<tr>
<td>Reset from normal</td>
<td>0000</td>
<td>000</td>
</tr>
<tr>
<td>Reset from normal</td>
<td>0000</td>
<td>000</td>
</tr>
<tr>
<td>Reset from normal</td>
<td>0000</td>
<td>000</td>
</tr>
<tr>
<td>Reset from normal</td>
<td>0000</td>
<td>000</td>
</tr>
<tr>
<td>Reset from normal</td>
<td>0000</td>
<td>000</td>
</tr>
</tbody>
</table>

Legend: n = background in standby mode

Note 1: See Table 6.2 for details.
### Table 4.3: Reset Conditions for All Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
<th>Function or Reset</th>
<th>Value on reset if power fault, etc., and internal reset not used</th>
<th>Value during normal operation or normally used internal reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCLR</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TATF</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TO</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKS</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROS</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPO</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DREG</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CORE</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCR</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSTAT</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EEDAT</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STAT</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P6</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note:
- If the microblock is to be an interrupt and the UE bit is set, the PC is loaded with the vector address.
- Do not rely on the EEDAT register contents.
- This is the value that will be loaded at reset.

---

© 2017 Microchip Technology Inc.
Appendix A

6.4 Power-on Reset (POR)
A Power-on Reset pulse is generated when the power supply voltage is 0.7 to 1.5 volts lower than the low-voltage trip point (TV) of Iddq-power-down circuitry. The POR pulse duration is 1.5 to 100 milliseconds.

6.6 Oscillators and Startup Time (OST)

![Diagram of Oscillator and Startup Time](image)

Note: Additional information and technical specifications can be found in the PIC16F84A datasheet.
Figure 6-5: Turn-on Sequence on Power-up).

TABLE 6-5: Turn-on Sequence

| Device Configuration | Power-on | Inhibit Power-on | Initial Power-on | Power-on After RST
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PTU, LP</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>STU, LP</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

6.7 Turn-on Sequence and Power-up Status Bits (P82)

On power-up (Figure 6-5 through 6-8), the internal sequence is as follows:

1. Power-up is checked after a POR has occurred.
2. Power-on is activated.
3. The POR is activated.
4. The power-up will only be done in the same configuration as the previous configuration of the device. For example, if the previous configuration was RST, the next configuration will not be RST.

TABLE 6-6: Status Bits and Their Significance

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Power-on</td>
</tr>
<tr>
<td>1</td>
<td>POR/Reset</td>
</tr>
<tr>
<td>2</td>
<td>POR/Reset (high, reset operation)</td>
</tr>
<tr>
<td>3</td>
<td>POR/Reset (low, no operation)</td>
</tr>
<tr>
<td>4</td>
<td>POR/Reset (low, no operation, timeout)</td>
</tr>
<tr>
<td>5</td>
<td>POR/Reset (low, no operation, timeout)</td>
</tr>
<tr>
<td>6</td>
<td>POR/Reset (low, no operation, timeout)</td>
</tr>
</tbody>
</table>
6.8 Interrupts

Appendix A

6.8.1 INT INTERRUPT

External interrupt pins INT0 or INT1 can be made into a source of interrupt or not, depending on the pin configuration. When configured as an external interrupt, an interrupt request is generated when the INT0 or INT1 pin is pulled low while the INT0 or INT1 pin is configured as an external interrupt. The INT0 or INT1 pin can be made into a source of interrupt or not by using the INTCON register.

6.8.2 SFR INTERRUPT

Interrupts from SFRs can be made into a source of interrupt or not, depending on the SFR configuration. When configured as an SFR interrupt, an interrupt request is generated when the SFR is read or written to while the SFR is configured as an SFR interrupt. The SFR can be made into a source of interrupt or not by using the INTCON register.

6.8.3 PORTB INTERRUPT

Interrupts from PORTB can be made into a source of interrupt or not, depending on the PORTB configuration. When configured as an PORTB interrupt, an interrupt request is generated when the PORTB is read or written to while the PORTB is configured as an PORTB interrupt. The PORTB can be made into a source of interrupt or not by using the INTCON register.

Note: Each interrupt source can be enabled or disabled independently, regardless of the state of other interrupt sources.
## 6.9 Context Saving During Interrupts

When an interrupt occurs, the microcontroller preserves the previous context using a specific sequence of instructions. The context stack pointer (SP) is used to manage the context save area, which includes the stack and other critical registers.

### Example 9.1: Saving Status and I/O Registers in RAM

<table>
<thead>
<tr>
<th>Register</th>
<th>Initial Value</th>
<th>After Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I/O Registers</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Example for the following:

1. Move to interrupt handler.
2. Save the program counter (PC) and other critical registers.
3. Execute the interrupt service routine.
4. Return to the previous context.
5. Restore the registers and return to normal operation.

## 6.10 Watchdog Timer (WDT)

The Watchdog Timer is an in-circuit debug feature that prevents the microcontroller from getting stuck in an infinite loop. The WDT is enabled by default and can be configured through the Configuration Register (CONFIG). The WDT period can be adjusted by software, allowing for flexibility in debugging and testing.

### 6.10.1 WDT Period

- The WDT operates with a pre-scaler of 150. This configuration provides a default period of approximately 1 second. The WDT can be disabled or re-configured to meet specific application needs.
- If the WDT times out, the microcontroller will perform a reset, allowing the developer to re-initiate the debugging process.
- The CONFIG register controls the WDT period, providing a balance between stability and flexibility in debugging.

The WDT reset in the PIC16F84A is a hardware action that can be cleared upon a WDT reset.
TABLE 6.1: SUMMARY OF REGISTERS ASSOCIATED WITH THE WACHEDOR TIMER

| Bit | Name          | Bit 1 | Bit 0 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|-----|---------------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| 01 | Port D PTE   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   | 01   |
| 02 | Port D PSW   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   | 02   |
| 03 | Port D PDI   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   | 03   |
| 04 | Port D PDI   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   | 04   |

Notes:
- Bit 1: Port D PTE
- Bit 0: Port D PSW
- Bit 3: Port D PDI

See Section 6.1 and Section 6.2 for the operation of the control and data connection bits.
6.11 Power-down Mode (SLEEP)

The PIC16F84A can be put into power-down mode (SLEEP) at any time to minimize power consumption. When a device is powered down, all I/O pin functions are disabled and the CPU stops executing its instructions. The WDT is also stopped and the oscillator circuit is slowed down. When the device is awakened from SLEEP, the oscillator circuit is re-activated and the program counter (PC) is returned to the address following the SLEEP instruction (see Table 6-10).

6.11.2 Wake-up from SLEEP

The device can wake-up from SLEEP through one of the following events:
1. External WDT reset (MCLR = 0).
2. RESET pulse if WDT was enabled.
3. Any I/O pin (except RB6 and RB7) going high when configured as an input with pull-up enabled.

Figure 6-12 illustrates the timing diagram for the device waking up from SLEEP through an I/O pin interrupt. The RST pin is pulled high to wake-up the device from SLEEP mode. The RST pin must be held high for at least two machine cycles (24 oscillator periods) to allow the device to exit SLEEP mode. Once the device wakes up from SLEEP, the program counter (PC) is set to the address following the SLEEP instruction. The device then executes the next instruction after exiting SLEEP mode.
6.12 Program Verification/Code Protections

If the code protection flags have not been programmed, the current program memory can be read and executed as any other memory. The program memory can be protected by setting the flags. If the VER field in Configuration Word is set, the code protection flags can be checked. The VER field is set and the PVD set to zeros.

6.14 In-Circuit Serial Programming

In-Circuit Serial Programming can be enabled via the EPP flag in the Configuration Word. This flag can be set by the user by using the Configuration Word Programming command. Details about the Configuration Word Programming command can be found in the "In-Circuit Serial Programming" section in "PIC16F84A Data Sheet."
7.0 INSTRUCTION SET SUMMARY

Each PIC16F84 instruction is a 16-bit word divided into an opcode (8-bits) and an 8-bit address. The
opcode determines the instruction type and the address specifies the execution location of the instruc-
tion. Instructions are organized into groups based on the number of operand registers and the
function performed. Each group is further divided into subgroups. For example, the instruction set
includes a single-register instruction group and a two-register instruction group. The instructions
are also separated into machine instructions and program instructions. Machine instructions con-
stitute the majority of the instruction set and are used for basic operations such as arithmetic and
logical functions. Program instructions are used to control the execution flow of the program.

Table 7-1: Opcode Field Descriptions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Add R1, R2</td>
</tr>
<tr>
<td>0001</td>
<td>Subtract R1, R2</td>
</tr>
<tr>
<td>0010</td>
<td>Multiply R1, R2</td>
</tr>
<tr>
<td>0011</td>
<td>Divide R1, R2</td>
</tr>
<tr>
<td>0100</td>
<td>Logical And R1, R2</td>
</tr>
<tr>
<td>0101</td>
<td>Logical Or R1, R2</td>
</tr>
<tr>
<td>0110</td>
<td>Logical Not R1</td>
</tr>
<tr>
<td>0111</td>
<td>Logical Xor R1, R2</td>
</tr>
<tr>
<td>1000</td>
<td>Jump to location R1</td>
</tr>
<tr>
<td>1001</td>
<td>Jump if carry</td>
</tr>
<tr>
<td>1010</td>
<td>Jump if negative</td>
</tr>
<tr>
<td>1011</td>
<td>Jump if zero</td>
</tr>
<tr>
<td>1100</td>
<td>Call subroutine</td>
</tr>
<tr>
<td>1101</td>
<td>Return from subroutine</td>
</tr>
<tr>
<td>1110</td>
<td>Set condition register</td>
</tr>
<tr>
<td>1111</td>
<td>Clear condition register</td>
</tr>
</tbody>
</table>

Note: In the above instruction tables, each instruction is represented in the format
"opcode - argument - operand - condition"
where:
- Opcode is an 8-bit code
- Argument is a 4-bit value
- Operand is a 1-bit value
- Condition is a 1-bit value

A detailed description of each instruction is available in the PIC16F84 data sheet (DS40034A).
## PIC16F84A

### Table 7-2: PIC16FXXX Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Opcode</th>
<th>8-bit Opcode</th>
<th>16-bit Opcode</th>
<th>Usage</th>
<th>Status Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add</td>
<td>0000</td>
<td>0000000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td>0001</td>
<td>0000000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
<td>0010</td>
<td>0000000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRZ</td>
<td>Branch if Zero</td>
<td>0011</td>
<td>0000000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRNE</td>
<td>Branch if Not Equal</td>
<td>0100</td>
<td>0000000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRH</td>
<td>Branch if High</td>
<td>0101</td>
<td>0000000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRL</td>
<td>Branch if Low</td>
<td>0110</td>
<td>0000000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSZ</td>
<td>Branch if Sign Zero</td>
<td>0111</td>
<td>0000000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRC</td>
<td>Branch if Carry</td>
<td>1000</td>
<td>0000000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRR</td>
<td>Branch if Reset</td>
<td>1001</td>
<td>0000000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRT</td>
<td>Branch if Tahn</td>
<td>1010</td>
<td>0000000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRF</td>
<td>Branch if Overflow</td>
<td>1011</td>
<td>0000000000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Additional information on the instruction set is available in the PICmicro Microcontroller Family Reference Manual (DS33024).
## 7.1 Instruction Descriptions

### ADDLH l, W

- **Syntax:** `ADDLH l, W`
- **Operands:** `l`, `W`
- **Description:**
  - **Operation:** `(l + W) & 0xff`
  - **Status Affected:** None
  - **Description:** Add lower 8 bits of `l` to `W`, and store the result in `l`.

### ADDLH l, W

- **Syntax:** `ADDLH l, W`
- **Operands:** `l`, `W`
- **Description:**
  - **Operation:** `(l + W) & 0xff`
  - **Status Affected:** None
  - **Description:** Add lower 8 bits of `l` to `W`, and store the result in `l`.

### ADCLH l, W

- **Syntax:** `ADCLH l, W`
- **Operands:** `l`, `W`
- **Description:**
  - **Operation:** `(l + W + carry) & 0xff`
  - **Status Affected:** carry
  - **Description:** Add lower 8 bits of `l` to `W`, and store the result in `l`. If the carry is set, it is stored in the carry bit.

### BIC l, W

- **Syntax:** `BIC l, W`
- **Operands:** `l`, `W`
- **Description:**
  - **Operation:** `(l & ~W)`
  - **Status Affected:** None
  - **Description:** Bit in register `l` is cleared.
### PIC16F84A

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRIF</td>
<td>Branch on Internal Flag</td>
<td>A branch is taken if the internal flag is set.</td>
</tr>
<tr>
<td>BRIP</td>
<td>Branch on Internal Flag</td>
<td>A branch is taken if the internal flag is clear.</td>
</tr>
<tr>
<td>BDF</td>
<td>Branch on Data</td>
<td>A branch is taken if the data is 0.</td>
</tr>
<tr>
<td>BDP</td>
<td>Branch on Data</td>
<td>A branch is taken if the data is 1.</td>
</tr>
</tbody>
</table>

#### BRIF

<table>
<thead>
<tr>
<th>Operands</th>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>ICF</td>
<td>A branch is taken if the internal flag is set.</td>
</tr>
</tbody>
</table>

#### BRIP

<table>
<thead>
<tr>
<th>Operands</th>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>ICFR</td>
<td>A branch is taken if the internal flag is clear.</td>
</tr>
</tbody>
</table>

#### BDF

<table>
<thead>
<tr>
<th>Operands</th>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>DF</td>
<td>A branch is taken if the data is 0.</td>
</tr>
</tbody>
</table>

#### BDP

<table>
<thead>
<tr>
<th>Operands</th>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>DF</td>
<td>A branch is taken if the data is 1.</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>PIC16F84A</td>
<td>Increment</td>
<td></td>
</tr>
</tbody>
</table>
### PIC16F84A

<table>
<thead>
<tr>
<th>Operand</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MOV</code></td>
<td>Y</td>
<td>Set Y to 1</td>
</tr>
<tr>
<td><code>LDF</code></td>
<td>R16</td>
<td>Load R16</td>
</tr>
<tr>
<td><code>SETB</code></td>
<td>E</td>
<td>Set bit E</td>
</tr>
<tr>
<td><code>JMP</code></td>
<td>0110</td>
<td>Jump to label</td>
</tr>
<tr>
<td><code>CLRF</code></td>
<td>R16</td>
<td>Clear R16</td>
</tr>
<tr>
<td><code>MOVWF</code></td>
<td>R16</td>
<td>Move W to R16</td>
</tr>
<tr>
<td><code>BSF</code></td>
<td>R16</td>
<td>Set bit R16</td>
</tr>
</tbody>
</table>

### Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLF</td>
<td>Rotate left</td>
</tr>
<tr>
<td>RRF</td>
<td>Rotate right</td>
</tr>
<tr>
<td>BCF</td>
<td>Clear bit</td>
</tr>
<tr>
<td>BTFN</td>
<td>Branch if negative</td>
</tr>
<tr>
<td>BFTH</td>
<td>Branch if positive</td>
</tr>
<tr>
<td>BTF</td>
<td>Branch if flag (overflow or carry)</td>
</tr>
<tr>
<td>BTFZ</td>
<td>Branch if zero</td>
</tr>
<tr>
<td>BTFN</td>
<td>Branch if negative</td>
</tr>
<tr>
<td>BTFNZ</td>
<td>Branch if non-zero</td>
</tr>
<tr>
<td>BRT</td>
<td>Branch if result</td>
</tr>
<tr>
<td>BSF</td>
<td>Set bit</td>
</tr>
<tr>
<td>BSFZ</td>
<td>Set bit if zero</td>
</tr>
<tr>
<td>BTFS</td>
<td>Set bit if signed zero</td>
</tr>
<tr>
<td>BTFT</td>
<td>Set bit if unsigned zero</td>
</tr>
<tr>
<td>BTR</td>
<td>Set bit if positive</td>
</tr>
<tr>
<td>BTRZ</td>
<td>Set bit if zero</td>
</tr>
<tr>
<td>BTD</td>
<td>Set bit if negative</td>
</tr>
<tr>
<td>BTDZ</td>
<td>Set bit if zero</td>
</tr>
<tr>
<td>BTRP</td>
<td>Set bit if positive</td>
</tr>
<tr>
<td>BTRNZ</td>
<td>Set bit if non-zero</td>
</tr>
<tr>
<td>BTRN</td>
<td>Set bit if negative</td>
</tr>
<tr>
<td>BTRNZ</td>
<td>Set bit if non-zero</td>
</tr>
<tr>
<td>BTB</td>
<td>Set bit if branch</td>
</tr>
<tr>
<td>BTBZ</td>
<td>Set bit if zero</td>
</tr>
<tr>
<td>BTBN</td>
<td>Set bit if negative</td>
</tr>
<tr>
<td>BTBNZ</td>
<td>Set bit if non-zero</td>
</tr>
<tr>
<td>BMI</td>
<td>Branch if minus</td>
</tr>
<tr>
<td>BMIH</td>
<td>Branch if high minus</td>
</tr>
<tr>
<td>BMIZ</td>
<td>Branch if zero</td>
</tr>
<tr>
<td>BIFZ</td>
<td>Branch if zero</td>
</tr>
<tr>
<td>BIFZ</td>
<td>Branch if zero</td>
</tr>
<tr>
<td>BIFN</td>
<td>Branch if negative</td>
</tr>
<tr>
<td>BIFNZ</td>
<td>Branch if non-zero</td>
</tr>
<tr>
<td>BIE</td>
<td>Branch if interrupt enabled</td>
</tr>
<tr>
<td>BIEH</td>
<td>Branch if high interrupt enabled</td>
</tr>
<tr>
<td>BIFZ</td>
<td>Branch if zero</td>
</tr>
<tr>
<td>BIFZ</td>
<td>Branch if zero</td>
</tr>
<tr>
<td>BIFN</td>
<td>Branch if negative</td>
</tr>
<tr>
<td>BIFNZ</td>
<td>Branch if non-zero</td>
</tr>
<tr>
<td>BMI</td>
<td>Branch if minus</td>
</tr>
<tr>
<td>BMIH</td>
<td>Branch if high minus</td>
</tr>
<tr>
<td>BMIZ</td>
<td>Branch if zero</td>
</tr>
<tr>
<td>BIFZ</td>
<td>Branch if zero</td>
</tr>
<tr>
<td>BIFZ</td>
<td>Branch if zero</td>
</tr>
<tr>
<td>BIFN</td>
<td>Branch if negative</td>
</tr>
<tr>
<td>BIFNZ</td>
<td>Branch if non-zero</td>
</tr>
</tbody>
</table>
### PIC16F84A

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>NOP</td>
<td>0000</td>
<td>0</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- None

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>01h</td>
<td>RST</td>
<td>0001</td>
<td>1</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Reset

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>10h</td>
<td>CWAIT</td>
<td>0010</td>
<td>0</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Wait command

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>11h</td>
<td>CLSWD</td>
<td>0011</td>
<td>2</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Clear system interrupt delays

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>12h</td>
<td>CIDF</td>
<td>0100</td>
<td>2</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Clear interrupt disable flag

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>13h</td>
<td>CIDF</td>
<td>0101</td>
<td>2</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Clear interrupt disable flag

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>14h</td>
<td>CIRF</td>
<td>0110</td>
<td>2</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Clear interrupt request flag

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>15h</td>
<td>CIRF</td>
<td>0111</td>
<td>2</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Clear interrupt request flag

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>16h</td>
<td>CLSWD</td>
<td>0100</td>
<td>2</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Clear system interrupt delays

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>17h</td>
<td>CLSWD</td>
<td>0101</td>
<td>2</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Clear system interrupt delays

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>18h</td>
<td>CLSWD</td>
<td>0110</td>
<td>2</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Clear system interrupt delays

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>19h</td>
<td>CLSWD</td>
<td>0111</td>
<td>2</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Clear system interrupt delays

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>20h</td>
<td>DDF</td>
<td>1000</td>
<td>0</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>21h</td>
<td>DDF</td>
<td>1001</td>
<td>0</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>22h</td>
<td>DDF</td>
<td>1010</td>
<td>0</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>23h</td>
<td>DDF</td>
<td>1011</td>
<td>0</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>24h</td>
<td>DDF</td>
<td>1100</td>
<td>0</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>25h</td>
<td>DDF</td>
<td>1101</td>
<td>0</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>26h</td>
<td>DDF</td>
<td>1110</td>
<td>0</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>27h</td>
<td>DDF</td>
<td>1111</td>
<td>0</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>28h</td>
<td>DI</td>
<td>1000</td>
<td>2</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>29h</td>
<td>DI</td>
<td>1001</td>
<td>2</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>30h</td>
<td>DI</td>
<td>1010</td>
<td>2</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>31h</td>
<td>DI</td>
<td>1011</td>
<td>2</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>32h</td>
<td>DI</td>
<td>1100</td>
<td>2</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>33h</td>
<td>DI</td>
<td>1101</td>
<td>2</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>34h</td>
<td>DI</td>
<td>1110</td>
<td>2</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>35h</td>
<td>DI</td>
<td>1111</td>
<td>2</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>36h</td>
<td>DI</td>
<td>1100</td>
<td>4</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>37h</td>
<td>DI</td>
<td>1101</td>
<td>4</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>38h</td>
<td>DI</td>
<td>1110</td>
<td>4</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>39h</td>
<td>DI</td>
<td>1111</td>
<td>4</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None

---

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Opcode</th>
<th>Elements</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>40h</td>
<td>DI</td>
<td>1100</td>
<td>8</td>
<td>None</td>
</tr>
</tbody>
</table>

**Operation:**

- Disable divided wait generation

**Status Affected:**

- None
### PIC16F84A

**Instruction:** EXORL

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>W</td>
</tr>
</tbody>
</table>

**Operation:**

- The EXORL instruction takes the source register and the destination register, performs an exclusive OR operation on them, and stores the result in the destination.

**Description:**

- The source register is XORed with the destination register. The result is stored in the destination register.
Appendix B
DIZI-2 Board and Lock Application

DIZI-2 Demonstration Board

A circuit was required to demonstrate a range of basic microcontroller programming techniques via a set of simple applications for the PIC 16F84. The DIZI (Display, buzzer and interrupt) board was designed to allow the special hardware features of the PIC chip to be exercised, including interrupts, timer and EEPROM memory. In-circuit programming was not incorporated, in order to emphasise the stand-alone operation of the microcontroller. The chip would be programmed separately and then physically transferred to the target system. The enhanced DIZI-2 board described in this appendix incorporates an on-board battery supply, a finger pot to provide an analog input and hardware switch debouncing to improve the reliability of the push button operation.

The DIZI-2 board measures 100 x 100 mm and has copper tracks for making the component connections on a standard 0.1" grid. The design includes a 2 x 1.5 V battery pack on the board. The power is switched on via a non-latching push button so that it cannot be left on accidently, and thereby exhaust the batteries, it must be held on manually while the circuit is in operation.

The basic demonstration programs in Part B of this book can be run on the DIZI-2 board, while the motor programs must be run on the MOTA demo board. The simple introductory circuits in Part B and the motor board can be constructed using the same techniques as will be described for the DIZI board. The reader who is inexperienced in prototype construction is encouraged to attempt these tasks. The binary output counts from the BINx programs will be seen on the corresponding LED segments of the DIZI display, although the binary number is not displayed so clearly as it would be on a set of eight discrete LEDs, or an LED bar graph module.

DIZI-2 Board Design

A seven-segment display, along with decimal and hexadecimal digits, is to be shown. A range of applications with a numerical output can then be demonstrated, for example, the electronic DICE in Chapter 12 and the LOCK application detailed below. Port B has eight I/O bits; seven are used for the LED display, leaving RB0 free for use as both an audio output and a push button interrupt input. A small audio transducer, a piezo electric buzzer, provides a simple and effective way of monitoring audio output frequencies or generating status signals.

Port A has five pins. RA0 can be used as an input to the TMR0 counter, while RA4 can be used as an input to the timer. RA1 and RA2 are used for the LED display, having RB0 free for use as both an audio output and a push button interrupt input. RA3 can be used as an input to the timer. RA5 can be used as an input for the binary output in the BINx programs.

The seven-segment display is used to show decimal and hexadecimal digits. It is composed of seven segments, each represented by a segment on the display. The segments are arranged in a specific pattern to form the desired digit. The display can be controlled by the microcontroller to show various numerical values.

The LED display uses seven segments to form digits. The segments are arranged in a specific pattern to represent the desired number. The display is controlled by the microcontroller, allowing the user to display a range of numerical values.

The DIZI board is designed to allow the special hardware features of the PIC chip to be exercised, including interrupts, timer and EEPROM memory. In-circuit programming was not incorporated, in order to emphasise the stand-alone operation of the microcontroller. The chip would be programmed separately and then physically transferred to the target system.
BCD inputs for the LOCK application, so RA0–RA3 were allocated for this purpose. The switch and push button inputs have 100k pull-up resistors, and the push buttons have 22μF debouncing capacitors. These are fitted to each input because, when a switch closes, the metal contacts can bounce open several times before finally closing. The CR network prevents this from occurring because after the capacitor has quickly discharged at 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 re-open. 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 the analogue input process in the LOCK program.

**Circuit Diagram**

Developing circuits for microcontrollers is not always too difficult, because the same circuit element can often be reused in different designs. Thus, in the DIZI circuit (Fig B.1), the switch inputs, display outputs and the clock circuit are standard arrangements of components. Remember however, that unlike the PIC, some microcontrollers cannot drive LEDs directly, but need a current driver stage inserted between each output and the LED.

**DIZI-2 Board Layout**

The layout of a PCB or prototype circuit is derived from the circuit diagram. The pins on DIL (Dual In-Line) chips are spaced 0.1" apart, so the circuits must be laid out on a 0.1" grid.
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 wordprocessor to do the same job. Examples of such connection diagrams are given in the text. A more refined layout drawing for the DIZI-2 board is shown in Fig. B.2.

The board is viewed from the front (component) side, with the tracks on the back shown vertically. The components are numbered for reference to the parts listed below. The chips are all placed in the same orientation, so that pin 1 is bottom left. The seven-segment display has the decimal point bottom right. The ICs must obviously be fitted across the tracks, so that their pin connections are separated. The PIC chip should 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; these positions are shown as small white circles. The tracks must also be cut between the opposite pairs of pins on each DIL component, and, in this case, under the clock circuit capacitors (11).

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 directing the layout, perhaps with some angling of board area.
Appendix B

Parts List

A parts list is required to specify the exact component when ordering from a suitable supplier. The availability of components varies over time, so updating is sometimes necessary. For example, the finger preset pot originally used in the design is no longer listed by the UK supplier, and had to be replaced with an equivalent part. The layout then had to be amended because the pin arrangement of the new component was different.

<table>
<thead>
<tr>
<th>Layout</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Battery box, 2 × AA cells, PCB mounting</td>
</tr>
<tr>
<td>2</td>
<td>Microcontroller, PIC 16LF84-04</td>
</tr>
<tr>
<td>3</td>
<td>Peizo electric sounder, PCB mounting</td>
</tr>
<tr>
<td>4</td>
<td>Seven-segment LED display, 0.5”, Common Cathode</td>
</tr>
<tr>
<td>5</td>
<td>Piano DIL switch, 4-way</td>
</tr>
<tr>
<td>6</td>
<td>Tactile switch, PCB mounting (3 off)</td>
</tr>
<tr>
<td>7</td>
<td>Preset potentiometer, 10k, H-mount</td>
</tr>
<tr>
<td>8</td>
<td>DIL isolated resistor network, 100k × 8</td>
</tr>
<tr>
<td>9</td>
<td>DIL isolated resistor network, 220 R × 8</td>
</tr>
<tr>
<td>10</td>
<td>Quartz crystal, general purpose, 4 MHz</td>
</tr>
<tr>
<td>11</td>
<td>Capacitor, 22 pF, ceramic (2 off)</td>
</tr>
<tr>
<td>12</td>
<td>Capacitor, 22 nF, polyester (3 off)</td>
</tr>
<tr>
<td>13</td>
<td>Stripboard, SRBP 3939 100 × 100 mm</td>
</tr>
<tr>
<td>14</td>
<td>Batteries, 1.5 V, size AA, Duracell (2 off)</td>
</tr>
<tr>
<td>15</td>
<td>18-pin DIL IC socket</td>
</tr>
</tbody>
</table>

Construction

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 corner pin outwards. All the pins should then be soldered to the tracks using the minimum amount of solder necessary, whilst 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 insulated with a small strip of tape to prevent any debris. Ratios between the tracks with a small width should be used to ensure that there are no short circuits, and that adjacent tracks and solder joints.

Static Testing

When the board has been completed, it should be tested against the circuit diagram, and then for correct connections, and that there are no debris, solder splashes or shorts on or inside the plugs. With the batteries are put back, 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 should light up, and check that the supply voltages on the supply tracks and PIC socket pin: Pin 5 = 0 V and Pin 14 = +3 V. Check that
Appendix B

35

the voltages at the PIC inputs change correctly as the switches are toggled. A DMM (Digital Multimeter) or 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.

Test Program

To complete testing of the DIZI-2 board, a program should be blown into the PIC which exercises all the hardware, while remaining as simple as possible so that there is no question

Program B.1 DIZI board test program

```
; Test DIZI board test program
; diz1.asm
;
; 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
; MOVF 05,W ; get DIL input &
; MOVWF 06 ; send to display
; RLF 06 ; rotate bits left

; continued...
```
of the software being faulty. A suitable program is listed in Program B.1; it does not test the analogue input operation, which will be covered later. When this program has been loaded, the
following test procedure will confirm correct hardware operation.

<table>
<thead>
<tr>
<th>Step</th>
<th>Test</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Power Button On</td>
<td>Decimal point ON</td>
</tr>
<tr>
<td>2</td>
<td>Button B pressed and released</td>
<td>All display segments ON</td>
</tr>
<tr>
<td>3</td>
<td>Button A pressed and released</td>
<td>Buzzer sounds</td>
</tr>
<tr>
<td>4</td>
<td>Operate DIL switches</td>
<td>Segments a, b, c, d change</td>
</tr>
</tbody>
</table>

If faults are found, it is quite possible that there are still short circuits on the board. Check also that all the tracks have been cut as required, and that all connections are correct.

Analogue to Digital Conversion

One feature of the DIZI board not described in the main text is the analogue input. A similar input is available on the MOTA board, so the method for using it will now be explained. Some PIC chips, and other microcontrollers, have built-in ADCs, which allow analogue voltages to be converted to binary form for input to the processor. An ADC would be needed, for example, if a temperature is to be monitored by the controller in a process system. The general block diagram for a counting ADC is shown in Fig. B.3.

Figure B.3  General analogue to digital converter.

The 8-bit ADC shown in Fig. B.3 converts the analogue voltage present at its input to an 8-bit binary number, which means 256 different voltage levels. If the input range is set to, say, 0–5.5 V, then 255 steps of 0.03125 V can be detected. When the ‘Start Conversion’ (GO bit)
Appendix B

signal goes active, the DAC converts the binary number into a corresponding voltage. The 'End of Conversion' signal to the processor is to indicate completion of the conversion process.

**CR-ADC**

The hardware for the standard ADC above is fairly complex, while the control process is relatively simple. An alternative is to use simple external hardware with a software conversion procedure, if there is no hardware ADC available. The CR-ADC is based on the measurement, using a counter register, of the rise time in a CR network connected to the processor system input. The CR counter will generally be slower and less accurate than a hardware-based ADC, but may be quite adequate in simple applications.

The components connected to RA4 in the DIZI circuit are shown in Fig. B.4. The capacitor-charging curve at Fig. B.5 shows the time constant of the circuit as 2.3 ms, assuming that the pot is set midway. The PIC chip is a 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. Therefore the time taken to reach this level, here called the charging time, 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 constant, it is not necessary for this application.

![Figure B.4](image-url)

**Figure B.4** CR conversion network.

![Figure B.5](image-url)

**Figure B.5** CR network characteristic.
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.

The waveform which will be seen at RA4 is illustrated in Fig. B.6 (not to scale). A register labelled PotVal is incremented, and RA4 checked, within a loop taking, in the LOCK program, 20µs to execute (see Fig. B.7). An adjustable delay routine allows the timing to be modified to suit the application and the component values.

The result of the process is that a count is obtained which represents the setting of the pot. This can 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 simply adjusted to give a one decade display when the pot was turned zero. This also simplified the conversion process, as any decade of values could be used. The upper end of the 4-bit range, hex numbers A–F, are displayed as '-'. These can be used as 'hidden' digits for extra security, if required.

![Figure B.6](image.png) Conversion waveform at RA4.

![Figure B.7](image.png) CR-ADC algorithm.
A similar method can be used to operate the analogue input on the MOTA board (Fig. 13.2). The motor speed could then be controlled by a voltage source or sensor, or analogue feedback employed to implement closed loop control.

EEPROM Memory

Non-volatile read and write memory is very useful because data input by the user or acquired by the processor during its operation can be retained while the power is off. One important application area is data security and encryption. PIC devices can be programmed, for example, to control access to satellite television broadcast channels. The LOCK application illustrates this feature of the PIC 16F84 by using the EEPROM memory to store a 4-digit security code.

The PIC 16F84 has 64 bytes of EEPROM with addresses 00–3F. The memory is accessed via EEDATA and EEADR in the SFR set. The EEPROM address is loaded into EEADR, and the data byte to be stored in EEDATA. An artificially complex write initialisation sequence is then executed to actually write to the EEPROM memory, using EECON1 and EECON2 (page 106). This sequence is designed to reduce the possibility of an accidental write to the EEPROM memory. The LOCK security application 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.

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 turned, and the input decimal digits are displayed and entered. If the sequence of four input digits matches those previously stored in EEPROM, a siren sound is made to indicate the opening of the lock.

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. Only the Power button, Enter button, Digit select pot and display would be accessible to the user in the final design. The hardware would need to be reconfigured so that the unit would appear as shown in Fig. B.8 to the user. The DIL switch bank and its button for setting the entry code would be concealed.
Appendix B

Program Structure

The 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 blocks should be ordered in such a way that labels referenced have already been defined when the program is assembled. Thus, the subroutines should be placed before the main sequences in the source code. However, when actually developing the code, if you are working ‘top down’, the subroutines may actually be written after the main sequences. To place them correctly in the source code, they can then be inserted when editing, or cut and pasted later.

The program has two main sequences, for inputting and checking a combination, and two alternate endings. The processor goes to sleep after completion of the input sequence, or an incorrect digit match. The DIGI board must be re-powered to try again, as there are no other interrupts enabled to restart it. If the combination checks out correctly, the siren ending is used, which continues until the power goes off.

Pseudocode

The program is outlined below using ‘pseudocode’, which is a text method of designing the program, which may be used instead of a flowchart. 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. In this case, it must be written in a form which allows it to be easily converted to PIC assembly language. The program structure and logic can then be worked out before attempting to write the source code itself. To use pseudocode effectively, the programmer must be reasonably expert in using the language syntax.

The conventions used in the pseudocode are as follows:
- block structure applied
- target hardware specified
- register and bit labels defined
- user inputs included in the sequence
- GOTO [deslab] – Jump to destination address label

---

The 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 blocks should be ordered in such a way that labels referenced have already been defined when the program is assembled. Thus, the subroutines should be placed before the main sequences in the source code. However, when actually developing the code, if you are working ‘top down’, the subroutines may actually be written after the main sequences. To place them correctly in the source code, they can then be inserted when editing, or cut and pasted later.

The program has two main sequences, for inputting and checking a combination, and two alternate endings. The processor goes to sleep after completion of the input sequence, or an incorrect digit match. The DIGI board must be re-powered to try again, as there are no other interrupts enabled to restart it. If the combination checks out correctly, the siren ending is used, which continues until the power goes off.

Pseudocode

The program is outlined below using ‘pseudocode’, which is a text method of designing the program, which may be used instead of a flowchart. 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. In this case, it must be written in a form which allows it to be easily converted to PIC assembly language. The program structure and logic can then be worked out before attempting to write the source code itself. To use pseudocode effectively, the programmer must be reasonably expert in using the language syntax.

The conventions used in the pseudocode are as follows:
- block structure applied
- target hardware specified
- register and bit labels defined
- user inputs included in the sequence
- GOTO [deslab] – Jump to destination address label
Appendix B

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 labelled 'regname'
program block type defined:
INIT = Initialise
MAIN = Main program
SEQn = Sequence ending with GOTO
ENDn = End operation
SUBn = Subroutine, optionally receiving and/or returning values

LOCK List File
The list file for the LOCK program contains the source code and machine code. If the reader
wishes to test the program, the machine code (column 2 of the list file) can be entered directly
into the program memory buffer in the programming software. This avoids the need to type in
the source code, if the hex file itself is not available.
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;
• capitalisation of user register labels;
• identification and separation of block types.
The pseudocode and list files are reproduced in Programs B.2 and B.3.

Program B.2
Lock program pseudocode

LOCK PROGRAM PSEUDOCODE 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
continued...
Appendix B

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

SEQ2: Check 4 digits from pot for match

potin

{Read potentiometer}
CALL [getpot] for (DigVal)
{Returns with (DigVal)=00-0F}
GOTO [potin] UNTIL (butB)=0

done

{End of program}
Appendix B

Read (EEDATA) at (EEADR)
Compare (EEDATA) with (DigVal)
If (Z)=0 GOTO \[done\]
waitb
{Release Button B}
GOTO \[waitb\] UNTIL (butB)=1
CALL \[beep\]
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 0utput
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

continued...
Algorithm: Get Pot Value using CR ADC method

Antena: (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 B.3
Lock program list file

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,
00021 ; and an audio output generated to indicate the correct sequence.
00022 ; The processor halts if any digit fails to match, and the
00023 ; program must be re-executed.
00024 ;
00025 ; To set a combination:
Appendix B

0026) 1. Hold Power On Button
0027) 2. Press Button A
0028) 3. Set a digit on DIL switches and Press A - beeps
0029) 4. Repeat step 3 for 3 more digits
0030) 5. Release Power Button

To check a combination:
0035) 1. Hold Power On Button
0036) 2. Press Button B
0037) 3. Set a digit on pot and Press B - beeps if matched
0038) 4. Repeat step 3 for 3 more digits
0039) - if digits all match, siren is sounded
0040) - if any digit fails to match, the processor

To return to Power On:
0045) 1. Hold Power On Button

.Processor 16F84

EQU: Special Function Register Equates....................

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

EQU: User Register Equates................................

Period EQU 0C ; Period of Output Sound
Count EQU 0D ; Delay Down Counter
PotVal EQU 0E ; Analogue Input Value
DigVal EQU 0F ; Current Digit Value 00 to 09

EQU: User Bit Equates.....................................

butA EQU 4 ; PORTA - RA4 Input Button
butB EQU 0 ; PORTB - RB0 Input Button
buzO EQU 0 ; PORTB - RB0 Output Buzzer

********************************************************

continued...
Appendix B

00078

; INIT: Initialise Port B (Port A defaults to inputs)
00079

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

00085

; SUBROUTINES *****************************************
00086

; SUB1: 7-Segment Code Table using PCL + offset in W
00087

Returns
00088
digit display codes, with '−' for numbers A to F
00089

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

;--------------------------------------------
00109 ; SUB2: Delay routine
00110 ; Receives delay count in W
00111

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 ; SUB3: Output One Beep Cycle to BuzO
00119

0019 30FF 00121 beep MOVLW 0FF ; Load FF into
001A 008C 00122 MOVWF Period ; Period counter
001B 3000 00124 MOVLW B'00000000' ; Set RB0
001C 0066 00125 TRIS PORTB ; as output
00126 ; Do one cycle of rising tone....
00127
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
Appendix B

0020 1006 00133 BCF PORTB, buz0 ; 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 ; Set RB0 to input again................................
00138
0024 3001 00140 MOVLW B'00000001' ; Reset RB0
0025 0066 00141 TRIS PORTB ; as input
0026 0008 00142 RETURN ; from tone cycle

00143 ;--------------------------------------------
00144 ; SUB4: Get pot value (Rv) using rise time due to C and R
00145 ; Returns with digit value (0-F) in DigVal
00146
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
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 ; Increment a counter until RA4 goes high due to
00158 ; 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
0032 1E05 00164 BTFSS PORTA, 4 ; Check input bit RA4
0033 282F 00165 GOTO check ; and repeat if not yet high

00166 ; Mask out high bits of count value, and store & display
00167 ; 4-bit digit value, 0-F
00168
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 ; MAIN SEQUENCES**************************************
00176 ;SEQ1: Store 4 Digits in non volatile EEPROM
00177 ; Beep after each digit, and twice when 4 done

continued...
Appendix B

00183: Complete Button A input operation

00184: Delay for about 1ms

00185: stocom MOVLW 0F00; Delay for about 1ms

00186: CALL delay; to avoid Button A switch bounce

00187: BTFSS PORTA, butA; Wait for Button A

00188: GOTO stocom; to be released

00189: Read 4-bit binary number from DIL switches into EEDATA and display

00190: CLRF EEADR; Zero EEPROM address register

00191: MOVF PORTA, W; Read DIL switches

00192: ANDLW 0F; and set high 4 bits to 0

00193: MOVWF EEDATA; Put DIL value in EEPROM data

00194: CALL codtab; Display DIL input as decimal

00195: MOVWF PORTB;

00196: BTFSC PORTA, butA; Check if Button A pressed

00197: GOTO getdil; If not, keep reading DIL input

00198: Store the current DIL input in EEPROM at current address

00199: CLRF EEADR; Zero EEPROM address

00200: CALL getpot; Get a digit value set on pot (Rv)

00201: BSF STATUS, RP0; Select Register Bank 1

00202: BSF EECON1, WREN; Enable EEPROM write

00203: MOVLW 055; Write initialisation sequence

00204: MOVWF EECON2;

00205: MOVLW 0AA;

00206: MOVWF EECON2;

00207: BSF EECON1, WR; Write data into current address

00208: BCF STATUS, RP0; Re-select Register Bank 0

00209: WAITA BTFSS PORTA, butA; Wait for Button A to be released

00210: GOTO WAITA;

00211: CALL beep; Beep twice when 4 digits stored

00212: GOTO done; Go to sleep when done

00213: ; --------------------------------------------

00214: SEQ2: Check PotVal v EEPROM

00215: MOVLW 0FF; Delay for about 1ms

00216: CALL delay; to avoid Button B switch bounce

00217: BTFSS PORTB, butB; Wait for Button B to be released

00218: GOTO checom; If not, keep reading DIL input

00219: ; Read the value set on the input pot

00220: CLRF EEADR; Zero EEPROM address

00221: CALL getpot; Get a digit value set on pot (Rv)
Appendix B

365 005D 1806 00241 BTFSC PORTB, butB ; Check if Button pressed again
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
00247 1408 00248 BSF EECON1, RD ; Read selected EEPROM location
00249 1283 00248 BCF STATUS, RP0 ; Re-select Register Bank 0
00250 0808 00251 MOVF EEDATA, W ; Copy EEPROM data to W
00252 068F 00253 XORWF DigVal ; Compare the input with EEPROM data
00254 1D03 00255 BTFSS STATUS, Z ; If it does not match, go to sleep
00256 2874 00257 GOTO done ;
00258 00259 ; If digit match obtained, check if 4 done and do next if not
00260 1C06 00261 waitb BTFSS PORTB, butB ; Wait for Button B to be released
00262 2866 00263 GOTO waitb ;
00263 2019 00264 CALL beep ; Beep to confirm successful match
00264 0D25 00265 INCF EEADR ; Select next EEPROM location
00266 1D09 00267 BTFSS EEADR, 2 ; 4 digits checked yet?
00268 285C 00269 GOTO potin ; If not, do the next
00270 2872 00271 GOTO siren ; When 4 digits done, run siren

00266 ; ****************************************************
00267 ; MAIN: Select Set or Check Combination
00268 1E05 00269 select BTFSS PORTA, butA ; Button A pressed?
00270 283A 00271 GOTO stocom ; If so, store a combination
00272 1C06 00273 BTFSS PORTB, butB ; Button B pressed?
00274 2857 00275 GOTO checom ; If so, check a combination
00276 286D 00277 GOTO select ; repeat endlessly

00275 ; ***************************************************
00276 ; END1: When combination successfully matched, make siren sound
00277 2019 00278 siren CALL beep ; Do a tone cycle
00278 2872 00279 GOTO siren ; and repeat endlessly

00276 ;--------------------------------------------
00277 ; END2: When a digit check fails, go to sleep, and try again
00278 0186 00279 done CLRF PORTB ; Switch off display
00280 0063 00281 SLEEP ; Processor halts

00276 ; ***************************************************
00277 END ; of program source code

continued...
<table>
<thead>
<tr>
<th>LABEL</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>0000000D</td>
</tr>
<tr>
<td>DigVal</td>
<td>0000000F</td>
</tr>
<tr>
<td>EEADR</td>
<td>00000009</td>
</tr>
<tr>
<td>EECON1</td>
<td>00000008</td>
</tr>
<tr>
<td>EECON2</td>
<td>00000009</td>
</tr>
<tr>
<td>EEDATA</td>
<td>00000008</td>
</tr>
<tr>
<td>Period</td>
<td>0000000C</td>
</tr>
<tr>
<td>RD</td>
<td>00000000</td>
</tr>
<tr>
<td>RP0</td>
<td>00000005</td>
</tr>
<tr>
<td>STATUS</td>
<td>00000003</td>
</tr>
<tr>
<td>WR</td>
<td>00000001</td>
</tr>
<tr>
<td>WREN</td>
<td>00000002</td>
</tr>
<tr>
<td>Z</td>
<td>00000002</td>
</tr>
<tr>
<td>__16C84</td>
<td>00000001</td>
</tr>
<tr>
<td>beep</td>
<td>00000019</td>
</tr>
<tr>
<td>butA</td>
<td>00000004</td>
</tr>
<tr>
<td>butB</td>
<td>00000000</td>
</tr>
<tr>
<td>buzO</td>
<td>00000000</td>
</tr>
<tr>
<td>check</td>
<td>0000002F</td>
</tr>
<tr>
<td>checom</td>
<td>00000057</td>
</tr>
<tr>
<td>codtab</td>
<td>00000004</td>
</tr>
<tr>
<td>cycle</td>
<td>0000001D</td>
</tr>
<tr>
<td>delay</td>
<td>00000015</td>
</tr>
<tr>
<td>done</td>
<td>00000074</td>
</tr>
<tr>
<td>getdil</td>
<td>0000003F</td>
</tr>
<tr>
<td>getpot</td>
<td>00000027</td>
</tr>
<tr>
<td>loop</td>
<td>00000016</td>
</tr>
<tr>
<td>potin</td>
<td>0000005C</td>
</tr>
<tr>
<td>select</td>
<td>0000006D</td>
</tr>
<tr>
<td>siren</td>
<td>00000072</td>
</tr>
<tr>
<td>start</td>
<td>00000000</td>
</tr>
<tr>
<td>stocom</td>
<td>0000003A</td>
</tr>
<tr>
<td>store</td>
<td>00000046</td>
</tr>
<tr>
<td>waita</td>
<td>0000004E</td>
</tr>
<tr>
<td>waitb</td>
<td>00000066</td>
</tr>
</tbody>
</table>

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

0000 : XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXX
0040 : XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXX-----
All other memory blocks unused.