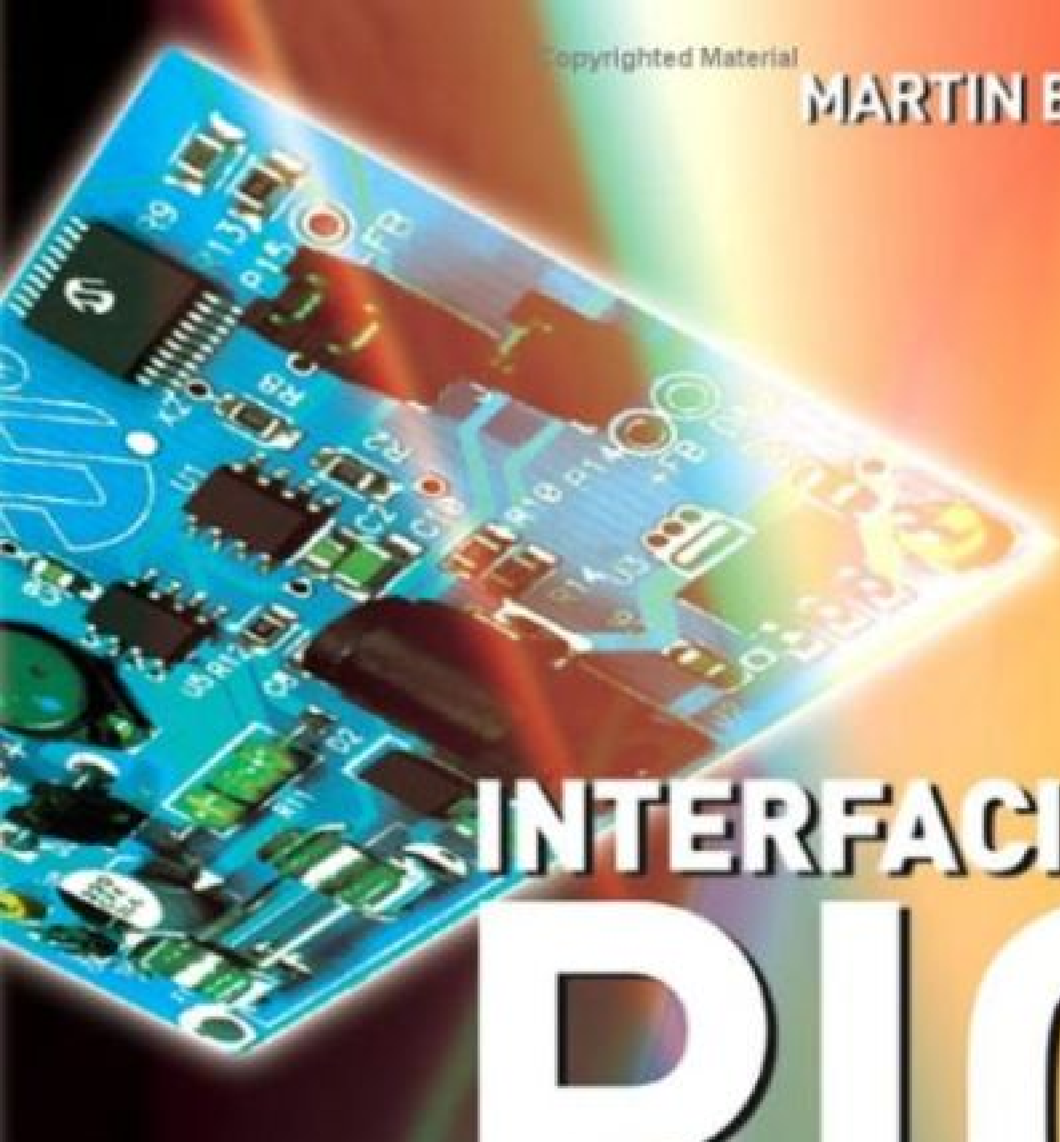
MARTIN BATES



INTERFACING PIC MICROCONTROLLERS

EMBEDDED DESIGN BY INTERACTIVE SIMULATION

Interfacing PIC Microcontrollers

Embedded Design by Interactive Simulation

This page intentionally left blank

Interfacing PIC Microcontrollers
Embedded Design by Interactive Simulation

Martin Bates

Amsterdam • Boston • Heidelberg • London • New York • Oxford
Paris • San Diego • San Francisco • Singapore • Sydney • Tokyo
Newnes is an imprint of Elsevier

Newnes is an imprint of Elsevier
Linacre House, Jordan Hill, Oxford OX2 8DP, UK
The Boulevard, Langford Lane, Kidlington, Oxford OX5 1GB, UK
84 Theobald's Road, London WC1X 8RR, UK
Radarweg 29, PO Box 211, 1000 AE Amsterdam, The Netherlands
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA
525 B Street, Suite 1900, San Diego, CA 92101-4495, USA

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

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the Library of Congress

For information on all Newnes publications
visit our web site at books.elsevier.com

Printed and bound in Great Britain
06 07 08 09 10 10 9 8 7 6 5 4 3 2 1

ISBN-13: 978-0-7506-8028-8
ISBN-10: 0-7506-80288-8

Preface

I have my students to thank for this book. Regardless of ability, each has had a role to play. The more able students have always helped, through their project work, to develop new ideas and solutions in electronic design. Some have displayed an astonishing instinctive understanding of engineering ideas, and some have been so keen to learn as to make teaching easy and rewarding. There is never enough time to give each individual student the time and help they deserve. So, one has to start writing to make sure that the essential technical information is at least accessible, and hope that students are able to make best use of it.

Another spur to writing this book has been the development of the interactive design software which has made the job of learning and teaching electronics that much more enjoyable. The Proteus software used in this book has been developed by a talented team at Labcenter Electronics in the UK, led by John Jameson and Iain Cliffe. They have a world beating product, and I wanted to make a small contribution to encouraging students and engineers to use it. It allows us to bring electronic circuits to life on the computer screen instantly.

It has always been a problem in electronics that you cannot see a circuit working in the same way that a mechanical engineer can see a steam engine pumping up and down. Sure, we can see the screen flickering on a television, or an electric motor spinning, but you cannot see electrons or volts. As a result, it has always been that much more difficult to teach electronics. Proteus is a big step towards bringing electronics alive, as such it helps us to participate more effectively in the communications and information revolution that is all around us.

This page intentionally left blank

Introduction

This book is a sequel to my first effort ‘PIC Microcontrollers, an Introduction to Microelectronics’. This attempted to provide a comprehensive introduction to the subject via a single type of microcontroller, which is essentially a complete computer on a chip. The PIC was the first widely available device to use flash memory, which makes it ideal for experimental work. Flash memory allows the program to be replaced quickly and easily with a new version. It is now commonplace, not least in our USB memory sticks, but also in a wide range of electronic systems where user data need to be retained during power down. Cheap flash memory microcontrollers have transformed the teaching of microelectronics – they are re-usable and the internal architecture is fixed, making them easier to explain. On the other hand, beginners can ignore the innards and treat them as a black box, and get on with the programming! The small instruction set of the PIC is also a major advantage – only 35 instructions to learn. Compare that with a complex processor such as the Pentium, which is quite terrifying compared with the PIC! The quality of the PIC technical documentation is also a major factor.

For these reasons, I set out to introduce the PIC into my teaching as widely as possible. At the same time, schools, universities and hobbyists were starting to use it, so continuity of learning was possible. Since there is never enough time to teach all the detail, I decided to set out a full description of the basic PIC device, the 16F84, and some representative applications. Although this particular chip is now redundant in terms of new products, the basic architecture is unchanged in current chips, so it is still a useful starting point.

My students and I soon graduated to the more powerful PIC 16F877. This is now used widely as a more advanced teaching device, because it has a full complement of interfaces: analogue input, serial ports, slave port and so on, plus a good range of hardware timers. A full description of this chip covers most of the features that higher level students need for project work with microcontrollers.

When interactive simulation of microcontrollers became available, a new dimension was added. We could now see them in action without having to spend a lot of time building and debugging hardware! These design tools allow even the inexperienced designer to create a working system relatively quickly. As a result, my next step was to document the 16F877 and its applications, through the medium of interactive simulation.

Proteus© from Labcenter Electronics© consists of two main parts, ISIS and ARES. ARES is a layout package, which is used to create a PCB when the circuit has been designed. ISIS is the schematic capture and interactive simulation software used to create the circuit drawing and to test the circuit prior to building the real hardware. SPICE is a mathematical circuit modelling system which has been developed over many years – these models can now be used to bring the drawing to life. Onscreen buttons and virtual signal sources, for example, provide inputs to the circuit. Output can be displayed on a voltage probe or on a virtual oscilloscope. Now that we have microcontroller simulation as well, we really are in business. The MCU can be dropped on the screen, a program attached and debugged instantly. Electronic design has never been so easy!

It is assumed that the reader is familiar with the basics of microcontroller systems, as covered in the first book. This one follows on, and is divided into three main parts. In the first part, the 16F877 hardware and programming and the simulation system are introduced. In the second part, a range of interfacing techniques are covered; switches, keypads, displays, digital and analogue interfacing, data conversion and so on. In the third part, power outputs, serial interfaces, sensors, and system design examples culminate in a design for a general purpose board which provides a platform for further development.

Each topic is illustrated by designs based on the 16F877, so that the reader can concentrate on the interfacing and not have to deal with different microcontrollers. All the circuits are available on the associated website (see links below). All schematics were produced using ISIS – and you can produce them to the same standard in your own reports. The designs can be downloaded and run along side the book. ISIS Lite, the introductory design package, can be downloaded free, with extra features available for a small registration fee. The 16F877 will simulate fully, and the software changed, but the hardware cannot be modified unless a licence is purchased for this device. The microcontroller models can be purchased for institution or professional use in packages – see the Labcenter website.

Get PICing!

Links, References and Acknowledgements

Support Website

www.picmicros.org.uk

This book is supported by the above website created by the author to provide

- application examples for downloading, listed by chapter
- applications to be displayed on screen for teaching purposes
- easy access to relevant data sheets
- links to relevant manufacturers websites, Microchip and Labcenter
- links to 'PIC Microcontrollers - an Introduction to Microelectronic Systems'

If you have Proteus Professional installed and a licence for the PIC 16 series microcontrollers, you will be able to modify the hardware and application program to your own requirements.

If not, Labcenter have agreed a **special offer** for readers of this book: a special low cost edition of ISIS Lite schematic capture, with PROSPICE Lite simulation tools and PIC 16F877 licence. A key will be e-mailed to you which will allow the demo programs to be fully tested and modified.

If you do not have a licensed copy of Proteus, you can download the demo version and run the applications and modify the code, but not the hardware.

Please log on to www.picmicros.org.uk for details; also visit www.labcenter.co.uk and www.proteuslite.com for Proteus/ISIS information and downloads.

Labcenter Electronics
www.labcenter.co.uk
Manufacturer and supplier of Proteus VSM electronic design system

Microchip Technology Inc.
www.microchip.com
Manufacturer of the PIC microcontroller range and MPLAB IDE

Custom Computer Services, Inc.
www.ccsinfo.com
Manufacturer and supplier of PIC CCS 'C' Compilers

Data References and Trademark Acknowledgements

Microchip Technology Inc., RS Components, Fairchild, Intel, Freescale (Motorola), National Semiconductor, Sensor Technics, Densitron, Honeywell, SGS Thomson, Maxim, ST Microelectronics, HBM, ARM, AVR Atmel, Texas, Vishay.

I would also like to thank the dedicated teachers of engineering that I have worked with, especially Melvyn Ball at Hastings College and Chris Garrett at the University of Brighton, and, of course, Julia Bates.

Martin Bates
Hastings, UK

Contents

Part 1

Microcontroller

This page intentionally left blank

1

PIC Hardware

The microcontroller is simply a computer on a chip. It is one of the most important developments in electronics since the invention of the microprocessor itself. It is essential for the operation of devices such as mobile phones, DVD players, video cameras, and most self-contained electronic systems. The small LCD screen is a good clue to the presence of an MCU (Microcontroller Unit) – it needs a programmed device to control it. Working sometimes with other chips, but often on its own, the MCU provides the key element in the vast range of small, programmed devices which are now commonplace.

Although small, microcontrollers are complex, and we have to look carefully at the way the hardware and software (control program) work together to understand the processes at work. This book will show how to connect the popular PIC range of microcontrollers to the outside world, and put them to work. To keep things simple, we will concentrate on just one device, the PIC 16F877, which has a good range of features and allows most of the essential techniques to be explained. It has a set of serial ports built in, which are used to transfer data to and from other devices, as well as analogue inputs, which allow measurement of inputs such as temperature. All standard types of microcontrollers work in a similar way, so analysis of one will make it possible to understand all the others.

The PIC 16F877 is also a good choice for learning about micro-controllers, because the programming language is relatively simple, as compared with a microprocessor such as the Intel Pentium™, which is used in the PC. This has a powerful, but complex, instruction set to support advanced multimedia applications. The supporting documentation for the PIC MCU is well designed, and a development system, for writing and testing programs, can be downloaded free from the Microchip website (www.microchip.com).

Processor System

The microcontroller contains the same main elements as any computer system:

- Processor
- Memory
- Input/Output

In a PC, these are provided as separate chips, linked together via bus connections on a printed circuit board, but under the control of the microprocessor (CPU). A bus is a set of lines which carry data in parallel form which are shared by the peripheral devices. The system can be designed to suit a particular application, with the type of CPU, size of memory and selection of input/output (I/O) devices tailored to the system requirements.

In the microcontroller, all these elements are on one chip. This means that the MCU for a particular application must be chosen from the available range to suit the requirements. In any given circuit, the microcontroller also tends to have a single dedicated function (in contrast to the PC); this type of system is described as an embedded application (Figure 1.1).

Processor

In a microprocessor system or a microcontroller, a single processor block is in charge of all input, output, calculations and control. This cannot operate without a program, which is a list of instructions that is held in memory. The

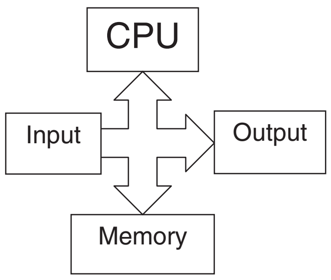


Figure 1.1 Block diagram of a basic microprocessor system

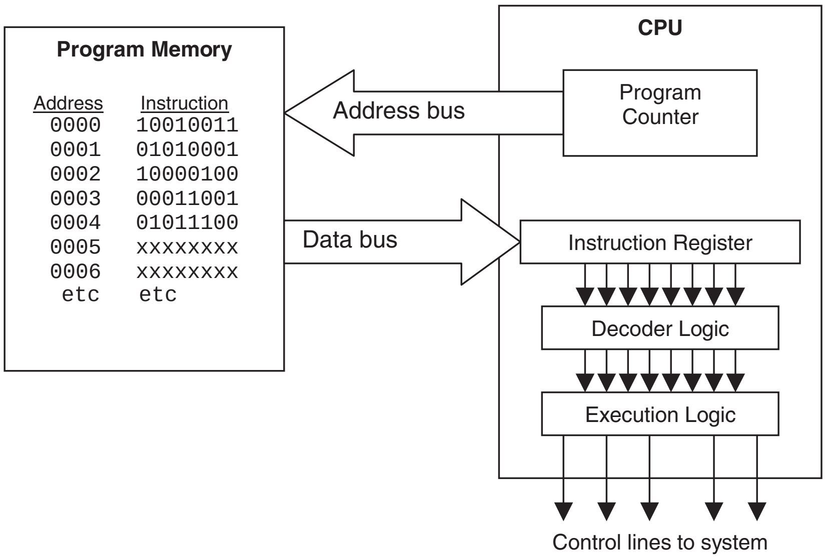


Figure 1.2 Processor program execution

program consists of a sequence of binary codes that are fetched from memory by the CPU in sequence, and executed (Figure 1.2).

The instructions are stored in numbered memory locations, and copied to an instruction register in the CPU via the data bus. Here, the instruction controls the selection of the required operation within the control unit of the processor. The program codes are located in memory by outputting the address of the instruction on an address bus. The address is generated in the program counter, a register that starts at zero and is incremented or modified during each instruction cycle. The busses are parallel connections which transfer the address or data word in one operation. A set of control lines from the CPU are also needed to assist with this process; these control lines are set up according to the requirements of the current instruction.

Decoding the instruction is a hardware process, using a block of logic gates to set up the control lines of the processor unit, and fetching the instruction operands. The operands are data to be operated on (or information about where to find it) which follows most instructions. Typically, a calculation or logical operation is carried out on the operands, and a result stored back in memory, or an I/O action set up. Each complete instruction may be 1, 2 or more bytes long, which includes the operation (instruction) code (op-code) itself and the operand/s (1 byte = 8 bits).

Thus, a list of instructions in memory is executed in turn to carry out the required process. In a word processor, for example, keystrokes are read in via the keyboard port, stored as character codes, and sent to a screen output for display. In a game, input from the switches on the control pad are processed and used to modify the screen. In this case, speed of the system is a critical factor.

Memory

There are two types of memory: volatile and non-volatile. Volatile memory loses its data when switched off, but can be written by the CPU to store current data; this is RAM (Random Access Memory). ROM (Read Only Memory) is non-volatile, and retains its data when switched off.

In a PC, a small ROM is used to get the system started when it is switched on; it contains the BIOS (Basic Input Output System) program. However, the main Operating System (OS), for example, Windows™ and application program (e.g. Word) have to be loaded into RAM from Hard Disk Drive (HDD), which takes some time, as you may have noticed!

So why not put the OS in ROM, where it would be instantly available? Well, RAM is faster, cheaper and more compact, and the OS can be changed or upgraded if required. In addition, an OS such as Windows is very large, and some elements are only loaded into RAM as needed. In addition, numerous applications can be stored on disk, and loaded only as required.

The ideal memory is non-volatile, read and write, fast, large and cheap. Unfortunately, it does not exist! Therefore, we have a range of memory technologies as shown in Table 1.1, which provide different advantages, which

	ROM	**Flash ROM**	**RAM**	**CD-ROM**	**DVD-RW**	**HDD**
Description	*Chip*	*Chip*	*Chip*	*Optical disk*	*Optical disk*	*Magnetic disk*
Sample size*	128 kb	128 Mb	512 Mb	650 Mb	4.7 Gb	30 Gb
Non-volatile	✓	✓	x	✓	✓	✓
Write (many)	Once	✓	✓	Once	✓	✓
Large (bytes)	x	?	x	✓	✓	✓
Cheap (per bit)	?	x	?	✓	✓	✓
Fast (access)	?	✓	✓	x	x	x

*1 byte = 8 bits
1 kb = 1 kilobyte = 1024 bytes
1 Mb = 1 megabyte = 1024 kb
1 Gb = 1 gigabyte = 1024 Mb

Table 1.1 Memory and data storage technologies

may all be used with a standard PC. The main trade-off is cost, size and speed of access. Flash ROM, as used in memory sticks and MP3 players, is closest to the ideal, having the advantages of being non-volatile and rewritable. This is why it is used as program memory in microcontrollers which need to be reprogrammed, such as the PIC 16F877.

Input and Output

Without some means of getting information and signals in and out, a data processing or digital control system would not be very useful. Ports are based on a data register, and set of control registers, which pass the data in and out in a controlled manner, often according to a standard protocol (method of communication).

There are two main types of port: parallel and serial. In a parallel port, the data is usually transferred in and out 8 bits at a time, while in the serial port it is transmitted 1 bit at a time on a single line. Potentially, the parallel port is faster, but needs more pins; on the other hand, the port hardware and driver software are simpler, because the serial port must organise the data in groups of bits, usually 1 byte at a time, or in packets, as in a network (Figure 1.3).

Taking printers as an example, the old standard is a parallel port (Centronics), which provides data to the printer 1 byte (8 bits) at a time via a multipin connector. The new standard, USB (Universal Serial Bus) is a serial data system, sending only 1 bit at a time. Potentially, the parallel connection is 8 times faster, but USB operates at up to 480 megabits (Mb) per second, and the printer is slow anyway, so there is no problem. One advantage of using

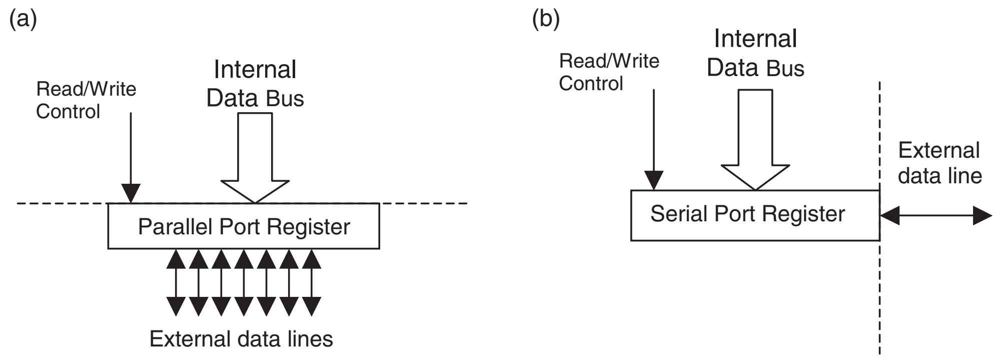


Figure 1.3 Parallel and serial data ports: (a) parallel; (b) serial

USB is that it provides a simple, robust connector and this outweighs the fact that the interface protocol (driver software) is relatively complex, because this is hidden from the user. USB also provides power to the peripheral, if required, and the printer can be daisy-chained with other devices. USB also automatically configures itself for different peripherals, such as scanners and cameras.

In the parallel port operating in output mode, the data byte is loaded from the internal data bus under the control of a read/write signal from the CPU. The data can then be seen on the output pins by the peripheral; for testing, a logic probe, logic analyser or just a simple LED indicator can be used. In input mode, data presented at the input pins from a set of switches or other data source are latched into the register when the port is read, and is then available on the data bus for collection by the CPU. One of the functions of the port is to separate the internal data bus from the external hardware, and the other is to temporarily store the data. The data can then be transferred to memory, or otherwise processed, as determined by the CPU program.

The serial port register also loads data from the internal bus in parallel, but then sends it out 1 bit at a time, operating as a shift register. If an asynchronous serial format is used, such as RS232 (COM ports on old PCs), start and stop bits are added so that bytes can be separated at the receiving end. An error check bit is also available, to allow the receiver to detect corrupt data. In receive mode, the register waits for a start bit, and then shifts in the data at the same speed as it is sent. This means the clock rate for the send and receive port must be the same. The USART (Universal Synchronous/Asynchronous Receive/Transmit) protocol will be described in more detail later.

A USB or network port is more sophisticated, and arranges the data bytes in packets of, say, 1k bytes, which are sent in a form which is self-clocking; that is, there is a transition within each bit (1 or 0), so each can be picked up individually. An error-correction code follows the data, which allows mistakes to be corrected, rather than just be detected. This reduces the need for retransmission of incorrectly received data, as required by simple error detection. Addressing information preceding the data allows multiple receivers to be used.

The PIC 16F877, in common with most current MCUs, does not have USB or network interfaces built in, so we can avoid detailed consideration of these complex protocols. It does, nevertheless, have a good selection of other interfaces, which will be discussed in detail and sample programs provided.

PIC 16F877 Architecture

Microcontrollers contain all the components required for a processor system in one chip: a CPU, memory and I/O. A complete system can therefore be built using one MCU chip and a few I/O devices such as a keypad, display and other interfacing circuits. We will now see how this is done in practice in our typical microcontroller.

PIC 16F877 Pin Out

Let us first consider the pins that are seen on the IC package, and we can then discover how they relate the internal architecture. The chip can be obtained in different packages, such as conventional 40-pin DIP (Dual In-Line Package), square surface mount or socket format. The DIP version is recommended for prototyping, and is shown in Figure 1.4.

Most of the pins are for input and output, and arranged as 5 ports: A(5), B(8), C(8), D(8) and E(3), giving a total of 32 I/O pins. These can all operate as simple digital I/O pins, but most have more than one function, and the mode of operation of each is selected by initialising various control registers within the chip. Note, in particular, that Ports A and E become ANALOGUE INPUTS by default (on power up or reset), so they have to set up for digital I/O if required.

Port B is used for downloading the program to the chip flash ROM (RB6 and RB7), and RB0 and RB4–RB7 can generate an interrupt. Port C gives access to timers and serial ports, while Port D can be used as a slave port, with Port E providing the control pins for this function. All these options will be explained in detail later.

Reset = 0, Run = 1	**MCLR**	1	40	**RB7**	Port B, Bit 7 (Prog. Data, Interrupt)
Port A, Bit 0 (Analogue AN0)	**RA0**	2	39	**RB6**	Port B, Bit 6 (Prog. Clock, Interrupt))
Port A, Bit 1 (Analogue AN1)	**RA1**	3	38	**RB5**	Port B, Bit 5 (Interrupt)
Port A, Bit 2 (Analogue AN2)	**RA2**	4	37	**RB4**	Port B, Bit 4 (Interrupt)
Port A, Bit 3 (Analogue AN3)	**RA3**	5	36	**RB3**	Port B, Bit 3 (LV Program)
Port A, Bit 4 (Timer 0)	**RA4**	6	35	**RB2**	Port B, Bit 2
Port A, Bit 5 (Analogue AN4)	**RA5**	7	34	**RB1**	Port B, Bit 1
Port E, Bit 0 (AN5, Slave control)	**RE0**	8	33	**RB0**	Port B, Bit 0 (Interrupt)
Port E, Bit 1 (AN6, Slave control)	**RE1**	9	32	**V_{DD}**	+5V Power Supply
Port E, Bit 2 (AN7, Slave control)	**RE2**	10	31	**Vss**	0V Power Supply
+5V Power Supply	**V_{DD}**	11	30	**RD7**	Port D, Bit 7 (Slave Port)
0V Power Supply	**Vss**	12	29	**RD6**	Port D, Bit 6 (Slave Port)
(CR clock) XTAL circuit	**CLKIN**	13	28	**RD5**	Port D, Bit 5 (Slave Port)
XTAL circuit	**CLKOUT**	14	27	**RD4**	Port D, Bit 4 (Slave Port)
Port C, Bit 0 (Timer 1)	**RC0**	15	26	**RC7**	Port C, Bit 7 (Serial Ports)
Port C, Bit 1 (Timer 1)	**RC1**	16	25	**RC6**	Port C, Bit 6 (Serial Ports)
Port C, Bit 2 (Timer 1)	**RC2**	17	24	**RC5**	Port C, Bit 5 (Serial Ports)
Port C, Bit 3 (Serial Clocks)	**RC3**	18	23	**RC4**	Port C, Bit 4 (Serial Ports)
Port D, Bit 0 (Slave Port)	**RD0**	19	22	**RD3**	Port D, Bit 3 (Slave Port)
Port D, Bit 1 (Slave Port)	**RD1**	20	21	**RD2**	Port D, Bit 2 (Slave Port)

Figure 1.4 PIC 16F877 pin out

The chip has two pairs of power pins ($V_{DD} = +5$ V nominal and $V_{ss} = 0$ V), and either pair can be used. The chip can actually work down to about 2 V supply, for battery and power-saving operation. A low-frequency clock circuit using only a capacitor and resistor to set the frequency can be connected to CLKIN, or a crystal oscillator circuit can be connected across CLKIN and CLKOUT. MCLR is the reset input; when cleared to 0, the MCU stops, and restarts when MCLR = 1. This input must be tied high allowing the chip to run if an external reset circuit is not connected, but it is usually a good idea to incorporate a manual reset button in all but the most trivial applications.

PIC 16F877 Block Diagram

A block diagram of the 16F877 architecture is given in the data sheet, Figure 1-2 (downloadable from www.microchip.com). A somewhat simplified version is given in Figure 1.5, which emphasises the program execution mechanism.

The main program memory is flash ROM, which stores a list of 14-bits instructions. These are fed to the execution unit, and used to modify the RAM file registers. These include special control registers, the port registers and a set of general purpose registers which can be used to store data temporarily. A separate working register (W) is used with the ALU (Arithmetic Logic Unit) to process data. Various special peripheral modules provide a range of I/O options.

There are 512 RAM File Register addresses (0–1FFh), which are organised in 4 banks (0–3), each bank containing 128 addresses. The default (selected on power up) Bank 0 is numbered from 0 to 7Fh, Bank 1 from 80h to FFh and so on. These contain both Special Function Registers (SFRs), which have a dedicated purpose, and the General Purpose Registers (GPRs). The file registers are mapped in Figure 2-3 of the data sheet. The SFRs may be shown in the block diagram as separate from the GPRs, but they are in fact in the same logical block, and addressed in the same way. Deducting the SFRs from the total number of RAM locations, and allowing for some registers which are repeated in more than one bank, leaves 368 bytes of GPR (data) registers.

Test Hardware

We need to define the hardware in which we will demonstrate PIC program operation. Initially, a block diagram is used to outline the hardware design (Figure 1.6). The schematic symbol for the MCU is also shown indicating the pins to be used. For this test program, we simply need inputs which switch between 0 V and +5 V, and a logic indication at the outputs. For simulation purposes, we will see that the clock circuit does not have to be included in the schematic; instead, the clock frequency must be input to the MCU properties dialogue. The power supply pins are implicit – the simulated MCU operates at +5 V by default. Unused pins can be left open circuit, as long as they are programmed as inputs.

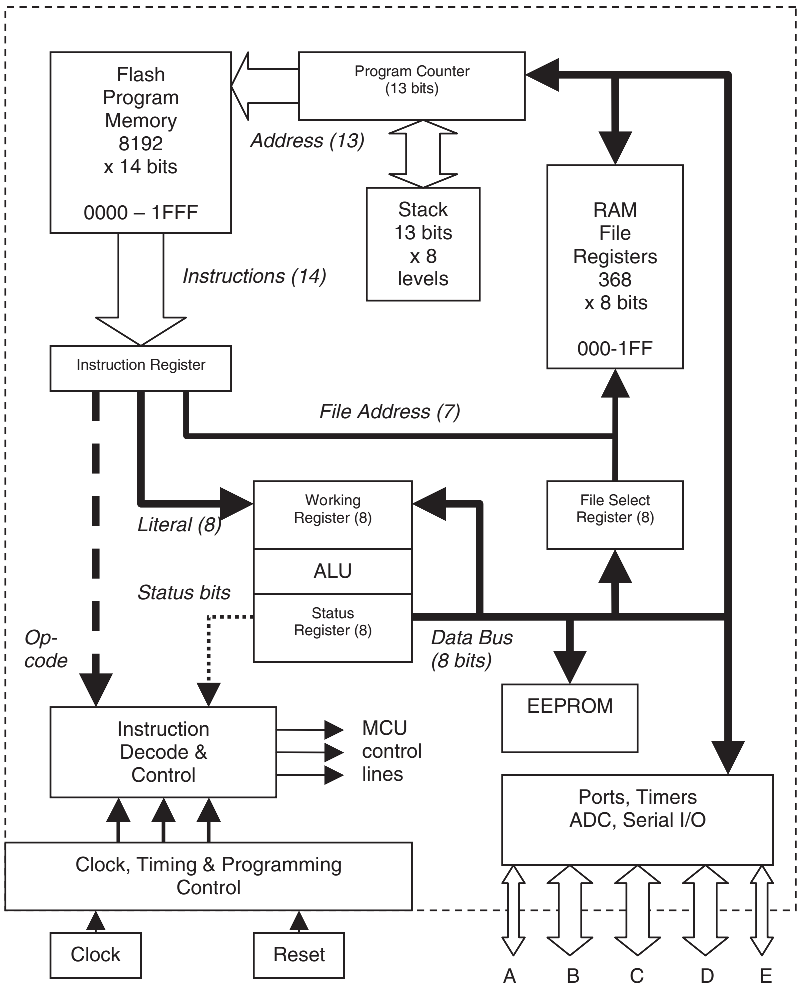


Figure 1.5 16F877 program execution block diagram

The full schematic is shown in Chapter 3 (Figure 3.1).

The first test program, BIN1, will simply light a set of LEDs connected to Port B in a binary count sequence, by incrementing Port B data register. The second program, BIN4, will use two input push buttons attached to Port D to control the output (start, stop and reset). The program will also include a delay so that the output is slower, and visible to the user. Detailed design of the interfacing will be covered later. A simple CR clock will be used, which is set to 40 kHz ($C = 4.7$ nF, $R \approx 5$ kΩ (preset), $CR = 25$ µs). This will give an instruction execution time of 100 µs.

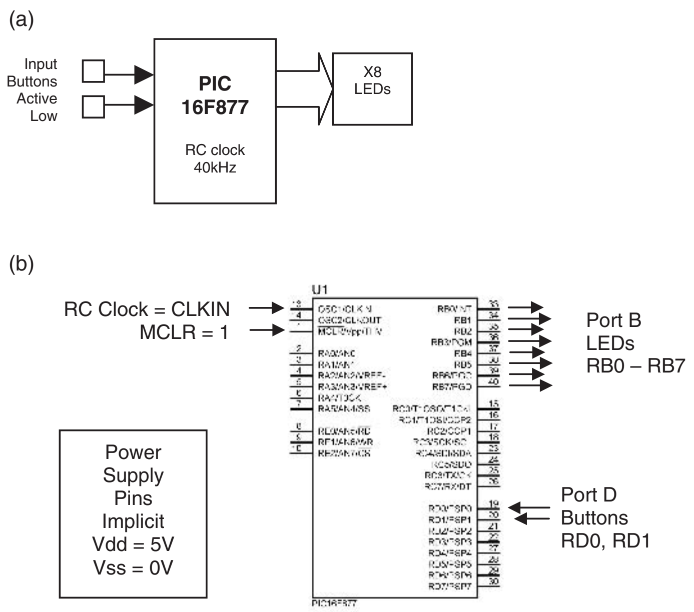


Figure 1.6 BINX hardware outline: (a) block diagram; (b) PIC 16F877 MCU connections

The PIC Program

The program is written as a source code (a simple text file) on a PC host computer. Any text editor such as Notepad™ can be used, but an editor is provided with the standard PIC development system software MPLAB (downloadable from www.microchip.com). The instructions are selected from the pre-defined PIC instruction set (Table 13-2 in the data sheet) according to the operational sequence required. The source code file is saved as PROGNAME.ASM. More details of the assembler program syntax are given later.

The source code is assembled (converted into machine code) by the assembler program MPASM, which creates the list of binary instruction codes. As this is normally displayed as hexadecimal numbers, it is saved as PROGNAME.HEX. This is then downloaded to the PIC chip from the PC by placing the MCU in a programming unit which is attached to the serial port of PC, or by connecting the chip to a programmer after fitting it in the application board (in-circuit programming). The hex code is transferred in serial form via Port B into the PIC flash program memory. A list file is created by the assembler, which shows the source code and machine code in one text file. The list file for a simple program which outputs a binary count at Port B is shown in Program 1.1.

```
Memory        Hex          Line          Address      Operation    Operand
Address       Code         Number        Label        Mnemonic
---------------------------------------------------------------------------------


                            00001                       PROCESSOR 16F877
                            00002
0000          3000          00003                       MOVLW         00
0001          0066          00004                       TRIS          06
                            00005
0002          0186          00006                       CLRF          06
0003          0A86          00007        again          INCF          06
0004          2803          00008                       GOTO          again
                            00009
                            00010                       END

Note: Lines 00001 and 00010 are assembler directives
```

Program 1.1 BIN1 list file

The program listing includes the source code at the right, with source line numbers, the hex machine code and the memory location where each instruction is stored (0000–0004). Notice that some statements are assembler directives, not instructions: PROCESSOR to specify the MCU type and END to terminate the source code. These are not converted into machine code.

The '877 has 8k of program memory, that is, it can store a maximum of $1024 \times 8 = 8192$ 14-bit instructions. By default, it is loaded, and starts executing, from address zero. In real-time (control) applications, the program runs continuously, and therefore loops back at the end. If it does not, be careful – it will run through the blank locations and start again at the beginning!

Let us look at a typical instruction to see how the program instructions are executed.

Source code:	MOVLW 05A
Hex code:	305A (4 hex digits)
Binary code:	0011 0000 0101 1010 (16 bits)
Instruction:	11 00xx kkkk kkkk (14 bits)

The instruction means: Move a Literal (given number, 5Ah) into the Working register.

The source code consists of a mnemonic MOVLW and operand 05A. This assembles into the hex code 305A, and is stored in binary in program memory as 11 0000 0101 1010. Since each hex digit represents four binary bits, the leading two bits are zero, and the leading digit will only range from 0 to 3 for a 14-bit number.

In the instruction set (data sheet, Table 13-2), it is seen that the first 4 bits (11 00) are the instruction code, the next two are unused (xx, appearing as 00 in the binary code) and the last 8 are the literal value (5A). The literal is represented as 'kkkk kkkk' since it can have any value from 00000000 to 11111111 (00–FF).

The format of other instructions depends mainly on the number of bits required for the operand (data to be processed). The number of op-code bits can vary from 3 to all 14, depending on the number of bits needed for the operand. This is different from a conventional processor, such as the Pentium, where the op-code and operand are each created as a whole number of bytes. The PIC instruction is more compact, as is the instruction set itself, for greater speed of operation. This defines it as a RISC (Reduced Instruction Set Computer) chip.

Program BIN4

The program BIN4 contains many of the basic program elements, and the list file (Program 1.2) shows the source code, machine code, memory address and list file line number as before. There are additional comments to aid program analysis and debugging.

Note that two types of labels are used in program to represent numbers. Label equates are used at the top of the program to declare labels for the file registers which will be used in the program. Address labels are placed in the first column to mark the destination for GOTO and CALL instructions.

Chip Configuration Word

In Program BIN4, the assembler directive __CONFIG is included at the top of the program, which sets up aspects of the chip operation which cannot be subsequently changed without reprogramming. A special area of program memory outside the normal range (address 2007h) stores a chip configuration word; the clock type, and other MCU options detailed below, are set by loading the configuration bits with a suitable binary code. The function of each bit is shown in Table 1.2, along with some typical configuration settings. Details can be found in the data sheet, Section 12.

CODE PROTECTION

Normally, the program machine code can be read back to the programming host computer, be disassembled and the original source program recovered. This can be prevented if commercial or security considerations require it. The

```
MPASM 03.00 Released            BIN4.ASM   8-28-2005  19:54:36         PAGE  1

LOC     OBJECT CODE         SOURCE TEXT
  VALUE               LINE

                      00001 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                      00002 ;
                      00003 ;       Source File:    BIN4.ASM
                      00004 ;       Author:         MPB
                      00005 ;       Date:           28-5-05
                      00006 ;
                      00007 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                      00008 ;
                      00009 ;       Slow output binary count is stopped, started
                      00010 ;       and reset with push buttons.
                      00011 ;
                      00012 ;       Processor:      PIC 16F877
                      00013 ;
                      00014 ;       Hardware:       PIC Demo System
                      00015 ;       Clock:          RC = 40kHz
                      00016 ;       Inputs:         Port D: Push Buttons
                      00017 ;                               RD0, RD1 (active low)
                      00018 ;       Outputs:        Port B: LEDs (active high)
                      00019 ;
                      00020 ;       WDTimer:        Disabled
                      00021 ;       PUTimer:        Enabled
                      00022 ;       Interrupts:     Disabled
                      00023 ;       Code Protect:   Disabled
                      00024 ;
                      00025 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                      00026
                      00027         PROCESSOR 16F877        ; Define MCU type
2007   3733           00028         __CONFIG 0x3733         ; Set config fuses
                      00029
                      00030 ; Register Label Equates....................................
                      00031
  00000006            00032 PORTB   EQU     06      ; Port B Data Register
  00000086            00033 TRISB   EQU     86      ; Port B Direction Register
  00000008            00034 PORTD   EQU     08      ; Port D Data Register
  00000020            00035 Timer   EQU     20      ; GPR used as delay counter
                      00036
                      00037 ; Input Bit Label Equates ..................................
                      00038
  00000000            00039 Inres   EQU     0       ; 'Reset' input button = RD0
  00000001            00040 Inrun   EQU     1       ; 'Run' input button = RD1
                      00041
                      00042 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                      00043
                      00044 ; Initialise Port B (Port A defaults to inputs)............
                      00045
0000   1683 1303      00046         BANKSEL TRISB           ; Select bank 1
0002   3000           00047         MOVLW   b'00000000'     ; Port B Direction Code
0003   0086           00048         MOVWF   TRISB           ; Load the DDR code into F86
0004   1283 1303      00049         BANKSEL PORTB           ; Select bank 0
0006   280B           00050         GOTO    reset           ; Jump to main loop
                      00051
                      00052
                      00053 ; 'delay' subroutine .......................................
                      00054
0007   00A0           00055 delay   MOVWF   Timer           ; Copy W to timer register
0008   0BA0           00056 down    DECFSZ  Timer           ; Decrement timer register
0009   2808           00057         GOTO    down            ; and repeat until zero
000A   0008           00058         RETURN                  ; Jump back to main program
                      00059
                      00060
                      00061 ; Start main loop ..........................................
                      00062
000B   0186           00063 reset   CLRF    PORTB           ; Clear Port B Data
                      00064
000C   1C08           00065 start   BTFSS   PORTD,Inres     ; Test reset button
000D   280B           00066         GOTO    reset           ; and reset Port B if pressed
000E   1888           00067         BTFSC   PORTD,Inrun     ; Test run button
000F   280C           00068         GOTO    start           ; and repeat if not pressed
                      00069
0010   0A86           00070         INCF    PORTB           ; Increment output at Port B
0011   30FF           00071         MOVLW   0FF             ; Delay count literal
0012   2007           00072         CALL    delay           ; Jump to subroutine 'delay'
0013   280C           00073         GOTO    start           ; Repeat main loop always
                      00074
                      00075         END                     ; Terminate source code
```

Program 1.2 BIN4 list file

Bit	Label	Function	Default	Enabled	Typical
15	–	None	0	x	0
14	–	None	0	x	0
13	CP1	Code protection	1	0	1
12	CP0	(4 levels)	1	0	1
11	DEBUG	In-circuit debugging (ICD)	1	0	0
10	–	None	1	x	1
9	WRT	Program memory write enable	1	1	1
8	CPD	EEPROM data memory write protect	1	0	1
7	LVP	Low-voltage programming enable	1	1	0
6	BODEN	Brown-out reset (BoR) enable	1	1	0
5	CP1	Code protection (CP)	1	0	1
4	CP0	(repeats)	1	0	1
3	PWRTE	Power-up timer (PuT) enable	1	0	0
2	WDTE	Watchdog timer (WdT) enable	1	1	0
1	FOSC1	Oscillator type select	1	x	0
0	FOSC0	RC = 11, HS = 10, XT = 01, LP = 00	1	x	1

Default = 3FFF (RC clock, PuT disabled, WdT enabled).
Typical RC clock = 3FF3 (RC clock, ICD disabled, PuT enabled, WdT disabled).
Typical XT clock = 3731 (XT clock, ICD enabled, PuT enabled, WdT disabled).

Table 1.2 Configuration bits

code protection bits (CP1:CP0) disable reads from selected program areas. Program memory may also be written from within the program itself, so that data tables or error checking data can be modified. Obviously, this needs some care, and this option can be disabled via the WRT bit. Data EEPROM may also be protected from external reads in the same way via the CPD bit, while internal read and write operations are still allowed, regardless of the state-of-the-code protection bits.

IN-CIRCUIT DEBUGGING

In-circuit debugging (ICD) allows the program to be downloaded after the chip has been fitted in the application circuit, and allows it to be tested with the real hardware. This is more useful than the previous method, which requires the chip to be programmed in a separate programmer unit before insertion in its socket on the board. With ICD, the chip can be programmed, and reprogrammed during debugging, while avoiding possible electrical and mechanical damage caused by removal from the circuit. The normal debugging techniques of single stepping, breakpoints and tracing can be applied in ICD mode. This allows a final stage of debugging in the prototype hardware, where problems with the interaction of the MCU with the real hardware can be resolved.

LOW VOLTAGE PROGRAMMING

Normally, when the chip is programmed, a high voltage (12–14 V) is applied to the PGM pin (RB3). To avoid the need to supply this voltage during in-circuit programming (e.g. during remote reprogramming), a low-voltage programming mode is available; however, using this option means that RB3 is not then available for general I/O functions during normal operation.

POWER-UP TIMER

When the supply power is applied to the programmed MCU, the start of program execution should be delayed until the power supply and clock are stable, otherwise the program may not run correctly. The power-up timer may therefore be enabled (PWRTE = 0) as a matter of routine. It avoids the need to reset the MCU manually at start up, or connect an external reset circuit, as is necessary with some microprocessors. An internal oscillator provides a delay between the power coming on and an internal MCU reset of about 72 ms. This is followed by an oscillator start up delay of 1024 cycles of the clock before program execution starts. At a clock frequency of 4 MHz, this works out to 256 µs.

BROWN-OUT RESET

Brown out refers to a short dip in the power-supply voltage, caused by mains supply fluctuation, or some other supply fault, which might disrupt the program execution. If the Brown-Out Detect Enable bit (BODEN) is set, a PSU glitch of longer than about 100 µs will cause the device to be held in reset until the supply recovers, and then wait for the power-up timer to time out, before restarting. The program must be designed to recover automatically.

WATCHDOG TIMER

The watchdog timer is designed to automatically reset the MCU if the program malfunctions, by stopping or getting stuck in loop. This could be caused by an undetected bug in the program, an unplanned sequence of inputs or supply fault. A separate internal oscillator and counter automatically generates a reset about every 18 ms, unless this is disabled in the configuration word. If the watchdog timer is enabled, it should be regularly reset by an instruction in the program loop (CLRWDT) to prevent the reset. If the program hangs, and the watchdog timer reset instruction not executed, the MCU will restart, and (possibly) continue correctly, depending on the nature of the fault.

RC OSCILLATOR

The MCU clock drives the program along, providing the timing signals for program execution. The RC (resistor–capacitor) clock is cheap and cheerful, requiring only these two inexpensive external components, operating with the internal clock driver circuit, to generate the clock. The time constant (product R × C) determines the clock period. A variable resistor can be used to give a manually adjustable frequency, although it is not very stable or accurate.

CRYSTAL OSCILLATOR

If greater precision is required, especially if the program uses the hardware timers to make accurate measurements or generate precise output signals, a crystal (XTAL) oscillator is needed. Normally, it is connected across the clock pins with a pair of small capacitors (15 pF) to stabilise the frequency. The crystal acts as a self-contained resonant circuit, where the quartz or ceramic crystal vibrates at a precise frequency when subject to electrical stimulation. The oscillator runs at a set frequency with a typical accuracy of better than 50 parts per million (ppm), which is equivalent to +/– 0.005%. A convenient value (used in our examples later) is 4 MHz; this gives an instruction cycle time of 1 μs, making timing calculations a little easier (each instruction takes four clock cycles). This is also the maximum frequency allowed for the XT configuration setting. The PIC 16FXXX series MCUs generally run at a maximum clock rate of 20 MHz, using a high-speed (HS) crystal which requires the selection of the HS configuration option.

CONFIGURATION SETTINGS

The default setting for the configuration bits is 3FFF, which means that the code protection is off, in-circuit debugging disabled, program write enabled, low-voltage programming enabled, brown-out reset enabled, power-up timer disabled, watchdog timer enabled and RC oscillator selected. A typical setting for basic development work would enable in-circuit debugging, enable the power-up timer for reliable starting, disable the watchdog timer and use the XT oscillator type.

By default, the watchdog timer is enabled. This produces an automatic reset at regular intervals, which will disrupt normal program operation. Therefore, this option will usually be disabled (bit 2 = 0). Conversely, it is generally desirable to enable the power-up timer, to minimise the possibility of a faulty start-up.

PIC Instruction Set

Each microcontroller family has its own set of instructions, which carry out essentially the same set of operations, but using different syntax. The PIC uses a minimal set of instructions, which makes it a good choice for learning.

A version of the PIC instruction set organised by functional groups is listed in Table 1.3. It consists of 35 separate instructions, some with alternate result destinations. The default destination for the result of an operation is the file

PIC INSTRUCTION SET

F = Any file register (specified by address or label), example is labelled GPR1
L = Literal value (follows instruction), example is labelled num1
W = Working register, W (default label)
Labels *Register labels must be declared in include file or by register label equate (e.g. GPR1 EQU 0C)*
Bit labels must be declared in include file or by bit label equate (e.g. bit1 EQU 3)
Address labels must be placed at the left margin of the source code file (e.g. start, delay)

Operation	Example	
Move		
Move data from F to W	`MOVF`	`GPR1,W`
Move data from W to F	`MOVWF`	`GPR1`
Move literal into W	`MOVLW`	`num1`
Test the register data	`MOVF`	`GPR1,F`
Register		
Clear W (reset all bits and value to 0)	`CLRW`	
Clear F (reset all bits and value to 0)	`CLRF`	`GPR1`
Decrement F (reduce by 1)	`DECF`	`GPR1`
Increment F (increase by 1)	`INCF`	`GPR1`
Swap the upper and lower four bits in F	`SWAPF`	`GPR1`
Complement F value (invert all bits)	`COMF`	`GPR1`
Rotate bits Left through carry flag	`RLF`	`GPR1`
Rotate bits Right through carry flag	`RRF`	`GPR1`
Clear (= 0) the bit specified	`BCF`	`GPR1,but1`
Set (= 1) the bit specified	`BSF`	`GPR1,but1`
Arithmetic		
Add W to F, with carry out	`ADDWF`	`GPR1`
Add F to W, with carry out	`ADDWF`	`GPR1,W`
Add L to W, with carry out	`ADDLW`	`num1`
Subtract W from F, using borrow	`SUBWF`	`GPR1`
Subtract W from F, placing result in W	`SUBWF`	`GPR1,W`
Subtract W from L, placing result in W	`SUBLW`	`num1`
Logic		
AND the bits of W and F, result in F	`ANDWF`	`GPR1`
AND the bits of W and F, result in W	`ANDWF`	`GPR1,W`
AND the bits of L and W, result in W	`ANDLW`	`num1`
OR the bits of W and F, result in F	`IORWF`	`GPR1`
OR the bits of W and F, result in W	`IORWF`	`GPR1,W`
OR the bits of L and W, result in W	`IORLW`	`num1`
Exclusive OR the bits of W and F, result in F	`XORWF`	`GPR1`
Exclusive OR the bits of W and F, result in W	`XORWF`	`GPR1,W`
Exclusive OR the bits of L and W	`XORLW`	`num1`
Test & Skip		
Test a bit in F and Skip next instruction if it is Clear (= 0)	`BTFSC`	`GPR1,but1`
Test a bit in F and Skip next instruction if it is Set (= 1)	`BTFSS`	`GPR1,but1`
Decrement F and Skip next instruction if F = 0	`DECFSZ`	`GPR1`
Increment F and Skip next instruction if F = 0	`INCFSZ`	`GPR1`
Jump		
Go to a labelled line in the program	`GOTO`	`start`
Jump to the label at the start of a subroutine	`CALL`	`delay`
Return at the end of a subroutine to the next instruction	`RETURN`	
Return at the end of a subroutine with L in W	`RETLW`	`num1`
Return From Interrupt service routine	`RETFIE`	
Control		
No Operation - delay for 1 cycle	`NOP`	
Go into standby mode to save power	`SLEEP`	
Clear watchdog timer to prevent automatic reset	`CLRWDT`	

Note 1: For MOVE instructions data is copied to the destination but retained in the source register.
Note 2: General Purpose Register 1, labelled ' GPR1', represents all file registers (00-4F). Literal value 'num1' represents all 8-bit values 00-FF. File register bits 0–7 are represented by the label 'but1'.
Note 3: The result of arithmetic and logic operations can generally be stored in W instead of the file register by adding, 'W' to the instruction. The full syntax for register operations with the result remaining in the file register F is ADDWF GPR1,F etc. F is the default destination, and W the alternative, so the instructions above are shortened to ADDWF, GPR1, etc. This will generate a message from the assembler that the default destination will be used.

Table 1.3 PIC instruction set by functional groups

register, but the working register W is sometimes an option. Each instruction is described in detail in the MCU data sheet, Section 13.

Instruction Types

The functional groups of instructions, and some points about how they work, are described below. The use of most of these instructions will be illustrated in due course within the demonstration programs for each type of interface.

MOVE

The contents of a register are copied to another. Notice that we cannot move a byte directly from one file register to another, it has to go via the working register. To put data into the system from the program (a literal) we must use MOVLW to place the literal into W initially. It can then be moved to another register as required.

The syntax is not symmetrical; to move a byte from W to a file register, MOVWF is used. To move it the other way, MOVF F,W is used, where F is any file register address. This means that MOVF F,F is also available. This may seem pointless, but in fact can be used to test a register without changing it.

REGISTER

Register operations affect only a single register, and all except CLRW (clear W) operate on file registers. Clear sets all bits to zero (00h), decrement decreases the value by 1 and increment increases it by 1. Swap exchanges the upper and lower four bits (nibbles). Complement inverts all the bits, which in effect negates the number. Rotate moves all bits left or right, including the carry flag in this process (see below for flags). Clear and set a bit operate on a selected bit, where the register and bit need to be specified in the instruction.

ARITHMETIC & LOGIC

Addition and subtraction in binary gives the same result as in decimal or hex. If the result generates an extra bit (e.g. FF + FF = 1FE), or requires a borrow (e.g. 1FE–FF = FF), the carry flag is used. Logic operations are carried out on bit pairs in two numbers to give the result which would be obtained if they were fed to the corresponding logic gate (e.g. 00001111 and 01010101 = 00000101). If necessary, reference should be made to an introductory text for further details of arithmetic and logical operations, and conversion between number systems. Some examples will be discussed later.

TEST, SKIP & JUMP

A mechanism is needed to make decisions (conditional program branches) which depend on some input condition or the result of a calculation. Programmed jumps are initiated using a bit test and conditional skip, followed by a GOTO or CALL. The bit test can be made on any file register bit. This could be a port bit, to check if an input has changed, or a status bit in a control register.

BTFSC (Bit Test and Skip if Clear) and BTFSS (Bit Test and Skip if Set) are used to test the bit and skip the next instruction, or not, according to the state of the bit tested. DECFSZ and INCFSZ embody a commonly used test – decrement or increment a register and jump depending on the effect of the result on the zero flag (Z is set if result = 0). Decrement is probably used more often (see BIN4 delay routine), but increment also works because when a register is incremented from the maximum value (FFh) it goes to zero (00h).

The bit test and skip may be followed by a single instruction to be carried out conditionally, but GOTO and CALL allow a block of conditional code. Using GOTO *label* simply transfers the program execution point to some other point in the program indicated by a label in the first column of the source code line, but CALL *label* means that the program returns to the instruction following the CALL when RETURN is encountered at the end of the subroutine.

Another option, which is useful for making program data tables, is RETLW (Return with Literal in W). See the KEYPAD program later for an example of this. RETFIE (Return From Interrupt) is explained below.

CONTROL

NOP simply does nothing for one instruction cycle (four clock cycles). This may seem pointless, but is in fact very useful for putting short delays in the program so that, for example, external hardware can be synchronised or a delay loop adjusted for an exact time interval. In the LCD driver program (Chapter 4), NOP is used to allow in-circuit debugging to be incorporated later when the program is downloaded, and to pad a timing loop so that it is exactly 1 ms.

SLEEP stops the program, such that it can be restarted with an external interrupt. It should also be used at the end of any program that does not loop back continuously, to prevent the program execution continuing into unused locations. The unused locations contain the code 3FFF (all 1 s), which is a valid instruction (ADDLW FF). If the program is not stopped, it will run through, repeating this instruction, and start again when the program counter rolls over to 0000.

CLRWDT means clear the watchdog timer. If the program gets stuck in a loop or stops for any other reason, it will be restarted automatically by the watchdog timer. To stop this happening when the program is operating normally, the watchdog timer must be reset at regular intervals of less than, say, 10 ms, within the program loop, using CLRWDT.

OPTIONAL INSTRUCTIONS

TRIS was an instruction originally provided to make port initialisation simpler (see program BIN1). It selects register bank 1 so that the TRIS data direction registers (TRISA, TRISB, etc.) can be loaded with a data direction code (0=output). The manufacturer no longer recommends use of this instruction, although it is still supported by the current assembler versions to maintain backward compatibility, and is useful when learning with very simple programs. The assembler directive BANKSEL can be used in more advanced programs, because it gives more flexible access to the registers in banks 1, 2, 3. It will be used here from Program BIN4 onwards. The other option is to change the bank select bits in the STATUS register direct, using BSF and BCF.

OPTION, providing special access to the OPTION register, is the other instruction, which is no longer recommended. It can be replaced by BANKSEL to select bank 1 which contains the OPTION register, which can then be accessed directly.

Program Execution

The PIC instruction contains both the op-code and operand. When the program executes, the instructions are copied to the instruction register in sequence, and the upper bits, containing the op-code, are decoded and used to set up the operation within the MCU. Figure 1.5, which illustrates the key hardware elements in this process, is derived from the system block diagram given in the data sheet.

The program counter keeps track of program execution; it clears to zero on power up or reset. With 8k of program memory, a count from 0000 to 1FFF (8191) is required (13 bits). The PCL (Program Counter Low) register (SFR 02) contains the low byte, and this can be read or written like any other file register. The high byte is only indirectly accessible via PCLATH (Program Counter Latch High, SFR 0Ah).

SUBROUTINES

Subroutines are used to create functional blocks of code, and provide good program structure. This makes it easier for the program to be understood, allows blocks of code to be re-used, and ultimately allows ready-made library routines to be created for future use. This saves on programming time and allows us to avoid ‘re-inventing the wheel’ when writing new applications.

A label is used at the start of the subroutine, which the assembler then replaces with the actual program memory address. When a subroutine is called, this destination address is copied into the program counter, and the program continues from the new address. At the same time, the return address (the one following the CALL) is pushed onto the stack, which is a block of memory dedicated to this purpose. In the PIC, there are 8 stack address storage levels, which are used in turn. The return addresses may thus be viewed as a stack of items, which must be added and removed in the same sequence.

The subroutine is terminated with a RETURN instruction, which causes the program to go back to the original position and continue. This is achieved by pulling the address from the top of the stack and replacing it in the program counter. It should be clear that CALL and RETURN must always be used in sequence to avoid a stack error, and a possible program crash. Conventional microprocessor systems often use general RAM as the stack, in which case it is possible to manipulate it directly. In the PIC, the stack is not directly accessible (Figure 1.7).

A delay subroutine is included in the program BIN4. The stack mechanism and program memory arrangement is shown in Figure 2.1 in the data sheet, and a somewhat simplified version is shown in Figure 1.6.

INTERRUPTS

The stack is also used when an interrupt is processed. This is effectively a call and return which is initiated by an external hardware signal which forces the processor to jump to a dedicated instruction sequence, an Interrupt Service Routine (ISR). For example, the MCU can be set up so that when a hardware timer times out (finishes its count), the process required at that time is called via a timer interrupt.

When an interrupt signal is received, the current instruction is completed and the address of the next instruction (the return address) is pushed into the first available stack location. The ISR is terminated with the instruction RETFIE (return from interrupt), which causes the return address to be pulled from the stack. Program execution then restarts at the original location. However, remember to take into account any changes in the registers which may have happened in the ISR. If necessary, the registers must be saved at the beginning of the ISR, and restored at the end, in spare set of file registers. A simple example using a timer interrupt is seen later in a test program which generates a pulse output.

PAGE BOUNDARIES

In normal program execution, the operation of the program counter is automatic, but there are potential problems when a program branch occurs. Jump instructions (CALL or GOTO) provide only an 11-bit destination address, so the program

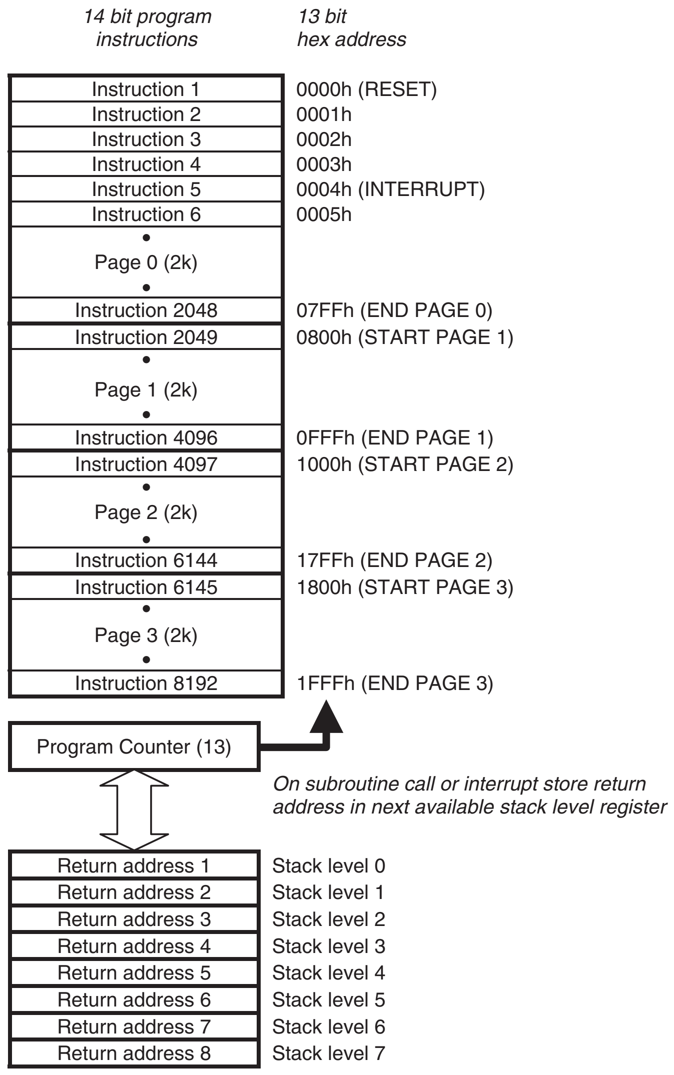


Figure 1.7 P16F877 program memory and stack

memory is effectively divided into four 2k blocks, or pages. A jump across the program memory page boundary requires the page selection bits (PCLATH 4:3) to be modified by the user program. In addition, if the 8-bit PCL is modified directly, as in table read, care must be taken if a jump is made from one 256-byte block to another; PCLATH again may need to be modified explicitly. Sections 2.3 and 2.4 in the 16F877 data sheet detail how to handle these problems.

Special Function Registers

As we have seen, the file register set is divided into special function registers and general purpose registers. The SFRs have predetermined functions, as specified in the 16F877 data sheet (Figure 2-3), and occupy locations 00-1F in bank 0, 80-9F in bank 1, 100-10F in bank 2 and 180-18F in bank 3. Many are repeated in more than one bank. Their functions will be explained below in order of significance.

Program Counter (PCL)

The function the program counter has been described above, under program execution. PCL contains the low 8 bits of the program counter, while the upper bits (PC<8–12>) are accessed via PCLATH. It is incremented during each instruction, and the contents replaced during a GOTO, CALL (program address) or RETURN (stack).

Status Register

The status register records the result of certain operations, MCU power status and includes the bank selection bits. The bit functions are detailed in the Table Register 2-1 in the data sheet.

ZERO FLAG (Z)

This is set when the result of a register operation is zero, and cleared when it is not zero. The full instruction set must be consulted to confirm which operations affect the Z flag. Bit test and skip instructions use this flag for conditional branching, but remember that there are dedicated instructions for decrement or increment and skip if zero. Curiously, these do not affect the zero flag itself. A typical use of the zero flag is to check if two numbers are the same by subtracting and applying bit test and skip to the Z bit.

CARRY FLAG (C)

This flag is only affected by add, subtract and rotate instructions. If the result of an add operation generates a carry out, this flag is set; that is, when two 8-bit numbers give a 9-bit sum. The carry bit must then be included in subsequent calculations to give the right result. When subtracting, the carry flag must be set initially, because it provides the borrow digit (if required) in the most significant bit of the result. If the carry flag is cleared after a subtraction, it means the result was negative, because the number being subtracted was the larger. An example of this is seen later in the calculator program.

Taken together, the zero and carry flags allow the result of an arithmetic operation to be detected as positive, negative or zero, as shown in Table 1.4.

Flag after Operation	Zero (Z)	Carry (C)	Result	Comment
ADD	0	0	A+B<256	8-bit sum, no carry
A+B	1	1	A+B=256 (100h)	Exactly, carry out
	0	1	A+B>256	9-bit sum, carry out
SUB	0	1*	A–B<256	8-bit difference, no borrow
A–B	1	1*	A–B=0	Numbers equal, no borrow
	0	0*	A–B<0	Borrow taken, result negative

*Set carry flag before subtracting.

Table 1.4 Arithmetic results

Again, remember that the carry flag must be set before a subtraction operation, so that a borrow can be detected as C = 0.

DIGIT CARRY (DC)

A file register can be seen as containing 8 individual bits, or 1 byte. It can also be used as 2 × 4-bit nibbles (a small byte!). Each nibble can be represented as 1 hex digit (0-F). The digit carry records a carry from the most significant bit of the low nibble (bit 3). Hence the digit carry allows 4-bit hexadecimal arithmetic to be carried out in the same way as 8-bit binary arithmetic uses the carry flag C.

REGISTER BANK SELECT (RP1:RP0)

The PIC 16F877 file register RAM is divided into four banks of 128 locations, banks 0–3 (Figure 2-3 in data sheet). At power on reset, bank 0 is selected by default. To access the others, these register bank select bits must be changed, as shown in Table 1.5.

It can be seen that some registers repeat in more than one bank, making it easier and quicker to access them when switched to that bank. For example, the status register repeats in all banks. In addition, a block of GPRs at the end of each bank repeat, so that their data contents are available without changing banks.

The register banks are selected by setting and clearing the bits RP0 and RP1 in the status register. More conveniently, the pseudo-operation BANKSEL can be used instead. The operand for BANKSEL is any register in that bank, or its label. In effect, BANKSEL detects the bank bits in the register address and copies them to the status register bank select bits.

RP1	RP0	Bank	Address	Total	Function
0	0	0	00 – 20	32	Special function registers
			20 – 7F	96	General purpose registers
0	1	1	80 – 9F	32	SFRs, some repeat
			A0 – EF	80	GPRs
			F0 – FF	16	Repeat 70-7F
1	0	2	100 – 10F	16	SFRs, some repeat
			110 – 16F	96	GPRs
			170 – 17F	16	Repeat 70-7F
1	1	3	180 – 18F	16	SFRs, some repeat
			190 – 1EF	96	GPRs
			1F0 – 1FF	16	Repeat 70-7F
			000 – 1FF	96	SFRs
				368	GPRs

Table 1.5 Register bank select

POWER STATUS BITS

There are two read only bits in the status register which indicate the overall MCU status. The Power Down (PD) bit is clear to zero when SLEEP mode is entered. The Time Out (TO) bit is cleared when a watchdog time out has occurred.

Ports

There are five parallel ports in the PIC 16F877, labelled A–E. All pins can be used as bit- or byte-oriented digital input or output. Their alternate functions are summarised in Table 1.6.

It can be seen that many of the port pins have two or more functions, depending on the initialisation of the relevant control registers. On power up or reset, the port control register bits adopt a default condition (see Table 2-1 in the data sheet, right hand columns). The TRIS (data direction) register bits in bank 1 default to 1, setting the ports B, C and D as inputs. If this is as required, no further initialisation is needed, since other relevant control registers are generally reset to provide plain digital I/O by default.

However, there is an IMPORTANT exception. Ports A and E are set to ANALOGUE INPUT by default, because the analogue control register ADCON1 in bank 1 defaults to 0 - - - 0000. To set up these ports for digital I/O, this register must be loaded with the code x - - - 011x (x = don’t care), say 06h. If analogue input is required only on selected pins, ADCON1 can be initialised with bit codes that give a mixture of analogue and digital I/O on

	Bits	Pins	Alternate function/s	Bit	Default
Port A	6	RA0–RA5	Analogue inputs	0,1,2,3,5	Analogue
			Timer0 clock input	4	Input
			Serial port slave select input	5	
Port B	8	RB0–RB7	External interrupt	0	Digital
			Low-voltage programming input	3	I/O
			Serial programming	6,7	
			In-circuit debugging	6,7	
Port C	8	RC0–RC7	Timer1 clock input/output	0,1	Digital
			Capture/Compare/PWM	1,2	I/O
			SPI, I^{2}C synchronous clock/data	3,4,5	
			USART asynchronous clock/data	6,7	
Port D	8	RD0–RD7	Parallel slave port data I/O	0–7	Digital I/O
Port E	3	RE0–RE2	Analogue inputs	0,1,2	Analogue
			Parallel slave port control bits	0,1,2	Input

Table 1.6 Port alternate functions

Ports A and E. Note that ADCON1 is in bank 1 so BANKSEL is needed to access it. Initialisation for analogue I/O will be explained in more detail later.

Timers

The PIC 16F877 has three hardware timers (data sheet, Sections 5, 6 and 7). These are used to carry out timing operations simultaneously with the program, to make the program faster and more efficient. An example would be generating a pulse every second at an output.

Timer0 uses an 8-bit register, TMR0, file register address 01. Its output is an overflow flag, T0IF, bit 2 in the Interrupt Control Register INTCON, address 0B. The timer register is incremented via a clock input which is derived either from the MCU oscillator (f_{OSC}) or an external pulse train at RA4. The register counts from 0 to 255d in binary, and then rolls over to 00 again. When the register goes from FF to 00, T0IF is set.

If the internal clock is used, the register acts as a timer. Each instruction in the MCU takes four clock cycles to execute, so the instruction clock is $f_{OSC}/4$. The timers are driven from the instruction clock, which can be monitored externally at CLKOUT, if the chip is operating with an RC clock. If preloaded with a value of say, 155d, TMR0 will count 100 clock pulses until T0IF is set. If the chip is driven from a crystal of 4 MHz, the instruction clock will be 1 MHz, and the timer will overflow after 100 μs. If this were used to toggle an output, a signal with a period of exactly $2 \times 100 = 200$ μs (frequency = 5 kHz) would be obtained.

Alternatively, a count of external pulses can be made, and read from the register when finished, or the read triggered by external signal. Thus, the timers can also be used as counters. Figure 5-1 in the data sheet shows the full block diagram of Timer0, which shows a pre-scale register and the watchdog timer. The pre-scaler is a divide by N register, where $N = 2, 4, 8, 16, 32, 64, 128$ or 256, meaning that the output count rate is reduced by this factor. This extends the count period or total count by the same ratio, giving a greater range to the measurement. The watchdog timer interval can also be extended, if this is selected as the clock source. The pre-scale select bits, and other control bits for Timer0 are found in OPTION_REG. Some typical Timer0 configurations are detailed in Table 1.7.

Timer1 is a 16-bit counter, consisting of TMR1H and TMR1L (0E AND 0F). When the low byte rolls over from FF to 00, the high byte is incremented. The maximum count is therefore 65535d, which allows a higher count without sacrificing accuracy.

Timer2 is an 8-bit counter (TMR2) with a 4-bit pre-scaler, 4-bit post-scaler and a comparator. It can be used to generate Pulse Width Modulated (PWM) output which is useful for driving DC motors and servos, among other things (see the data sheet, section 7, for more details). These timers can also be used in capture and compare modes, which allow external signals to be more easily measured. There will be further detail provided with demonstration programs on timed I/O.

OPTION_REG	Configuration	Effect	Applications
11**0**10**000** *Active bits in bold*	Internal clock ($f_{OSC}/4$) No pre-scale	Timer mode using instruction clock	1. Preload Timer0 with initial value, and count up to 256 2. Clear Timer0 initially and read count later to measure time elapsed
11**0**10**011**	Internal clock ($f_{OSC}/4$) Pre-scale = 16	Timer mode using instruction clock with pre-scale	Extend the count period × 16 for applications 1 and 2
11**110111**	External clock T0CKI pin	Counter mode Pre-scale = 256	Count one pulse in 256 at RA4
1111**1110**	Watchdog timer selected pre-scale = 64	Extend watchdog reset period to $18 \times 64 = 1152$ ms	Watchdog timer checks program every second

Table 1.7 Typical configurations for Timer0

Indirect File Register Addressing

File register 00 (INDF) is used for indirect file register addressing. The address of the register is placed in the file select register (FSR). When data is written to or read from INDF, it is actually written to or read from the file register pointed to by FSR. This is most useful for carrying out a read or write on a continuous block of GPRs, for example, when saving data being read in from a port over a period of time. Since nine bits are needed to address all file registers (000–1FF), the IRP bit in the status register is used as the extra bit. Direct and indirect addressing of the file registers are compared in the data sheet (Figure 2–6).

Interrupt Control Registers

The registers involved in interrupt handling are INTCON, PIR1, PIR2, PIE1, PIE2 and PCON. Interrupts are external hardware signals which force the MCU to suspend its current process, and carry out an Interrupt Service Routine (ISR). An interrupt can be generated in various ways, but, in the PIC, the result is always to jump to program address 004. If more than one interrupt source is operational, then the source of the interrupt must be detected and the corresponding ISR selected.

By default, interrupts are disabled, so programs can be loaded with their origin (first instruction) at address 0000, and the significance of address 0004 can be ignored. If interrupts are to be used, the main program start address needs to be 0005, or higher, and a ‘GOTO start’ (or similar label) placed at address 0000. A ‘GOTO ISR’ instruction can then be placed at 004, using the ORG directive, which sets the address at which the instruction will be placed by the assembler.

The Global Interrupt Enable bit (INTCON, GIE) must be set to enable the interrupt system. The individual interrupt source is then enabled. For example, the bit INTCON, T0IE is set to enable the Timer0 overflow to trigger the interrupt sequence. When the timer overflows, INTCON, T0IF (Timer0 Interrupt Flag) is set to indicate the interrupt source, and the ISR called. The flags can be checked by the ISR to establish the source of the interrupt, if more than one is enabled. A list of interrupt sources and their control bits is given in Table 1.8.

The primary interrupt sources are Timer0 and Port B. Input RB0 is used for single interrupts, and pins RB4–RB7 can be set up so that any change on these inputs initiates the interrupt. This could be used to detect when a button on a keypad connected to Port B has been pressed, and the ISR would then process the input accordingly.

The remaining interrupt sources are enabled by the Peripheral Interrupt Enable bit (INTCON, PEIE). These are then individually enabled and flagged

Source	Enable Bit Set	Flag Bit Set	Interrupt Trigger Event
TMR0	INTCON,5	INTCON,2	Timer0 count overflowed
RB0	INTCON,4	INTCON,1	RB0 input changed (also uses INTEDG)
RB4-7	INTCON,3	INTCON,0	Port B high nibble input changed
Peripherals	INTCON,6		
TMR1	PIE1,0	PIR1,0	Timer1 count overflowed
TMR2	PIE1,1	PIR1,1	Timer2 count matched period register PR2
CCP1	PIE1,2	PIR1,2	Timer1 count captured in or matched CCPR1
SSP	PIE1,3	PIR1,3	Data transmitted or received in Synchronous Serial Port
TX	PIE1,4	PIR1,4	Transmit buffer empty in Asynchronous Serial Port
RC	PIE1,5	PIR1,5	Receive buffer full in Asynchronous Serial Port
AD	PIE1,6	PIR1,6	Analogue to Digital Conversion completed
PSP	PIE1,7	PIR1,7	A read or write has occurred in the Parallel Slave Port
CCP2	PIE2,0	PIR2,0 CCPR2	Timer2 count captured in or matched
BCL	PIE2,3	PIR2,3	Bus collision detected in SSP (I^2C mode)
EE	PIE2,4	PIR2,4	Write to EEPROM memory completed

Table 1.8 Interrupt sources and control bits

in PIE1, PIE2, PIR1 and PIR2. Many of these peripherals will be examined in more detail later, but the demonstration programs do not generally use interrupts, to keep them as simple as possible. However, if these peripherals are used in more complex programs where multiple processes are required, interrupts are useful. The program designer then has to decide on interrupt priority. This means selectively disabling lower priority interrupts, using the enable bits, when a more important process is in progress. For example, when reading a serial port, the data has to be picked up from the port before being overwritten by the next data to arrive.

In more complex processors, a more sophisticated interrupt priority system may be available, so that interrupts have to be placed in order of priority, and those of a lower priority automatically disabled during a high-priority process. The limited stack depth (8 return addresses) in the PIC must also be taken into account, especially if several levels of subroutine are implemented as well as multiple interrupts.

Peripheral Control Registers

The function of most of the peripheral blocks and their set up will be explained as each is examined in turn, with a sample program.

The only peripheral which does not require external connections is the Electrically Erasable Programmable Read Only Memory (EEPROM). This is a block of non-volatile read and write memory which stores data during power down; for example, a security code or combination for an electronic lock. A set of registers in banks 2 and 3 are used to access this memory as well as a special EEPROM write sequence designed to prevent accidental overwriting of the secure data. See Section 4 of the data sheet for details.

SUMMARY 1

- The microcontroller contains a processor, memory and input/output devices
- The program is stored in ROM memory in numbered locations (addresses)
- The P16F877 stores a maximum of 8k × 14 instructions in flash ROM
- The P16FXXX family uses only 35 instructions
- The P16F877 has 368 bytes of RAM and 5 ports (33 I/O pins)
- The ports act as buffers between the MCU and external systems
- The program is executed in sequence, unless there is a jump instruction
- The program counter tracks the current instruction address
- A configuration word is needed to select the clock type and other chip options
- The program source code (.ASM) is assembled into machine code (.HEX)
- Subroutines are used to structure the program
- Interrupts are used to give I/O processing priority
- Hardware timers operate simultaneously with the program
- The serial ports require parallel to serial data conversion

ASSESSMENT 1

1 State the three main elements in any microprocessor system. *(3)*

2 State the difference between a microprocessor and microcontroller. *(3)*

3 Describe briefly the process of fetching an instruction. *(3)*

4 State the advantages of flash ROM, compared with other memory types. *(3)*

5 Explain why serial data communication is generally slower than parallel. *(3)*

6 State why Ports A and E in the PIC 16F877 cannot be used for digital input without initialisation. *(3)*

7 How many bits does the 8k MCU program memory contain? *(3)*

8 Outline the process of installing a program in the PIC MCU. *(3)*

9 Explain the function of each bit in the binary code for the instruction 'MOVWF 0C'. *(3)*

10 Work out the configuration word required to initialise the P16F877 as follows: RC clock, ICD enabled, PuT on, WDT off, BoD on, code protection off. *(3)*

11 Briefly compare the operation of a subroutine and an interrupt, explaining the role of the stack, return address, interrupt flag and the special significance of address 004 in the P16XXX. *(5)*

12 Explain how conditional program jumps are implemented in the PIC MCU. *(5)*

ASSIGNMENTS 1

1.1 Program Execution

Describe the process of program instruction execution in a P16F877 MCU by reference to its block diagram in the data sheet. Explain the role of each block, and how the instructions and data is moved around. Use the instructions MOVLW XX, ADDWF XX and CALL XXX as examples to explain the execution sequence and the nature of the operands of these instructions.

1.2 Instruction Analysis

Analyse the machine code for program BIN1. List the instructions in binary and explain the function of each individual bit, or group of bits, within each instruction, by reference to the instruction set summary in the data sheet. In particular, identify the operand bits, and the operand type (literal, file address or program address).

1.3 Led Scanner

Study program BIN4. Suggest how the main loop could be modified to light the least significant LED, and then rotate it through each bit so that the lit LED appears to scan. Explain what will happen after it reaches the last position, referring to role of the carry flag in the rotate command. Suggest how the scan direction can be reversed at the end of the row of LEDs.

2

PIC Software

The PIC microcontroller architecture has been introduced in Chapter 1, so we now turn to the software, or firmware, as it should more accurately be known, since it is stored in non-volatile memory. The source code is written on a PC host using a text editor, or the edit window in MPLAB (the standard development system), assembled and downloaded to the chip. The entry-level development system hardware normally used until recently is shown in Figure 2.1 (a). It consists of a host PC and programming unit, connected via a serial link.

The PC is running MPLAB, and when the program has been written and assembled, it is downloaded by placing the chip in programming unit connected to a PC COM (RS232) port. The RS232 protocol, the simplest serial data format, will be described later since it is available as a serial port on the PIC 16F877 itself. The chip is programmed via pins RB6 (clock) and RB7 (data).

The programming unit supplied by Microchip is called PICSTART Plus. It has a zero insertion force (ZIF) socket in which the chip is placed, and contains another PIC within to handle the programming. The firmware in the programmer control chip can be updated in line with updates of the development system itself, since new MCU types are added continuously to the range. Third party-companies also produce inexpensive programming modules, some of which are supplied in a kit form to further reduce the cost.

A more recent, and versatile, method of program downloading uses ICD (In-Circuit Debugging) mode (Figure 2.1 (b)). Here, the PC is connected to an ICD module, which controls communication directly with the target chip. The

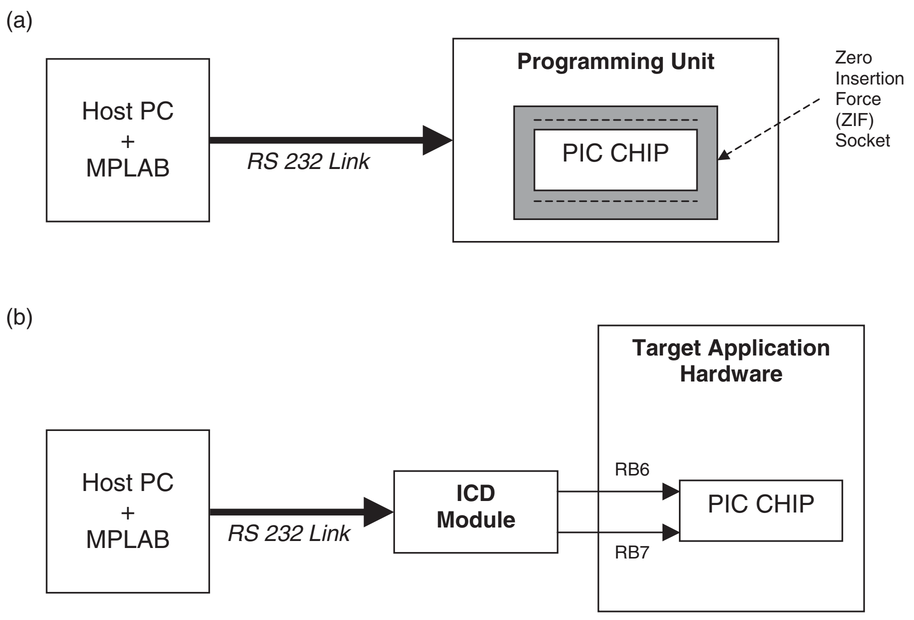
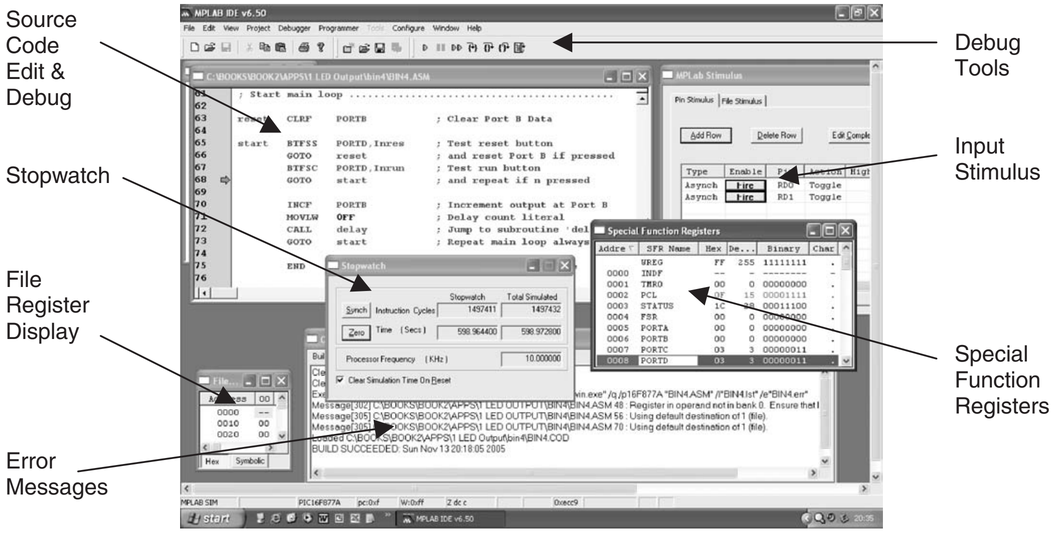


Figure 2.1 PIC development systems: (a) programming unit; (b) ICD system



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

reset   CLRF    PORTB           ; Clear Port B Data

start   BTFSS   PORTD,Inres     ; Test reset button
        GOTO    reset           ; and reset Port B if pressed
        BTFSC   PORTD,Inrun     ; Test run button
        GOTO    start           ; and repeat if n pressed

        INCF    PORTB           ; Increment output at Port B
        MOVLW   OFF             ; Delay count literal
        CALL    delay           ; Jump to subroutine 'del
        GOTO    start           ; Repeat main loop always

        END
```

Type	Enable	Pi	Action	High
Asynch	Fire	RD0	Toggle	
Asynch	Fire	RD1	Toggle	

	Stopwatch	Total Simulated
Instruction Cycles	1497411	1497432
Time (Secs)	598.964400	598.972800

Processor Frequency (KHz) 10.000000

☑ Clear Simulation Time On Reset

Addre	SFR Name	Hex	De...	Binary	Char
	WREG	FF	255	11111111	.
0000	INDF	--	-	--------	-
0001	TMR0	00	0	00000000	.
0002	PCL	0F	15	00001111	.
0003	STATUS	1C	28	00011100	.
0004	FSR	00	0	00000000	.
0005	PORTA	00	0	00000000	.
0006	PORTB	00	0	00000000	.
0007	PORTC	03	3	00000011	.
0008	PORTD	03	3	00000011	.

Address	00
0000	--
0010	00
0020	00

```
win.exe" /q /p16F877A "BIN4.ASM" /l"BIN4.lst" /e"BIN4.err"
Message[302] C:\BOOKS\BOOK2\APPS\1 LED OUTPUT\BIN4\BIN4.ASM 48 : Register in operand not in bank 0.  Ensure that I
Message[305] C:\BOOKS\BOOK2\APPS\1 LED OUTPUT\BIN4\BIN4.ASM 56 : Using default destination of 1 (file).
Message[305] C:\BOOKS\BOOK2\APPS\1 LED OUTPUT\BIN4\BIN4.ASM 70 : Using default destination of 1 (file).
Loaded C:\BOOKS\BOOK2\APPS\1 LED Output\bin4\BIN4.COD
BUILD SUCCEEDED: Sun Nov 13 20:18:05 2005
```

```
MPLAB SIM    PIC16F877A    pc:0xf    W:0xff    Z dc c    0xecc9
```

Figure 2.2 Screenshot of MPLAB

MCU is left in circuit at all times, which minimises the possibility of damage, and is reprogrammed via the same pins as before in Port B. However, the chip can now be programmed to run in ICD mode; this means, program execution in the chip itself can be controlled from MPLAB. The program can be stopped, started and single stepped in the same way as in software simulation mode. The program can thus be tested within the target hardware, a much more realistic and powerful option.

MPLAB can be downloaded free from www.microchip.com (development tools). The essential elements are an editor, assembler, simulator and programmer. The interface is shown in a typical configuration in the screenshot (Figure 2.2).

Assembly Language

MPLAB has an integrated text editor designed for entering PIC programs. An assembly language program can be entered using the instruction set defined for the MCU chosen, plus any assembler directives required.

Assembler Code

The general syntax requirements for the typical PIC program have already been seen in Program 1.2 (BIN4). The source code occupies the columns to the right of the line numbers in the list file (Column 2). The columns to the left of the line numbers are the machine code, and the memory locations at which each code is stored. A typical instruction is analysed in Table 2.1.

The machine code instruction provides the information to the execution unit of the MCU to carry out the required operation (move, calculate, test, etc.). It could be entered in plain binary, but this would require us to look up the code

List File Column	Example Content	Meaning
0	000C	Memory location at which machine code instruction is stored
1	1C08	Machine code instruction, including op-code and operands
2	00065	Source code line number
3	start	Address label marking jump destination
4	BTFSS	Instruction mnemonic
5	PORTD,Inres	Instruction operand labels
6	;Test reset button	Comment delimited by semi-colon

Table 2.1 List file elements

each time. A more user-friendly option is the instruction mnemonic. Labels are used to represent the op-code and operands, and these are replaced by the assembler program with the corresponding binary codes, to produce the machine code program (hex file). Our sample instruction is further dissected in Table 2.2.

It can be seen that the 14-bit instruction code can be broken into three functional parts. The first two bits are always zero, the unused bits 14 and 15. The next four give the code for BTFSS, the next three identify the bit to be tested (0) and the last seven the file register address (08). This structure can be seen in the instruction set in the data sheet (Table 13-2).

Other instructions do not necessarily have all these operands. CLRW has no operands, and CLRF has one, the file register address. In most of the byte-oriented file register operations, the destination for the result is switched to W by setting bit 7 = 0.

Note that an unused memory location normally contains all 1s (3FFF). However, in the instruction set code 3FFF = ADDLW 0FF, meaning add the literal FF to W. Blank locations will repeat this operation until the program counter rolls over to zero and the program restarts! Therefore, if the program does not loop continuously, it should be terminated with a SLEEP instruction to stop the program at that point. Note that code 0000 = NOP, no operation.

In the official instruction set in the data sheet, the elements are identified as follows:

f = file register (00-7F)

d = destination (1=file register, 0=W)

k = literal (00-FF)

b = bit number (0-7)

Note also that the effect on the flags of each instruction is given.

Label	Hex	Binary	Meaning	Range
`start`	`000C`	`0000 0000 0000 1100`	Program memory address label	`0000` – `1FFF(8K)`
`BTFSS`		`00 01 11-- ---- ----`	Op-code (bits 13,12,11,10 only)	-
`PORTD`	`1C08`	`-- -- ---- -000 1000`	File register address = 08	`00 – 7F (128)`
`Inres`		`-- -- --00 0--- ----`	File register bit = 0	`0 – 7`

Table 2.2 Instruction analysis

The assembly language program is therefore written using pre-defined labels for the instruction (mnemonics) with user-defined labels for destination addresses, registers, bits and literals. In general, any number that appears in the program can be replaced by a label.

Assembler Syntax

The essentials of program syntax (grammar and spelling) are also illustrated in BIN4. The instructions themselves are placed in the second and third columns; here they are separated by a tab for clarity, but a space is also acceptable to the assembler. The line comments are delimited by a semicolon. The comment should describe the effect of the instruction, rather than simply repeating the meaning of the mnemonics and labels.

In MPLAB, the line numbering can be switched on in the source code edit window. When creating a source code file, always 'Save As..' immediately the edit window has been opened, to make sure that the file path is established. Note that MPLAB has a limit on the number of characters in the file path, so you may not be able to save in the MPLAB folder itself. In any case, we will be accessing the files from the interactive simulation software as well, so a directory near the root of C: is preferable. Do not forget to keep backup versions of your work on a network, USB flash ROM or other available drive!

A large comment block at the top of the program is desirable, but this should be in proportion to the complexity of the program. The source code file name, author and date or version number are essential. A program description should then follow, and details of the MCU and its configuration. Details of the target hardware should also be included, especially the I/O usage.

This is then followed by the MCU selection and configuration word. These are assembler directives, and are not converted to machine code, as can be seen in the left hand column of the list file. The PROCESSOR directive tells the assembler which MCU will be used, because there is some variation between them; most obviously, the number of valid port addresses. The 18FXXX series of chips also have a more extensive instruction set.

The __CONFIG directive sets the programmable fuses in the chip, which cannot be changed except by reprogramming. These have been described in detail in Chapter 1, but include the options listed in the program header: clock type, code protection and watchdog and power-up timer enable. The configuration code (3733h) and its location (2007h) appear in the left column of the list file.

Another commonly required directive is EQU. This allows any number in the program to be represented by a label, notifying the assembler that the label given should be replaced by the number to which it is equated. File register (PORTB, TRISB, PORTD, Timer), bit numbers (Inrun, Inres) and literals may be represented in this way. The literal b'00000000' in the port initialisation block could also have been replaced by a suitable label by adding this to the label equates. However, the only directive that is absolutely essential is END, to indicate the end of the source code to the assembler.

By contrast, program memory address labels are declared implicitly by their placement at the beginning of the relevant source code line (delay, down, reset, start). Short labels are used here so they fit into an 8-character column in the source code. Longer labels (e.g. start_of_main_program) may be used, in which case the label can be one line and the associated code on the next; the line return is not significant to the assembler. However, no spaces should be left in the label, as these are significant. Avoid characters other than letters, numbers and underscore in labels.

Layout and Structure

The source code shown in BIN4 is organised in blocks, to make it easier to understand. Good layout and readability are important as it is quite possible that another software engineer will want to repair, modify or otherwise update your program. As much information as possible should be included to assist in this process, not always a priority when software production schedules are tight! However, it will be worthwhile in the long term.

Each block is described in a full line block comment, and each instruction explained in a line comment. Functional parts of the code are separated into the following blocks:

1. Header comment
2. MCU configuration
3. Label equates
4. Port initialisation
5. Subroutines/macros
6. Main program

The subroutines are placed before the main program so that the destination address labels are declared before being encountered as operands in subsequent blocks, which could give rise to a syntax error 'label not recognised' or similar. In fact, in two pass assemblers, the problem does not arise, since the labels are established before the final assembly of the object code. Therefore, if you wish to write the main block first, and follow it with the component subroutines, that is not a problem, and possibly more logical.

The use of subroutines encourages structured programming, where distinct operations are created in separate blocks and then called as necessary. This provides re-usable code: a subroutine may be called as many times as required, but only needs to be written once.

These blocks can be included as separate source code files, or as pre-compiled blocks, as is the case if a higher-level language such as ‘C’ is used (see below).

Standard special function register names can also be included in the form of a standard header file that is supplied with MPLAB for each PIC chip. This is called P16F877A.INC for our examples here; it contains a complete list of the SFRs and their addresses. These standard names then have to be used, providing a standard labelling system, which aids communication between developers. These labels are all in upper case e.g. PORTA. The use of the standard SFR label file is recommended for all PIC programs. Incidentally, the ‘A’ designation at the end of the PIC chip number indicates that the maximum clock rate is increased from 10 to 20 MHz.

Macros, Special Instructions, Assembler Directives

Another structured programming technique is the macro. This is similar to the subroutine, but the block of code is pre-defined, and then inserted as source code whenever required. This increases the source code size, but reduces the program execution time by eliminating the time taken to jump to the subroutine, and back, when CALL and RETURN are executed. Each of these takes an extra clock cycle, which may be significant in high-speed applications.

There are also special or supplementary instructions recognised by the assembler. Essentially, this is a macro, which is predefined within the assembler, so providing extra instructions on the basic set. There is a collection of these listed in the assembler help file supplied with MPLAB (HelpFiles\hlpMPASMAsm). Selected examples are given in Table 2.3.

Assembler directives are also provided in addition to the instruction set to improve the efficiency and flexibility of the programming process for experienced developer. One is essential (END), a few others are usually necessary, but most are optional at this stage. Some of the more commonly used directives are listed in Table 2.4.

Program BIN4 source code can now be rewritten using some of these directives (BIN4D) to illustrate their use (Program 2.1).

Note the following features of the BIN4D list file:

- List file options allow unwanted elements of printout to be suppressed.
- Configuration settings are detailed in Chapter 1.

S. Instruction	Meaning	Assembler Code
`BZ addlab`	Branch to destination (address label) if result of previous operation zero	`BTFSC STATUS,Z` `GOTO addlab`
`BNZ addlab`	Branch to destination (address label) if result of previous operation not zero	`BTFSS STATUS,Z` `GOTO addlab`
`BC addlab`	Branch to destination (address label) if carry set	`BTFSC STATUS,C` `GOTO addlab`
`BNC addlab`	Branch to destination (address label) if carry not set	`BTFSS STATUS,C` `GOTO addlab`
`NEG num1`	Negate (2s complement) a file register (labelled num1)	`COMF num1` `INCF num1`
`TSTF num1`	Test a file register (labelled num1) to modify status bits	`MOVF num1`

Table 2.3 Supplementary instructions

Directive Example	Meaning
`LIST p=16f877a, w=2, st=off`	Listing options: e.g. select MCU, print errors only, no symbol table
`ORG 05`	Set the first program memory address for the code that follows
`END`	End of program source code
`PORTA EQU 05`	Declare a label (assembler constant)
`Max SET 200`	Declare a label value which may be changed later (assembler variable)
`PROCESSOR 16F877A`	Select the MCU type
`CONSTANT Hours_in_day=24`	Declare a constant
`__CONFIG 0x3731`	Set processor configuration word
`BANKSEL TRISC`	Select file register bank containing the register specified
`#INCLUDE "C:\PIC\P16F877A.INC"`	Include additional source file from directory specified
`#DEFINE Cflag 3,0`	Substitute text 'Cflag' with '3,0' e.g. 'BSF Cflag'
`MACRO pulse`	Declare macro definition with address label
`ENDM`	End a macro definition
`NOEXPAND`	Do not print macro each time used

Table 2.4 Selected assembler directives

- Equate is used for file register addresses.
- Data Direction code is SET, and may be changed later.
- Timer count value is defined as a constant.
- Text substitution is used for input bits.
- File path for standard P16F877A include file is specified in double quotes.

```
MPASM 03.00 Released            BIN4D.ASM   11-13-2005  19:43:31         PAGE  1

LOC  OBJECT CODE     LINE SOURCE TEXT
  VALUE
                      00001 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                      00002 ;
                      00003 ;       Source File:    BIN4D.ASM
                      00004 ;       Author:         MPB
                      00005 ;       Date:           13-11-05
                      00006 ;
                      00007 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                      00008 ;
                      00009 ;       Slow output binary count is stopped, started
                      00010 ;       and reset with push buttons.
                      00011 ;       Modified with extra directives
                      00012 ;
                      00013 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                      00014
                      00015 ;       Declare processor, supress messages and warnings,
                      00016 ;       do not print symbol table, and configure (RC clock)
                      00017
                      00018         LIST p=16f877a, w=2, st=off, mm=off
2007   3733           00019         __CONFIG 0x3733
                      00020
                      00021 ;       Declare GPR label and literal constant
                      00022 ;       Define input labels
                      00023 ;       Include standard SFR label file
                      00024 ;       Include PortB initialisation file
                      00025
  00000020            00026 Timer EQU 20
  00000000            00027 DDCodeB SET b'00000000'
  00FF                00028 CONSTANT Count=0FF
                      00029 #DEFINE RunBut PORTD,1
                      00030 #DEFINE ResBut PORTD,0
                      00031 #INCLUDE "C:\books\book2\apps\p16f877a.inc"
                      00001         LIST
                      00002 ; P16F877A.INC  Standard Header File, Version 1.00    Microchip
Technology, Inc.
                      00398         LIST
                      00032 #INCLUDE <bout.ini>
0000   1683 1303      00001         BANKSEL TRISB           ; Select bank 1
0002   3000           00002         MOVLW   DDCodeB         ; Port B Direction Code
0003   0086           00003         MOVWF   TRISB           ; Load the DDR code into F86
0004   1283 1303      00004         BANKSEL PORTB           ; Select bank 0
                      00033
                      00034 ; Program code ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                      00035
                      00036 ; 'delay' macro ...........................................
                      00037
                      00038 delay   MACRO                   ; macro definition starts
                      00039         MOVWF   Timer           ; Copy W to timer register
                      00040 down    DECF    Timer           ; Decrement timer register
                      00041         BNZ     down            ; and repeat until zero
                      00042         ENDM                    ; macro definition ends
                      00043
                      00044 ; Main loop ...............................................
                      00045
0006   0186           00046 reset   CLRF    PORTB           ; Clear Port B Data
                      00047
0007   1C08           00048 start   BTFSS   ResBut          ; Test reset button
0008   2806           00049         GOTO    reset           ; and reset Port B if pressed
0009   1888           00050         BTFSC   RunBut          ; Test run button
000A   2807           00051         GOTO    start           ; and repeat if n pressed
                      00052
000B   0A86           00053         INCF    PORTB           ; Increment output at Port B
000C   30FF           00054         MOVLW   Count           ; Delay count literal
                      00055         delay                   ; Jump to subroutine 'delay'
000D   00A0               M         MOVWF   Timer           ; Copy W to timer register
000E   03A0               M down    DECF    Timer           ; Decrement timer register
000F   1D03 280E          M         BNZ     down            ; and repeat until zero
0011   2807           00056         GOTO    start           ; Repeat main loop always
                      00057
                      00058         END                     ; Terminate source code......

Errors   :     0
Warnings :     0 reported,     1 suppressed
Messages :     0 reported,     3 suppressed
```

Program 2.1 BIN4D list file using assembler directives

- Microchip header file title line is too long in this format.
- List file print for standard P16F877A include file is suppressed (398 lines).
- Port B initialisation include file path is not specified, uses default (<>).
- Delay macro does not need to be skipped in program execution.
- Special instruction BNZ is used in delay macro.
- Text substitution is used for input testing.
- Macro expansion is indicated by 'M' instead of line number.
- END is the only essential assembler directive.
- Warnings and messages are suppressed.
- Memory map and symbol list are suppressed.

In order to keep the sample programs, provided later, on interfacing as easy to understand as possible, most of these assembler options are to be avoided. Once the essentials of the assembly language have been mastered, the more powerful features of the assembler can be incorporated in your applications, based on a close study of the description of the PIC assembler directives given the help files provided with MPLAB.

Software Design

When the principles of assembly language programming are reasonably well understood, methods of software design need to be considered. This involves taking the program specification and working out how to construct the program; a design method is needed to outline the program structure and logic.

The language to be used to write the program will influence how this is done. The main alternative to assembly language is 'C', which has a more user-friendly syntax than assembler. It uses functions to represent blocks of assembly language; the C compiler converts source code functions and statements into the pre-defined machine code blocks. The component functions may readily be compiled in a library (collection of commonly used functions) or written by the user.

A C program can be best represented by pseudocode, a program description using structured text statements. By contrast, assembly language is more conveniently represented by flowcharts, especially for learning purposes, as it is a graphical design aid. More complex programs can be represented by structure charts. To illustrate the use of these techniques, we will analyse program BIN4.

The program specification can be stated as follows:

A portable, self-contained unit is required for educational purposes, at minimum cost, which displays a repeating 8-bit binary count on a set of LEDs. The count should start at zero, and be displayed when a non-latching push button (RUN) is held down. The count should stop when RUN is released, and reset to zero with another push button (RESET). The output sequence should be easily visible, with each full count cycle taking at least 10 s.

Flowcharts

There are two main forms of flowchart. Data flowcharts may be used to represent complex data processing systems, but here we will use a minimal set of symbols to represent an assembly language program. Program flowcharts may be used to represent overall program structure and sequence, but not the details. An example is given in Figure 2.3, which represent the program BIN4, as listed in Chapter 1.

The name of the program or project is given in the *start* symbol at the top of the flowchart. This is followed by the initialisation sequence in a plain rectangular *process* symbol, and an *input/output* operation (clear the LEDs) in the parallelogram-shaped symbol. A program *decision* (button pressed?) is enclosed in a diamond shape, with two possible outputs. The selection test is expressed as a question; the active decision is labelled yes or no, so it is unnecessary to label the default path.

The flow is implicitly down the page, so plain connected lines may be used, with the branch forward or backward using an arrow line style. The subroutine name is enclosed in a double line box, and expanded into a separate flowchart below. A parameter (the delay time) is passed to the subroutine as the register variable ‘count’.

These operations can be translated into PIC assembler as shown in Table 2.5. Obviously, the precise implementation will depend on the exact sequence required, but generally

- A process is a sequence with no external branches
- An I/O operation uses the ports
- A branch will use bit test and skip
- A subroutine uses CALL and RETURN

In most of the programs given here, ‘End’ is not needed in the flowchart, as the main sequence usually loops continuously until reset. However, the END directive is still needed to terminate the source code file.

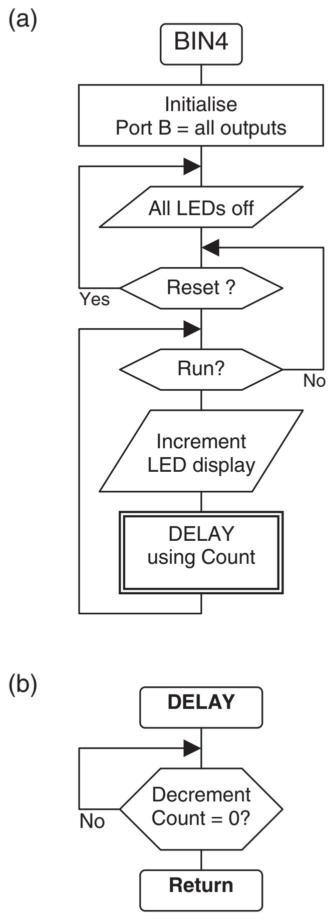
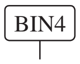


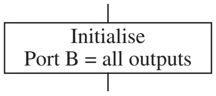
Figure 2.3 BIN4 flowcharts: (a) main routine; (b) delay subroutine

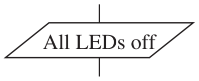
Flowcharts are useful for providing a graphical representation of the program, for example, for a presentation, but they are time consuming to create. Nevertheless, the flowcharts shown here were drawn just using the drawing tools in Word™, so the creation of flowcharts to a reasonable standard is not difficult for the occasional user. Specialist drawing packages are also available, which make the process quicker and easier for the professional software engineer.

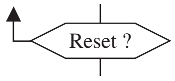
Pseudocode

Pseudocode shows the program as a text outline, using higher-level language constructs to represent the basic processes of sequential processing, selection and repetition. BIN4 is represented in Table 2.6.

Operation	Symbol	Implementation
Start End		Source code file/project name in start box. End not needed if program loops endlessly

Process
Sequence
		`BANKSEL   TRISB`
		`MOVLW     B'00000000'`
		`MOVWF     PORTB`

Input or Output		`CLRF      PORTB`

Branch
Selection
		`BTFSS     PORTD,Inres`
		`GOTO      reset`

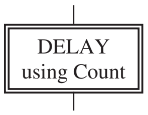
Subroutine
Procedure
or
Function
		`MOVLW     0FF`
		`CALL      delay`

Table 2.5 Flowchart implementation

The program outline uses high level key words such as IF and DO…WHILE to control the sequence. It is not an ideal method for a very simple program like this, but is useful for more complex programs. In particular, it translates directly into 'C', if the high-level language is preferred. Note that in this case, the program outline does not make any assumptions about the hardware implementation.

Structure Charts

Structure charts are also more suited to more complex programs, but the concept can be illustrated as in Figure 2.4.

Each program component is included under standard headings: inputs, processes and outputs, and can be broken down further in more complex programs, so that components can be created independently and then integrated.

'C' Programming

The 'C' programming language allows applications to be written using syntax whose meaning is a little easier to understand than assembly code. Programs

```
Project:       BIN4    MPB    22-3-06          Ver 1.0
Hardware:      BINX    MCU = P16F877           RC clock = 40 KHz
Description:   LED binary counter with stop and reset buttons

        Declare
            Registers               Input, Output, Count
            Bits                    Reset, Run

        Initialise
            Inputs (default)        Reset, Run
            Outputs                 LEDS

        Main
            DO
                IF Reset pressed
                    Switch off LEDs
                DO
                    Increment LEDS
                    Load Count
                    DELAY using Count
                WHILE Run pressed
            ALWAYS

        Subroutine
            DELAY
                DO
                    Decrement Count
                WHILE Count not zero
            RETURN
```

Table 2.6 BIN4 pseudocode

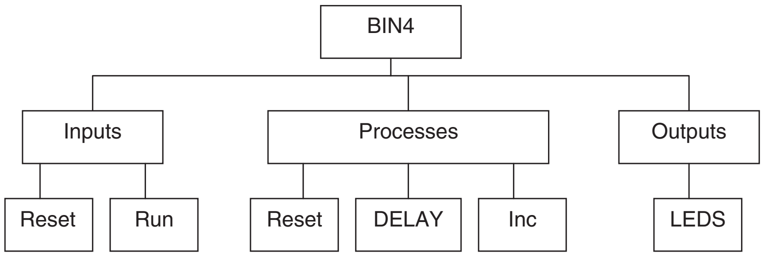


Figure 2.4 BIN4 structure chart

in C are converted into assembly language by a compiler, and assembled into machine code, in a two-stage process.

'C' is the high-level language of choice for microcontrollers. A range of different development systems and compilers are available, but most use the same basic syntax defined as ANSI (American National Standards Institute) C. Assembly language is syntax which is unique to each type of processor, while C provides a common language for all MCU types.

BIN4C Program

A version of BIN4, BIN4C, is shown in Figure 2.5 as an example. The function is the same as BIN4.

It can be seen that the program is simpler, because each C statement is converted into several assembler instructions. As a result, the program written in C will normally occupy more memory than the equivalent assembler version, so microcontrollers with larger memory are typically needed. Therefore, the more powerful 18XXXX series of PIC chips are usually used for C applications. They also have additional instructions, such as multiply, which makes the conversion more compact.

```
//    BIN4C.C     ************************      MPB   19-11-05

#include <16F877.h>           // Include standard MCU labels
#byte PortB=6                 // Output port data type and address


void main()                   // Start of program
{
   set_tris_b(0);             // Initialise output port
   PortB=0;                   // Initial output value

   while(1)                   // Endless loop between braces
   {
      if (!input(PIN_D0))     // Reset button pressed?
            PortB=0;          // if so, switch off LEDs

      if (!input(PIN_D1))     // Run button pressed?
            PortB++;          // if so, increment binary display
   }
}                             // End of program
```

Figure 2.5 BIN4C source code

In the BIN4C source code, the header file with the standard register labels for the 16F877 is included in the same way as in the assembler version. The output port is declared as an 8-bit variable (PortB), and its address assigned (6).

The main program block starts with the statement 'void main()' and is enclosed in braces (curly brackets). The output port is then initialised using a library function provided with the compiler 'set_tris_b(0)', where 0 is the data direction code in decimal form. An initial value of 0 is output to switch off the LEDs.

The control loop starts with the loop condition statement 'while(1)', which means repeat the statements between the braces endlessly. The buttons are tested using 'if (condition)' statements, and the actions following carried out if the condition is true. The condition is that the input is low (! = not), and pin labels as defined in the header file are used.

BIN4C Assembler Code

The C source code is compiled into assembler code, and then into machine code. The list file in Figure 2.6 shows the assembler version of BI4C.

It can be seen that some statements are converted into a single instruction, for example,

```
PortB++;  >>>  INCF   06,F
```

Others need several instructions,

```
if (!input(PIN_D0))   >>>     BSF    03.5
                              BSF    08.0
                              BCF    03.5
                              BTFSS  08.0
```

The total number of instructions for the C version is 28. The original assembler version used 20, giving an increase of 40% for the C version, in this case.

We are not going to look at the C language in any further detail here, but this example is given so that the advantages of C programming for microcontrollers can be appreciated. When assembly language has been mastered, the developer can then decide if C would be a better choice for given applications. For those needing complex mathematical calculations, for example, C is a better choice. For simpler programs comprising more bit

```
CCS PCM C Compiler, Version 3.207      19-Nov-05 15:01

               Filename: C:\PIC\bin4c\bin4c.LST
               ROM used: 28 words (0%)
                         Largest free fragment is 2048
               RAM used: 5 (3%) at main() level
                         5 (3%) worst case
               Stack:    0 locations
*
0000:  MOVLW  00
0001:  MOVWF  0A
0002:  GOTO   004
0003:  NOP
....................  //BIN4C.C
....................
.................... #include <16F877.h>
.................... #device PIC16F877
.................... #list
....................
.................... #byte PortB=6
....................
.................... void main()
.................... {
0004:  CLRF   04
0005:  MOVLW  1F
0006:  ANDWF  03,F
0007:  BSF    03.5
0008:  BSF    1F.0
0009:  BSF    1F.1
000A:  BSF    1F.2
000B:  BCF    1F.3
....................    set_tris_b(0);
000C:  MOVLW  00
000D:  MOVWF  06
....................    PortB=0;
000E:  BCF    03.5
000F:  CLRF   06
....................
....................    while(1)
....................    {
....................       if (!input(PIN_D0))
0010:  BSF    03.5
0011:  BSF    08.0
0012:  BCF    03.5
0013:  BTFSS  08.0
....................              PortB=0;
0014:  CLRF   06
....................       if (!input(PIN_D1))
0015:  BSF    03.5
0016:  BSF    08.1
0017:  BCF    03.5
0018:  BTFSS  08.1
....................          PortB++;
0019:  INCF   06,F
....................    }
001A:  GOTO   010
.................... }
....................
001B:  SLEEP
```

Figure 2.6 BIN4C list file

I/O operations and fewer calculations, assembler is generally faster and more compact. C and assembler can be mixed in the same program to retain the advantages of both.

SUMMARY 2

- The standard development system consists of a source code editor, assembler, simulator and programmer
- Machine code instructions can be broken down into operation and operand
- Programs should be well commented and structured for ease of analysis and debugging
- Assembler directives can be used to improve the efficiency and flexibility of code production

ASSESSMENT 2

1 Describe the advantages of in-circuit programming and debugging over the corresponding conventional development process. *(3)*

2 Refer to the instruction set in the PIC16F877 data sheet. State the binary codes for the operation and operands in the instruction DECFSZ 0C. *(3)*

3 State three commonly used assembler directives. *(3)*

4 Identify two instructions, one of which must be placed last in the PIC source code. What happens if one of these is not used? *(3)*

5 Identify two types of label used assembly language programming. *(3)*

6 State three PIC chip options, which are determined by the configuration code. *(3)*

7 State the function of the EQU directive. *(3)*

8 State the difference between the subroutine and macro, and one advantage of each. *(3)*

9 Describe the function of the standard header file “P16F877A.INC”. *(3)*

10 State the only assembler directive, which is essential in any program, and its function. *(3)*

11 Identify the five main symbols, which are used in a flowchart. *(5)*

12 Rewrite the BIN4 program pseudocode outline with the delay in-line (eliminate the subroutine). *(5)*

ASSIGNMENTS 2

2.1 MPLAB Test

Download and install the current version of the MPLAB development system (if necessary). Enter or download program BIN4. If entered manually, leave out the comments. Assemble (V7 Quickbuild) and run the program. Set up the input simulator buttons to represent the push buttons at Port D (toggle mode). Set the MCU clock to 40 kHz. Display Port B and the Timer register in a suitable window. Demonstrate that the program runs correctly.

2.2 MPLAB Debugging

Use the MPLAB debugging tools to single step the program BIN4 and observe the changes in the MCU registers. Operate the simulated inputs to enable the output count to Port B. Set a break point at the output instruction and run one loop at a time, checking that Port B is incremented. Use the stopwatch to measure the loop time. Comment out the delay routine call in the source code, re-assemble and check that the delay does not execute, and note the effect on the loop time. Re-instate the delay, change the delay count to 03 and note the effect on the loop time.

2.3 C Program

Write a minimal ‘C’ program, which will perform the same function as BIN1, and save as a plain text file BIN1.C. Discuss the advantages and disadvantages of programming in ‘C’ and assembler. Obtain access to a suitable ‘C’ development system and test your program. Predict the assembler code, which will be produced by the same compiler that was used to produce the list file BIN4C.LST. Add comments to explain the meaning of each assembler statement produced by the compiler.

This page intentionally left blank

3

Circuit Simulation

In the past, the electronics engineers needed to have a fairly comprehensive knowledge of both electronic component operation and circuit analysis, before setting out to design new applications. The circuit would be designed on paper and a prototype built to test the design, using a hardware prototyping technique such as stripboard; further refinement of the design would often then be required. When the circuit was fully functional, a production version could be developed, with the printed circuit board (PCB) being laid out by hand. Further testing would then be needed on the production prototype to make sure that the layout was correct, and that the variation in component values due to tolerances would not prevent the circuit from functioning correctly. Learning how electronics systems worked also required a good imagination! Unlike mechanical systems, it is not obvious how a circuit works from simple observation. Instruments (voltmeters, oscilloscopes, etc.) must be used to see what is happening, and these also need complex skills to use them effectively.

We now have computer-based tools that make the job easier, and perhaps more enjoyable. An early ECAD (Electronic Computer-Aided Design) tool was a system of mathematical modelling used to predict circuit behaviour. SPICE was developed at University of Berkeley, California, to provide a consistent and commonly understood set of models for components, circuits and signals. This system uses nodal analysis to predict the signal flow between each point in a circuit, based on the connections between the components. The results would be displayed or printed numerically.

The simplest component is the resistor, and the simplest mathematical model Ohm's law, $V = IR$, which relates the current and voltage in the resistor. For two resistors in series this becomes $V = I(R_1+R_2)$. The power dissipated in the resistor is given by $P = IV$. For AC signals, RMS voltages are used so that the same model can be applied. Reactive components need the frequency of the signal to be included, so $V = IX$ is used, where X is the reactance. For a capacitor, the magnitude of the reactance is $1/\omega C$, for an inductor ωL, where C is the capacitance and L the inductance. We then need to model the phase relationship between voltage and current, using complex numbers, and so on.

Digital circuits are in principle easier, since they are modelled using simple logical relationships, such as A = B.C, where the dot represents the ‘and’ operation. The other main operators are ‘+’ representing logical ‘or’, and ‘!’ logical invert. Thus, a simple logic function may appear as A = (B.C + !D).

The next step is to model mixed mode circuits, with analogue and digital components connected together. Then microprocessors needed to be added, which needed a programmed model to represent program execution. Computer graphics have now developed to the point that the modelling can be done in conjunction with a circuit drawn on the computer screen, and a simulation generated interactively. Components placed in the drawing have their models attached, and the nodes are identified from the connections on the schematic. Inputs can be supplied from simulated signal sources, and virtual instruments and on-screen graphics are used to display the outputs obtained.

Interactive circuit simulation now makes the job of analysing and designing electronic circuits quicker, easier (therefore cheaper) and more fun! The circuit can be drawn and tested on screen, and a PCB layout also generated from the schematic. Simple (one or two sided) PCBs can now be produced directly by a machine tool attached to the same computer, avoiding the usual chemical process. Once in production, assembly and testing can also be automated.

Proteus VSM (Virtual System Modelling), from Labcenter Electronics in the UK, has been used to create the circuit diagrams and test the designs in this book. It is currently the only package available with a comprehensive range of microcontroller models. The schematic capture and interactive simulator component is ISIS; a PCB layout can be created from the same drawing using the associated application ARES. It is the most complete package available at the current time for designing and testing embedded applications, providing an extensive range of passive and active components, mixed mode simulation and interactive peripheral hardware. Details of Proteus can be found at www.labcenter.co.uk.

Basic Circuit

A circuit to demonstrate the operation of BIN4 program is shown in Figure 3.1, designated as BINX since it can be used for a range of programs. The circuit

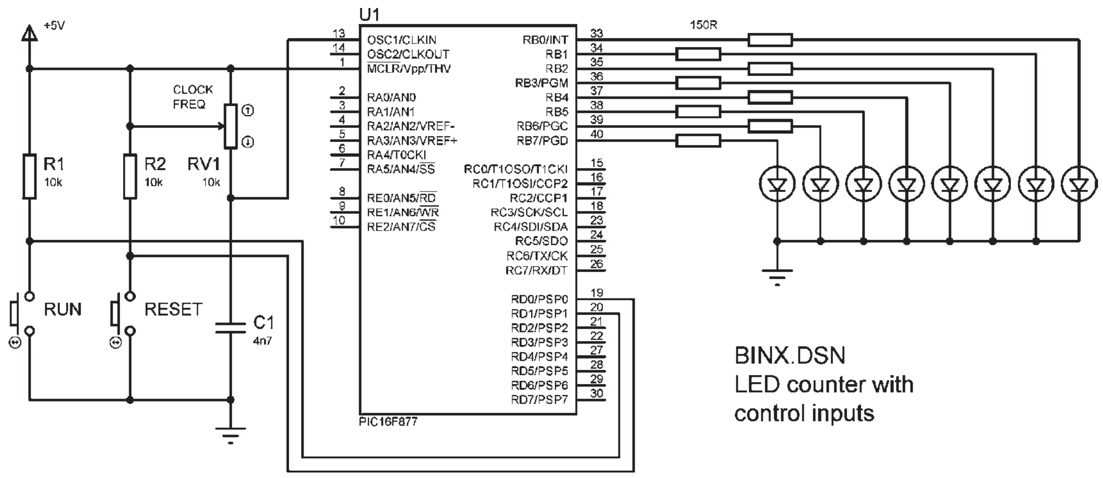


Figure 3.1 BINX schematic

was drawn using ISIS and exported as a bitmap for insertion into a document. As can be seen, ISIS also allows circuit diagrams to be readily presented to a professional standard.

The microcontroller is a PIC 16F877, our reference device; other PIC chips will be described later. A set of LEDs is connected to Port B, with push buttons on RD0 and RD1. A CR clock circuit is shown connected to CLKIN, with a pre-set pot providing variable resistance, which allows the clock frequency to be adjusted. Remember the clock frequency is inversely proportional to the CR product. Note that for simulation purposes the external clock circuit does not control the operating frequency of the PIC; this must be set in the properties dialogue for the MCU component (see below). Similarly, the MCLR (Master Clear) input does not have to be connected for the program to run in simulation mode, whereas this is essential in the real circuit.

The inputs are pulled up to 5 V via 10k resistors, although this value is not critical. The inputs are thus high by default, as is the case if they are not connected. The port does not need to be initialised for input, as this is also the default condition. On the other hand, the outputs do need to be initialised by the MCU program by loading the data direction register with zeros. The PIC outputs can typically provide up to 25 mA, which is enough to light the LEDs without any additional current drivers. 150 R resistors limit the current in the LEDs to about 20 mA.

The programs BIN1 and BIN4 can be tested in this simulated hardware. When creating a new application, a suitable folder should be made to contain the project files, since the design file for the schematic will be accompanied by several files associated with the attached program (source code, hex code, list file, etc.).

Drawing the Schematic

A screenshot of the ISIS schematic capture and interactive simulation environment is shown in Figure 3.2. The main schematic edit window is accompanied by an overview window showing the whole drawing and an object select window, which normally contains a list of components. However, it also shows lists of other available devices for use in the edit window when specific modes are selected.

The main editing window includes a sheet outline, which shows the edge of the drawing area, within which components must be placed. The component button is normally selected by default in the mode toolbar. With this mode selected, components are fetched for placing on the schematic by hitting the P (pick devices) button, and selecting the required category of components. The individual device type can then be chosen from a list (Figure 3.3).

The components are categorised as microprocessors (includes microcontrollers), resistors, capacitors, etc. with variants within each. Subcategories can be selected. Interactive components, such as push buttons, are grouped in the ACTIVE library.

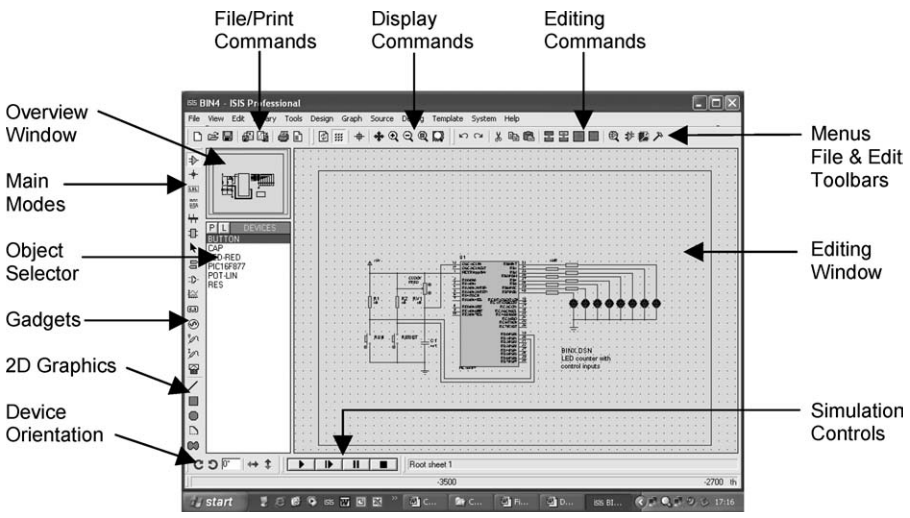
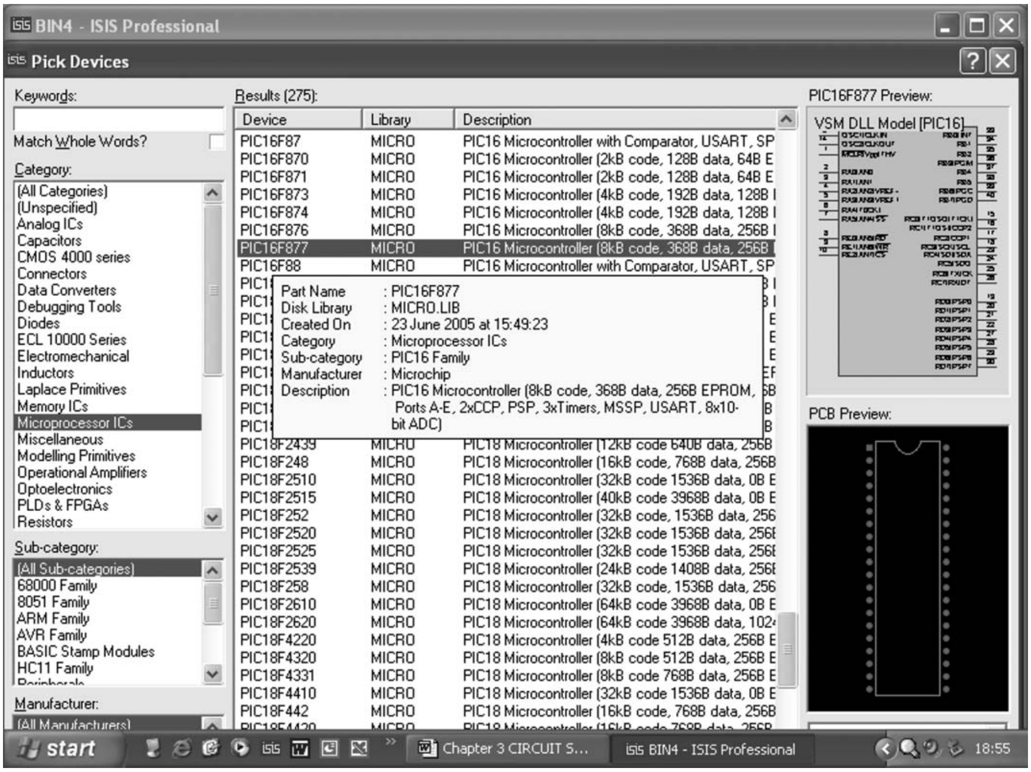


Figure 3.2 ISIS screenshot



BIN4 - ISIS Professional

Pick Devices

Keywords:

Match Whole Words?

Category:

(All Categories)
(Unspecified)
Analog ICs
Capacitors
CMOS 4000 series
Connectors
Data Converters
Debugging Tools
Diodes
ECL 10000 Series
Electromechanical
Inductors
Laplace Primitives
Memory ICs
Microprocessor ICs
Miscellaneous
Modelling Primitives
Operational Amplifiers
Optoelectronics
PLDs & FPGAs
Resistors

Sub-category:

(All Sub-categories)
68000 Family
8051 Family
ARM Family
AVR Family
BASIC Stamp Modules
HC11 Family

Manufacturer:

(All Manufacturers)

Results (275):

Device	Library	Description
PIC16F87	MICRO	PIC16 Microcontroller with Comparator, USART, SP
PIC16F870	MICRO	PIC16 Microcontroller (2kB code, 128B data, 64B E
PIC16F871	MICRO	PIC16 Microcontroller (2kB code, 128B data, 64B E
PIC16F873	MICRO	PIC16 Microcontroller (4kB code, 192B data, 128B I
PIC16F874	MICRO	PIC16 Microcontroller (4kB code, 192B data, 128B I
PIC16F876	MICRO	PIC16 Microcontroller (8kB code, 368B data, 256B I
PIC16F877	MICRO	PIC16 Microcontroller (8kB code, 368B data, 256B
PIC16F88	MICRO	PIC16 Microcontroller with Comparator, USART, SP
PIC18F2439	MICRO	PIC18 Microcontroller (12kB code 640B data, 256B
PIC18F248	MICRO	PIC18 Microcontroller (16kB code, 768B data, 256B
PIC18F2510	MICRO	PIC18 Microcontroller (32kB code 1536B data, 0B E
PIC18F2515	MICRO	PIC18 Microcontroller (40kB code 3968B data, 0B E
PIC18F252	MICRO	PIC18 Microcontroller (32kB code, 1536B data, 256
PIC18F2520	MICRO	PIC18 Microcontroller (32kB code 1536B data, 256
PIC18F2525	MICRO	PIC18 Microcontroller (32kB code 1536B data, 256
PIC18F2539	MICRO	PIC18 Microcontroller (24kB code 1408B data, 256
PIC18F258	MICRO	PIC18 Microcontroller (32kB code, 1536B data, 256
PIC18F2610	MICRO	PIC18 Microcontroller (64kB code 3968B data, 0B E
PIC18F2620	MICRO	PIC18 Microcontroller (64kB code 3968B data, 102
PIC18F4220	MICRO	PIC18 Microcontroller (4kB code 512B data, 256B E
PIC18F4320	MICRO	PIC18 Microcontroller (8kB code 512B data, 256B E
PIC18F4331	MICRO	PIC18 Microcontroller (8kB code 768B data, 256B E
PIC18F4410	MICRO	PIC18 Microcontroller (32kB code 1536B data, 0B E
PIC18F442	MICRO	PIC18 Microcontroller (16kB code, 768B data, 256B

Part Name	: PIC16F877
Disk Library	: MICRO.LIB
Created On	: 23 June 2005 at 15:49:23
Category	: Microprocessor ICs
Sub-category	: PIC16 Family
Manufacturer	: Microchip
Description	: PIC16 Microcontroller (8kB code, 368B data, 256B EPROM, Ports A-E, 2xCCP, PSP, 3xTimers, MSSP, USART, 8x10-bit ADC)

PIC16F877 Preview:

VSM DLL Model [PIC16]

PCB Preview:

start

Chapter 3 CIRCUIT S...

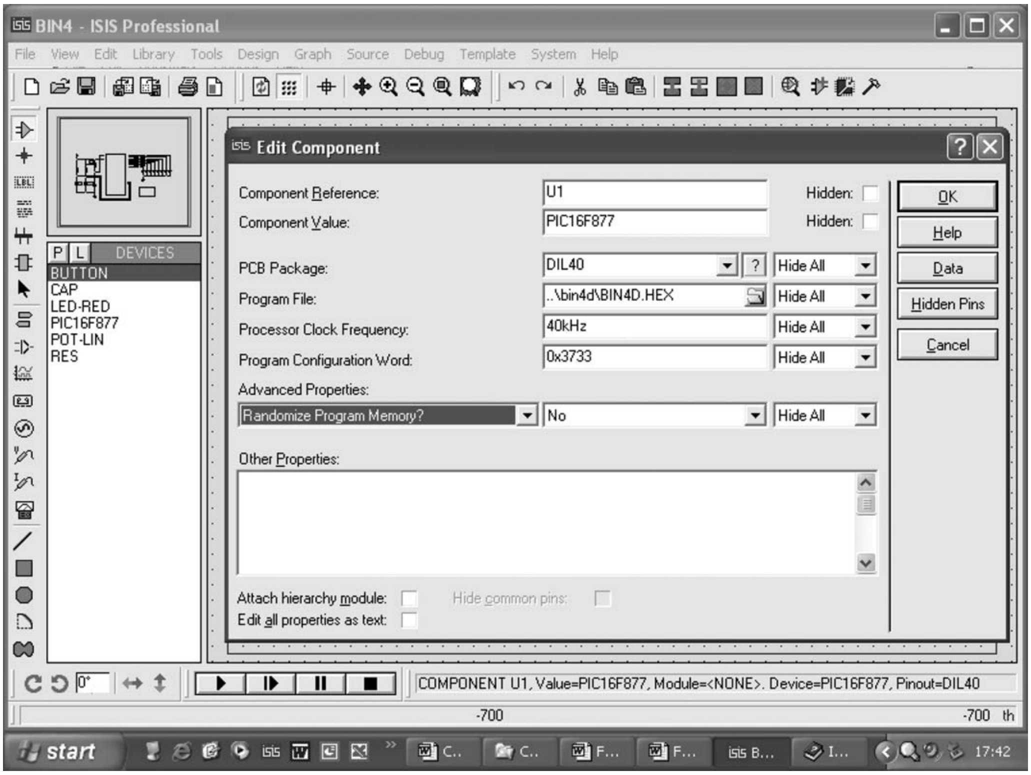
BIN4 - ISIS Professional

18:55

Figure 3.3 Picking a device

The selected components appear in the device list, and when highlighted can be placed on the schematic with a single mouse click. The pins are then connected as required by clicking on the component leads and dragging a wire. Right-click highlights a connection or component, and further right-click deletes it. Right-click and left-click open a connection or component properties dialogue. The PIC chip property edit window, for example, allows the program hex file to be selected, and the simulation clock frequency and configuration word to be entered (Figure 3.4).

To complete the BINX schematic, power terminals must be added. Select the terminal button in the objects toolbar, and a list of terminal types is displayed in the device list. Power and ground terminals may then be added to the drawing. The power terminal voltage needs to be defined via its properties' dialogue. Entering +5 V as the label actually defines the operating voltage as well. Note that the MCU does not need a power supply connection – it is assumed to operate at 5 V.



Component Reference:	U1	Hidden: ☐
Component Value:	PIC16F877	Hidden: ☐
PCB Package:	DIL40	Hide All
Program File:	..\bin4d\BIN4D.HEX	Hide All
Processor Clock Frequency:	40kHz	Hide All
Program Configuration Word:	0x3733	Hide All
Advanced Properties:		
Randomize Program Memory?	No	Hide All

Other Properties:

Attach hierarchy module: ☐ Hide common pins: ☐
Edit all properties as text: ☐

```
COMPONENT U1, Value=PIC16F877, Module=<NONE>, Device=PIC16F877, Pinout=DIL40
```

Figure 3.4 MCU properties

Circuit Simulation

An MCU-based simulation will not run without a program attached to the micro-controller chip. When the program has been edited and assembled, a hex (machine code) file is created. In the MPLAB development system, it is tested with simulated inputs and numerical outputs; for example, the state of Port B is displayed as a hex or binary number. Inputs are generated as asynchronous events by assigning on screen buttons, or using a stimulus file to generate the same input sequence each time the simulation is run. It is a purely software simulator, but with some advantages for the experienced developer.

ISIS provides a more user-friendly development environment, particularly for the inexperienced designer, by providing interactive, on-screen, inputs and outputs, so that the circuit can be seen operating in the same way as it will in the real hardware (we hope!). It also provides debugging features, which are also good for learning, that is, not too complicated.

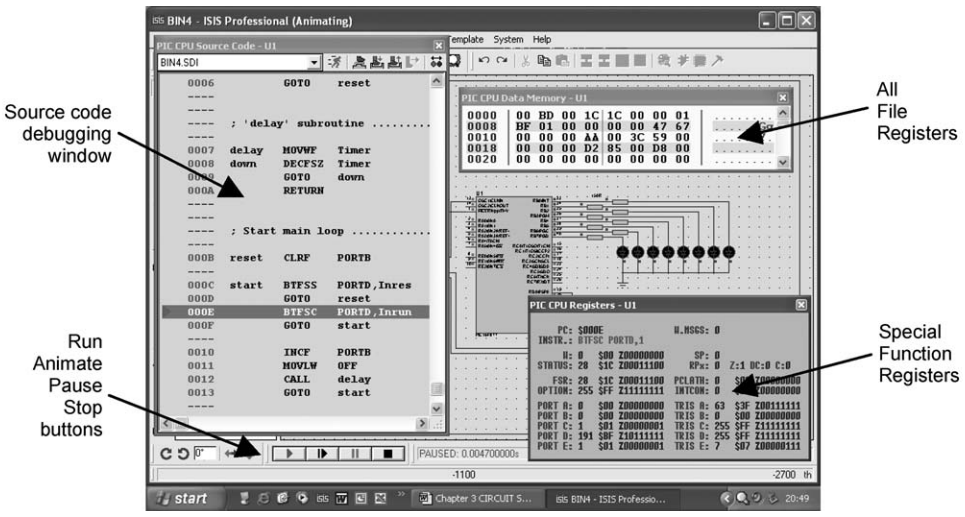
If the hex file has already been created, this can be attached to the MCU via the processor properties dialogue and the circuit will operate. However, it is generally more useful to attach the source code as well, since this enables source code debugging. In this mode, the source code listing is displayed and the programme stopped, started and single stepped with the execution point showing in the source code window. If there are problems with the program sequence (logical errors), the source code debugger allows them to be more easily resolved.

The source code debugging window is seen in Figure 3.5. The execution point in the program listing is highlighted. The buttons at the top of the source code window allow the program to be stepped, or run between breakpoints. The CPU (special function) registers are also displayed, so that the MCU internal changes can be tracked. The CPU data memory window shows all the file registers, so that general-purpose register contents can be monitored.

Setting up the Simulation

There are three main stages for testing an MCU design:

- Create the schematic drawing around the selected MCU
- Write the program source code and build (assemble) it
- Attach the resulting machine code to the MCU



```
0006          GOTO    reset
----
----
----  ; 'delay' subroutine .........
----
0007  delay   MOVWF   Timer
0008  down    DECFSZ  Timer
0009          GOTO    down
000A          RETURN
----
----
----  ; Start main loop ...........
----
000B  reset   CLRF    PORTB
----
000C  start   BTFSS   PORTD,Inres
000D          GOTO    reset
000E          BTFSC   PORTD,Inrun
000F          GOTO    start
----
0010          INCF    PORTB
0011          MOVLW   OFF
0012          CALL    delay
0013          GOTO    start
----
```

```
0000 | 00 BD 00 1C | 1C 00 00 01
0008 | BF 01 00 00 | 00 00 47 67
0010 | 00 00 00 AA | 00 3C 59 00
0018 | 00 00 00 D2 | 85 00 D8 00
0020 | 00 00 00 00 | 00 00 00 00
```

```
     PC: $000E                W.MSGS: 0
 INSTR.: BTFSC PORTD,1

      W: 0   $00 Z00000000       SP: 0
 STATUS: 28  $1C Z00011100      RPx: 0  Z:1 DC:0 C:0

    FSR: 28  $1C Z00011100  PCLATH: 0   $00 Z00000000
 OPTION: 255 $FF Z11111111  INTCON: 0   $00 Z00000000

 PORT A: 0   $00 Z00000000  TRIS A: 63  $3F Z00111111
 PORT B: 0   $00 Z00000000  TRIS B: 0   $00 Z00000000
 PORT C: 1   $01 Z00000001  TRIS C: 255 $FF Z11111111
 PORT D: 191 $BF Z10111111  TRIS D: 255 $FF Z11111111
 PORT E: 1   $01 Z00000001  TRIS E: 7   $07 Z00000111
```

Figure 3.5 Source code debugging

SOURCE CODE

The source code file is written in the source text editor from the Source menu, select Add/Remove Source Files, New. Create the new source file in the project folder containing the design file. MPASM should be selected as the Code Generation Tool (the standard Microchip assemblers are included in ISIS). The source code file now appears in the Source menu, and the edit window can be opened by clicking on it.

The program is written using the defined PIC assembler syntax. Write a basic header consisting of the source code file name, author, date, plus version number, target design file and hardware information (e.g. clock type and speed) as required. The MCU type should then be specified, the configuration word provided and the standard label file included, as necessary. Add the END directive, which is always needed for correct assembly. The header can now be saved and built, and attached, and the simulation run. The program will not yet do anything, but this ensures the file paths are correct before proceeding with the full source code. As with most projects, it is advisable to develop the application stage by stage, ensuring correct function at each stage before proceeding to the next.

Build All is used to assemble the program and create the hex file. Obviously, if there are any syntax errors, they must be corrected at each stage. BIN4 source code (Chapter 1) provides an example of the kind of header information which should be included.

MACHINE CODE

To attach the machine (hex) code, click right, then left, on the MCU and the component properties dialogue should open. The folder tab by the Program File box gives access to the project files, and allows the hex file to be opened (attached). The MCU clock frequency and configuration word can be entered at the same time. For the clock RC components shown in the schematic for BINX, the time constant is about 5k $\times$ 4*n*7 $\cong$ 25 μs, giving a frequency of 1/25 MHz = 40 kHz. Any clock value can be set in the properties dialogue, regardless of the components in the drawing, but normally they should match. The variable resistance in the schematic allows the clock frequency to be adjusted around this value in the final hardware. 40 kHz gives an instruction frequency of 10 kHz, and an instruction cycle time of 100 μs. The configuration word can be set to 0x3FFB (RC clock, watchdog disabled).

Running the Simulation

The simulation is run by clicking on the run button in the set of control buttons at the bottom of the screen. If the source code has been changed, it is automatically saved and re-assembled at this command, which provides one-click retesting. This saves a lot of time during the development process, especially for the inexperienced programmer.

The circuit will operate in real time if the animation settings are correct, and the simulation is not too complex. The default animation settings (System, Set Animation Options) are 20 frames per second and 50 ms per frame, giving real-time operation (20×50 ms = 1 s). For other applications and clock speeds, these settings may need to be modified to see the circuit operation clearly. In this circuit, the output LEDs should show a visible binary count; as the delay between each increment is about 75 ms, the whole count will take about 20 s.

The count is started by 'pressing' the run button with the mouse. It should stop when released, and start again at the same count. The reset button should clear the count to zero. Note that there is a problem if mouse must be used to select other operations – it cannot be used to hold the button. The answer is to temporarily link across the run button in the circuit, to keep the circuit in run mode for testing or replace the buttons on the drawing with switches. While the simulation is running, the logic state of each line can be indicted as a red (1) or blue (0) square; this feature is enabled via the System menu, Animation Options.

Software Debugging

The main objective of simulation testing is to fault-find the software before downloading it to the real hardware. In ISIS, the hardware design is tested at the same time. Changes to the hardware simply require editing the component properties (e.g. to change a resistance), rewiring or changing components.

Syntax errors (e.g. misspelling an instruction) and semantic errors (e.g. missing a label out) should have been identified at the initial program assembly stage. Simulation allows logical errors to be detected, that is, mistakes in the operation of the program when executed. Source code debugging enables the source code execution sequence to be examined, and changed to eliminate errors. The main debugging information is provided from

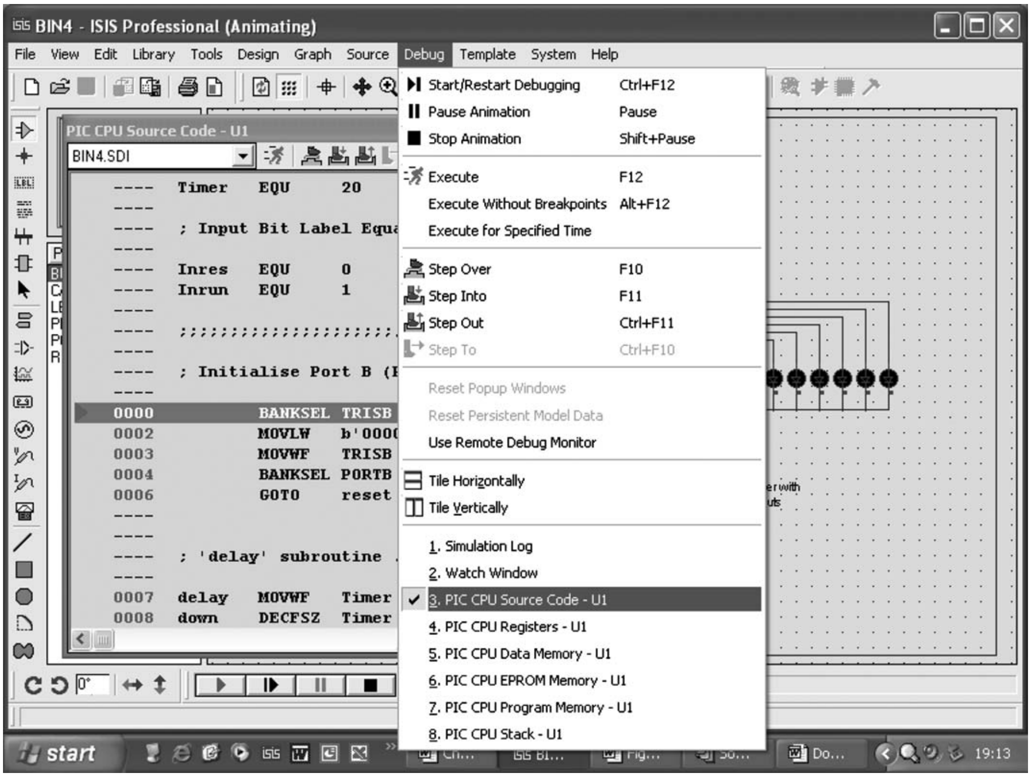
- Source code debug window
- CPU register display (SFRs)
- CPU data memory display (file registers)
- Watch window

SOURCE CODE DEBUG WINDOW

This shows the source code, alongside the program memory locations and hex code when these options are selected; the program execution can be controlled from this window. It is called up by pausing the program, and selecting PIC CPU Source Code in the Debug menu (Figure 3.6).

When paused, the current program execution point is shown in the source debug window. The buttons at the top of the window are used to run or single-step the program, as follows:

Run..	*at full speed (window closes)*
Step Over…	*Step through instructions only in the current routine, and execute subroutines at full speed*
Step Into…	*Step through all instructions, including subroutines*


BIN4 - ISIS Professional (Animating)
File View Edit Library Tools Design Graph Source Debug Template System Help
PIC CPU Source Code - U1
BIN4.SDI
Start/Restart Debugging Ctrl+F12
Pause Animation Pause
Stop Animation Shift+Pause
Execute F12
Execute Without Breakpoints Alt+F12
Execute for Specified Time
Step Over F10
Step Into F11
Step Out Ctrl+F11
Step To Ctrl+F10
Reset Popup Windows
Reset Persistent Model Data
Use Remote Debug Monitor
Tile Horizontally
Tile Vertically
1. Simulation Log
2. Watch Window
3. PIC CPU Source Code - U1
4. PIC CPU Registers - U1
5. PIC CPU Data Memory - U1
6. PIC CPU EPROM Memory - U1
7. PIC CPU Program Memory - U1
8. PIC CPU Stack - U1
start
19:13

```
----    Timer    EQU     20
----
----    ; Input Bit Label Equa
----
----    Inres    EQU     0
----    Inrun    EQU     1
----
----    ;;;;;;;;;;;;;;;;;;;;;;
----
----    ; Initialise Port B (I
----
0000             BANKSEL TRISB
0002             MOVLW   b'0000
0003             MOVWF   TRISB
0004             BANKSEL PORTB
0006             GOTO    reset
----
----
----    ; 'delay' subroutine .
----
0007    delay    MOVWF   Timer
0008    down     DECFSZ  Timer
```

Figure 3.6 Select source debug window

Step Out of..	*Run at full speed out of current subroutine, then step through the calling routine*
Run to…	*current cursor position*

These controls allow the program sequence to be inspected, skipping subroutines if these are OK or will be debugged later, or escaping from a subroutine loop (e.g. delay). Breakpoints allow the program to be run and stopped at a selected point. For example, if a break point is set at the beginning of a loop, it can be executed one loop at a time. Additional options are available with a right-click on the source debug window (e.g. clear all breakpoints). If the program sequence is incorrect, the source code must be corrected in the edit window, which should be kept minimised for quick access while debugging.

CPU REGISTER WINDOW

This displays selected special function registers, including the port data and direction registers, plus the working register, status flags, etc. It also shows the stack pointer, which is not normally accessible in the real chip. This shows which of the 8 return address locations is next available, that is, how many levels of subroutine have been used up.

CPU DATA MEMORY

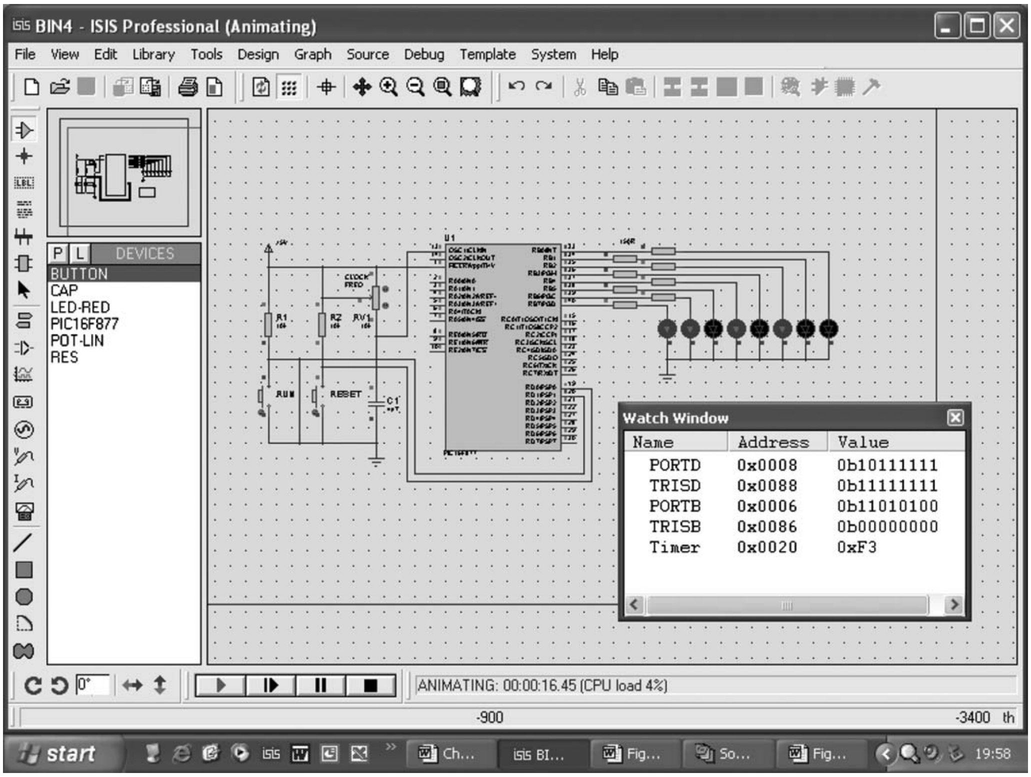
This shows all the file registers, so it is a quick way to check on a general purpose register. For example, in BIN4, the Timer count register can be seen. When a register changes, it is highlighted, which helps to keep a track of them.

WATCH WINDOW

This window allows user-selected registers to be monitored, in a variety of data formats. By right-clicking on the window, the SFRs can be picked up from a list by name, or GPRs added by address (number) and named. This allows only those registers which are of particular interest to be viewed. Unlike the other debug windows, this window remains visible when the simulator is in run mode, which allows the registers to be monitored in real time (Figure 3.7).

Hardware Testing

We have seen how to create the circuit in schematic form, and to test the program. The simulation software also provides virtual instruments which can be used to measure circuit performance, just as in the real world. The Instruments button in the Gadgets toolbar provides a list of these in the device window. It includes an Oscilloscope, Logic Analyser, Signal Generator, Voltmeters and


BIN4 - ISIS Professional (Animating)
File View Edit Library Tools Design Graph Source Debug Template System Help
DEVICES
BUTTON
CAP
LED-RED
PIC16F877
POT-LIN
RES
RUN
RESET
Watch Window
ANIMATING: 00:00:16.45 (CPU load 4%)
start

Name	Address	Value
PORTD	0x0008	0b10111111
TRISD	0x0088	0b11111111
PORTB	0x0006	0b11010100
TRISB	0x0086	0b00000000
Timer	0x0020	0xF3

Figure 3.7 Watch window

Ammeters. The complete list of tools available and how to use them is provided in the Proteus VSM help files.

Meters

Voltmeters and ammeters may be added to the circuit to measure volt drop across a component or absolute voltage at a point (relative to 0 V) (Figure 3.8). The ammeter measures current through a component, or along a ‘wire’. As an example, the current and voltage around one of the LEDs is shown with the LED on. The range and other properties of a meter can be changed by right-clicking on the instrument.

Oscilloscope

A virtual oscilloscope allows signals to be displayed in the same way as a real oscilloscope. Select it from the Instruments list and left-click to drop the

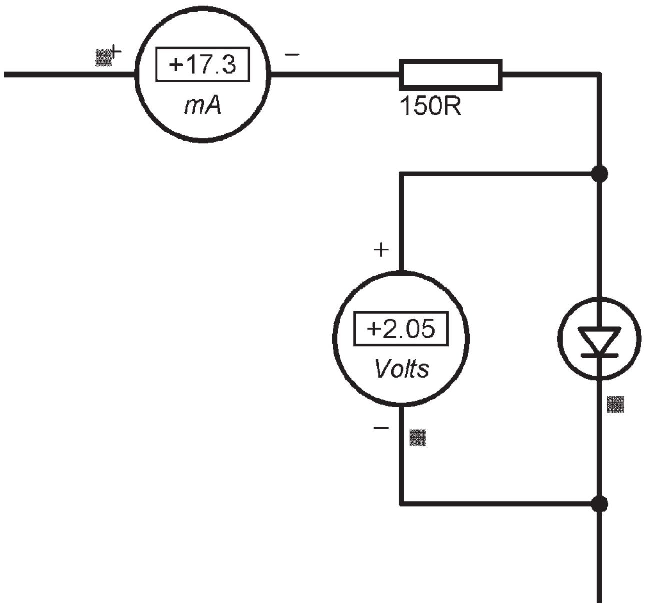


Figure 3.8 Voltmeter and ammeter

minimised version on the drawing. Two channels, A and B, are available for connection to different points in the circuit, with the 0 V being implicit. A larger display version is enabled by pausing the simulation and selecting the oscilloscope in the Debug menu. It is not displayed until the program is run; with a signal in view, the scope controls may be adjusted to obtain the best display, and measure signal amplitude or frequency. Figure 3.9 shows the BIN4 application with a scope and logic analyser attached. A special version of the program with the delay commented out to speed it up was used to give a suitable scope display (BIN4F). Figure 3.10 shows the clock output (CLKOUT) from the PIC chip alongside the LSB output at RB0 (CLKOUT has to be enabled in the MCU Properties, Advanced Properties dialogue).

Logic Analyser

The logic analyser allows multiple digital signals to be displayed simultaneously. They are captured by sampling a set of lines at regular intervals, and storing the samples as binary bits. A typical mid-range analyser might have 48 inputs, sampled at 25 MHz. Thus, 6 bytes are stored every 40 µs, continuously. Unlike the oscilloscope, when the analyser is triggered, the data from 'before' the trigger event, as well as after, can be displayed.

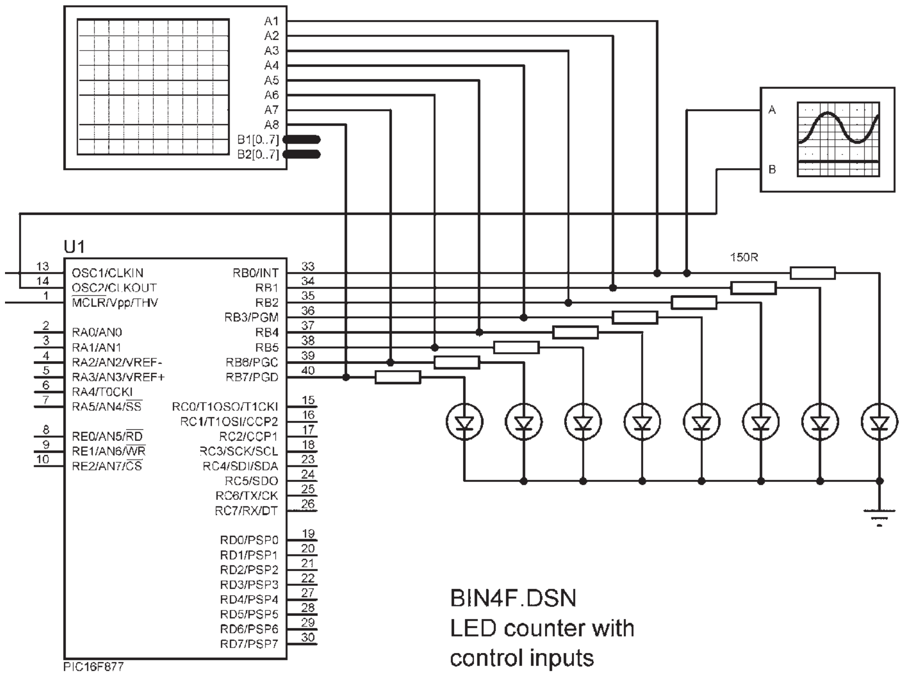
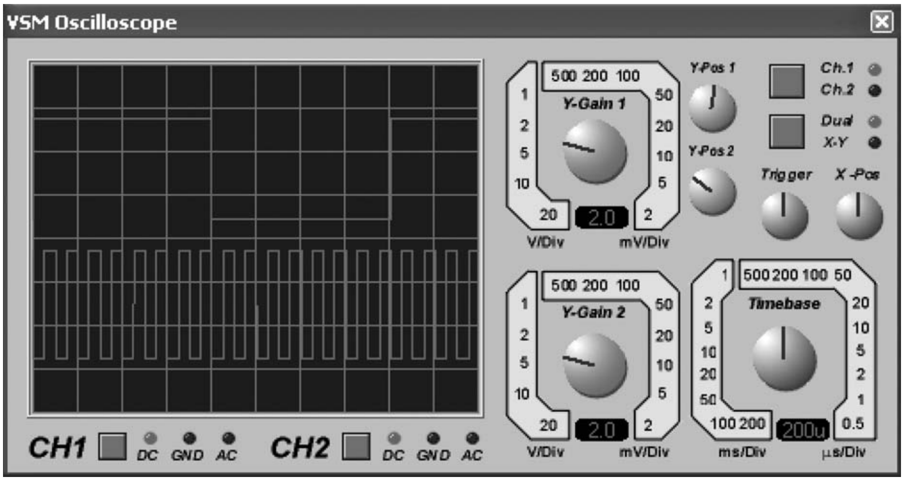


Figure 3.9 Oscilloscope and Logic Analyser



Y-Gain 1: 2.0 V/Div
Y-Gain 2: 2.0 V/Div
Timebase: 200u

Figure 3.10 Oscilloscope display

The logic analyser is particularly useful in testing conventional microprocessor systems, as it allows the data signals flowing between the CPU, memory and I/O devices to be captured from the address and data busses. This can then be compared with the program list file, to identify any discrepancies. Individual signals can also be displayed. In real logic analysers, the data can be displayed as a timing diagram, as in the oscilloscope, or as binary or hexadecimal in a table, which shows each sample numerically. This is unnecessary in the simulation, as this information can be captured from the CPU or MCU itself.

In Figure 3.9, the logic analyser is capturing the 8-bit output at Port B. This is useful if the output is changing at high speed; program BIN4F provides the test output. The analyser trigger input must be operated while the program is running or paused. Note that there may be a significant delay before the data is displayed. Time intervals can be measured using the two marker controls (Figure 3.11).

Graphs

Another powerful feature of the Proteus simulation is the graph display. Select the Simulation Graph option in the Mode toolbar, and a graph box can be drawn in a convenient position on the schematic. Attach voltage probes to the binary output at Port B, as seen in Figure 3.12. If these are highlighted and dragged onto the graph area, they are assigned to the next available graph line on the chart. They can be deleted by right-clicking twice.

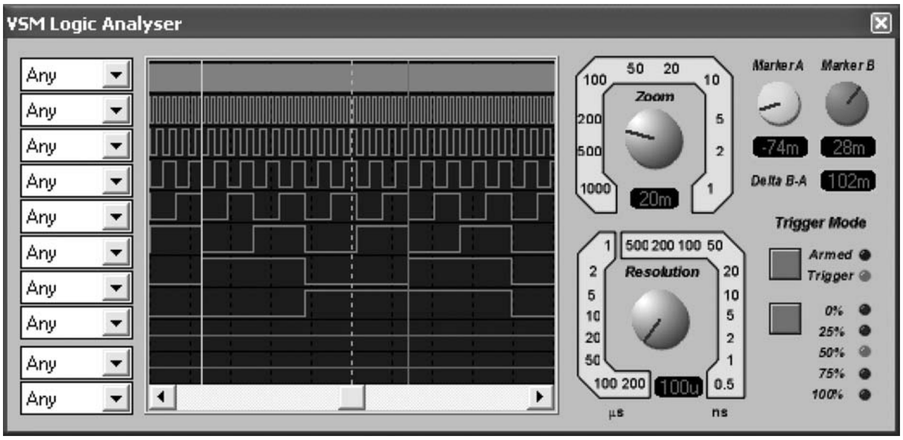


Figure 3.11 Logic analyser display

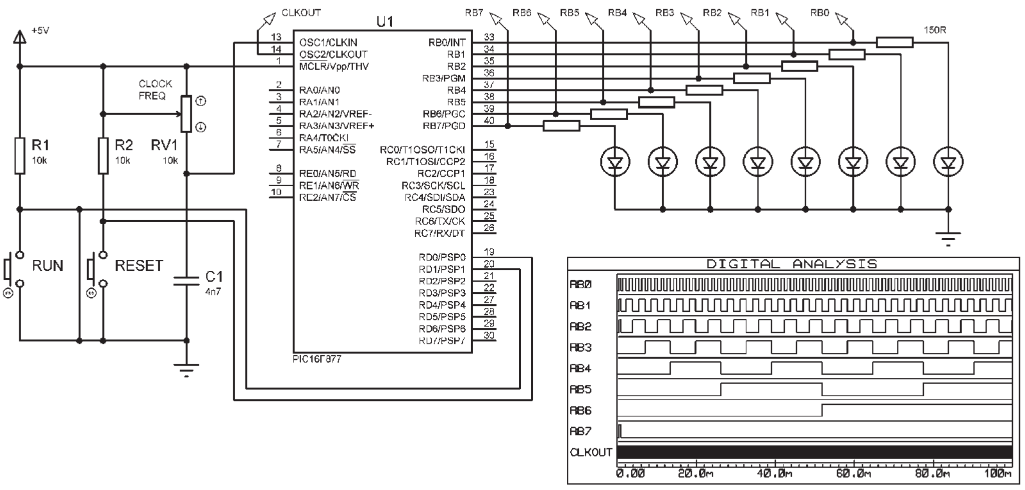


Figure 3.12 Digital graph

Now run the simulation and stop. Hit the spacebar – the signals should appear in the graph window. If necessary, right, then left, click on the graph area to change the timescale settings. For this example, running BIN4F, a maximum time of 100 ms is suitable. The graph can be enlarged to full screen size for detailed analysis and printing, just click on the title bar of the window.

Hardware Implementation

When the application design has been tested and proved in simulation mode, it can be converted into prototype hardware. The ISIS schematic can be exported to ARES, the PCB layout part of the Proteus package, as a netlist. This is a list of the components in the circuit, with SPICE model definitions attached, and a list of the circuit nodes which describes how the components are connected. Each of these identifies the terminals of the components attached to that node, so that the signal flow can be calculated for simulation, and to define the components and connections needed in a PCB layout (Figure 3.13).

The nodes in the netlist can be identified by comparison with the circuit schematic. For example, the clock input CLKIN is defined as node 18:

`#00018,3`	Node number 18, 3 connections to:
`C1,PS,2`	Cap C1, passive terminal, pin 2

```
ISIS SCHEMATIC DESCRIPTION FORMAT 6.1
=====================================
Design:   C:\BOOKS\BOOK2\APPS\1 LED Output\bin4\BIN4.DSN
Doc. no.: <NONE>
Revision: <NONE>
Author:   <NONE>
Created:  12/02/05
Modified: 25/11/05

*PROPERTIES,0

*MODELDEFS,0

*PARTLIST,21
C1,CAP,4n7,EID=16,PACKAGE=CAP10,PINSWAP="1,2"
D1,LED-RED,LED-RED,BV=4V,EID=6,IMAX=10mA,PACKAGE=DIODE25,ROFF=100k,RS=3,STATE=0,VF=2V
D2,LED-RED,LED-RED,BV=4V,EID=7,IMAX=10mA,PACKAGE=DIODE25,ROFF=100k,RS=3,STATE=0,VF=2V
D3,LED-RED,LED-RED,BV=4V,EID=8,IMAX=10mA,PACKAGE=DIODE25,ROFF=100k,RS=3,STATE=0,VF=2V
D4,LED-RED,LED-RED,BV=4V,EID=9,IMAX=10mA,PACKAGE=DIODE25,ROFF=100k,RS=3,STATE=0,VF=2V
D5,LED-RED,LED-RED,BV=4V,EID=A,IMAX=10mA,PACKAGE=DIODE25,ROFF=100k,RS=3,STATE=0,VF=2V
D6,LED-RED,LED-RED,BV=4V,EID=B,IMAX=10mA,PACKAGE=DIODE25,ROFF=100k,RS=3,STATE=0,VF=2V
D7,LED-RED,LED-RED,BV=4V,EID=C,IMAX=10mA,PACKAGE=DIODE25,ROFF=100k,RS=3,STATE=0,VF=2V
D8,LED-RED,LED-RED,BV=4V,EID=D,IMAX=10mA,PACKAGE=DIODE25,ROFF=100k,RS=3,STATE=0,VF=2V
R1,RES,10k,EID=4,PACKAGE=RES40,PINSWAP="1,2",PRIMTYPE=RESISTOR
R2,RES,10k,EID=5,PACKAGE=RES40,PINSWAP="1,2",PRIMTYPE=RESISTOR
R3,RES,220R,EID=E,PACKAGE=RES40,PINSWAP="1,2",PRIMTYPE=RESISTOR
R4,RES,220R,EID=F,PACKAGE=RES40,PINSWAP="1,2",PRIMTYPE=RESISTOR
R5,RES,220R,EID=10,PACKAGE=RES40,PINSWAP="1,2",PRIMTYPE=RESISTOR
R6,RES,220R,EID=11,PACKAGE=RES40,PINSWAP="1,2",PRIMTYPE=RESISTOR
R7,RES,220R,EID=12,PACKAGE=RES40,PINSWAP="1,2",PRIMTYPE=RESISTOR
R8,RES,220R,EID=13,PACKAGE=RES40,PINSWAP="1,2",PRIMTYPE=RESISTOR
R9,RES,220R,EID=14,PACKAGE=RES40,PINSWAP="1,2",PRIMTYPE=RESISTOR
R10,RES,150R,EID=15,PACKAGE=RES40,PINSWAP="1,2",PRIMTYPE=RESISTOR
RV1,POT-LIN,10k,EID=17,STATE=5
U1,PIC16F877,PIC16F877,ADC_ACQUISITION_TIME=20u,ADC_RCCLOCK_PERIOD=4u,ADC_SAMPLE_DELAY=100n,CFGWORD=0x3FFB,CLOCK=40
kHz,DBG_ADC_BREAK=0,DBG_ADC_WARNINGS=0,DBG_ADDRESSES=0,DBG_DUMP_CFGWORD=0,DBG_GENERATE_CLKOUT=0,DBG_I2C_OPERATIONS=
1,DBG_RANDOM_DMEM=0,DBG_RANDOM_PMEM=0,DBG_STACK=1,DBG_STARTUP_DELAY=0,DBG_UNIMPLEMENTED_MEMORY=1,DBG_UNIMPLEMENTED_
OPCODES=1,DBG_WAKEUP_DELAY=0,EID=1A,EPR_WRITECODE_DELAY=10m,EPR_WRITEDATA_DELAY=10m,ITFMOD=PIC,MODDATA="256,255",MO
DDLL=PIC16,PACKAGE=DIL40,PORTTDHL=0,PORTTDLH=0,PROGRAM=..\bin4f\BIN4F.HEX,WDT_PERIOD=18m

*NETLIST,47
#00000,2                        #00012,2                        #00025,1                        #00041,1
R1,PS,1                         R5,PS,1                         U1,IO,9                         U1,IO,5
U1,IO,20                        U1,IO,38
                                                                #00026,1                        #00042,1
#00001,2                        #00013,2                        U1,IO,10                        U1,IO,15
R2,PS,1                         R6,PS,1
U1,IO,19                        U1,IO,37                        #00027,1                        +5V,6
                                                                U1,OP,14                        +5V,PT
#00002,2                        #00014,2                                                        R1,PS,2
D1,PS,A                         R7,PS,1                         #00028,1                        R2,PS,2
R3,PS,2                         U1,IO,36                        U1,IO,16                        RV1,PS,3
                                                                                                RV1,PS,1
#00003,2                        #00015,2                        #00029,1                        U1,IP,1
D2,PS,A                         R8,PS,1                         U1,IO,17
R4,PS,2                         U1,IO,35                                                        GND,10
                                                                #00030,1                        GND,PT
#00004,2                        #00016,2                        U1,IO,18                        C1,PS,1
D3,PS,A                         R9,PS,1                                                         D1,PS,K
R5,PS,2                         U1,IO,34                        #00031,1                        D8,PS,K
                                                                U1,IO,30                        D7,PS,K
#00005,2                        #00017,2                                                        D6,PS,K
D4,PS,A                         R10,PS,1                        #00032,1                        D5,PS,K
R6,PS,2                         U1,IO,33                        U1,IO,29                        D4,PS,K
                                                                                                D3,PS,K
#00006,2                        #00018,3                        #00033,1                        D2,PS,K
D5,PS,A                         C1,PS,2                         U1,IO,28
R7,PS,2                         RV1,PS,2                                                        VDD,3
                                U1,IP,13                        #00034,1                        VDD,PT
#00007,2                                                        U1,IO,27                        U1,PP,11
D6,PS,A                         #00019,1                                                        U1,PP,32
R8,PS,2                         U1,IO,2                         #00035,1
                                                                U1,IO,22                        VSS,3
#00008,2                        #00020,1                                                        VSS,PT
D7,PS,A                         U1,IO,3                         #00036,1                        U1,PP,12
R9,PS,2                                                         U1,IO,21                        U1,PP,31
                                #00021,1
#00009,2                        U1,IO,4                         #00037,1                        *GATES,0
D8,PS,A                                                         U1,IO,26
R10,PS,2                        #00022,1
                                U1,IO,6                         #00038,1
#00010,2                                                        U1,IO,25
R3,PS,1                         #00023,1
U1,IO,40                        U1,IO,7                         #00039,1
                                                                U1,IO,24
#00011,2                        #00024,1
R4,PS,1                         U1,IO,8                         #00040,1
U1,IO,39                                                        U1,IO,23
```

Figure 3.13 BINX netlist

`RV1,PS,2`	Pot RV1, passive terminal, pin 2
`U1,IP,13`	Chip U1, input terminal, pin 13

The netlist data is used by the layout package to generate a list of components with specified pinouts. If there is a choice of physical packages, ARES will allow the user to select the most suitable. These are placed on the layout, with temporary connections shown as straight lines between the pins, producing a ratsnest display. Autorouting can then be invoked and a layout produced which can be converted into a PCB. In Figure 3.14, the layout for circuit BINX is shown for a double-sided board, so some tracks are overlaid on others.

If the PCB is to be produced automatically, a standard Gerber output file can be generated to be passed to a prototyping system. Alternatively, the PCB can be produced in the traditional way, by printing the layout as a transparency, transferring it to the board photographically and etching the board layout chemically.

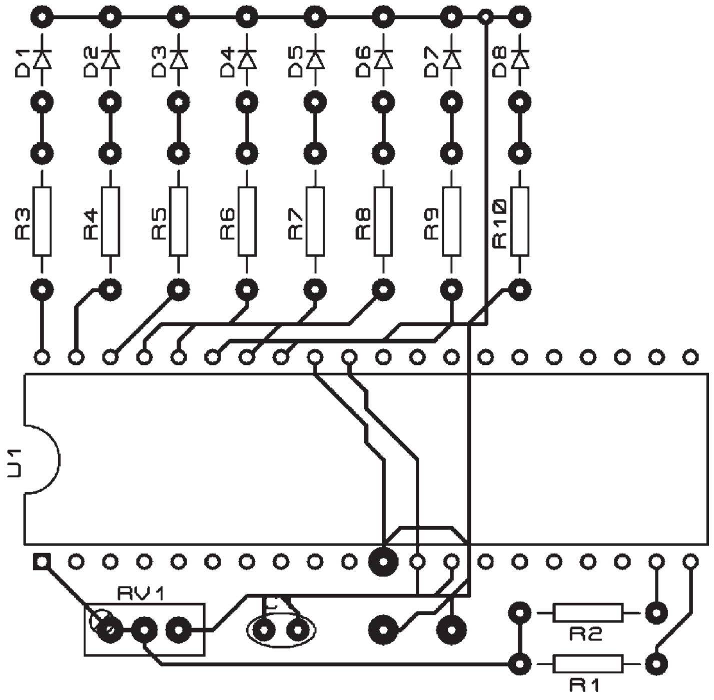


Figure 3.14 PCB layout

Program Downloading

When the application hardware has been produced, the MCU program must be downloaded into the chip. The traditional method is to remove the chip from the circuit and place it in a separate programming unit (therefore must be fitted in a socket, not soldered in). The program is then downloaded from the development system PC host, using the programming utility in MPLAB. This has obvious disadvantages – removing the chip risks physical and electrical damage, is not possible to reprogram in-circuit or remotely.

Microchip have therefore come up with an in-circuit programming method, which is inexpensive and provides the opportunity to debug the system while running in real hardware. ICD (In-Circuit Debugging) uses the same programming pins on the chip, but they are connected to a 6-pin programming connector fitted to the application hardware. An ICD programming module is then connected between the host PC and the target application board, and the program downloaded to the chip in circuit. This also provides the possibility of remote reprogramming after an application has been commissioned on site. A slight disadvantage when using Proteus VSM for debugging is that, at present, the program must be returned to MPLAB for downloading (Figure 3.15).

ICD allows the program to run from MPLAB, and debugged in hardware, with the usual single stepping and breakpoint control. To achieve this, the chip has special debugging features built in so that the program can be interrupted as required; also, a block of debug code is loaded into the top of memory, slightly reducing the maximum possible program size. A NOP (No Operation) must be placed at location 0000 (first instruction in the program) to allow access to the debug code before the program starts executing. Since RB3, RB6

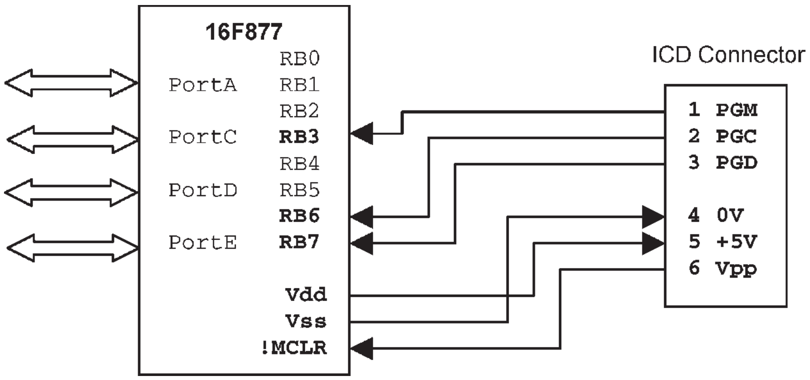


Figure 3.15 ICD connections

and RB7 are needed for programming, it is best not to use these for conventional I/O, if possible. This would mean re-designing the BINX circuit and using another port for the LEDs. Note also that a pull-up resistor should be used on !MCLR to enable normal running, while allowing the programming voltage (about 13 V) to be applied without damaging the target system 5 V supply.

Testing the program in the real hardware allows any final bugs to be squashed. These could arise from slight performance discrepancies between the hardware and the simulated circuit, last minute modifications to the design and so on. When the program has been fully tested and passed as functioning fully to specification, the program is downloaded again with the ICD option turned off, so that it will run independently.

SUMMARY 3

- Circuits are tested by simulation based on mathematical modelling
- Proteus VSM provides interactive simulation in the schematic
- Source code and the hex file must be attached to the MCU
- The primary debug tool is source code stepping with register display
- Simulated test instruments are used for virtual testing
- A netlist is exported to ARES for the PCB layout

ASSESSMENT 3

1 State the three main steps in testing an MCU design in Proteus VSM. *(3)*

2 Why does the clock circuit not have to be drawn for simulation? *(3)*

3 Calculate the instruction cycle time if the clock is 10 MHz *(3)*

4 What type of errors are detected by the assembler and simulation respectively? *(3)*

5 Explain the difference between step into, step over and step out. *(3)*

6 Explain why breakpoints are useful in debugging. *(3)*

7 Explain the functions of a logic analyser. *(3)*

8 Explain the user actions required to generate a graph in Proteus VSM. *(3)*

9 Explain the function of a netlist. *(3)*

10 Explain the advantages of ICD over traditional hardware debugging. *(3)*

11 Outline the process for testing a design by interactive simulation, and state its advantages over conventional development techniques. *(5)*

12 State the type of signal that would be typically measured using a voltmeter, oscilloscope and logic analyser, respectively. State two advantages of a simulation graph. *(5)*

ASSIGNMENTS 3

3.1 Bin4 Simulation

Test the application program BIN4 in simulation mode by attaching it to the BINX design. Carry out the same, or equivalent, tests as those detailed in Chapter 2 (Assignment 1). Comment out the delay and modify the schematic as necessary to display a full speed count. Use the virtual instruments in ISIS simulation mode to display the output: the LSB output on the scope, all outputs on the logic analyser and graph. Confirm correct operation of the application in the interactive simulation environment.

3.2 System Comparison

Compare in detail the functionality of the MPLAB and Proteus simulation environments, and identify the advantages of each.

3.3 Proteus Debugging

Load the BIN4 project into the ISIS simulator environment. Introduce the following errors into BIN4.ASM:

Omit (comment out) the PROCESSOR directive

Omit (comment out) the label equate for 'Timer'

Replace 'CLRF' with 'CLR' (invalid mnemonic)

Delete label 'Start' (label missing)

Replace '0' with 'O' in the literal 0FF.

Omit (comment out) 'END' directive

Note the effect of each error type and the message produced by the assembler. What general type of error are they? Warning ‘default destination being used’ should be received in the list file. What does this mean? Eliminate it by changing the assembler error level to suppress messages and warnings.

With the program restored so that it assembles correctly:

Replace ‘BTFSS’ with ‘BTFSC’

Omit (comment out) ‘GOTO reset’

Note the effect of these errors. What general type of error are they? Describe the process used to detect each one.

Part 2

Interfacing

This page intentionally left blank

4

Input & Output

We can now proceed to the main business of this book – describing a range of input & output techniques which will help you to design microcontroller circuits. Today, the typical MCU-based consumer product can be extremely complex, and contains a wide range of technologies around the main controller. The mobile phone is a good example; in addition to the sophisticated digital communications subsystem which provides its main function, it can also have a full-colour, medium resolution liquid crystal display (LCD) screen, camera, sound system and so on. A detailed understanding of these technologies requires a very high level of engineering skill. To help develop this skill, some simpler equivalent technologies must be studied – for example, we will see how to display character-based information on a low-resolution monochrome alphanumeric LCD, and ignore its graphics capabilities for now.

Switch Input

The simplest input is a switch or push button. This can operate with just one additional support component, a pull-up resistor, but there are still some significant issues to consider, such as input loading and debouncing. We have seen in the BINX hardware how a simple push button or switch is interfaced with a pull-up resistor. Let us make sure we understand how this works (Figure 4.1).

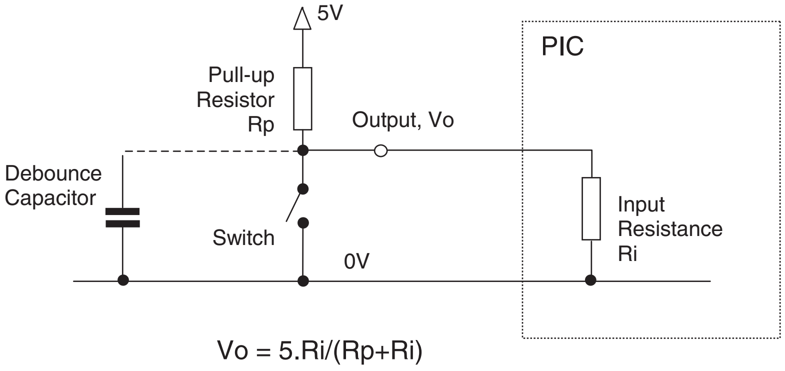


Figure 4.1 Input switch

When the switch is open, the output voltage of the circuit is pulled up to +5 V via the resistor. Another way to look at it is that there is no current in the resistor (assuming there is no load on the output), so there is no volt drop, and the output voltage must be the same as the supply (+5 V). When the switch is closed the output is connected direct to 0 V; the resistor prevents the supply being shorted to ground.

The resistor value is not critical, but if there is any load on the output, the pull-up must be significantly lower in value than the load, for the digital voltage levels to be valid. On the other hand, its value should be as high as possible to minimise wasted power, especially if the circuit is battery-powered. Let us say the load on the output (R_i) was equivalent to 400k (100 μA with a 5 V supply), the pull-up resistor (R_p) should be 100k or less, so that the output voltage with the switch open would be at least 4 V. The minimum voltage which will be reliably recognised as logic 1 at a PIC input is 2.0 V, and the maximum voltage recognised as a logic 0 is 0.8 V, assuming a +5 V supply. The input leakage current is actually only about 1 μA. If power conservation is not critical, a 10k resistor is a reasonable choice.

If a PIC input is open circuit, it is pulled up to Vdd (normally +5 V) internally, that is, it floats high. On some ports (Port B), weak pull-ups can be enabled to eliminate the need for external pull-up resistors. The switch symbol assumes toggle mode operation – the switch remains in the set position until changed. Push buttons normally assumed to be closed only when held – the toggle operation can be implemented in software if required, to obtain a push-on, push-off operation.

Switch Debouncing

When the contacts close in any mechanical switch or push button, they tend to bounce open before settling in closed position. In normal life, this is not noticeable or significant, but in microsystems it is liable to cause circuit misbehaviour if ignored. The effect generally lasts a few milliseconds, but if the switch is sampled at high speed, it can appear that it has been operated several times.

On the other hand, the sequence of the program may be such that the switch bounce does not adversely affect the correct operation of the system. For example, if the program does not recheck the input until the bouncing has finished anyway, no specific debouncing is needed.

In Figure 4.2 (a), the output voltage from a switch jumps back up to 5 V due to the switch contacts bouncing open. If a suitable capacitor is connected across the switch, as in Figure 4.2 (b) it is initially charged up to 5 V. When the contacts close, it is quickly discharged by the short circuit. However, it can only recharge via the pull-up resistor, which takes more time. If the switch closes again before the logic 0 minimum threshold is crossed (0.8 V), the voltage is prevented from going back to logic 1. The capacitor needs to be large enough to give a slow voltage rise in the charging phase, while not being so high in value as to cause a large discharge current through the switch contacts when the contacts close, or making the rise time too long when the switch is opened. With a 10k pull-up resistor, a 10 nF capacitor would give a time constant of 100 ms, which should be more than adequate.

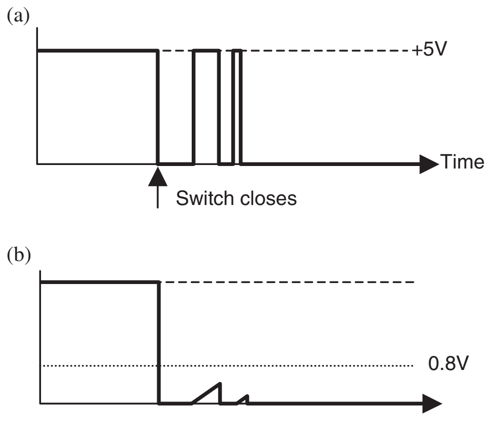


Figure 4.2 Switch hardware debounce (a) without debounce capacitor; (b) with debounce capacitor

The circuit shown in Figure 4.3 will be used to illustrate debouncing. A binary display count is to be incremented under manual control, via a switch input, one step at a time. The circuit now includes the ICD programming connections with a 6-way connector, as would be required in the actual hardware. These will not always be shown, but will be required if the ICD method is to be used when debugging the real circuit. To accommodate these connections, the LEDs have been moved to Port D, and spare lines in Port B used for the inputs.

If the switch input is not debounced, several counts might be registered. The debounce process shown in Program 4.1 (LED1S) uses the same software delay routine previously used to provide delays between each output step. Note that the simulated switch model has a delay of 1 ms to represent the bounce effect, which does not, unfortunately, accurately model the real switch behaviour. However, it can still be seen that if the software delay is commented out, the count is not reliable.

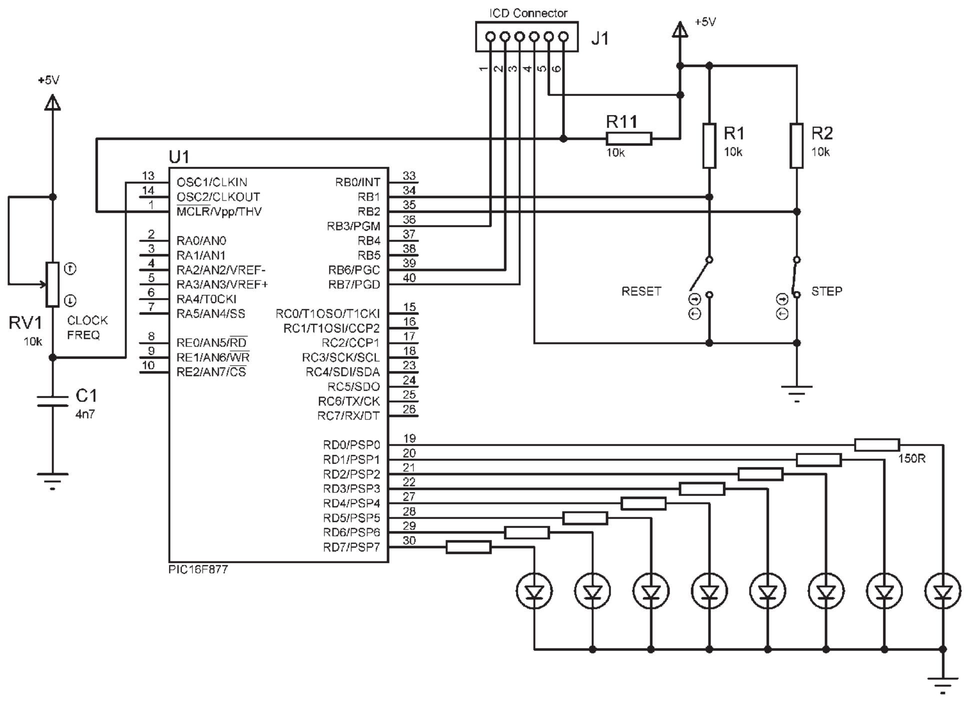


Figure 4.3 LED counter with ICD interface

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;       Source File:     LED1S.ASM
;       Author:          MPB
;       Date:            2-12-05
;
;       Output binary count is stepped manually
;       and reset with push buttons.
;       Demonstrates software delay switch debounce
;       Hardware: BIN4X simulation
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        PROCESSOR 16F877         ; Define MCU type
        __CONFIG 0x3733          ; Set config fuses

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

PORTB   EQU     06      ; Port B Data Register
PORTD   EQU     08      ; Port D Data Register
TRISD   EQU     88      ; Port B Direction Register
Timer   EQU     20      ; GPR used as delay counter

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

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

        BANKSEL TRISD            ; Select bank 1
        MOVLW   b'00000000'      ; Port B Direction Code
        MOVWF   TRISD            ; Load the DDR code into F86
        BANKSEL PORTD            ; Select bank 0
        GOTO    reset            ; Jump to main loop

; '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

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

reset   CLRF    PORTD            ; Clear LEDs

start   BTFSS   PORTB,1          ; Reset?
        GOTO    reset            ; Yes - clear LEDs
        BTFSC   PORTB,2          ; Step on?
        GOTO    start            ; No - wait

        MOVLW   0FF              ; Delay count literal
        CALL    delay            ; Wait for count
        BTFSS   PORTB,2          ; Step on?
        GOTO    start            ; Yes – wait
        INCF    PORTD            ; Increment LEDs
        GOTO    start            ; Repeat always

        END                      ; Terminate source code
```

Program 4.1 Software debouncing

Timer and Interrupts

The main problem with the delay loop method is that MCU time is being wasted (at 5 Mhz instruction rate, 5000 instruction cycles could be completed in 1 ms). In a high performance system (high speed, large data throughput), this is very inefficient, so the use of a hardware timer would be preferred. This option allows the MCU to proceed with other tasks while carrying out a timing operation concurrently. In addition, the switch problem is a good opportunity to examine the use of hardware timers and interrupts in general.

If a hardware timer is started when the switch is first closed, the closure can be confirmed by retesting the input after a time delay to check if it is still closed. Alternatively, the switch input can be processed after the button is released, rather than when it is closed. The system responds when the switch is released rather than when it is pressed, again avoiding the bounce problem.

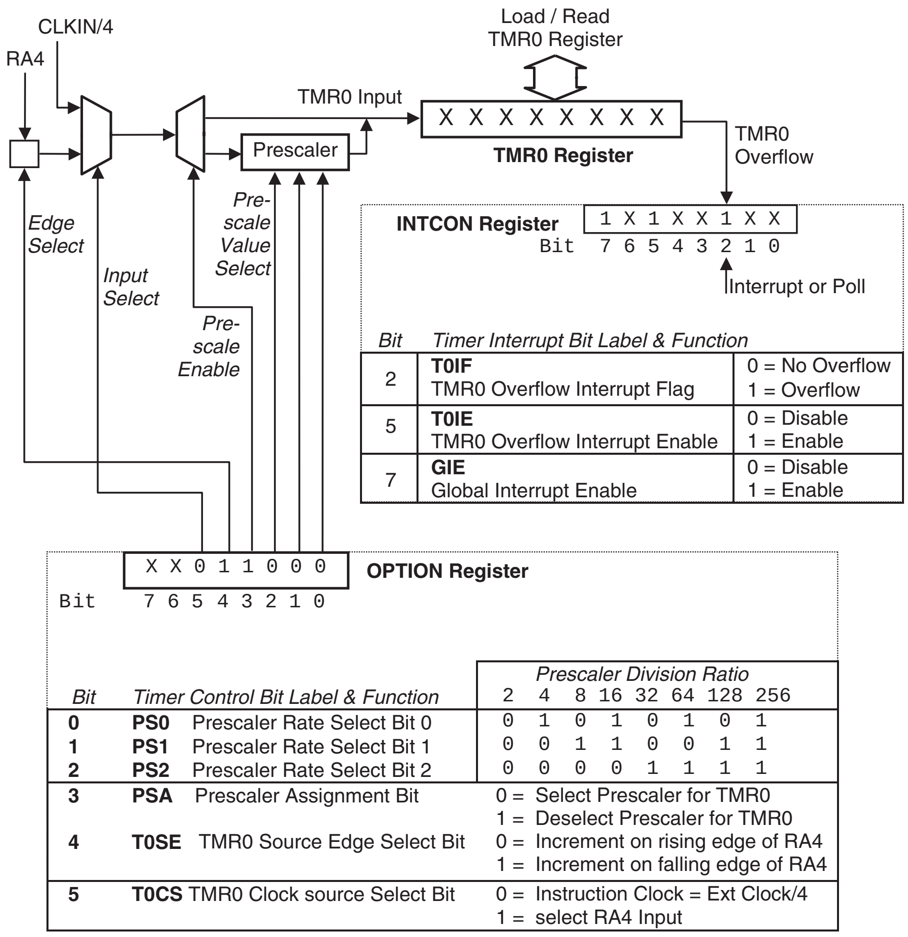
The same virtual hardware (Figure 4.3) will be used for Program 4.2 (LED1H) which illustrates the use of a hardware timer. TMR0 (Timer0) is located at file register address 01. It operates as an 8-bit binary up counter, driven from an external or internal clock source. The count increments with each input pulse, and a flag is set when it overflows from FF to 00. It can be pre-loaded with a value so that the time out flag is set after the required interval. For example, if it is pre-loaded with the number 156_{10}, it will overflow after 100 counts (256_{10}). A block diagram of Timer0 is shown in Figure 4.4.

Timer0 has a pre-scaler available at its input, which divides the number of input pulses by a factor of 2, 4, 8, 16, 32, 64, 128 or 256, which increases the range of the count but reduces its accuracy. For timing purposes, the internal clock is usually selected, which is the instruction clock seen at CLKOUT in RC mode ($f_{osc}/4$). Thus the register is incremented once per instruction cycle (there are 1 or 2 cycles per instruction), without the pre-scaler. At 10 kHz, it will count in steps of 100 µs.

Interrupts are another way of increasing the program efficiency. They allow external devices or internal events to force a change in the execution sequence of the MCU. When an interrupt occurs, in the PIC the program jumps to program address 004, and continues from there until it sees a Return From Interrupt (RETFIE) instruction. It then resumes at the original point, the return address having been stored automatically on the stack as part of the interrupt process. Typically, a GOTO ISR (Interrupt Service Routine) is placed at the interrupt address 004, and the ISR placed elsewhere in memory using ORG. The interrupt source is identified by the associated flag (in this case, Timer0 overflow flag). This has an associated interrupt enable bit to enable the MCU to respond to this particular source, and a global enable bit to disable all interrupts

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;       Source File:     LED1H.ASM
;       Author:          MPB
;       Date:            2-12-05
;
;       Output binary count incremented
;       and reset with push buttons.
;       Uses hardware timer to debounce input switch
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        PROCESSOR 16F877          ; Define MCU type
        __CONFIG 0x3733           ; Set config fuses

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

PORTB   EQU     06      ; Port B Data Register
PORTD   EQU     08      ; Port D Data Register
TRISD   EQU     88      ; Port D Direction Register

TMR0    EQU     01      ; Hardware Timer Register
INTCON  EQU     0B      ; Interrupt Control Register
OPTREG  EQU     81      ; Option Register

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        ORG     000               ; Start of program memory
        NOP                       ; For ICD mode
        GOTO    init              ; Jump to main program

; Interrupt Service Routine ...............................

        ORG     004
        BCF     INTCON,2          ; Reset TMR0 interrupt flag
        RETFIE                    ; Return from interrupt

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

init    NOP                       ; BANKSEL cannot be labelled
        BANKSEL TRISD             ; Select bank 1
        MOVLW   b'00000000'       ; Port B Direction Code
        MOVWF   TRISD             ; Load the DDR code into F86

; Initialise Timer0 .......................................

        MOVLW   b'11011000'       ; TMR0 initialisation code
        MOVWF   OPTREG            ; Int clock, no prescale
        BANKSEL PORTD             ; Select bank 0
        MOVLW   b'10100000'       ; INTCON init. code
        MOVWF   INTCON            ; Enable TMR0 interrupt

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

reset   CLRF    PORTD             ; Clear Port B Data

start   BTFSS   PORTB,1           ; Test reset button
        GOTO    reset             ; and reset Port B if pressed
        BTFSC   PORTB,2           ; Test step button
        GOTO    start             ; and repeat if not pressed

        CLRF    TMR0              ; Reset timer
wait    BTFSS   INTCON,2          ; Check for time out
        GOTO    wait              ; Wait if not
stepin  BTFSS   PORTB,2           ; Check step button
        GOTO    stepin            ; and wait until released
        INCF    PORTD             ; Increment output at Port B
        GOTO    start             ; Repeat main loop always

        END                       ; Terminate source code......
```

Program 4.2 Hardware timer debouncing



INTCON Register

Bit	7	6	5	4	3	2	1	0
	1	X	1	X	X	1	X	X

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

OPTION Register

Bit	7	6	5	4	3	2	1	0
	X	X	0	1	1	0	0	0

Bit	Timer Control Bit Label & Function		Prescaler Division Ratio 2	4	8	16	32	64	128	256
0	**PS0**	Prescaler Rate Select Bit 0	0	1	0	1	0	1	0	1
1	**PS1**	Prescaler Rate Select Bit 1	0	0	1	1	0	0	1	1
2	**PS2**	Prescaler Rate Select Bit 2	0	0	0	0	1	1	1	1

Bit	Timer Control Bit Label & Function		
3	**PSA**	Prescaler Assignment Bit	0 = Select Prescaler for TMR0 1 = Deselect Prescaler for TMR0
4	**T0SE**	TMR0 Source Edge Select Bit	0 = Increment on rising edge of RA4 1 = Increment on falling edge of RA4
5	**T0CS**	TMR0 Clock source Select Bit	0 = Instruction Clock = Ext Clock/4 1 = select RA4 Input

Figure 4.4 Timer0 operation

by default. If more than one interrupt source is enabled, the program must test the flags to see which is active, as part of the ISR.

In program LED1H, Timer0 is initialised to make a full count of 256 instruction cycles. At 10 kHz, this gives a delay of about 26 ms. This is more than enough for debouncing. The OPTION register is set up for the timer to operate from the instruction clock with no pre-scaling. The INTCON register is initialised to enable the timer interrupt. When the timer times out, the program jumps to the interrupt service routine at address 004, resets the time out flag, and returns. The program then waits for the button to be released, and increments the LED count.

Keypad Input

A keypad is simply an array of push buttons connected in rows and columns, so that each can be tested for closure with the minimum number of connections (Figure 4.5). There are 12 keys on a phone type pad (0–9, #, ∗), arranged in a 3×4 matrix. The columns are labelled 1, 2, 3 and the rows A, B, C, D. If we assume that all the rows and columns are initially high, a keystroke can be detected by setting each row low in turn and checking each column for a zero.

In the KEYPAD circuit in Figure 4.6, the 7 keypad pins are connected to Port D. Bits 4–7 are initialised as outputs, and bits 0–2 used as inputs. These input pins are pulled high to logic 1. The output rows are also initially set to 1. If a 0 is now output on row A, there is no effect on the inputs unless a button in row A is pressed. If these are checked in turn for a 0, a button in this row which is pressed can be identified as a specific combination of output and input bits.

A simple way to achieve this result is to increment a count of keys tested when each is checked, so that when a button is detected, the scan of the keyboard is terminated with current key number in the counter. This works because the (non-zero) numbers on the keypad arranged in order:

Row A = 1, 2, 3

Row B = 4, 5, 6

Row C = 7, 8, 9

Row D = *, 0, #

Following this system, the star symbol is represented by a count of 10 (0Ah), zero by 11(0Bh) and hash by 12 (0C).

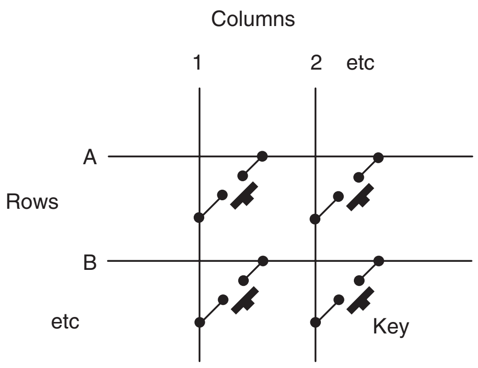


Figure 4.5 Keypad connections

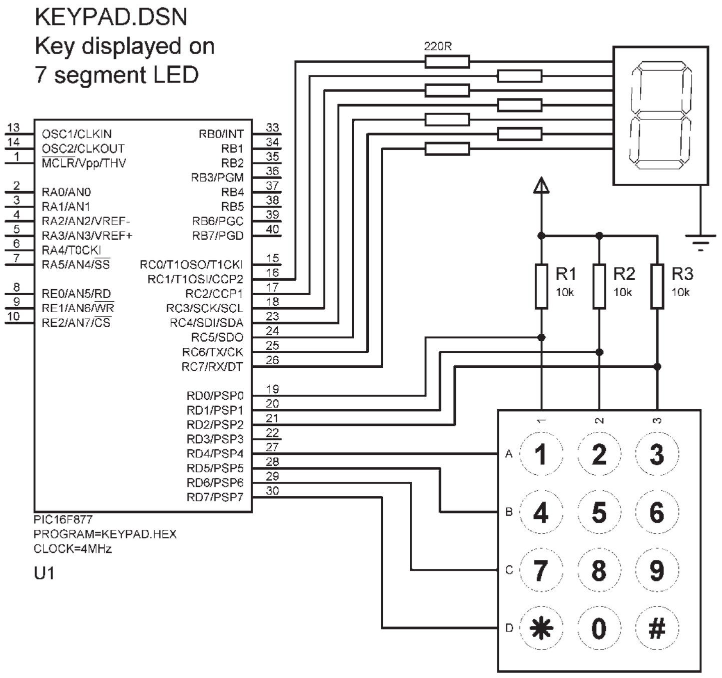


Figure 4.6 Keypad circuit

The keypad read operation (Program 4.3) steps through the buttons in this way, incrementing a key count, and quits the scanning routine when a button is detected, with the corresponding count of keys stored. If no button is pressed, it repeats. The program then displays the button number on a 7-segment display, with arbitrary symbols representing star and hash.

7-Segment LED Display

The standard 7-segment LED display in the keypad application consists of illuminated segments arranged to show numerical symbols when switched on in the appropriate combination. Each segment is driven separately from Port C via a current-limiting resistor. Numbers 0–9 can be displayed, but for a full range of alphanumeric characters, more segments (e.g. starburst LED) or a dot matrix display is more versatile. The codes for 0–9, ∗ and # are shown in Table 4.1.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;         KEYPAD.ASM          MPB  Ver 1.0 28-8-05
;
;         Reads keypad and shows digit on display
;         Design file KEYPAD.DSN
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
          PROCESSOR 16F877

PCL       EQU       002                 ; Program Counter
PORTC     EQU       007                 ; 7-Segment display
PORTD     EQU       008                 ; 3x4 keypad

TRISC     EQU       087                 ; Data direction
TRISD     EQU       088                 ; registers

Key       EQU       020                 ; Count of keys

; Initialise ports.......................................

          BANKSEL   TRISC               ; Display
          CLRW                          ; all outputs
          MOVWF     TRISC               ;
          MOVLW     B'00000111'         ; Keypad
          MOVWF     TRISD               ; bidirectional

          BANKSEL   PORTC               ; Display off
          CLRF      PORTC               ; initially
          GOTO      main                ; jump to main

; Check a row of keys ...................................

row       INCF      Key                 ; Count first key
          BTFSS     PORTD,0             ; Check key
          GOTO      found               ; and quit if on

          INCF      Key                 ; and repeat
          BTFSS     PORTD,1             ; for second
          GOTO      found               ; key

          INCF      Key                 ; and repeat
          BTFSS     PORTD,2             ; for third
          GOTO      found               ; key
          GOTO      next                ; go for next row

; Scan the keypad........................................

scan      CLRF      Key                 ; Zero key count
          BSF       3,0                 ; Set Carry Flag
          BCF       PORTD,4             ; Select first row
newrow    GOTO      row                 ; check row

next      BSF       PORTD,3             ; Set fill bit
          RLF       PORTD               ; Select next row
          BTFSC     3,0                 ; 0 into carry flag?
          GOTO      newrow              ; if not, next row
          GOTO      scan                ; if so, start again

found     RETURN                        ; quit with key count
; Display code table.....................................

table     MOVF      Key,W               ; Get key count
          ADDWF     PCL                 ; and calculate jump
          NOP                           ; into table
          RETLW     B'00001100'         ; Code for '1'
          RETLW     B'10110110'         ; Code for '2'
          RETLW     B'10011110'         ; Code for '3'
          RETLW     B'11001100'         ; Code for '4'
          RETLW     B'11011010'         ; Code for '5'
          RETLW     B'11111010'         ; Code for '6'
          RETLW     B'00001110'         ; Code for '7'
          RETLW     B'11111110'         ; Code for '8'
          RETLW     B'11001110'         ; Code for '9'
          RETLW     B'10010010'         ; Code for '*'
          RETLW     B'01111110'         ; Code for '0'
          RETLW     B'11101100'         ; Code for '#'
; Output display code....................................

show      CALL      table               ; Get display code
          MOVWF     PORTC               ; and show it
          RETURN
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Read keypad & display....

main      MOVLW     0FF                 ; Set all outputs
          MOVWF     PORTD               ; to keypad high
          CALL      scan                ; Get key number
          CALL      show                ; and dsiplay it
          GOTO      main                ; and repeat

          END       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Program 4.3 Keypad program

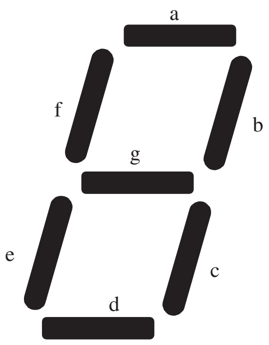
Key	Segment	Hex	Segment Labels
	g f e d c b a -	LSD=0	
1	0 0 0 0 1 1 0 0	0C	
2	1 0 1 1 0 1 1 0	B6	
3	1 0 0 1 1 1 1 0	9E	
4	1 1 0 0 1 1 0 0	CC	
5	1 1 0 1 1 0 1 0	DA	
6	1 1 1 1 1 0 1 0	FA	
7	0 0 0 0 1 1 1 0	0E	
8	1 1 1 1 1 1 1 0	FE	
9	1 1 0 0 1 1 1 0	CE	
#	1 1 1 0 1 1 0 0	EC	
0	0 1 1 1 1 1 1 0	7E	
*	1 0 0 1 0 0 1 0	92	

Table 4.1 7-Segment codes

The segments are labelled a–g, and are assumed to operate active high (1 = ON). The binary code required must then be worked out for each character to be displayed, depending on the order in which the outputs are connected to the segments. In this case, bit 1 = a, through to bit 7 = g, with bit 0 not used. Hash is displayed as 'H' and star as three horizontal bars (S is already used for 5). As only 7 bits are needed, the LSB is assumed to be 0 when converting into hexadecimal. In any case, it is preferable to put the binary code in the program. Codes for other types of display can be worked out in the same way.

The PIC output provides enough current to drive an LED (unlike standard logic outputs), but for a display element requiring more than 20 mA to operate, additional current drive must be added to the hardware, usually in the form of a bipolar transistor. This interface will be covered later. An alternative to the plain 7-segment display is a binary code decimal (BCD) module; this receives BCD input and displays the corresponding number. In BCD $0 = 0000_2$, $1 = 0001_2$ and so on to $9 = 1001_2$, and therefore only needs 4 inputs (plus a common terminal).

These displays are usually provided with one common terminal, connected to the anodes or cathodes of the LEDs. An active high display will have a common cathode and individual anodes, an active low-type (ON = 0) will have a common anode.

Liquid Crystal Display

The LCD is now a very common choice for graphical and alphanumeric displays. These range from small, 7-segment monochrome numerical types such as those used in digital multimeters (typically 3 ½ digits, maximum reading 1.999) to large, full colour, high-resolution screens which can display full video. Here we shall concentrate on the small monochrome, alphanumeric type which displays alphabetical, numerical and symbolic characters from the standard ASCII character set. This type can also display low-resolution graphics, but we will stick to simple examples. A circuit is shown in Figure 4.7.

The display is a standard LM016L which displays 2 lines of 16 characters (16×2). Each character is 5×8 pixels, making it 80×16 pixels overall. In the demo program, a fixed message is displayed on line 1, showing all the numerical digits. The second line finishes with a character that counts up from 0 to 9 and repeats, to demonstrate a variable display. The display receives ASCII codes for each character at the data inputs (D0–D7). The data is presented to the display inputs by the MCU, and latched in by pulsing the E (Enable) input. The RW (Read/Write) line can be tied low (write mode), as the LCD is receiving data only.

The RS (Register Select) input allows commands to be sent to the display. RS=0 selects command mode, RS=1 data mode. The display itself contains a microcontroller; the standard chip in this type of display is the Hitachi HD44780. It must be initialised according to the data and display options required. In this example, the data is being sent in 4-bit mode. The 8-bit code for

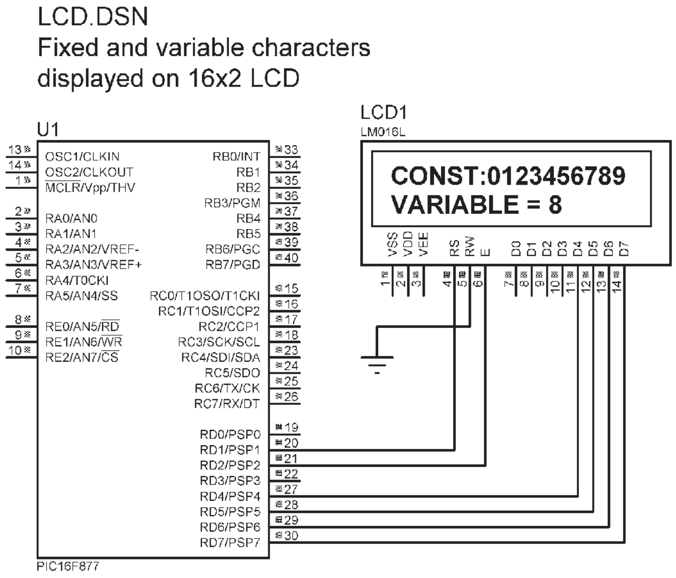


Figure 4.7 LCD display connections

each ASCII character is sent in two halves; high nibble (a small byte!) first, low nibble second. This saves on I/O pins and allows the LCD to be driven using only 6 lines of a single port, while making the software only slightly more complex. The command set for the display controller is shown in Table 4.2.

If we now turn to the demonstration program, we can interpret the commands sent during the display initialisation sequence, by comparing the hex codes with the binary commands. It can be seen that the display must be initially set to default operating mode, before selecting the required mode (4-bit, 2 lines), and resetting. Note that the commands are differentiated by the number of leading zeros (Table 4.3) (Program 4.4).

We will analyse the LCD program in detail as it contains a number of features which we will see again. In order to facilitate this, a pseudocode outline is given in Figure 4.8

(a) Commands

Instruction	Code	Description
Clear display	`0000 0001`	Clear display and reset address
Home cursor	`0000 001x`	Reset display location address
Entry mode	`0000 01MS`	Set cursor move and display shift
Display control	`0000 1DCB`	Display & cursor enable
Shift control	`0001 PRxx`	Moves cursor and shifts display
Function control	`001L NFxx`	Data mode, line number, font
CGRAM address	`01gg gggg`	Send character generator RAM address
DDRAM address	`1ddd dddd`	Send display data RAM address

X	Don’t care
M	Cursor move direction 1 = right 0 = left
S	Enable whole display shift = 1
D	Whole display on = 1
C	Cursor on = 1
B	Blinking cursor on = 1
P	Display shift = 1, cursor move = 0
R	Shift right = 1, shift left = 0
L	8-Bits = 1, 4-bits = 0
N	2 Lines = 1, 1 line = 0
F	5 × 10 character = 1, 5 × 8 = 0
g	Character generator RAM address bit
d	Data RAM address bit

(b) Character addresses (16 × 2 display)

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F

Table 4.2 LCD operation

Hex	Binary	Type	Meaning
32	0011 0010	Function control	8-bit data, 1 line, 5×8 character
28	0010 1000	Function control	4-bit data, 2 lines, 5×8 character
0C	0000 1100	Display control	Enable display, cursor off, blink off
06	0000 0110	Entry mode	Cursor auto-increment right, shift off
01	0000 0001	Clear display	Clear all characters
80	1000 0000	DDRAM address	Reset display memory address to 00

Table 4.3 LCD initialisation command code sequence

The program has three main processes:

- Output line 1 fixed message ‘CONST:0123456789’
- Output line 2 fixed message ‘VARIABLE =’
- Output variable count 0-9 at line 2, position 12

The main program (last in the source code list) is very short, comprising:

- Initialise the MCU and LCD
- Output fixed messages
- Output count

The program is divided into functional blocks accordingly. Note that standard register labels are defined by including the standard file P16F877.INC, which contains a list of all labels for the SFRs and control bits, for example, PORTD, STATUS, Z. This is more convenient than having to declare them in each program, and the include files supplied by Microchip define a standard set of labels which all programmers can use.

To send the data and commands to the display, the output data is initially masked so that only the high nibble is sent. The low bits are cleared. However, since the low bits control the display (RS and E), these have to be set up after the data have been output in the port high bits. In particular, an RS flag bit is set up in a dummy register ‘Select’ to indicate whether the current output is command or data, and copied to RD1 after the data set-up.

After each output, a 1 ms delay is executed to allow the LCD controller time to process the input and display it. An exact timing loop (Onems) is achieved by padding the delay loop to 4 cycles with a NOP, and executing it 249 times. With the additional instructions and subroutine jumps, the delay is exactly $250 \times 4 = 1000$ µs. This is then used by another loop (Xms) to obtain delays in whole milliseconds. It is also used to generate a 1 ms pulse at E to latch the data and commands into the LCD controller input port.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;       LCD.ASM         MPB   4-12-05
;
;       Outputs fixed and variable characters
;       to 16x2 LCD in 4-bit mode
;
;       Version 2.0: Initialisation modified
;       Status: Tested OK in simulation mode
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        PROCESSOR 16F877
;       Clock = XT 4MHz, standard fuse settings
        __CONFIG 0x3731

; LABEL EQUATES ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        INCLUDE "P16F877.INC"   ; Standard labels

Timer1  EQU     20              ; 1ms count register
TimerX  EQU     21              ; Xms count register
Var     EQU     22              ; Output variable
Point   EQU     23              ; Program table pointer
Select  EQU     24              ; Copy of RS bit
OutCod  EQU     25              ; Temp store for output

RS      EQU     1               ; Register select bit
E       EQU     2               ; Display enable

; PROGRAM BEGINS ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        ORG     0               ; Place machine code
        NOP                     ; for ICD mode

        BANKSEL TRISD           ; Select bank 1
        CLRW                    ; All outputs
        MOVWF   TRISD           ; Initialise display port
        BANKSEL PORTD           ; Select bank 0
        CLRF    PORTD           ; Clear display outputs

        GOTO    Start           ; Jump to main program
; SUBROUTINES ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; 1ms delay with 1us cycle time (1000 cycles)-------------------

Onems   MOVLW   D'249'          ; Count for 1ms delay
        MOVWF   Timer1          ; Load count
Loop1   NOP                     ; Pad for 4 cycle loop
        DECFSZ  Timer1          ; Count
        GOTO    Loop1           ; until Z
        RETURN                  ; and finish
; Delay Xms, X received in W -----------------------------------

Xms     MOVWF   TimerX          ; Count for X ms
LoopX   CALL    Onems           ; Delay 1ms
        DECFSZ  TimerX          ; Repeat X times
        GOTO    LoopX           ; until Z
        RETURN                  ; and finish
; Generate data/command clock siganl E -------------------------

PulseE  BSF     PORTD,E         ; Set E high
        CALL    Onems           ; Delay 1ms
        BCF     PORTD,E         ; Reset E low
        CALL    Onems           ; Delay 1ms
        RETURN                  ; done
; Send a command byte in two nibbles from RB4 - RB7 ------------

Send    MOVWF   OutCod          ; Store output code
        ANDLW   0F0             ; Clear low nybble
        MOVWF   PORTD           ; Output high nybble
        BTFSC   Select,RS       ; Test RS bit
        BSF     PORTD,RS        ; and set for data
        CALL    PulseE          ; and clock display
        CALL    Onems           ; wait 1ms for display
```

Program 4.4 LCD source code

```
        SWAPF   OutCod          ; Swap low/high nybbles
        MOVF    OutCod,W        ; Retrieve output code
        ANDLW   0F0             ; Clear low nybble
        MOVWF   PORTD           ; Output low nybble
        BTFSC   Select,RS       ; Test RS bit
        BSF     PORTD,RS        ; and set for data
        CALL    PulseE          ; and clock display
        CALL    Onems           ; wait 1ms for display
        RETURN                  ; done

; Table of fixed characters to send ---------------------------

Line1   ADDWF   PCL             ; Modify program counter
        RETLW   'C'             ; Pointer = 0
        RETLW   'O'             ; Pointer = 1
        RETLW   'N'             ; Pointer = 2
        RETLW   'S'             ; Pointer = 3
        RETLW   'T'             ; Pointer = 4
        RETLW   ':'             ; Pointer = 5
        RETLW   '0'             ; Pointer = 6
        RETLW   '1'             ; Pointer = 7
        RETLW   '2'             ; Pointer = 8
        RETLW   '3'             ; Pointer = 9
        RETLW   '4'             ; Pointer = 10
        RETLW   '5'             ; Pointer = 11
        RETLW   '6'             ; Pointer = 12
        RETLW   '7'             ; Pointer = 13
        RETLW   '8'             ; Pointer = 14
        RETLW   '9'             ; Pointer = 15

Line2   ADDWF   PCL             ; Modify program counter
        RETLW   'V'             ; Pointer = 0
        RETLW   'A'             ; Pointer = 1
        RETLW   'R'             ; Pointer = 2
        RETLW   'I'             ; Pointer = 3
        RETLW   'A'             ; Pointer = 4
        RETLW   'B'             ; Pointer = 5
        RETLW   'L'             ; Pointer = 6
        RETLW   'E'             ; Pointer = 7
        RETLW   ' '             ; Pointer = 8
        RETLW   '='             ; Pointer = 9
        RETLW   ' '             ; Pointer = 10

; Initialise the display ---------------------------------------

Init    MOVLW   D'100'          ; Load count 100ms delay
        CALL    Xms             ; and wait for display
        MOVLW   0F0             ; Mask for select code
        MOVWF   Select          ; High nybble not masked

        MOVLW   0x30            ; Load initial nibble
        MOVWF   PORTD           ; and output it to display
        CALL    PulseE          ; Latch initial code
        MOVLW   D'5'            ; Set delay 5ms
        CALL    Xms             ; and wait
        CALL    PulseE          ; Latch initial code again
        CALL    Onems           ; Wait 1ms
        CALL    PulseE          ; Latch initial code again
        BCF     PORTD,4         ; Set 4-bit mode
        CALL    PulseE          ; Latch it

        MOVLW   0x28            ; Set 4-bit mode, 2 lines
        CALL    Send            ; and send code
        MOVLW   0x08            ; Switch off display
        CALL    Send            ; and send code
        MOVLW   0x01            ; Clear display
        CALL    Send            ; and send code
        MOVLW   0x06            ; Enable cursor auto inc
        CALL    Send            ; and send code
        MOVLW   0x80            ; Zero display address
        CALL    Send            ; and send code
        MOVLW   0x0C            ; Turn on display
        CALL    Send            ; and send code

        RETURN                  ; Done

; Send the fixed message to the display ------------------------

OutMes  CLRF    Point           ; Reset table pointer
        BSF     Select,RS       ; Select data mode
```

Program 4.4 *Continued*

```
Mess1   MOVF    Point,W         ; and load it
        CALL    Line1           ; Get ASCII code from table
        CALL    Send            ; and do it
        INCF    Point           ; point to next character
        MOVF    Point,W         ; and load the pointer
        SUBLW   D'16'           ; check for last table item
        BTFSS   STATUS,Z        ; and finish if 16 done
        GOTO    Mess1           ; Output character code

        MOVLW   0xC0            ; Move cursor to line 2
        BCF     Select,RS       ; Select command mode
        CALL    Send            ; and send code
        CLRF    Point           ; Reset table pointer
Mess2   MOVF    Point,W         ; and load it
        CALL    Line2           ; Get fixed character
        BSF     Select,RS       ; Select data mode
        CALL    Send            ; and send code
        INCF    Point           ; next character
        MOVF    Point,W         ; Reload pointer
        SUBLW   D'11'           ; and check for last
        BTFSS   STATUS,Z        ; Skip if last
        GOTO    Mess2           ; or send next
        RETURN                  ; done

; Output variable count to display (0-9) endlessly ------------

OutVar  CLRF    Var             ; Clear variable number
        MOVLW   0X30            ; Load offset to be added
        ADDWF   Var             ; to make ASCII code (30-39)

Next    MOVF    Var,W           ; Load the code
        BSF     Select,RS       ; Select data mode
        CALL    Send            ; and send code

        MOVLW   0xCB            ; code to move cursor back
        BCF     Select,RS       ; Select command mode
        CALL    Send            ; and send code
        MOVLW   D'250'          ; Load count to wait 250ms
        CALL    Xms             ; so numbers are visible

        INCF    Var             ; Next number
        MOVF    Var,W           ; Load number
        SUBLW   0x3A            ; Check for last (10=A)
        BTFSS   STATUS,Z        ; and skip if last
        GOTO    Next            ; or do next number
        GOTO    OutVar          ; Repeat from number Z


; MAIN PROGRAM ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

Start   CALL    Init            ; Initialise the display
        CALL    OutMes          ; Display fixed characters
        GOTO    OutVar          ; Display an endless count

        END                     ; of source code ;;;;;;;;
```

Program 4.4 *Continued*

The fixed messages are generated from program data tables using ADDWF PCL to jump into the table using the number in W, and RETLW to return the ASCII codes. The assembler generates the ASCII code in response to the single quotes which enclose the character. To fetch the required code, the table pointer is added to the program counter at the top of the table. The table pointer is also checked each time to see the end of the table has been reached.

The output variable is just a count from 0 to 9, but to obtain the corresponding ASCII code, 30h must be added, because the ASCII for 0 is 30h, for 1 is 31h and so on. The ASCII code table is shown in Table 4.4. A 250 ms delay is executed between each output to make the count visible.

```
Project LCD
Program to demonstrate fixed and
variable output of alphanumeric characters
 to 16x2 LCD (simulation only)

HARDWARE
    ISIS simulation file LCD.DSN
    MCU    16F877A
           Clock = XT 4MHz
    LCD    Data = RD4 – RD7
           RS = RD1
           E = RD2
           RW = 0

FIRMWARE

    Initialise
        LCD output = Port D
        Wait 100ms for LCD to start
        LCD: 4-bit data, 2 lines, auto cursor
        Reset LCD

    Display message line 1
        Reset table pointer
        REPEAT
            Get next code
            Send ASCII code
        UNTIL 16 characters done

    Display message line 2
        Position cursor
        Reset table pointer
        REPEAT
            Get next code
            Send ASCII code
        UNTIL 11 characters done

    Display incrementing count
        REPEAT
            Set Count = 0
                LOOP
                    Calculate ASCII
                    Send ASCII code
                    Increment Count
                    Reset cursor
                    Delay 250ms
                UNTIL Count = 9
        ALWAYS

    Send ASCII code
        Mask low nibble
        Output high nibble
        Pulse E
        Wait 1ms
        Swap nibbles
        Output low nibble
        Pulse E
        Wait 1ms
        Return
```

Figure 4.8 LCD program outline

			High Bit			
Low Bits	0010	0011	0100	0101	0110	0111
0000	Space	0	@	P	`	p
0001	!	1	A	Q	a	q
0010	“	2	B	R	b	r
0011	#	3	C	S	c	s
0100	$	4	D	T	d	t
0101	%	5	E	U	e	u
0110	&	6	F	V	f	v
0111	‘	7	G	W	g	w
1000	(	8	H	X	h	x
1001	)	9	I	Y	i	y
1010	*	:	J	Z	j	z
1011	+	;	K	[	k	{
1100	,	<	L	\	l	|
1101	-	=	M	]	m	}
1110	.	>	N	∧	n	~
1111	/	?	O	–	o	

Table 4.4 ASCII character codes

The primary set of characters consists of upper and lower case letters, numbers and most of the other characters found on a standard keyboard. This comes to 95 characters, represented by codes from 32_{10} to 126_{10} inclusive. With an 8-bit code, 256 characters are possible, so the remaining ones are used for special language character sets, for example accented characters, Greek, Arabic or Chinese. The character set can be programmed into the display RAM as required, or ordered as a ROM implementation. Further details can be obtained from the display data sheet.

SUMMARY 4

- Switch inputs generally use pull-ups and debouncing
- The hardware timer operates independently of the program
- The keypad is a set of switches scanned in rows and columns
- 7-segment LEDs are the simplest form of alphanumeric display
- The alphanumeric LCD receives ASCII codes for display in rows
- The LCD also needs command codes for control operations

ASSESSMENT 4

1	Explain why a pull-up resistor needed with a switch input.	*(3)*
2	State three methods of switch debouncing.	*(3)*
3	Briefly explain why hardware timers are useful in MCUs.	*(3)*
4	Explain how a pre-scaler extends the timer period.	*(3)*
5	Explain why the plain 7-segment LED display needs a code table.	*(3)*
6	Explain why a BCD encoded display does not need a code table.	*(3)*
7	Briefly explain the scanning process used to read a keypad button.	*(3)*
8	Explain briefly how the LM016L LCD can be driven with only six outputs.	*(3)*
9	State the function of inputs RS and E in the LM016L LCD.	(3)
10	State the ASCII code for '#' in hexadecimal.	*(3)*
11	Draw a block diagram of the circuit with a keypad and 7-segment display, indicating the main components and signals in the system.	*(5)*
12	Describe the software precautions needed to obtain correct operation of the LCD in 4-bit mode when the same port is used for control and data connections.	*(5)*

ASSIGNMENTS 4

4.1 Keypad Test

Run the keypad system simulation, KEYPAD.DSN. Ensure that the correct display is obtained when the keypad is operated. Represent the program using a flowchart and pseudocode. Explain the advantages of each. Explain why switch debouncing is not necessary in this particular application.

4.2 LCD Test

Run the LCD system simulation, LCD.DSN. Ensure that correct operation is obtained; fixed messages should be followed by an incrementing count. Enable debug mode by hitting pause instead of run, and select PIC CPU Source Code in the debug menu; also select PIC CPU Registers. Ensure that the debug windows are opened.

Single step in the source code window using the ‘step into’ button. Follow through the initialisation until the delay routine is entered. Why is the ‘step out’ button now useful? Start again using ‘step over’ – note the effect. Why is ‘step over’ useful?

Single step through the output sequence, displaying the fixed characters one at a time, then the count. Note the codes output at Port D and check them against the ASCII table given.

4.3 LCD Control

Construct a timing diagram for the output sequence obtained to the LCD in LCD.DSN. for one complete character, by recording the changes at Port D, and referring to the simulation timer. Show traces for all six outputs. Obtain a timing diagram using the simulation graph feature, and check that this is consistent with your manually constructed version.

5

Data Processing

Most microcontroller programs need to process data using arithmetic or logical operations. The main types of data in general processor systems are numbers and characters. Characters are not too much of a problem as there are only 26 letters in the alphabet and 10 numerical characters. Even allowing for upper and lower case, that is only 62 codes are required. The basic ASCII character set thus only requires a 7-bit code, which also provides for most other keyboard symbols (see previous chapter for table of ASCII codes). Use of the eighth bit allows special character sets to be added.

Numerical data is a bit more of a problem, as an 8-bit code can only represent numbers 0–255; so some additional methods are needed to handle larger (and smaller) numbers, so that calculations can be performed with a useful degree of precision.

Number Systems

Computers work with binary numbers, but humans prefer to work in decimal. However, these number systems both follow the same basic rules:

1. Select a range of digit symbols to use (e.g. 10 symbols in decimal)
2. Count from zero up to the maximum value in single digits (0–9 in decimal)

3. Increment the digit to the left of the first digit (to 1)
4. Reset the previous column to zero
5. Count up again in the units column to the maximum
6. Increment the next digit again
7. When each column is at the maximum value, increment the next and reset
8. Use more columns as necessary for larger numbers

Hopefully we all know how to count in decimal (denary is the official name). The base of the number system is the number of digits used (10). Binary is base 2, hex base 16, octal base 8. Any base can be use in theory, but in practice some are more useful than others.

Historically, base 12 has been used extensively (hours, minutes, angles, old English money), and is useful because it can be divided by 2, 3, 4 and 6. But this is not a true number system because there are no discrete symbols for 10 and 11. Similarly, BCD is not a proper number system (see later for details) because its binary count stops at 9. Hexadecimal is a true number system because it uses discrete symbols for 10(A) – 15(F). It is useful because it provides a compact way of representing binary, the native number system of all digital computers and controllers.

Denary

Zero was a very important invention, because number systems depend on it. Another important idea is column weighting, and you can see the significance by simply analysing how the denary system works (Table 5.1).

The number is calculated as follows:

Total value $= \Sigma$ (column weighting $\times$ digit value) [$\Sigma =$ sum of]

This may seem to be obvious, but it is important to state it clearly, as we will be implementing numerical processing which uses this structure. Denary is also the reference system, that is other systems are described by reference to it.

Column weight	$1000 = 10^3$	$100 = 10^2$	$10 = 10^1$	$1 = 10^0$
Example	**7**	**3**	**9**	**5**
Value	7×10^3	3×10^2	9×10^1	5×10^0
Total value	7000 + 300 + 90 + 5 = **7395**			

Table 5.1 Structure of a denary number

Binary

Computers work in binary because it was found to be the most efficient and precise way to represent numbers electronically. Before the necessary digital hardware was developed, computers were used in which analogue voltages represented denary values. Linear amplifier (op-amp) circuits provided the processing, and these are well suited to mathematical processes such as integration and differentiation, but the accuracy was limited by the signal quality. On the other hand, the accuracy of the digital computer can be increased (in theory) to any required degree of precision by simply increasing the number of binary digits. Processing in 32 bits (now common), for example, provides a potential degree of precision of 1 part in 2^{32} (4294967296), and even a modest 8 bits gives an accuracy of 1 part in 256, better than 0.5% error at full scale (Table 5.2).

Column weight	$8 = 2^3$	$4 = 2^2$	$2 = 2^1$	$1 = 2^0$
Example	**1**	**0**	**0**	**1**
Value	$1 \times 8 = 8$	$0 \times 4 = 0$	$0 \times 2 = 0$	$1 \times 1 = 1$
Total value	$8 + 0 + 0 + 1 = 9$			

Table 5.2 Structure of a binary number

The result of the analysis of the structure of a typical binary number shows that the decimal value can be worked out as follows:

Total value $= \Sigma$ (column weight of non-zero bits)

Notice the pattern of the column weightings – it is the base number to the power 0, 1, 2, 3, etc. This is the meaning of the base number.

The maximum number for a given number of bits is obtained when all the bits are 1. In a 8-bit number, the maximum value is $11111111_2 = 255_{10}$ (the subscript indicates the number base). This is calculated as 2^8-1, that is, two to the power of the base minus 1.

The number of different codes, including zero, is $2^8 = 256$. This is important in defining memory capacity, where an extra bit on the address doubles the memory. Important reference points are $2^{10} = 1024$ bytes (1 kb), 2^{16} (64 kb), 2^{20} (1 Mb) and 2^{30} (1 Gb). The highest address in a 1 kb memory, for example, is 1023.

Hexadecimal

The same principle is applied to the number system with base 16. The problem here was that extra numerical symbols were required, so symbols which are

Example	**9**	**B**	**0**	**F**
Column weight	$1000_{16} = 4096 = 16^3$	$100_{16} = 256 = 16^2$	$10_{16} = 16 = 16^1$	$1_{16} = 1 = 16^0$
Value	$9 \times 4096 = 36864$	$B = 11 \times 256 = 2816$	$0 \times 16 = 0$	$F = 15 \times 1 = 15$
Total value	36864 + 2816 + 0 + 15 = **39695**			

Table 5.3 Structure of a hex number

normally used as letters were simply adopted as numbers: $A_{16} = 10_{10}$ to $F_{16} = 15_{10}$ (Table 5.3).

Note the pattern in the progression of the hex column weight – the weighting is 0_{16}, 10_{16}, 100_{16}, 1000_{16}, etc. This will apply to all number systems – the column weight is a progression of 0_n, 10_n, 100_n, 1000_n, etc where *n* is the base. It can also be seen that the conversion from hex to denary is not simple, but the conversion from hex to binary is, which is why hex is useful.

Other Number Systems

Numbers can be represented using any base by following the rules outlined above. Octal (base 8) is sometimes used in computing, but will not be considered here. Numbers with a base greater than 16 would need additional symbols, so, in theory, one could carry on using letters up to Z.

Binary Coded Decimal

As mentioned previously, BCD is not a proper number system, but it is useful as an intermediate system between binary and decimal. In BCD, the numbers 0–9 are represented by their binary equivalent, and stored as 4-bit numbers. These may then be converted into ASCII code to send to a display (e.g. the alphanumeric LCD seen later) or into pure binary for processing. Alternatively, the data can be processed in BCD by using appropriate algorithms (Table 5.4).

By studying these examples, general rules for BCD processing can be deduced. When adding two single digit numbers (Table 5.5), the first (Num1) incremented and the second (Num2) decremented. If the Num2 reaches zero, Num1 can be stored as the single digit result. However, if Num1 reaches 10, Num2 is stored as the least significant digit of the result, and the MSD set to 1.

This type of process can be devised and implemented for all arithmetic operations. The advantage is that the results can be input from a keyboard as BCD, and output as ASCII without conversion. However, if the arithmetic is

Calculation	Add	Add with carry	Multiply
In decimal	$2 + 3 = 5$	$7 + 8 = 15$	$3 \times 7 = 21$
In BCD	0010	$(7 + 3) + 5$	$7 + 7 + 7$
	$+ \underline{0011}$	$= 10 + 5$	$= (7 + 3) + (4 + 6) + 1$
	$= 0101$	$= 15$	$= 21$

Table 5.4 BCD calculations

BCDADD

Process to add two single digit BCD numbers with one or two digit result

```
Load Num1, Num2                          ; Get initial values
Clear MSD, LSD                           ; Assign two digit store for result

REPEAT
        Increment Num1                   ; Start adding
        Decrement Num2                   ; Adjust number being added

        IF Num2 = 0                      ; Single digit result
          LSD = Num1                     ; Store result
          RETURN                         ; and quit

UNTIL Num1 = 10                          ; Check for carry out of LSD

LSD = Num2                               ; Two digit result
MSD = 1                                  ; Result of carry out
RETURN                                   ; done
```

Table 5.5 Process for BCD add with carry

more complex, conversion into pure binary or floating point (FP) format may be necessary.

Floating Point Formats

Numbers have a limited range in the context of computer storage and processing, depending on how they are represented and stored. The number of bytes which will be allocated to store a number must be set up in advance of running a program. For example, a plain 16-bit binary number ranges from 0 to 65535 ($2^{16}-1$), and requires 2 bytes oxf memory. Since negative numbers are usually required, the MSB = 1 may be used to indicate a negative integer, leaving only 15 bits for the number. The range is then from − 32767 to + 32767.

However, as numbers get bigger (and smaller), the number of bits needed will be excessive.

An alternative format is needed to represent large and small numbers. On a calculator, scientific notation is used, base 10. For example, 2.3615×10^{73} is a large number, 6.9248×10^{-23} is a small one. The decimal part is called the mantissa and the range multiplier the exponent.

Computers work in base 2, so an example of a large negative number could be $-1.011001010 \times 2^{10011}$, and a small positive one $1.0101001101 \times 2^{-01001}$. We have to decide on the number of bits to use, which in turn determines the precision and range of the number itself. 32-bit numbers are sufficient for most purposes, but the bits available must be allocated in groups as mantissa, sign, exponent and exponent sign. This gives the FP numerical type (meaning the decimal point can move according to the exponent). Standard forms use 1 bit to represent the sign, 23 bits for the mantissa and 8 bits for the exponent and its sign. To illustrate the form of a floating-point number, the conversion of this type into decimal will be detailed in section 'Floating Point'.

Conversion

Conversion between numerical types is often required in microprocessor systems. Data may be input in ASCII, processed in binary or FP format, and output in BCD. Machine code is normally displayed in hexadecimal, since binary is cumbersome, so we need to know how to perform this conversion. FP formats are needed to extend the range and precision of numerical data.

Binary to Decimal

The structure of binary numbers has been described above. The value of a number is found by multiplying each digit by its decimal column weight and adding. The weighting of the digits in binary is (from the LSB) 1, 2, 4, 8, 16 … or 2^0, 2^1, 2^2, 2^3 … that is, the base of the number system is raised to the power 1, 2, 3 … The conversion process for a sample 8-bit binary number is therefore:

$$\begin{aligned} 1001\ 0110_2 &= (128 \times 1) + (64 \times 0) + (32 \times 0) + (16 \times 1) \\ &\quad + (8 \times 0) + (4 \times 1) + (2 \times 1) + (1 \times 0) \\ &= 128 + 16 + 4 + 2 \\ &= 150_{10} \end{aligned}$$

We can see that the process can be simplified to just adding the column weight for the bits that are not zero.

Decimal to Binary

The process is reversed for conversion from decimal to binary. The binary number is divided by two, the remainder recorded as a digit, and the result divided by two again, until the result is zero. For the same number:

```
150/2   =    75     rem     0(LSB)
75/2    =    37     rem     1
37/2    =    18     rem     1
18/2    =    9      rem     0
9/2     =    4      rem     1
4/2     =    2      rem     0
2/2     =    1      rem     0
1/2     =    0      rem     1(MSB)
```

We then see that the binary result is obtained by transcribing the column of remainder bits from the bottom up (MSB to LSB).

Binary and Hex

Binary to hex conversion is simple – that is why hex is used. Each group of four bits are converted into the corresponding hex digit, starting with the least significant four, and padding with leading zeros if necessary:

```
1001   1111   0011    1101     =   9F3D16
  9      F      3       D
```

The reverse process is just as trivial, where each hex digit is converted into a group of four bits, in order.

The result can be checked by converting both into decimal. First binary to decimal:

```
Bit    15 14 13 12 11 10 9  8  7  6  5  4  3  2  1  0
       1  0  0  1  1  1  1  1  0  0  1  1  1  1  0  1
```

$$= 2^{15} + 2^{12} + 2^{11} + 2^{10} + 2^{9} + 2^{8} + 2^{5} + 2^{4} + 2^{3} + 2^{2} + 2^{0}$$

$$= 32768 + 4096 + 2048 + 1024 + 512 + 256 + 32 + 16 + 8 + 4 + 1$$

$$= \quad = \quad 40765_{10}$$

Now hex to decimal:

$$
\begin{aligned}
& \mathbf{9F3D_{16}} \\
= \quad & (9 \times 16^3) + (15 \times 16^2) \\
& + (3 \times 16^1) + (13 \times 16^0) \\
= \quad & 36864 + 3840 + 48 + 13 \\
= \quad & \mathbf{40765_{10}}
\end{aligned}
$$

Binary and BCD

This conversion is often needed in MCU systems, since the manual input and displayed output tend to use BCD (or ASCII, which can be derived from it), while the internal calculations are performed in binary.

The input from a numeric keypad may often be in BCD, that is, binary numbers 0–9 representing each key. When multidigit numbers are input, the keys are pressed in the sequence from the highest significant digit to lowest. This sequence must be converted into the corresponding binary after the sequence is complete, typically by pressing an enter key. The decimal input number may have any number of digits, from zero to the maximum allowed by the binary number to which it is converted.

Let us assume the system handles 16-bit positive integers only; the range will be $0-65535_{10}$. We will therefore limit the input to four digits, with a maximum of 9999_{10}. The key inputs will be stored in temporary registers, and then converted into the equivalent 16-bit binary when an enter key is pressed.

The process will have to detect if four, or fewer, digits have been entered. It must then add the lowest digit (last key) to a previously cleared register pair (2×8 bits), multiply the next digit by 10, add the result to the running total, multiply the next by 100, add it to the total, and multiply the highest digit (first key) by 1000, and add it to the total. It is described in Figure 5.1 for 4-digit input, but this process can be extended as far as the integer size allows.

A set of registers is assigned and cleared, and a keypad scanning routine reads in keys as 4-bit BCD codes stored in the low nibble of the BCD registers. The codes are shifted after each input stroke, from BCD3 to BCD4, BCD2 to BCD3, BCD1 to BCD2 and BCD0 to BCD1, to allow the next input digit to be stored in BCD0. A maximum of four digits are stored in BCD4–BCD1 as result. If return code is detected as input before 4 keys have been entered, the loop quits with the digits in the correct registers, with the leading digits left at zero.

If a 12-button telephone style keypad is used, ‘*’ could be used as return (enter), and ‘#’ to restart the input sequence (clear). These could be assigned codes A_{16} and B_{16}, with the keys 0–9 assigned the corresponding BCD code. The digits are then multiplied by their digit weighting and added to a running

```
BCDTOBIN

Converts 4 digits BCD keypad input into 16-bit binary
Inputs: Up to 4 BCD codes 0 – 9
Output: 16-bit binary code

Declare Registers

          BCD0,BCD1,BCD2,BCD3,BCD4                     ; BCD digits
          BINHI,BINLO                                  ; 16 bit result
          Keycount                                     ; Count of keys
          Clear all registers                          ; Result = 0000

Read in BCD digits from keypad

          REPEAT
                    Read number key into BCD0          ; Get button
                    Shift all BCD digits left          ; Store it
                    Increment Keycount                 ; How many?
          UNTIL Return OR Keycount = 4                 ; Done

Calculate binary

          Add BCD1 to BINLO                            ; Add ones (max 9)
          Multiply BCD2 by 10                          ; Calc tens
          Add BCD2 to BINLO                            ; Add tens (max 99
          Multiply BCD3 by 100                         ; Calc hundreds
          Add BCD3 to BINHI+BINLO                      ; Add huns. (max 999)
          Multiply BCD4 by 1000                        ; Calc thousands
          Add BCD4 to BINHI+BINLO                      ; Add thous. (max 9999)

RETURN
```

Figure 5.1 BCD to binary conversion

total in a pair of 8-bit registers. The multiplication can be implemented by a simple adding loop, or shifting and adding if speed is important (see below). Note that the calculated subtotals must be added in low byte, high byte order, and the carry flag handled to obtain the correct 16-bit total.

Binary to BCD conversion for output may be implemented as the inverse process: divide by 1000, 100 and 10 and store the results as BCD digits. The last remainder is the units digit. These processes are illustrated in the calculator program in Chapter 6.

BCD and ASCII

The ASCII code for ‘0’ is 30h, the code for ‘1’ is 31h and so on until up to 39h for ‘9’. Therefore to convert the BCD or binary code for a number into ASCII, add 30h. To convert ASCII into BCD, subtract 30h. This process is used to display BCD data on an LCD display which receives characters as ASCII code, as in the calculator program.

Variable Types

In high-level languages, such as C, the variables to be used in a program must be declared in advance, and the correct storage space allocated before running the program. In PIC programs, all data locations (GPRs) are 8 bits, so they will need to be used in groups to represent 16-bit (2 bytes) or even 32-bit (4 bytes) numbers. In assembly language, these registers can be assigned using label equates at the top of the program.

Integer

An 8-bit location can only store numbers from 0 to 255 in binary. This is an obvious limitation in programs that may need to calculate results up to say, four significant decimal digits (0–9999), with negative as well as positive numbers. 16-bit integers can represent decimal values from $+32767$ to -32767, which cover this range comfortably. The main problem with 16-bit calculations is that the carry/borrow bit must be handled correctly during addition and subtraction to give the right result. Once this is achieved, multiplication and division can be implemented. Long integers use 32-bits for a greater range.

Floating Point

If the range available with integers is insufficient, or decimal numbers must be represented, FP format must be used. A common standard is IEEE 754 format, which allows 32-bit single precision and 64-bit double precision representations. In the 32-bit form, the sign is the MSB (Bit 31), the exponent the next eight bits (30–23), and the mantissa the remaining 23 bits $(22-0)$. The exponent represents binary numbers from -126 to $+127$, with 01111111 (127_{10}) representing zero, to provide for negative exponents. The mantissa always starts with 1, so this does not need to be encoded. The bits have the fractional bit weightings of 0.5, 0.25, 0.125… as shown in Table 5.6.

The exponent value is calculated by converting the binary exponent value into decimal, adding 127 to normalise the range, and raising 2 to this power. The mantissa is found by adding the weightings of the fraction bits to form a number in the range 0 to 1. Then 1 is added to give a mantissa range of 1.0000… to 1.9999… The result can be calculated as a decimal, and converted into scientific notation.

The example given uses relatively few mantissa bits to keep the calculation simple; more decimal places will be added to the number by using the smaller fraction bits. The range of the 32-bit FP point number is greater than 10^{-44} to 10^{+38}. An alternative format uses bit 23 as the sign bit, leaving the complete high byte representing the exponent, which is probably easier to process.

Bit	31	30	29	28	27	26	25	24	23	
	Sign	Exponent								
Weight (n)		$+7$	$+6$	$+5$	$+4$	$+3$	$+2$	$+1$	0	Weight $= 2^n$
	1	1	0	0	1	0	0	1	0	

Bit	Fraction																						
Bit	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Weight(n)	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Example

	Sign $= 1 \rightarrow$ negative mantissa	
Exponent:	Binary $= 128+16+2 = 142$ Exponent $= 142–127 = +15$ Exponent multiplier $= 2^{+15}$	
Fraction:	$2^{-1} + 2^{-3} + 2^{-4} + 2^{-6}$	$= 1/2 + 1/8 + 1/16 + 1/64$ $= 0.5 + 0.125 + 0.0625 + 0.015625$ $= 0.703125$
Mantissa:	Fraction $+ 1$	$= 1.703125$
Number:	$- 1.703125 \times 2^{+15}$	$= -1.703125 \times 32768$ $= -55808$
Result:	$\mathbf{-5.5808 \times 10^4}$	

Table 5.6 Structure of a 32-bit floating point number

Only advanced programmers are likely to be writing code to manipulate FP numbers direct, and when programming in higher-level languages, the format is pre-defined and functions are provided to handle FP numbers. To perform arithmetic operations, the usual rules can be applied to numbers in this form; for example, to multiply, the exponents are added and the mantissas multiplied.

Characters and Strings

The standard coding of characters for text storage and transmission is ASCII; the code table has already been provided in Table 4.4. These codes are recognised by standard alphanumeric displays and in serial communications interfaces, and is the default storage format for plain text files.

A string is a sequence of characters which might be handled as a complete data object. A list of characters may be stored in a contiguous (adjacent) set of memory locations and accessed using a table pointer which is incremented automatically for each memory read. The processor may then only need the first location, and the number of locations to be read, to access the string.

The simplest method of string access is illustrated in the LCD demo Program 4.4 using the program counter as a pointer. Alternatively, the File Select Register can be used to access a block of GPRs containing character codes. Note that the PIC assembler generates ASCII codes automatically when the operand is given as a character in single quotes.

Arithmetic

Some form of calculation is needed in most programs, even if it is a simple subtraction to determine whether an input is greater or less than a required level. At the other extreme, a computer-aided design program may carry out millions of operations per second when drawing a 3-D graphic. Games programs are also among the most demanding processor powers, because 3-D graphics must be generated at maximum speed. Here, we will cover just the basics so that simple control and communication processes that include common arithmetic operations can be attempted.

Add

A simple calculation is adding the two numbers whose result is 255 or less, the maximum for an 8-bit location. In PIC assembler, ADDWF will give the right result with no further adjustment required. The conversion into decimal of each binary number is also shown to confirm that the result is correct.

ADD (Result < 256)

```
                   0111 0100  =  64+32+16+4  =  116
                 + 0011 0101  =  32+16+4+1   =   53
                   1010 1001  =  128+32+8+1  =  169
   Carry bits        11   1
```

Note the carry from some bits to the next, which are internally handled, until there is a carry out from the most significant bit. This occurs when the result is higher than 255.

ADD (Result > 255)

```
                   0111 0100  =   116
                 + 1001 0000  =   144
      Carry out  1 0000 0100  =   260₁₀
```

This carry out is recorded in the Carry bit of the status register. It is accessed there as necessary, for example, added to the next most significant byte of a multibyte number. The carry bit from the low byte addition must be added to the next byte to obtain the right result. The sample calculation is also shown in hex:

```
ADD (Multiple bytes)
                      0111 0101 0101  0111   =  7557
                      0001 1000 1100  1011   =  18CB
                     +1000 1110 0010  0010   =  8E22₁₆
      Carry bits       111   11 1 11  111        11
      Carry from low to high byte in bold
```

Subtract

Subtract is straightforward if a number is subtracted from a larger one. A borrow from one column becomes a 2 in the previous column, and as the answer is positive, no further processing is needed.

```
     1100   1011  =   203
   − 0110   0010  =  − 98
     0110   1001  =   105
```

If a borrow is required into the MSB, the carry flag is used. Therefore, the carry flag must be set before a subtract operation so that 1 is available to borrow:

```
  1   1100  1011   =   256 + 203
  −   1110  0010   =       − 226
      1110  1001   =         233
```

In this example, the borrow bit represents the least significant bit of the next byte, which has a value of 256_{10}. In multibyte subtraction the carry flag is used to transfer the borrow from one byte to the next. If the borrow is taken, the next highest byte must be decremented to 'take' the borrow from it.

Another method of subtraction uses the 2s complement form, which is outlined below, in Section 'Negative Integers'.

Multiply

A simple algorithm for multiplication is successive addition. For example:

$$3 \times 4 = 4 + 4 + 4$$

That is, add 4 three times. The process is detailed below:

```
MULTIPLY BY ADDING

       Clear a Result register
       Load a Count register with Num1

       Loop
             Add Num2 to Result
             Decrement Count
       Until Count = 0
```

The result of the multiplication is then left in the Result register.

An alternative method is shift and add, which is more efficient for larger numbers. This is based on conventional long multiplication. However, when implemented in binary, the process can be simplified because the multiplier contains only 1s and 0s:

```
     1101  =   13 (multiplicand)
   × 0110  =  × 6 (multiplier)
     0000
    11010
   110100
  0000000
 01001110  =   78
```

Where the multiplier is a 0, the result must be 0, so that operation can be skipped, and the non-zero subtotals are obtained by shifting, and then adding to a running total:

```
MULTIPLY BY SHIFTING

   Clear a Result register
   Point to Bit0 in multiplier

   Loop
      If multiplier bit is 1, add multiplicand to Result
      Shift multiplicand left
      Increment multiplier bit pointer
   Until last bit done
```

This process can be implemented in hardware within the MCU for higher processing speed; the 18-series PIC chips, for example, have a multiply operation in the instruction set.

Divide

Divide is the inverse of multiply, so can be implemented using successive subtraction. The divisor is subtracted from the dividend, and a counter incremented. This process is repeated until the result goes negative; this is detected by the carry flag being cleared, so it must be set before the process starts. The remainder is then corrected by adding the divisor back on to the negative dividend, leaving a positive remainder in the dividend register, and decrementing the result in the counter to compensate for going one step too far.

```
DIVIDE
      Load Dividend register
      Load Divisor register
      Set Carry flag

Loop
      Subtract Divisor from Dividend
      Increment Result
Until Carry flag clear

Add Divisor back onto Dividend
Decrement Result
```

Again, a divide instruction may be provided in higher performance MCUs.

Negative Integers

We have seen above that negative numbers can be represented by the positive number, accompanied by an extra bit to represent the sign. Usually, 0 = positive and 1 = negative. However, this does not allow the full range of arithmetic operations to be applied. A more coherent system is needed for complex calculations.

In general, when a number is subtracted from a smaller one, a negative result is obtained. When implemented in a binary counter, the negative numbers are represented by a register being decremented from zero. Assuming an 8-bit register is used, the negative going count from zero is shown below in the left column, with its hex equivalent:

```
0000  0000₂   ==   00₁₆  ==   000₁₀
----------------------------------
1111  1111    ==   FF    ==  −001
1111  1110    ==   FE    ==  −002
1111  1101    ==   FD    ==  −003
1111  1100    ==   FC    ==  −004
1111  1011    ==   FB    ==  −005
----  ----         --          ---
----  ----         --          ---
```

```
1000  0010   ==   82   ==  −126
1000  0001   ==   81   ==  −127
1000  0000   ==   80   ==  −128
0111  1111   ==   7F   ==  −129
----  ----        --        ---
----  ----        --        ---
0000  0010   ==   02   ==  −254
0000  0001   ==   01   ==  −255
```

The number obtained by decrementing a register from 0 is called the 2s complement of the corresponding positive number seen in the right column. This method can be used to represent a valid negative subtraction result. For example, to calculate $2-7 = -5$:

```
            HEX                   BINARY

      (1)   02                1   0000 0010
       −    07                −   0000 0111
            FB (−5)               1111 1011
```

The 7 is subtracted from 102_{16} (02_{16} plus 100_{16} borrow in) to give the 2s complement result.

If a positive number is added to a 2s complement negative number, the answer is correct if the carry out of the register is ignored. For example, to calculate –4+7=3:

```
(−4) = FC          FC                1111 1100
                 + 07                0000 0111
                 1 03              1 0000 0011
```

2s Complement Conversion

The corresponding positive number can be calculated from the 2s complement negative number by applying the process:

INVERT ALL BITS, THEN ADD 1

For example:

```
                             Invert bits     Add 1
−003₁₀ == FD == 1111 1101 → 0000 0010 → 0000 0011 == 3₁₀
−127₁₀ == 81 == 1000 0001 → 0111 1110 → 0111 1111 == 127₁₀
−129₁₀ == 7F == 0111 1111 → 1000 0000 → 1000 0001 == 129₁₀
−255₁₀ == 01 == 0000 0001 → 1111 1110 → 1111 1111 == 255₁₀
```

Conversion to the 2s complement negative form from the positive number is achieved by the inverse process:

SUBTRACT 1, THEN INVERT ALL BITS

For example:

```
          Subtract 1   Invert bits
  1 == 0000 0001 → 0000 0000 → 1111 1111 == FF == −001
 63 == 0011 1111 → 0011 1110 → 1100 0001 == C1 == −063
128 == 1000 0000 → 0111 1111 → 1000 0000 == 80 == −128
255 == 1111 1111 → 1111 1110 → 0000 0001 == 01 == −255
```

In this way, 2s complement form allows the usual arithmetic operations to be applied to negative binary numbers with the correct result. −255 is the limit of the 8-bit 2s complement range, but more bits can be used if necessary.

SUMMARY 5

- Digital information is stored as numerical or character data
- A number system uses a base set of digits and column weighting
- Base 2, 10 and 16 are most useful in microsystems
- BCD and FP numerical formats are often required
- The decimal equivalent is the sum of column-weighted products
- The number equivalent is the remainders of division by the base
- The main numerical types are integers and FP
- The standard character code is ASCII, with text stored as strings
- Multiplication and division can be implemented using add and subtract
- Shift and add or subtract, or hardware, is more efficient
- Negative numbers can be represented by a sign bit or 2s complement

ASSESSMENT 5

1 Calculate the range of integers represented by an unsigned 64-bit number. *(3)*

2 State the ASCII codes for ‘A’, ‘z’ and ‘*’ in hexadecimal. *(3)*

3 Convert 10010011_2 into decimal. *(3)*

4 Convert 1234_{10} into binary. *(3)*

5 Convert $3FB0_{16}$ into binary and decimal. *(3)*

6 List the 4 components of a standard 32-bit floating point number, and the number of bits used for each. *(3)*

7 Multiply $1001_2 \times 0101_2$ in binary. Do not write down the zero products. *(3)*

8 Check the answer to (7) by converting into integer decimal. *(3)*

9 Divide 145 by 23 by successive subtraction in integer decimal. *(3)*

10 Calculate the binary 2s complement form of -99_{10} in hexadecimal. *(3)*

11 Outline a process to multiply two 8-bit numbers and store the result in another register pair. Label the four registers clearly. *(5)*

12 Outline a process to convert an 8-bit integer to its 2s complement negative representation, and subtract it from another number. *(5)*

ASSIGNMENTS 5

5.1 Floating Point Multiplication

(a) Multiply the following 32-bit floating point numbers, in binary, showing all the steps. Do not use a calculator at this stage. The sign bit is the MSB, the exponent the next eight bits.

1 00101101 01101010000000000000000

0 00001011 10011000000000000000000

(b) Convert both numbers and the result into decimal scientific form and check the result using a calculator.

5.2 8-Bit Multiplication

(a) Write a subroutine for the for the PIC 16F877 to multiply two 8-bit binary numbers using simple adding loop, and store the result using suitably labelled registers.

(b) Write a subroutine for the for the PIC 16F877 to multiply two 8-bit binary numbers by shifting and adding, and store the result using suitably labelled registers.

(c) Calculate the total time taken (in instruction cycles) by each method and show which is more efficient.

5.3 C Variable Types

Investigate and tabulate the numeric and character types supported by a standard ‘C’ language compiler, indicating the bit usage and memory requirements of each type.

This page intentionally left blank

6

CALCULATE, COMPARE & CAPTURE

In this chapter, applications will be described which illustrate the use of the keypad, LCD display and hardware timers as well as some of the virtual instruments available for simulated circuit operation.

Calculator

The circuit for a calculator which will perform simple arithmetic operations in the 16F877 MCU, using a calculator keypad and 16×2 LCD display, is shown in Figure 6.1.

The keypad has 16 keys: 10 numeric buttons, 4 arithmetic operations, equals and clear. The results obtained are displayed on the first line of the LCD display, which receives the characters as ASCII codes in 4-bit mode (see Chapter 4). To keep it simple, the program is limited to single digit input and double-digit results. This allows the algorithms for the arithmetic operations to be more easily understood, while the same principles can be extended to multi-digit calculations.

The calculator operates as follows:

- To perform a calculation, press a number key, then an operation key, then another number and then equals.
- The calculation and result are displayed. For the divide operation, the result is displayed as result and remainder.

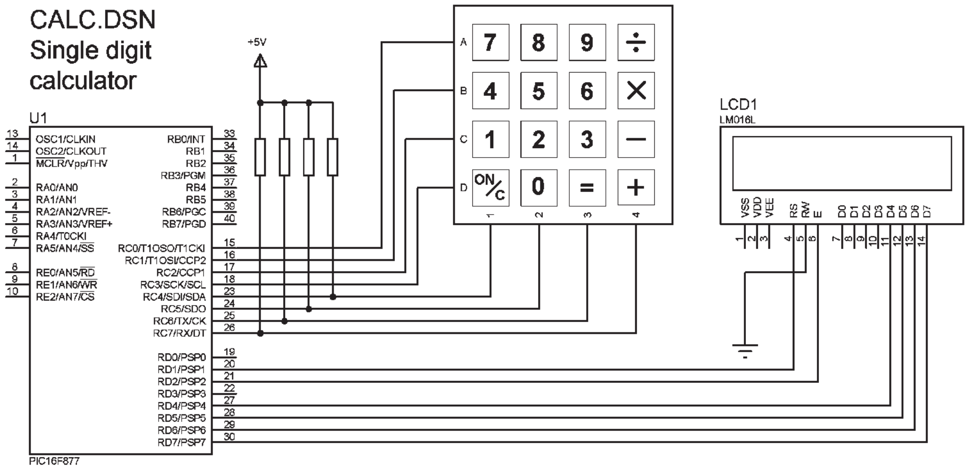


Figure 6.1 Calculator schematic

- The clear key will then erase the current display, and a new calculation can be entered. If an invalid key sequence is entered, the program should be restarted.

In order to leave Port B available for in-circuit programming and debugging (ICD), the peripheral devices are connected to Ports C and D. Remember that Ports A and E default to analogue inputs, and so have to be initialised for use as digital I/O, and that a clock circuit is not necessary in simulation mode. In the real hardware, clock and power supply must be added to the circuit, plus the ICD connections, if this is to be the programming method. The 16-button keypad is scanned by row and column as previously described (Chapter 4). The row outputs are programmed to default high. Each is then taken low in turn by outputting a 0 at RC0 to RC3. The column inputs default high due to pull-up resistors. If a button is pressed in a particular row, it can be identified by the combination of zeros on row and column. In this example, the ASCII code for the key is generated for each individual key, giving a rather lengthy scanning process, but one that is simple to understand. All keys are passed to the display routines so that they appear on the LCD immediately.

The LCD operates in 4-bit mode, as before (Chapter 4). The ASCII codes are sent in high nibble, low nibble order, and each nibble is latched into the display by pulsing input E. The R/W (read/not write) line is tied low, as reading from the display is not required. RS (Register Select) is set high for data input and low for commands.

Now refer to the program outline and source code listing, Figure 6.2 and Program 6.1. The standard P16F877 register label file is included, and the

```
CALC
Single digit calculator produces two digit results.
Hardware: x12 keypad, 2x16 LCD, P16F887 MCU

MAIN
    Initialise
        PortC = keypad
            RC0 – RC3 = output rows
            RC4 – RC7 = input columns
        PortD = LCD
            RD1, RD2 = control bits
            RD4– RD7 = data bits
    CALL Initialise display

    Scan Keypad
        REPEAT
            CALL Keypad input, Delay 50ms for debounce
            CALL Keypad input, Check key released

            IF first key, load Num1, Display character and restart loop
            IF second key, load sign, Display character and restart loop
            IF third key, load Num2 Display character and restart loop
            IF fourth key, CALL Calculate result
            IF fifth key, Clear display
        ALWAYS


SUBROUTINES

    Included LCD driver routines
        Initialise display
        Display character

    Keypad Input
        Check row A, IF key pressed, load ASCII code
        Check row B, IF key pressed, load ASCII code
        Check row C, IF key pressed, load ASCII code
        Check row D, IF key pressed, load ASCII code
        ELSE load zero code

    Calculate result
        IF key = '+', Add
        IF key = '-', Subtract
        IF key = 'x', Multiply
        IF key = '/', Divide

        Add  Add Num1 + Num2
            Load result, CALL Two digits

        Subtract  Subtract Num1 – Num2
            IF result negative, load minus sign, CALL Display character
            Load result, CALL Display character

        Multiply
            REPEAT
                Add Num1 to Result
                Decrement Num2
            UNTIL Num2 = 0
            Load result, CALL Two digits

        Divide
            REPEAT
                Subtract Num2 from Num1
                Increment Result
            UNTIL Num1 negative
            Add Num2 back onto Num1 for Remainder
            Load Result, CALL Display character
            Load Remainder, CALL Display character

    Two digits
        Divide result by 10, load MSD, CALL Display character
        Load LSD, CALL Display character
```

Figure 6.2 Calculator program outline

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;         CALC.ASM  MPB       Ver 1.0             28-8-05
;
;         Simple calculator
;         Single digit input, two digit results
;         Integer handling only
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
          PROCESSOR 16F877
;         Clock = XT 4MHz, standard fuse settings
          __CONFIG 0x3731

;         LABEL EQUATES       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          INCLUDE "C:\BOOK2\APPS\P16F877A.INC"

Char      EQU       30        ; Display character code
Num1      EQU       31        ; First number input
Num2      EQU       32        ; Second number input
Result    EQU       33        ; Calculated result
Oper      EQU       34        ; Operation code store
Temp      EQU       35        ; Temporary register for subtract
Kcount    EQU       36        ; Count of keys hit
Kcode     EQU       37        ; ASCII code for key
Msd       EQU       38        ; Most significant digit of result
Lsd       EQU       39        ; Least significant digit of result
Kval      EQU       40        ; Key numerical value

RS        EQU       1         ; Register select output bit
E         EQU       2         ; Display data strobe

; Program begins ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          ORG       0                   ; Default start address
          NOP                           ; required for ICD mode

          BANKSEL   TRISC               ; Select bank 1
          MOVLW     B'11110000'         ; Keypad direction code
          MOVWF     TRISC               ;
          CLRF      TRISD               ; Display port is output

          BANKSEL PORTC                 ; Select bank 0
          MOVLW     0FF                 ;
          MOVWF     PORTC               ; Set keypad outputs high
          CLRF      PORTD               ; Clear display outputs
          GOTO      start               ; Jump to main program

; MAIN LOOP ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

start     CALL      Init                ; Initialise the display
          MOVLW     0x80                ; position to home cursor
          BCF       Select,RS           ; Select command mode
          CALL      Send                ; and send code

          CLRW      Char                ; ASCII = 0
          CLRW      Kval                ; Key value = 0
          CLRW      DFlag               ; Digit flags = 0
scan      CALL      keyin               ; Scan keypad
          MOVF      Char,1              ; test character code
          BTFSS     STATUS,Z            ; key pressed?
          GOTO      keyon               ; yes - wait for release
          GOTO      scan                ; no - scan again
keyon     MOVF      Char,W              ; Copy..
          MOVWF     Kcode               ; ..ASCIIcode
          MOVLW     D'50'               ; delay for..
          CALL      Xms                 ; ..50ms debounce
wait      CALL      keyin               ; scan keypad again
          MOVF      Char,1              ; test character code
          BTFSS     STATUS,Z            ; key pressed?
          GOTO      wait                ; no - rescan
          CALL      disout              ; yes - show symbol
          INCF      Kcount              ; inc count..
          MOVF      Kcount,W            ; ..of keys pressed
          ADDWF     PCL                 ; jump into table
          NOP
          GOTO      first               ; process first key
          GOTO      scan                ; get operation key
          GOTO      second              ; process second symbol
          GOTO      calc                ; calculate result
          GOTO      clear               ; clear display
first     MOVF      Kval,W              ; store..
          MOVWF     Num1                ; first num
          GOTO      scan                ; and get op key
second    MOVF      Kval,W              ; store..
          MOVWF     Num2                ; second number
          GOTO      scan                ; and get equals key
```

Program 6.1 Single-digit calculator

```
; SUBROUTINES ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Include LCD driver routine

          INCLUDE   "C:\BOOK2\APPS\SUBS\LCD.INC"

; Scan keypad ...........................................

keyin     MOVLW     00F                 ; deselect..
          MOVWF     PORTC               ; ..all rows
          BCF       PORTC,0             ; select row A
          CALL      Onems               ; wait output stable
          BTFSC     PORTC,4             ; button 7?
          GOTO      b8                  ; no
          MOVLW     '7'                 ; yes
          MOVWF     Char                ; load key code
          MOVLW     07                  ; and
          MOVWF     Kval                ; key value
          RETURN
b8        BTFSC     PORTC,5             ; button 8?
          GOTO      b9                  ; no
          MOVLW     '8'                 ; yes
          MOVWF     Char
          MOVLW     08
          MOVWF     Kval
          RETURN
b9        BTFSC     PORTC,6             ; button 9?
          GOTO      bd                  ; no
          MOVLW     '9'                 ; yes
          MOVWF     Char
          MOVLW     09
          MOVWF     Kval
          RETURN
bd        BTFSC     PORTC,7             ; button /?
          GOTO      rowb                ; no
          MOVLW     '/'                 ; yes
          MOVWF     Char                ; store key code
          MOVWF     Oper                ; store operator symbol
          RETURN
rowb      BSF       PORTC,0             ; select row B
          BCF       PORTC,1
          CALL      Onems
          BTFSC     PORTC,4             ; button 4?
          GOTO      b5                  ; no
          MOVLW     '4'                 ; yes
          MOVWF     Char
          MOVLW     04
          MOVWF     Kval
          RETURN
b5        BTFSC     PORTC,5             ; button 5?
          GOTO      b6                  ; no
          MOVLW     '5'                 ; yes
          MOVWF     Char
          MOVLW     05
          MOVWF     Kval
          RETURN
b6        BTFSC     PORTC,6             ; button 6?
          GOTO      bm                  ; no
          MOVLW     '6'                 ; yes
          MOVWF     Char
          MOVLW     06
          MOVWF     Kval
          RETURN
bm        BTFSC     PORTC,7             ; button x?
          GOTO      rowc                ; no
          MOVLW     'x'                 ; yes
          MOVWF     Char
          MOVWF     Oper
          RETURN
rowc      BSF       PORTC,1             ; select row C
          BCF       PORTC,2
          CALL      Onems
          BTFSC     PORTC,4             ; button 1?
          GOTO      b2                  ; no
          MOVLW     '1'                 ; yes
          MOVWF     Char
          MOVLW     01
          MOVWF     Kval
          RETURN
b2        BTFSC     PORTC,5             ; button 2?
          GOTO      b3                  ; no
          MOVLW     '2'                 ; yes
          MOVWF     Char
          MOVLW     02
          MOVWF     Kval
          RETURN
```

Program 6.1 *Continued*

```
b3        BTFSC     PORTC,6               ; button 3?
          GOTO      bs                    ; no
          MOVLW     '3'                   ; yes
          MOVWF     Char
          MOVLW     03
          MOVWF     Kval
          RETURN
bs        BTFSC     PORTC,7               ; button -?
          GOTO      rowd                  ; no
          MOVLW     '-'                   ; yes
          MOVWF     Char
          MOVWF     Oper
          RETURN
rowd      BSF       PORTC,2               ; select row D
          BCF       PORTC,3
          CALL      Onems
          BTFSC     PORTC,4               ; button C?
          GOTO      b0                    ; no
          MOVLW     'c'                   ; yes
          MOVWF     Char
          MOVWF     Oper
          RETURN
b0        BTFSC     PORTC,5               ; button 0?
          GOTO      be                    ; no
          MOVLW     '0'                   ; yes
          MOVWF     Char
          MOVLW     00
          MOVWF     Kval
          RETURN
be        BTFSC     PORTC,6               ; button =?
          GOTO      bp                    ; no
          MOVLW     '='                   ; yes
          MOVWF     Char
          RETURN
bp        BTFSC     PORTC,7               ; button +?
          GOTO      done                  ; no
          MOVLW     '+'                   ; yes
          MOVWF     Char
          MOVWF     Oper
          RETURN
done      BSF       PORTC,3               ; clear last row
          CLRF      Char                  ; character code = 0
          RETURN
; Write display ..........................................
disout    MOVF      Kcode,W               ; Load the code
          BSF       Select,RS             ; Select data mode
          CALL      Send                  ; and send code
          RETURN
; Process operations .....................................

calc      MOVF      Oper,W                ; check for add
          MOVWF     Temp                  ; load input op code
          MOVLW     '+'                   ; load plus code
          SUBWF     Temp                  ; compare
          BTFSC     STATUS,Z              ; and check if same
          GOTO      add                   ; yes, jump to op
          MOVF      Oper,W                ; check for subtract
          MOVWF     Temp
          MOVLW     '-'
          SUBWF     Temp
          BTFSC     STATUS,Z
          GOTO      sub
          MOVF      Oper,W                ; check for multiply
          MOVWF     Temp
          MOVLW     'x'
          SUBWF     Temp
          BTFSC     STATUS,Z
          GOTO      mul
          MOVF      Oper,W                ; check for divide
          MOVWF     Temp
          MOVLW     '/'
          SUBWF     Temp
          BTFSC     STATUS,Z
          GOTO      div
          GOTO      scan                  ; rescan if key invalid
```

Program 6.1 *Continued*

```
; Calclate results from 2 input numbers ..................

add       MOVF      Num1,W              ; get first number
          ADDWF     Num2,W              ; add second
          MOVWF     Result              ; and store result
          GOTO      outres              ; display result

sub       BSF       STATUS,C            ; Negative detect flag
          MOVF      Num2,W              ; get first number
          SUBWF     Num1,W              ; subtract second
          MOVWF     Result              ; and store result

          BTFSS     STATUS,C            ; answer negative?
          GOTO      minus               ; yes, minus result
          GOTO      outres              ; display result

minus     MOVLW     '-'                 ; load minus sign
          BSF       Select,RS           ; Select data mode
          CALL      Send                ; and send symbol

          COMF      Result              ; invert all bits
          INCF      Result              ; add 1
          GOTO      outres              ; display result

mul       MOVF      Num1,W              ; get first number
          CLRF      Result              ; total to Z
add1      ADDWF     Result              ; add to total
          DECFSZ    Num2                ; num2 times and
          GOTO      add1                ; repeat if not done
          GOTO      outres              ; done, display result
div       CLRF      Result              ; total to Z
          MOVF      Num2,W              ; get divisor
          BCF       STATUS,C            ; set C flag
sub1      INCF      Result              ; count loop start
          SUBWF     Num1                ; subtract
          BTFSS     STATUS,Z            ; exact answer?
          GOTO      neg                 ; no
          GOTO      outres              ; yes, display answer
neg       BTFSC     STATUS,C            ; gone negative?
          GOTO      sub1                ; no - repeat
          DECF      Result              ; correct the result
          MOVF      Num2,W              ; get divisor
          ADDWF     Num1                ; calc remainder
          MOVF      Result,W            ; load result
          ADDLW     030                 ; convert to ASCII
          BSF       Select,RS           ; Select data mode
          CALL      Send                ; and send result

          MOVLW     'r'                 ; indicate remainder
          CALL      Send
          MOVF      Num1,W
          ADDLW     030                 ; convert to ASCII
          CALL      Send
          GOTO      scan
; Convert binary to BCD ..................................

outres    MOVF      Result,W            ; load result
          MOVWF     Lsd                 ; into low digit store
          CLRF      Msd                 ; high digit = 0
          BSF       STATUS,C            ; set C flag
          MOVLW     D'10'               ; load 10

again     SUBWF     Lsd                 ; sub 10 from result
          INCF      Msd                 ; inc high digit
          BTFSC     STATUS,C            ; check if negative
          GOTO      again               ; no, keep going
          ADDWF     Lsd                 ; yes, add 10 back
          DECF      Msd                 ; inc high digit
; display 2 digit BCD result .............................

          MOVF      Msd,W               ; load high digit result
          BTFSC     STATUS,Z            ; check if Z
          GOTO      lowd                ; yes, dont display Msd

          ADDLW     030                 ; convert to ASCII
          BSF       Select,RS           ; Select data mode
          CALL      Send                ; and send Msd

lowd      MOVF      Lsd,W               ; load low digit result
          ADDLW     030                 ; convert to ASCII
          BSF       Select,RS           ; Select data mode
          CALL      Send                ; and send Msd

          GOTO      scan                ; scan for clear key
; Restart ...............................................
clear     MOVLW     01                  ; code to clear display
          BCF       Select,RS           ; Select data mode
          CALL      Send                ; and send code
          CLRF      Kcount              ; reset count of keys
          GOTO      scan                ; and rescan keypad

          END       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Program 6.1 *Continued*

initialisation of the ports as required by the hardware connections is carried out. A separate file for the LCD initialisation and operation has been created (LCD.INI) to keep the source code size down and to provide a re-usable file for future programs. This is included at the top of the subroutine section. It contains the LCD initialisation sequence (inid) and code transmission block (send), as seen in the LCD demo program in Chapter 4.

The main program sequence calls the keypad scanning routine to detect a key, in which the key code is stored, and then delays for switch debouncing and release. The input key is displayed, and the program then jumps to a routine to handle each input in a sequence of five buttons (Num1, Operation, Num2, Equals and Clear). The calculation routine uses the operation input code to select the required process: add, subtract, multiply or divide. The binary result of the calculation is passed to a routine to convert it into BCD, then ASCII, and send it to the display. The result of the divide, being a single digit result and remainder, is sent direct to the display. The clear operation sends a command to the display to clear the last set of characters.

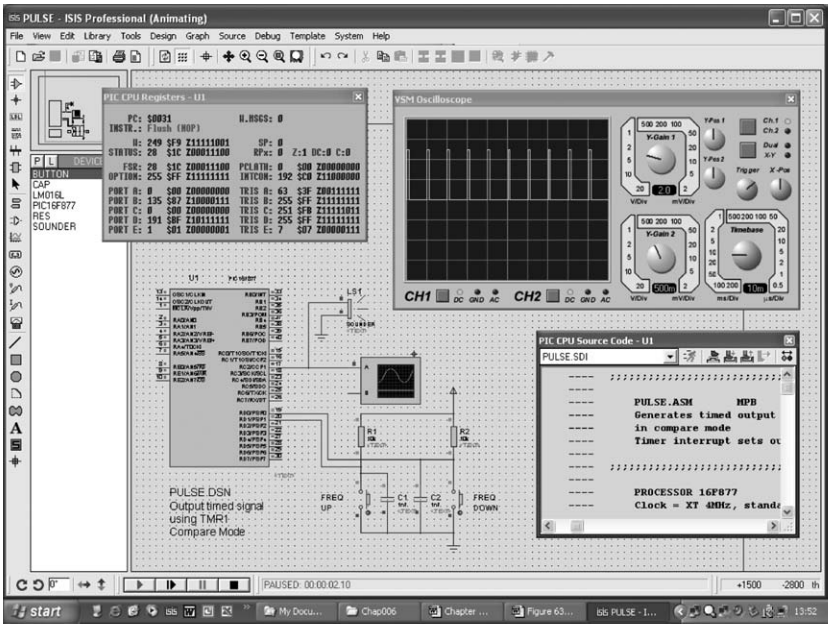
The program is highly structured to make it easier to understand (hopefully!). In longer programs, care must be taken not to exceed the stack depth when using multiple levels of subroutine in structured programs.

Pulse Output

This program illustrates the use of a hardware timer to generate an output waveform whose period can be controlled by push buttons.

The hardware configuration is shown in Figure 6.3, as a screenshot. The pulse output is generated at the output of Timer1, RC2, which is initialised for output. A pulse output is generated with a fixed 1 ms positive pulse, and a variable interval between pulses that can be adjusted manually. A virtual oscilloscope displays the output waveform, which initially runs at 100 Hz. The output is fed to a sounder, which causes the simulator to generate an audible output via the PC soundcard. The effect of pressing each button can thus be heard as well as displayed. Note that hardware debouncing has been used (capacitors across the buttons) to simplify the software.

The main purpose of this example is to illustrate the use of Timer1 compare mode. This requires pre-loading a register with a reference binary number, with which a count register is continuously compared in the timer hardware. When a match is detected, an interrupt is generated which calls the interrupt service routine (ISR). This allows the input to be processed immediately, preserving the accurate timing of program operation.



```
     PC: $0031              W.MSGS: 0
 INSTR.: Flush (NOP)

      W: 249 $F9 Z11111001      SP: 0
 STATUS: 28  $1C Z00011100     RPx: 0  Z:1 DC:0 C:0

    FSR: 28  $1C Z00011100  PCLATH: 0   $00 Z00000000
 OPTION: 255 $FF Z11111111  INTCON: 192 $C0 Z11000000

 PORT A: 0   $00 Z00000000  TRIS A: 63  $3F Z00111111
 PORT B: 135 $87 Z10000111  TRIS B: 255 $FF Z11111111
 PORT C: 0   $00 Z00000000  TRIS C: 251 $FB Z11111011
 PORT D: 191 $BF Z10111111  TRIS D: 255 $FF Z11111111
 PORT E: 1   $01 Z00000001  TRIS E: 7   $07 Z00000111
```

```
 ----   ;;;;;;;;;;;;;;;;;;;;;;;;;;
 ----
 ----       PULSE.ASM       MPB
 ----       Generates timed output
 ----       in compare mode
 ----       Timer interrupt sets ou
 ----
 ----   ;;;;;;;;;;;;;;;;;;;;;;;;;;
 ----
 ----       PROCESSOR 16F877
 ----       Clock = XT 4MHz, standa
```

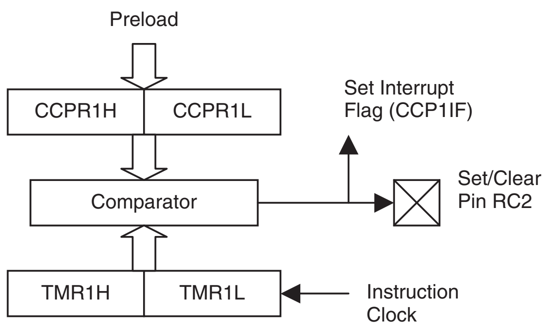
Figure 6.3 Pulse output simulation

The Timer1 compare mode is illustrated in Figure 6.4. It uses a pair of registers, TMR1H (high byte) and TMR1L (low byte), to record a 16-bit count, driven by the MCU instruction clock. In the system simulation, the clock is set to 4 MHz, giving a 1 MHz instruction clock (1 instruction takes four clock cycles). The timer therefore counts in microseconds. The reference register pair, CCPR1H and CCPR1L, is pre-loaded with a value which is continuously compared with the 16-bit timer count (default $2710_{16} = 10\,000_{10}$). With this value loaded, the compare becomes true after 10 ms, the interrupt generated, and the output set high. The ISR resets the interrupt, tests the buttons to see if the pre-set value should be changed, waits 1 ms and then clears the output to zero. The default output is therefore a 1 ms high pulse, followed by a 9 ms interval. This process repeats, giving a pulse waveform with an output period of 10 ms overall.

The source code shows the initialisation required for the interrupt operation. The interrupt vector (GOTO isr) is loaded at address 004, so the initial execution sequence has to jump over this location. The port, timer and interrupt registers are then set up (Figure 6.4 (b)). The timer is started, and the single instruction main loop then runs, waiting for the interrupt.

Timer1 counts up to 10000, and the interrupt is triggered. The interrupt flag is first cleared, and the counter reset to zero. The next 10 ms period starts

(a)



(b)

Register	**Load**	**Effect**
PIE1	0000 0**1**00	Enable Timer 1 interrupt
INTCON	**11**00 0000	Enable peripheral interrupts
CCP1CON	0000 **1000**	Compare mode – set output pin on match
CCPR1H	0**27**H	Initial value for high byte compare
CCPR1L	0**10**H	Initial value for low byte compare
T1CON	0000 000**1**	Enable timer with internal clock

Figure 6.4 Timer1 compare registers: (a) timer1 block diagram; (b) control registers

immediately, because the counter runs continuously. The buttons are checked, and the compare register value incremented or decremented to change the output period if one of them is pressed. A check is also made for zero at the upper and lower end of the period adjustment range, to prevent the compare value rolling over or under between 00 00 and FF FF. This would cause the output frequency to jump between the minimum and maximum value, which is undesirable in this case.

The 1 ms pulse period is generated as a software delay, which runs in parallel with the hardware timer count. After 1 ms, the output is cleared to zero, but the hardware count continues until the next interrupt occurs. This is an important point – the hardware timer continues independently of the program sequence, until the next interrupt is processed, allowing the timing operation and program to be executed simultaneously (Figure 6.5) (Program 6.2).

Period Measurement

An alternative mode of operation for Timer1 is capture mode. This allows counter value in the 16-bit timer register (TMR1H + TMR1L) to be captured in mid-count; the capture is triggered by the input RC2 changing state. A pre-scaler can be included between the input and capture enable so that the capture is only triggered every 4th or 16th pulse at the input, thereby reducing the capture rate.

```
PULSE
Generates a variable interval pulse output
controlled by up/down buttons
Hardware: P16F877 (4MHz), sounder

MAIN
        Initialise
                RC2/CCP1 = Pulse output
                RDO,RD1 = Up/Down buttons
                Timer1 Compare Mode & Interrupt

        Wait for interrupt

SUBROUTINE
        1ms delay

INTERRUPT SERVICE ROUTINE
        Reset interrupt
        IF Increase Frequency button pressed
                Decrement pulse interval
        IF Decrease Frequency button pressed
                Increment pulse interval
        Generate 1ms pulse
```

Figure 6.5 Pulse program outline

This method is used here to measure the period of a pulse waveform, which is fed in at RC2, the CCP1 module input. The module is set up to generate an interrupt when the input changes from high to low, once per cycle. The timer counts instruction clock cycles continuously, and the count reached is stored in the CCP1 register pair on each interrupt, and the count then restarted.

The main program itself continuously converts the 16-bit contents of CCP1 into 5-digit BCD, and displays the result on the LCD. It could wait for the next interrupt in an idle loop, as in the pulse output program, but this program highlights that the MCU can continue processing while the timer counts. The binary is converted into BCD using a simple subtraction loop for each digit, from ten thousands to tens (see Chapter 5). The remainder at the end is the unit's value.

The simulation uses a virtual signal generator to provide a variable frequency square wave at RC2. The 16×2 LCD is connected and driven as detailed in Chapter 4. The signal generator appears full size when the simulation is running, as long as it is selected in the debug menu. The frequency can be adjusted as the simulation runs, and the display responds accordingly, displaying the period in microseconds. Auto-ranging the display to milliseconds is one possible program enhancement which could be attempted.

The system operates correctly over the audio range, 15 Hz–50 kHz. The absolute maximum count is 65536 μs (16-bit count), and the minimum period is limited by the time taken to reset the interrupt and start counting again (less than 20 μs) (Figures 6.6–6.8) (Program 6.3).

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;         PULSE.ASM           MPB                 21-8-05
;
;         Generates timed output interval
;         using Timer 2 in compare mode
;         Timer interrupt sets output, cleared after 1ms
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          PROCESSOR 16F877
;         Clock = XT 4MHz, standard fuse settings
          __CONFIG 0x3731
;         LABEL EQUATES       ...................................

          INCLUDE "P16F877.INC"         ; Standard register labels

Count     EQU       20                  ; soft timer

; Program begins ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          ORG       0                   ; Place machine code
          NOP                           ; for ICD mode
          GOTO      init                ; Jump over ISR vector

          ORG       4                   ; ISR vector address
          GOTO      isr                 ; run ISR
init      NOP
          BANKSEL   TRISC               ; Select bank 1
          MOVLW     B'11111011'         ; RC2 = output
          MOVWF     TRISC               ; Initialise display port
          MOVLW     B'00000100'         ; Timer1 interrupt..
          MOVWF     PIE1                ; ..enable

          BANKSEL   PORTC               ; Select bank 0
          CLRF      PORTC               ; Clear output
          MOVLW     B'11000000'         ; Peripheral interupt..
          MOVWF     INTCON              ; ..enable
          MOVLW     B'00001000'         ; Compare mode..
          MOVWF     CCP1CON             ; ..set output on match
          MOVLW     027                 ; Initial value..
          MOVWF     CCPR1H              ; .. for high byte (10ms)
          MOVLW     010                 ; Initial value..
          MOVWF     CCPR1L              ; .. for low byte (10ms)
          MOVLW     B'00000001'         ; Timer1 enable..
          MOVWF     T1CON               ; internal clock (1MHz)

          GOTO      start               ; Jump to main program

;         SUBROUTINES..........................................

;         1ms delay with 1us cycle time (1000 cycles)

onems     MOVLW     D'249'              ; Count for 1ms delay
          MOVWF     Count               ; Load count
loop      NOP                           ; Pad for 4 cycle loop
          DECFSZ    Count               ; Count
          GOTO      loop                ; until Z
          RETURN                        ; and finish

;         INTERRUPT SERVICE ROUTINE.............................

;         Reset interrupt, check buttons, generate 1ms pulse

isr       CLRF      PIR1                ; clear interrupt flags
          CLRF      TMR1H               ; clear timer high..
          CLRF      TMR1L               ; ..and low byte

          BTFSC     PORTD,0             ; dec frequency button?
          GOTO      other               ; no
          INCFSZ    CCPR1H              ; yes, inc period, zero?
          GOTO      other               ; no
          DECF      CCPR1H              ; yes, step back

other     BTFSC     PORTD,1             ; inc frequency button?
          GOTO      wait                ; no
          DECFSZ    CCPR1H              ; yes, inc period, zero?
          GOTO      wait                ; no
          INCF      CCPR1H              ; yes, step back
wait      CALL      onems               ; wait 1ms
          BCF       CCP1CON,3           ; clear output
          BSF       CCP1CON,3           ; re-enable timer mode

          RETFIE                        ; return to main program

;-----------------------------------------------------------------
;         Main loop
;-----------------------------------------------------------------

start     GOTO      start               ; wait for timer interrupt
          END                           ; of source code
```

Program 6.2 Pulse output

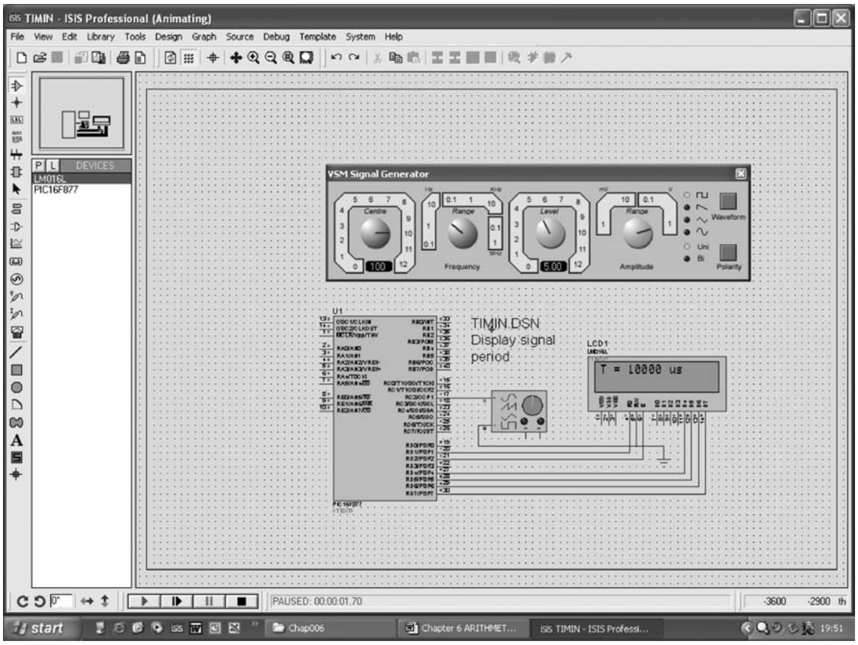
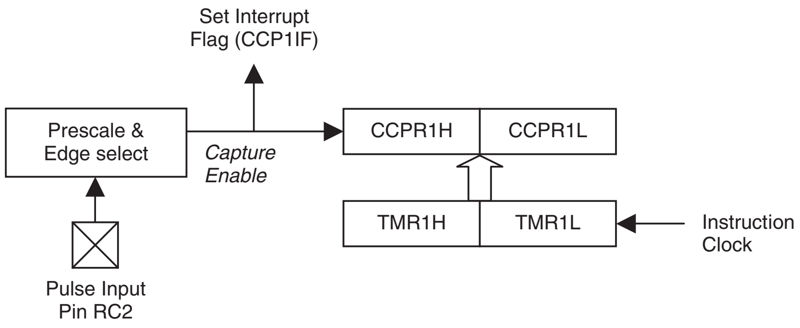


Figure 6.6 Input timing simulation

(a)



(b)

Register	Setting	Flags	Function
PIE1	0000 0**1**00	CCP1IE	Enable CCP1 interrupt
INTCON	**11**00 0000	GIE, PEIE	Enable peripheral interrupts
CCP1CON	0000 **0100**	CCP1M0 - 3	Capture mode – every falling edge
T1CON	0000 000**1**	TMR1ON	Enable timer with internal clock
PIR1	0000 0**X**00	CCP1IF	CCP1 interrupt flag

Figure 6.7 Capture registers (a) timer registers; (b) control registers

```
TIMIN
    Measure pulse waveform input period and display
    P16F877 (4MHz), audio signal source, !6x2 LCD

MAIN
    Intialise
        PortD = LCD outputs
        Capture mode & interrupt
        Initalise LCD
        Enable capture interrupt

    REPEAT
        Convert 16-bit count to 5 BCD digits
        Display input square wave period
    ALWAYS


SUBROUTINES

    Convert 16-bit count to 5 BCD digits
        Load 16-bit number
        Clear registers
            Tents, Thous, Hunds, Tens, Ones
        REPEAT
            Subtract 10000 from number
        UNTIL Tents negative
        Restore Tents and remainder
        REPEAT
            Subtract 1000 from remainder
        UNTIL Thous negative
        Restore Thous and remainder
        REPEAT
            Subtract 100 from remainder
        UNTIL Hunds negative
        Restore Hunds and remainder
        REPEAT
            Subtract 10 from remainder
        UNTIL Tens negative
        Restore Tens and store remainder Ones

    Display input square wave period
        Display 'T='
        Supress leading zeros
        Display digits in ASCII
        Display 'us'

INTERRUPT SERVICE ROUTINE
        Clear Timer 1 Count Registers
        Reset interrupt flag
```

Figure 6.8 Input period program outline

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;       TIMIN.ASM               MPB             25-8-05
;
;       Measure input period using Timer1 16-bit capture
;       and display in microseconds
;
;       STATUS: Capture working
;               Display working
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        PROCESSOR 16F877

;       Clock = XT 4MHz, standard fuse settings:

        __CONFIG 0x3731

;       LABEL EQUATES   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        INCLUDE "P16F877.INC"   ; Standard register labels

;       Local label equates.....................................

Hibyte  EQU     020
Lobyte  EQU     021

Tents   EQU     022
Thous   EQU     023
Hunds   EQU     024
Tens    EQU     025
Ones    EQU     026


; Program begins ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        ORG     0               ; Place machine code
        NOP                     ; Required for ICD mode
        GOTO    init

        ORG     4               ; Interrupt vector adress
        GOTO    ISR             ; jump to service routine

init    NOP
        BANKSEL TRISD           ; Select bank 1
        CLRF    TRISD           ; Initialise display port
        CLRF    PIE1            ; Disable peripheral interrupts

        BANKSEL PORTD           ; Select bank 0
        CLRF    PORTD           ; Clear display outputs

        MOVLW   B'11000000'     ; Enable..
        MOVWF   INTCON          ; ..peripheral interrupts
        MOVLW   B'00000100'     ; Capture mode:
        MOVWF   CCP1CON         ; ..every falling edge
        MOVLW   B'00000001'     ; Enable..
        MOVWF   T1CON           ; ..Timer 1

        GOTO    start           ; Jump to main program

; INTERRUPT SERVICE ROUTINE ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

ISR     CLRF    TMR1L
        CLRF    TMR1H
        BCF     PIR1,CCP1IF     ; Reset interrupt flag
        RETFIE
```

Program 6.3 Input period measurement

```
; SUBROUTINES   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        INCLUDE "LCD2.INC"      ; Include display routines
;--------------------------------------------------------------
; Convert 16 bit binary result to 5 digits
;--------------------------------------------------------------

conv    MOVF    CCPR1L,W        ; Get high byte
        MOVWF   Lobyte          ; and store
        MOVF    CCPR1H,W        ; Get low byte
        MOVWF   Hibyte          ; and store

        MOVLW   06              ; Correction value
        BCF     STATUS,C        ; prepare carry flag
        ADDWF   Lobyte          ; add correction
        BTFSC   STATUS,C        ; and carry
        INCF    Hibyte          ; in required

        CLRF    Tents           ; clear ten thousands register
        CLRF    Thous           ; clear thousands register
        CLRF    Hunds           ; clear hundreds register
        CLRF    Tens            ; clear tens register
        CLRF    Ones            ; clear ones register

; Subtract 10000d (2710h) and count ..........................

sub10   MOVLW   010             ; get low byte to sub
        BSF     STATUS,C        ; get ready to subtract
        SUBWF   Lobyte          ; sub 10h from low byte
        BTFSC   STATUS,C        ; borrow required?
        GOTO    sub27           ; no - sub high byte

        MOVF    Hibyte,F        ; yes - check high byte
        BTFSS   STATUS,Z        ; zero?
        GOTO    take1           ; no - take borrow

        MOVLW   010             ; yes - load low byte to add
        BCF     STATUS,C        ; get ready to add
        ADDWF   Lobyte          ; restore low byte
        GOTO    subE8           ; next digit

take1   DECF    Hibyte          ; take borrow

sub27   MOVLW   027             ; get high byte to sub
        BSF     STATUS,C        ; get ready to subtract
        SUBWF   Hibyte          ; sub from high byte
        BTFSS   STATUS,C        ; borrow taken?
        GOTO    done1           ; yes - restore remainder
        INCF    Tents           ; no - count ten thousand
        GOTO    sub10           ; sub 10000 again

done1   MOVLW   010             ; restore..
        BCF     STATUS,C        ; get ready to add
        ADDWF   Lobyte          ; restore low byte
        BTFSC   STATUS,C        ; Carry into high byte?
        INCF    Hibyte          ; yes - add carry to high byte
        MOVLW   027             ; restore..
        ADDWF   Hibyte          ; ..high byte

; Subtract 1000d (03E8) and count................................

subE8   MOVLW   0E8             ; get low byte to sub
        BSF     STATUS,C        ; get ready to subtract
        SUBWF   Lobyte          ; sub from low byte
        BTFSC   STATUS,C        ; borrow required?
        GOTO    sub03           ; no - do high byte

        MOVF    Hibyte,F        ; yes - check high byte
        BTFSS   STATUS,Z        ; zero?
        GOTO    take2           ; no - take borrow

        MOVLW   0E8             ; load low byte to add
        BCF     STATUS,C        ; get ready to add
        ADDWF   Lobyte          ; restore low byte
        GOTO    sub64           ; next digit

take2   DECF    Hibyte          ; take borrow

sub03   MOVLW   03              ; get high byte
        BSF     STATUS,C        ; get ready to subtract
        SUBWF   Hibyte          ; sub from high byte
        BTFSS   STATUS,C        ; borrow taken?
        GOTO    done2           ; yes - restore high byte
        INCF    Thous           ; no - count ten thousand
        GOTO    subE8           ; sub 1000 again

done2   MOVLW   0E8             ; restore..
        BCF     STATUS,C        ; get ready to add
        ADDWF   Lobyte          ; restore low byte
        BTFSC   STATUS,C        ; Carry into high byte?
        INCF    Hibyte          ; yes - add carry to high byte
        MOVLW   03              ; restore..
        ADDWF   Hibyte          ; ..high byte
```

Program 6.3 *Continued*

```
; Subtract 100d (064h) and count................................

sub64   MOVLW   064             ; get low byte
        BSF     STATUS,C        ; get ready to subtract
        SUBWF   Lobyte          ; sub from low byte
        BTFSC   STATUS,C        ; borrow required?
        GOTO    inchun          ; no - inc count

        MOVF    Hibyte,F        ; yes - check high byte
        BTFSS   STATUS,Z        ; zero?
        GOTO    take3           ; no - take borrow

        MOVLW   064             ; load low byte to add
        BCF     STATUS,C        ; get ready to add
        ADDWF   Lobyte          ; restore low byte
        GOTO    subA            ; next digit

take3   DECF    Hibyte          ; take borrow

inchun  INCF    Hunds           ; count hundred
        GOTO    sub64           ; sub 100 again

; Subtract 10d (0Ah) and count, leaving remainder................

subA    MOVLW   0A              ; get low byte to sub
        BSF     STATUS,C        ; get ready to subtract
        SUBWF   Lobyte          ; sub from low byte
        BTFSS   STATUS,C        ; borrow required?
        GOTO    rest4           ; yes - restore byte
        INCF    Tens            ; no - count one hundred
        GOTO    subA            ; and repeat

rest4   ADDWF   Lobyte          ; restore low byte
        MOVF    Lobyte,W        ; copy remainder..
        MOVWF   Ones            ; to ones register

        RETURN                  ; done
;-----------------------------------------------------------------
; Display period in microseconds
;-----------------------------------------------------------------
disp    BSF     Select,RS       ; Set display data mode

        MOVLW   'T'             ; Time period
        CALL    send            ; Display it
        MOVLW   ' '             ; Space
        CALL    send            ; Display it
        MOVLW   '='             ; Equals
        CALL    send            ; Display it
        MOVLW   ' '             ; Space
        CALL    send            ; Display it

; Supress leading zeros.........................................

        MOVF    Tents,F         ; Check digit
        BTFSS   STATUS,Z        ; zero?
        GOTO    show1           ; no - show it

        MOVF    Thous,F         ; Check digit
        BTFSS   STATUS,Z        ; zero?
        GOTO    show2           ; no - show it

        MOVF    Hunds,F         ; Check digit
        BTFSS   STATUS,Z        ; zero?
        GOTO    show3           ; no - show it

        MOVF    Tens,F          ; Check digit
        BTFSS   STATUS,Z        ; zero?
        GOTO    show4           ; no - show it

        MOVF    Ones,F          ; Check digit
        BTFSS   STATUS,Z        ; zero?
        GOTO    show5           ; no - show it

; Display digits of period....................................

show1   MOVLW   030             ; Load ASCII offset
        ADDWF   Tents,W         ; Add digit value
        CALL    send            ; Display it

show2   MOVLW   030             ; Load ASCII offset
        ADDWF   Thous,W         ; Add digit value
        CALL    send            ; Display it

show3   MOVLW   030             ; Load ASCII offset
        ADDWF   Hunds,W         ; Add digit value
        CALL    send            ; Display it

show4   MOVLW   030             ; Load ASCII offset
        ADDWF   Tens,W          ; Add digit value
        CALL    send            ; Display it

show5   MOVLW   030             ; Load ASCII offset
        ADDWF   Ones,W          ; Add digit value
        CALL    send            ; Display it
```

Program 6.3 *Continued*

```
; Show fixed characters.........................................

        MOVLW   ' '             ; Space
        CALL    send            ; Display it
        MOVLW   'u'             ; micro
        CALL    send            ; Display it
        MOVLW   's'             ; secs
        CALL    send            ; Display it
        MOVLW   ' '             ; Space
        CALL    send            ; Display it
        MOVLW   ' '             ; Space
        CALL    send            ; Display it

; Home cursor .................................................

        BCF     Select,RS       ; Set display command mode
        MOVLW   0x80            ; Code to home cursor
        CALL    send            ; Do it
        RETURN                  ; done

;----------------------------------------------------------------
; MAIN LOOP
;----------------------------------------------------------------
start   CALL    inid            ; Initialise display
        BANKSEL PIE1            ; Select Bank 1
        BSF     PIE1,CCP1IE     ; Enable capture interrupt
        BANKSEL PORTD           ; Select Bank 0
        BCF     PIR1,CCP1IF     ; Clear CCP1 interrupt flag

loop    CALL    conv            ; Convert 16 bits to 5 digits
        CALL    disp            ; Display period in microsecs
        GOTO    loop

        END                     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Program 6.3 *Continued*

SUMMARY 6

- The calculator demo performs single-digit arithmetic, using a keypad input and LCD display
- The output pulse generator uses hardware timer compare mode and a virtual oscilloscope to display the output
- Input period measurement uses the timer in capture mode and a virtual signal generator to provide the input

ASSESSMENT 6 — Total *(40)*

1 Describe briefly how the keypad code is generated in the CALC application. *(3)*

2 State the advantages of using an include file for the LCD driver routines in the CALC application. *(3)*

3 Outline the process required to display the negative result of a subtraction correctly on an LCD. *(3)*

4 State the meaning of the term ‘capture’ mode in an MCU timer. *(3)*

5 Explain why the main PULSE program consists of only one statement. *(3)*

6 Explain why the initial value in the timer preload registers in the PULSE program is 2710h. *(3)*

7 Explain why the ‘INCF CCPR1H’ instruction is needed in PULSE. *(3)*

8 Describe briefly the role of the carry flag in the division process. *(3)*

9 Explain the meaning of the term ‘compare’ mode in an MCU timer. *(3)*

10 Identify the Timer 1 interrupt flag by register and bit label. *(3)*

11 Describe the process for converting an 8-bit binary number to three ASCII digits for display. *(5)*

12 Describe the process for converting 16-bit binary into 5-digit BCD. *(5)*

ASSIGNMENTS 6

6.1 BCD Addition

Outline a process to read a sequence of decimal keys into the MCU which is terminated with the ‘+’ key. A further sequence of digits is terminated with the ‘=’ key. The numbers must then be added and the result displayed. Take account of the fact that the numbers may not be the same length. Do not write a source code program.

6.2 CCP Control

Refer to the PIC 16F877 data manual. Check the setup codes for capture and compare modes, and identify the function of each bit in the control registers initialised in the demo programs. Then check the setup of the interrupt in each program, and again identify the function of each relevant bit in the control registers. Explain the significance of each setup option. Why is the use of the hardware timer helpful in these applications? Consider if there are any alternative methods to achieve the same results.

6.3 Pulse Detection

A remote control receiver module generates a pulse whose length is controlled by the lever position on the transmitter console. The controller in the receiving system needs to determine the pulse length as short, mid-length or long. Outline a routine to check if an input pulse is longer than 1.4 ms, shorter that 1.2 ms, or in between these limits, switching on a ‘long’ or ‘short’ output bit accordingly, with neither for a mid-length pulse.

7

Analogue Interfacing

Many control applications require the measurement of analogue variables, such as voltage, temperature, pressure, speed and so on. Selected PIC MCUs incorporate analogue inputs, which are connected to an analogue to digital converter (ADC); this outputs a 10-bit binary representation of an input voltage. This result is then accurate to 1 part in 1024 (2^{10}), better than 0.1% at full scale, and precise enough for most purposes. In some cases, it is only necessary to use 8 bits of the conversion, which gives an accuracy of 1 part in 256 (<0.5%).

In this chapter, programs to handle 8-bit and 10-bit data will be presented, and the additional software overhead required to achieve the higher accuracy can be seen. The ADC is controlled from special function registers ADCON0 and ADCON1, and can generate a peripheral interrupt if required. The output from the converter is stored in ADRESH (analogue to digital conversion result, high byte) and ADRESL (low byte).

8-bit Conversion

The processing for an 8-bit result is simpler, so this will be described first. The ADC converts an analogue input voltage in the range 0–2.55 V to 10-bit binary, but only the upper 8 bits of the result are used, giving a resolution of 10 mV per bit ($1/256 \times 2.56$ V).

8-bit Conversion Circuit

A test circuit to demonstrate 8-bit conversion and display is shown in Figure 7.1. The 16F877 MCU has eight analogue inputs available, at RA0, RA1, RA2, RA3, RA5, RE0, RE1 and RE2. These have alternate labels AN0–AN7 for this function. RA2 and RA3 may be used as reference voltage inputs, setting the minimum and maximum values for the measured voltage range. These inputs default to analogue operation, so the register ADCON1 has to be initialised explicitly to use these pins for digital input or output.

INPUT & OUTPUT

The test voltage input at RA0 (analogue input AN0) is derived from a pot across the 5 V supply. A reference voltage is provided at RA3 (AN3), which sets the maximum voltage to be converted, and thus the conversion factor required in the software. The minimum value defaults to 0 V. The 2.7 V zener diode provides a constant reference voltage; it is supplied via a current limiting resistor, so that the zener operates at the current specified for optimum voltage stability. This is then divided down across the reference voltage pot RV1 and a 10k fixed resistor. The range across the pot is about 2.7–2.4 V, and is adjusted for 2.56 V, which gives a convenient conversion factor. The LCD is connected to Port D to operate in 4-bit mode and display the voltage, as described in Chapter 4.

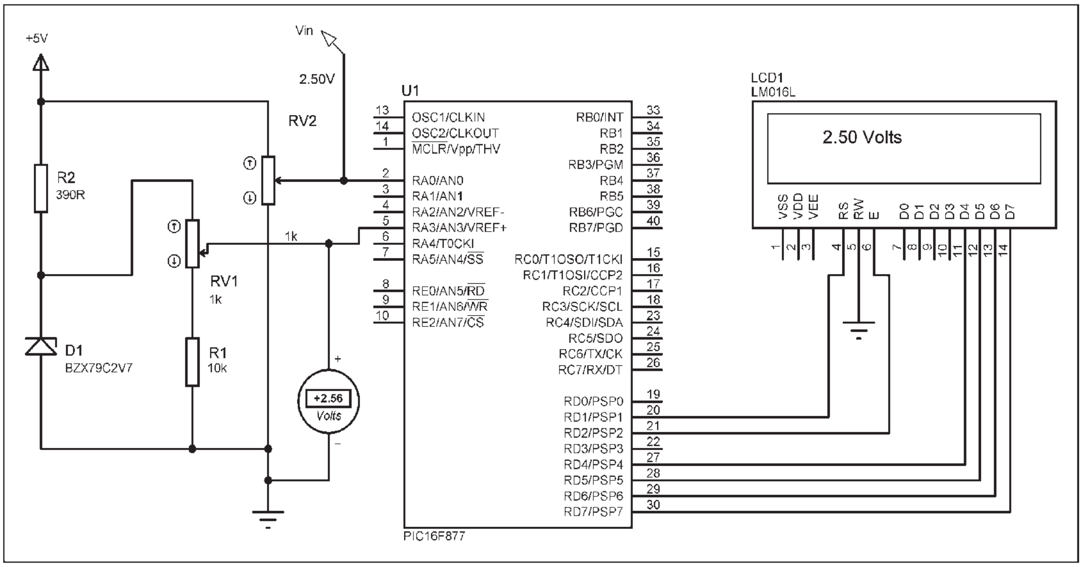


Figure 7.1 8-bit analogue input test circuit

ADC OPERATION

A block diagram of the ADC module is shown in Figure 7.2. The inputs are connected to a function selector block which sets up each pin for analogue or digital operation according to the 4-bit control code loaded into the A/D port configuration control bits, PCFG0–PCFG3 in ADCON1. The code used, 0011, sets Port E as digital I/O, and Port A as analogue inputs with AN3 as the positive reference input.

The analogue inputs are then fed to a multiplexer which allows one of the eight inputs to be selected at any one time. This is controlled by the three analogue channel select bits, CHS0–CHS2 in ADCON0. In this case, channel 0 is selected (000), RA0 input. If more than one channel is to be sampled, these select bits need to be changed between ADC conversions. The conversion is triggered by setting the GO/DONE bit, which is later cleared automatically to indicate that the conversion is complete.

ADC CLOCK

The speed of the conversion is selected by bits ADSC1 and ADSC0. The ADC operates by successive approximation; this means that the input voltage is fed to a comparator, and if the voltage is higher than 50% of the range, the MSB of the result is set high. The voltage is then checked against the mid-point of the remaining range, and the next bit set high or low accordingly, and so on for 10 bits. This takes a significant amount of time: the minimum conversion time is 1.6 μs per bit, making 16 μs for a 10-bit conversion. The ADC clock speed must be selected such that this minimum time requirement is satisfied; the MCU clock is divided by 2, 8 or 32 as necessary. Our simulated test circuit is clocked at 4 MHz. This gives a clock period of 0.25 μs. We need a conversion time of at least 1.6 μs; if we select the divide by 8 option, the ADC clock period will then be $8 \times 0.25 = 2$ μs, which is just longer than the minimum required. The select bits are therefore set to 01 (Figure 7.2 (b)).

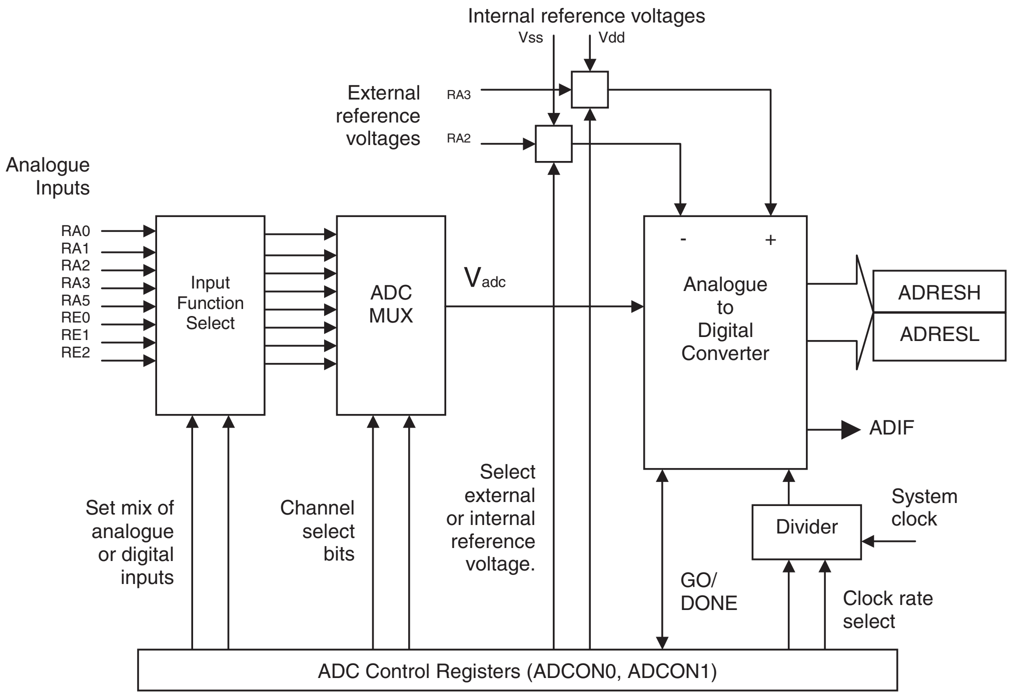
SETTLING TIME

The input of the ADC has a sample and hold circuit, to ensure that the voltage sampled is constant during the conversion process. This contains an RC low-pass filter with a time constant of about 20 μs. Therefore, if the input voltage changes suddenly, the sample and hold circuit will take time to respond. This needs to be taken into account, depending on the type of signal being measured. If sampling speed is not critical, a settling time delay of at least 20 μs should be included in the conversion sequence. In the test circuit, this is not a problem.

RESULT REGISTERS

When the conversion is complete, the result is placed in the result register pair, ADRESH and ADRESL, the GO/DONE bit cleared by the ADC controller,

(a)


Internal reference voltages
Vss
Vdd
External reference voltages
RA3
RA2
Analogue Inputs
RA0
RA1
RA2
RA3
RA5
RE0
RE1
RE2
Input Function Select
ADC MUX
Vadc
Analogue to Digital Converter
-
+
ADRESH
ADRESL
ADIF
Set mix of analogue or digital inputs
Channel select bits
Select external or internal reference voltage.
GO/ DONE
Divider
System clock
Clock rate select
ADC Control Registers (ADCON0, ADCON1)

(b)

Register	Setting	Flags	Function
ADRESH	**XXXX XXXX**		ADC result high byte
ADRESL	**XXXX XXXX**		ADC result low byte
ADCON0	**01**00 0**X**0**1**	ADCS1,0 GO/DONE, ADON	Conversion frequency select ADC start, ADC enable
ADCON1	**0**000 **0011**	ADFM, PCFG3-0	Result justify, ADC input mode control
INTCON	**11**00 0000	GIE,PEIE	Peripheral interrupt enable
PIE1	0**1**00 0000	ADIE	ADC interrupt enable
PIR1	0**1**00 0000	ADIF	ADC interrupt flag

(c)

		ADRESH	ADRESL
ADFM = 1	Right justified	0000 00**RR**	**RRRR RRRR**
ADFM = 0	Left justified	**RRRR RRRR**	**RR**00 0000

R = Result bits

Figure 7.2 ADC operation: (a) ADC block diagram; (b) ADC control registers; (c) result registers configuration

and the ADIF interrupt flag is set. Since the result is only 10 bits, the positioning in the 16-bit result register pair can be selected, so that the high 8 bits are in ADRESH (left justified), or the low 8 bits are in ADRESL (right justified) (Figure 7.2 (c)). Obviously, to retain 10-bit resolution, both parts must be processed, so right justification will probably be more convenient in this case.

If only 8 bits resolution is required, the process can be simplified. If the result is right justified, the low 8 bits in ADRESL will record the low bits of the conversion, meaning that only voltages up to 25% of the full range will be processed, but at full resolution. If the result is left justified, the high byte will be processed, which will represent the full voltage range, but at reduced resolution.

In our test circuit, the reference voltage is 2.56 V, and the justify bit ADFM = 0, selecting left justify. Only ADRESH then needs to be processed, giving results for the full range at 8-bit resolution, which is about 1% at mid-range. The result will be shown on the LCD as 3 digits, 0.00–2.55. The test input pot gives 0–5 V, but only 0–2.50 will be displayed. Over range inputs will be displayed as 2.55 V.

8-bit Conversion Program

The test program is outlined in Figure 7.3, and the source code listed in Program 7.1. The output port and ADC control registers are initialised in the first block, with the LCD include file providing the display initialisation, and driver routines. The main loop contains subroutine calls to read the ADC input, convert from binary to BCD and display it. The routine to read the ADC sets the GO/DONE bit and then polls it until it is cleared at the end of the conversion. The 8-bit result from ADRESH is converted to three BCD digits by the subtraction algorithm described previously. Full-scale input is 255, which is displayed as 2.55 V.

10-bit Conversion

Figure 7.4 shows a circuit, which demonstrates 10-bit, full resolution, analogue to digital conversion. The reference voltage circuit now provides a reference of 4.096 V, giving a wider range of 0–4.095 V. With this reference voltage, and a maximum binary result of 1023 (2^{10}–1), the conversion output will increase at 4 mV per bit. The result is displayed as a 4-digit fixed point decimal.

The reference voltage circuit is a little different from the 8-bit circuit. The zener voltage is divided down using fixed-value resistors, and the final voltage tweaked by adjusting the current to the zener. This gives a finer adjustment than using the calibration pot in the voltage divider chain.

```
ADC8
    Convert the analogue input to 8-bits and display
    Hardware: P16F877 (4MHz), Vref+ = 2.56, 16x2 LCD

Initialise
    PortA = Analogue inputs (default)
    PortC = LCD outputs
    ADC = Select f/8, RA0 input, left justify result, enable
    LCD = default setup (include LCD driver routines)

Main
    REPEAT
        Get ADC 8-bit input
        Convert to BCD
        Display on LCD
    ALWAYS


Subroutines

    Get ADC 8-bit input
        Start ADC and wait for done
        Store result

    Convert to BCD
        Calculate hundreds digit
        Calculate tens digit
        Remainder = ones digit

    Display on LCD
        Home cursor
        Convert BCD to ASCII
        Send hundreds, point, tens, ones
        Send 'Volts'

Include
    LCD routines
```

Figure 7.3 ADC test program outline

The data acquisition process is similar to the 8-bit system above, but the binary to BCD conversion process is rather more complicated. The result is required in the range 0–4095, so the original result (0–1023) is shifted left twice to multiply it by four. One thousand (03E8) is then loop subtracted from the result to calculate the number of thousands in the number. Correct borrow handling between the high and low byte is particularly important. The process stops when the remainder is less that 1000. The hundreds digit is calculated in a similar way, but the tens calculation is a little easier as the maximum remainder from the previous stage is 99, so the high byte borrow handling is not necessary. This process is outlined in Figure 7.5, and the source code shown in Program 7.2.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;         Project:                      Interfacing PICs
;         Source File Name:             VINTEST.ASM
;         Devised by:                   MPB
;         Date:                         19-12-05
;         Status:                       Final version
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;         Demonstrates simple analogue input
;         using an external reference voltage of 2.56V
;         The 8-bit result is converted to BCD for display
;         as a voltage using the standard LCD routines.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          PROCESSOR 16F877
;         Clock = XT 4MHz, standard fuse settings
          __CONFIG 0x3731

;         LABEL EQUATES         ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          #INCLUDE "P16F877A.INC"       ; standard labels

; GPR 70 - 75 allocated to included LCD display routine

count     EQU       30        ; Counter for ADC setup delay
ADbin     EQU       31        ; Binary input value
huns      EQU       32        ; Hundreds digit in decimal value
tens      EQU       33        ; Tens digit in decimal value
ones      EQU       34        ; Ones digit in decimal value

; PROGRAM BEGINS ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          ORG       0                   ; Default start address
          NOP                           ; required for ICD mode

; Port & display setup ------------------------------------------------

          BANKSEL   TRISC               ; Select bank 1
          CLRF      TRISD               ; Display port is output
          MOVLW     B'00000011'         ; Analogue input setup code
          MOVWF     ADCON1              ; Left justify result,
                                        ; Port A = analogue inputs

          BANKSEL   PORTC               ; Select bank 0
          CLRF      PORTD               ; Clear display outputs
          MOVLW     B'01000001'         ; Analogue input setup code
          MOVWF     ADCON0              ; f/8, RA0, done, enable

          CALL      inid                ; Initialise the display

; MAIN LOOP ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

start     CALL      getADC              ; read input
          CALL      condec              ; convert to decimal
          CALL      putLCD              ; display input
          GOTO      start               ; jump to main loop

; SUBROUTINES ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Read ADC input and store --------------------------------------------

getADC    BSF       ADCON0,GO ; start ADC..
wait      BTFSC     ADCON0,GO ; ..and wait for finish
          GOTO      wait
          MOVF      ADRESH,W            ; store result high byte
          RETURN

; Convert input to decimal --------------------------------------------

condec    MOVWF     ADbin               ; get ADC result
          CLRF      huns                ; zero hundreds digit
          CLRF      tens                ; zero tens digit
          CLRF      ones                ; zero ones digit
```

Program 7.1 8-bit analogue input

```
; Calclulate hundreds -------------------------------------------------

         BSF       STATUS,C           ; set carry for subtract
         MOVLW     D'100'             ; load 100
sub1     SUBWF     ADbin              ; and subtract from result
         INCF      huns               ; count number of loops
         BTFSC     STATUS,C           ; and check if done
         GOTO      sub1               ; no, carry on

         ADDWF     ADbin              ; yes, add 100 back on
         DECF      huns               ; and correct loop count

; Calculate tens digit ------------------------------------------------

         BSF       STATUS,C           ; repeat process for tens
         MOVLW     D'10'              ; load 10
sub2     SUBWF     ADbin              ; and subtract from result
         INCF      tens               ; count number of loops
         BTFSC     STATUS,C           ; and check if done
         GOTO      sub2               ; no, carry on

         ADDWF     ADbin              ; yes, add 100 back on
         DECF      tens               ; and correct loop count
         MOVF      ADbin,W            ; load remainder
         MOVWF     ones               ; and store as ones digit

         RETURN                       ; done

; Output to display ---------------------------------------------------

putLCD   BCF       Select,RS ; set display command mode
         MOVLW     080                ; code to home cursor
         CALL      send               ; output it to display
         BSF       Select,RS ; and restore data mode

; Convert digits to ASCII and display ---------------------------------

         MOVLW     030                ; load ASCII offset
         ADDWF     huns               ; convert hundreds to ASCII
         ADDWF     tens               ; convert tens to ASCII
         ADDWF     ones               ; convert ones to ASCII

         MOVF      huns,W             ; load hundreds code
         CALL      send               ; and send to display
         MOVLW     '.'                ; load point code
         CALL      send               ; and output
         MOVF      tens,W             ; load tens code
         CALL      send               ; and output
         MOVF      ones,W             ; load ones code
         CALL      send               ; and output
         MOVLW     ' '                ; load space code
         CALL      send               ; and output
         MOVLW     'V'                ; load volts code
         CALL      send               ; and output
         MOVLW     'o'                ; load volts code
         CALL      send               ; and output
         MOVLW     'l'                ; load volts code
         CALL      send               ; and output
         MOVLW     't'                ; load volts code
         CALL      send               ; and output
         MOVLW     's'                ; load volts code
         CALL      send               ; and output

         RETURN                       ; done


; INCLUDED ROUTINES ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Include LCD driver routines
;
         #INCLUDE "LCDIS.INC"
;        Contains routines:
;        inid:     Initialises display
;        onems:    1 ms delay
;        xms:      X ms delay
;                  Receives X in W
;        send:     Sends a character to display
;                  Receives: Control code in W (Select,RS=0)
;                                ASCII character code in W (RS=1)

         END       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Program 7.1 *Continued*

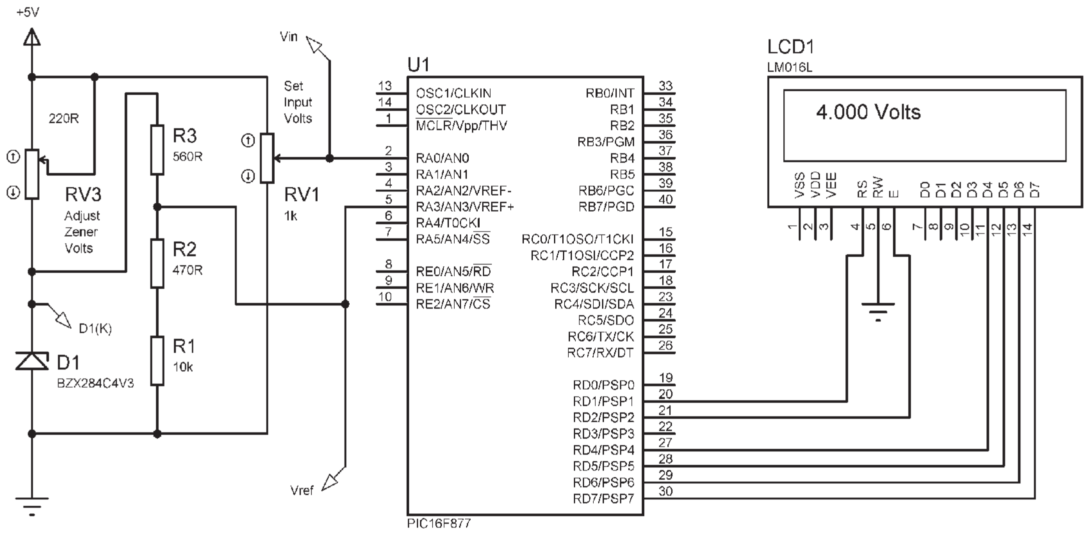


Figure 7.4 10-bit conversion circuit

Amplifier Interfaces

Having developed the analogue input conversion and display process, we can move on to the interfacing hardware itself. Input signals often need conditioning before being fed to the MCU analogue inputs. This can involve amplifiers to increase the signal amplitude, attenuators to reduce it, and filters to change the frequency response. For now, we will limit ourselves to DC signals and amplifiers, as these are most frequently used in MCU applications and are more straightforward. For processing of AC signals, standard references should be consulted.

Figure 7.6 shows a range of different amplifiers connected to the PIC MCU. They are connected to RA0 by a multi-way switch, so that the output of each may be displayed, using the previously developed 8-bit conversion and display program (Program 7.1). The basic op-amp configurations are summarised in Figure 7.7.

The op-amp (IC amplifier) is a high-gain amplifier with inverting and non-inverting inputs, with the output voltage controlled by the input differential voltage. However, since the differential gain is very high, typically >1,00,000, the operating input differential voltage is very small. As a result, we can assume that the gain and bandwidth (frequency response) are controlled by the external components only, and are independent of the amplifier itself.

```
ADC10
        Load 10-bit, right justified binary (0-1023)
        Multiply by 4 (0-4092) by shift left
        Clear BCD registers

        REPEAT
                Subtract $E8_{16}$ from low byte
                Subtract $3_{16}$ from high byte
                Increment thousands digit
        UNTIL remainder < $03E8_{16}$ (1000)

        REPEAT
                Subtract $64_{16}$ from low byte
                Borrow from high byte
                Increment hundreds digit
        UNTIL remainder < $64_{16}$ (100)

        REPEAT
                Subtract 10 from low byte
                Increment tens digit
        UNTIL remainder < 10

        Remainder = ones digits

RETURN
```

Figure 7.5 10-bit binary conversion routine outline

When used as a linear amplifier, the feedback must be negative. Essentially, this means the feedback signal path must be connected to the minus input terminal. The basic rules for ideal op-amp circuit analysis are as follows:

- Differential gain = ∞ (for voltage applied between + and – terminals)
- Differential voltage = 0 (terminals + and – are at the same voltage)
- Input resistance = ∞ (zero input current at + and – terminals)
- Output impedance = 0 (infinite current can be sunk or source at the output)
- Bandwidth = ∞ (all frequencies are amplified equally)
- Feedback is negative (signal connected from output to – terminal)

These rules allow amplifier circuit analysis to be greatly simplified, and give results which are accurate enough for most applications.

IC (integrated circuit) amplifiers can operate with dual or single supplies. Dual supplies, which are the norm, make the circuit design easier, because the output can swing positive and negative around 0 V. +/− 15 V and +/− 5 V are typical supply values, with 15 V supplies giving a higher output voltage swing.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;       Project:                Interfacing PICs
;       Source File Name:       TENBIT.ASM
;       Devised by:             MPB
;       Date:                   20-12-05
;       Status:                 Final
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;       Demonstrates 10-bit voltage measurement
;       using an external reference voltage of 4.096V,
;       giving 4mV per bit, and an resolution of 0.1%.
;       The result is converted to BCD for display
;       as a voltage using the standard LCD routines.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        PROCESSOR 16F877
;       Clock = XT 4MHz, standard fuse settings
        __CONFIG 0x3731

;       LABEL EQUATES   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        INCLUDE "P16F877A.INC"
        ; standard register labels

;-----------------------------------------------------------
; User register labels
;-----------------------------------------------------------
; GPR 20 - 2F allocated to included LCD display routine

count   EQU     30      ; Counter for ADC setup delay
ADhi    EQU     31      ; Binary input high byte
ADlo    EQU     32      ; Binary input low byte
thos    EQU     33      ; Thousands digit in decimal
huns    EQU     34      ; Hundreds digit in decimal value
tens    EQU     35      ; Tens digit in decimal value
ones    EQU     36      ; Ones digit in decimal value

;-----------------------------------------------------------
; PROGRAM BEGINS
;-----------------------------------------------------------

        ORG     0               ; Default start address
        NOP                     ; required for ICD mode

;-----------------------------------------------------------

; Port & display setup

        BANKSEL TRISC           ; Select bank 1
        CLRF    TRISD           ; Display port is output
        MOVLW   B'10000011'     ; Analogue input setup code
        MOVWF   ADCON1          ; Right justify result,
                                ; Port A = analogue inputs
                                ; with external reference

        BANKSEL PORTC           ; Select bank 0
        CLRF    PORTD           ; Clear display outputs
        MOVLW   B'01000001'     ; Analogue input setup code
        MOVWF   ADCON0          ; f/8, RA0, done, enable

        CALL    inid            ; Initialise the display

;-----------------------------------------------------------
; MAIN LOOP
;-----------------------------------------------------------

start   CALL    getADC          ; read input
        CALL    con4            ; convert to decimal
        CALL    putLCD          ; display input
        GOTO    start           ; jump to main loop
```

Program 7.2 10-bit conversion

```
;----------------------------------------------------------
; SUBROUTINES
;----------------------------------------------------------
; Read ADC input and store
;----------------------------------------------------------
getADC  MOVLW   007             ; load counter
        MOVWF   count
down    DECFSZ  count           ; and delay 20us
        GOTO    down

        BSF     ADCON0,GO       ; start ADC..
wait    BTFSC   ADCON0,GO       ; ..and wait for finish
        GOTO    wait
        RETURN
;----------------------------------------------------------
; Convert 10-bit input to decimal
;----------------------------------------------------------
con4    MOVF    ADRESH,W        ; get ADC result
        MOVWF   ADhi            ; high bits
        BANKSEL ADRESL          ; in bank 1
        MOVF    ADRESL,W        ; get ADC result
        BANKSEL ADRESH          ; default bank 0
        MOVWF   ADlo            ; low byte

; Multiply by 4 for result 0 - 4096 by shifting left.........

        BCF     STATUS,C        ; rotate 0 into LSB and
        RLF     ADlo            ; shift low byte left
        BTFSS   STATUS,C        ; carry out?
        GOTO    rot1            ; no, leave carry clear
        BSF     STATUS,C        ; rotate 1 into LSB and
rot1    RLF     ADhi            ; shift high byte left

        BCF     STATUS,C        ; rotate 0 into LSB
        RLF     ADlo            ; rotate low byte left again
        BTFSS   STATUS,C        ; carry out?
        GOTO    rot2            ; no, leave carry clear
        BSF     STATUS,C        ; rotate 1 into LSB and
rot2    RLF     ADhi            ; shift high byte left


; Clear BCD registers.......................................

clrbcd  CLRF    thos            ; zero thousands digit
        CLRF    huns            ; zero hundreds digit
        CLRF    tens            ; zero tens digit
        CLRF    ones            ; zero ones digit


; Calclulate thousands low byte .............................

tholo   MOVF    ADhi,F          ; check high byte
        BTFSC   STATUS,Z        ; high byte zero?
        GOTO    hunlo           ; yes, next digit

        BSF     STATUS,C        ; set carry for subtract
        MOVLW   0E8             ; load low byte of 1000
        SUBWF   ADlo            ; and subtract low byte
        BTFSC   STATUS,C        ; borrow from high bits?
        GOTO    thohi           ; no, do high byte
        DECF    ADhi            ; yes, subtract borrow

; Calculate thousands high byte..............................
thohi   BSF     STATUS,C        ; set carry for subtract
        MOVLW   003             ; load high byte of 1000
        SUBWF   ADhi            ; subtract from high byte
        BTFSC   STATUS,C        ; result negative?
        GOTO    incth           ; no, inc digit and repeat

        ADDWF   ADhi            ; yes, restore high byte

; Restore remainder when done ...............................
        BCF     STATUS,C        ; clear carry for add
        MOVLW   0E8             ; load low byte of 1000
        ADDWF   ADlo            ; add to low byte
        BTFSC   STATUS,C        ; carry out?
        INCF    ADhi            ; yes, inc high byte
        GOTO    hunlo           ; and do next digit

; Increment thousands digit and repeat.......................
incth   INCF    thos            ; inc digit
        GOTO    tholo           ; and repeat
```

Program 7.2 *Continued*

```
; Calclulate hundreds .......................................

hunlo   MOVLW   064             ; load 100
        BSF     STATUS,C        ; set carry for subtract
        SUBWF   ADlo            ; and subtract low byte
        BTFSC   STATUS,C        ; result negative?
        GOTO    inch            ; no, inc hundreds & repeat

        MOVF    ADhi,F          ; yes, test high byte
        BTFSC   STATUS,Z        ; zero?
        GOTO    remh            ; yes, done
        DECF    ADhi            ; no, subtract borrow
inch    INCF    huns            ; inc hundreds digit
        GOTO    hunlo           ; and repeat

remh    ADDWF   ADlo            ; restore onto low byte
; Calculate tens digit......................................

subt    MOVLW   D'10'           ; load 10
        BSF     STATUS,C        ; set carry for subtract
        SUBWF   ADlo            ; and subtract from result
        BTFSS   STATUS,C        ; and check if done
        GOTO    remt            ; yes, restore remainder
        INCF    tens            ; no, count number of loops
        GOTO    subt            ; and repeat
; Restore remainder.........................................

remt    ADDWF   ADlo            ; yes, add 10 back on
        MOVF    ADlo,W          ; load remainder
        MOVWF   ones            ; and store as ones digit

        RETURN                  ; done
;----------------------------------------------------------
; Output to display
;----------------------------------------------------------

putLCD  BCF     Select,RS       ; set display command mode
        MOVLW   080             ; code to home cursor
        CALL    send            ; output it to display
        BSF     Select,RS       ; and restore data mode
; Convert digits to ASCII and display......................
        MOVLW   030             ; load ASCII offset
        ADDWF   thos            ; convert thousands to ASCII
        ADDWF   huns            ; convert hundreds to ASCII
        ADDWF   tens            ; convert tens to ASCII
        ADDWF   ones            ; convert ones to ASCII
        MOVF    thos,W          ; load thousands code
        CALL    send            ; and send to display
        MOVLW   '.'             ; load point code
        CALL    send            ; and output
        MOVF    huns,W          ; load hundreds code
        CALL    send            ; and send to display
        MOVF    tens,W          ; load tens code
        CALL    send            ; and output
        MOVF    ones,W          ; load ones code
        CALL    send            ; and output
        MOVLW   ' '             ; load space code
        CALL    send            ; and output
        MOVLW   'V'             ; load volts code
        CALL    send            ; and output
        MOVLW   'o'             ; load volts code
        CALL    send            ; and output
        MOVLW   'l'             ; load volts code
        CALL    send            ; and output
        MOVLW   't'             ; load volts code
        CALL    send            ; and output
        MOVLW   's'             ; load volts code
        CALL    send            ; and output
        RETURN                  ; done
;---------------------------------------------------------
; INCLUDED ROUTINES
;---------------------------------------------------------
; Include LCD driver routine
;
        INCLUDE "LCDIS.INC"
;
;       Contains routines:
;       init:   Initialises display
;       onems:  1 ms delay
;       xms:    X ms delay
;               Receives X in W
;       send:   sends a character to display
;               Receives: Control code in W (Select,RS=0)
;                         ASCII character code in W (RS=1)
;
;---------------------------------------------------------
        END                     ; of source code
```

Program 7.2 *Continued*

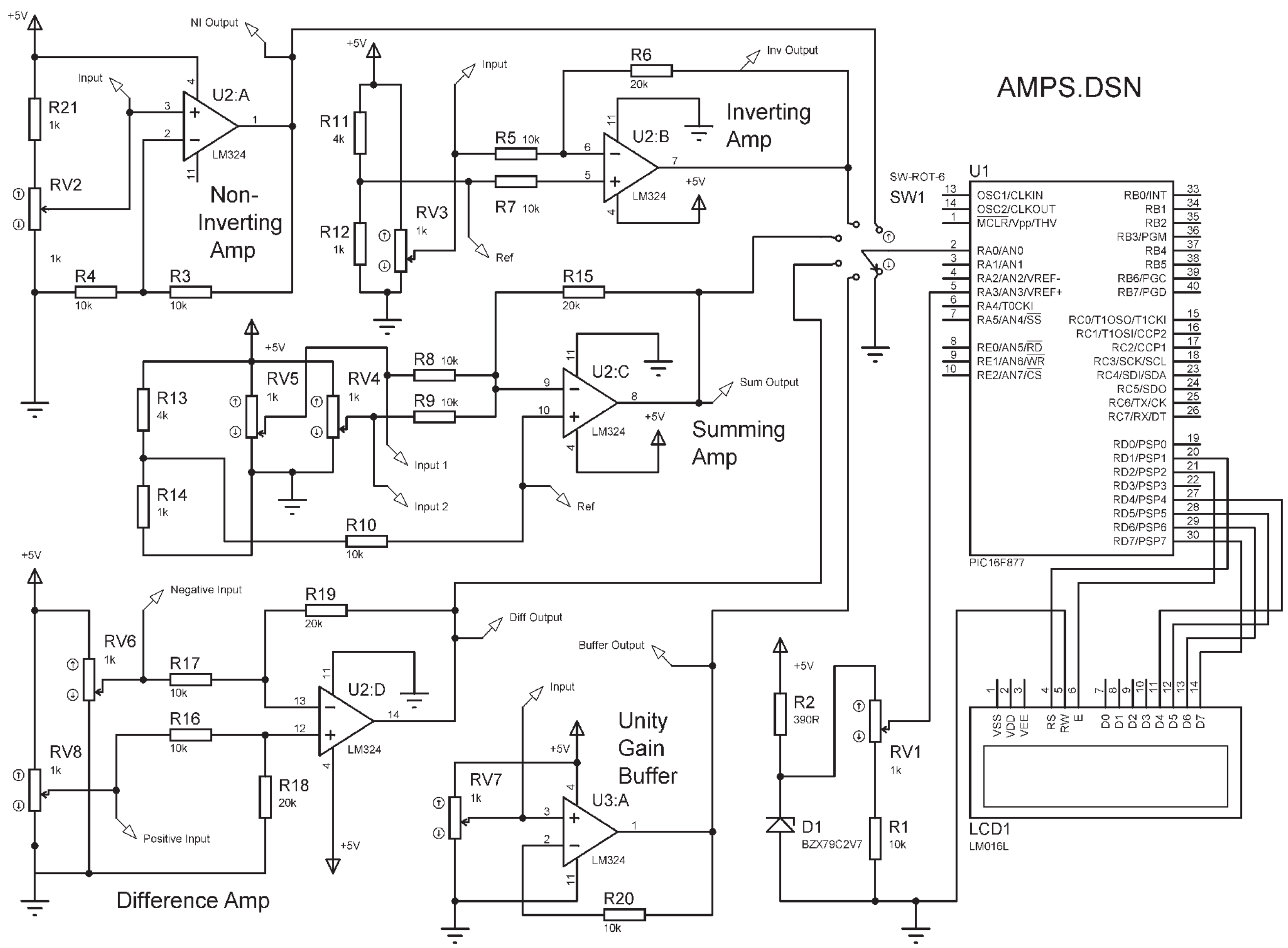


Figure 7.6 Basic amplifier interface circuits

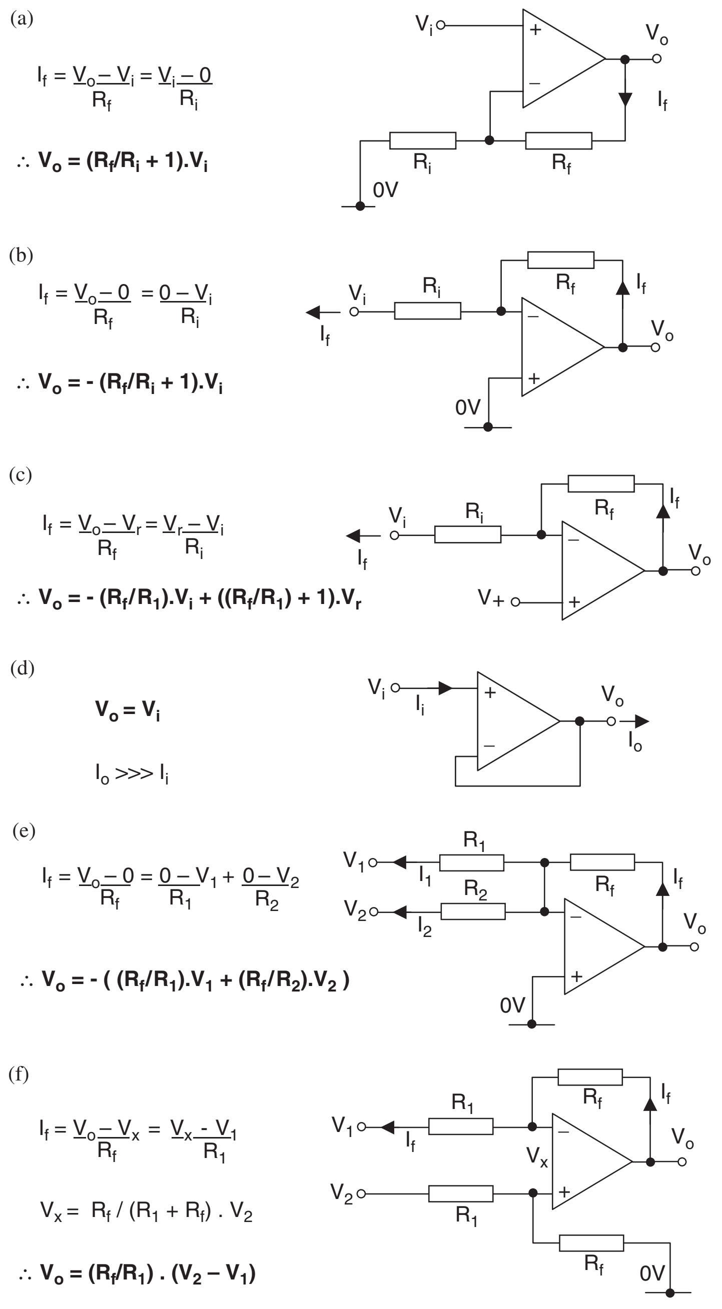

(a)
$I_f = \frac{V_o - V_i}{R_f} = \frac{V_i - 0}{R_i}$
$\therefore\ \mathbf{V_o = (R_f/R_i + 1).V_i}$
V_i
V_o
I_f
R_i
R_f
0V
(b)
$I_f = \frac{V_o - 0}{R_f} = \frac{0 - V_i}{R_i}$
$\therefore\ \mathbf{V_o = -\ (R_f/R_i + 1).V_i}$
V_i
R_i
R_f
I_f
V_o
0V
(c)
$I_f = \frac{V_o - V_r}{R_f} = \frac{V_r - V_i}{R_i}$
$\therefore\ \mathbf{V_o = -\ (R_f/R_1).V_i + ((R_f/R_1) + 1).V_r}$
V_i
R_i
R_f
I_f
V_o
V+
(d)
$\mathbf{V_o = V_i}$
$I_o >>> I_i$
V_i
I_i
V_o
I_o
(e)
$I_f = \frac{V_o - 0}{R_f} = \frac{0 - V_1}{R_1} + \frac{0 - V_2}{R_2}$
$\therefore\ \mathbf{V_o = -\ (\ (R_f/R_1).V_1 + (R_f/R_2).V_2\ )}$
V_1
V_2
R_1
R_2
I_1
I_2
R_f
I_f
V_o
0V
(f)
$I_f = \frac{V_o - V_x}{R_f} = \frac{V_x - V_1}{R_1}$
$V_x = R_f\,/\,(R_1 + R_f)\,.\,V_2$
$\therefore\ \mathbf{V_o = (R_f/R_1)\,.\,(V_2 - V_1)}$
V_1
V_2
R_1
R_f
I_f
V_x
V_o
0V

Figure 7.7 Basic amplifier configurations: (a) non-inverting amplifier; (b) inverting amplifier; (c) inverting amplifier with offset; (d) unity gain buffer; (e) summing amplifier; (f) difference amplifier

On the other hand, it is convenient in microprocessor systems to use the same single supply used by the digital circuits, +5 V, and to avoid the need to provide separate dual op-amp supplies. Some op-amps are designed specifically to operate with a single supply, such as the LM324 type used in the examples here. However, +5 V provides only a limited voltage swing; the lowest output may not reach 0 V, and the maximum will typically be even more limited, possibly less than 4 V, depending on the op-amp type.

The output does not usually reach the supply values due to residual volt drops across internal components. The 324, for example, can only reach about 3.5 V, so circuits have to operate within this limited output swing. The 8-bit range in the demo circuits is limited to 2.56 V, so that the amp outputs are operating comfortably within the upper limit, but we cannot assume that voltages near zero will be represented accurately. In the test circuits, therefore, output swing will be limited, and offsets introduced to allow the amplifier to operate within its limits.

Non-inverting Amplifier

The basic configuration for the non-inverting amp is shown in Figure 7.7 (a). The input is applied to the + terminal, and feedback and gain controlled by the resistor network R_f and R_1. If we assume that the voltage between the terminals is zero (rule 2), the voltage at the – terminal must be the same as the voltage at the + terminal. We can then write down an equation for the feedback network using Ohm's law applied to each resistor, assuming the current flow is from the output through the resistors to ground. This is possible because it is assumed that none of the current is lost at the input terminal, as it has infinite input resistance (rule 3). A simple re-arrangement of the equation allows us to predict the output voltage in terms of the resistor values.

$$V_0 = (R_f/R_1 + 1)V_1$$

The main advantage of this configuration is that the input impedance is very high (in theory, infinite). The loading on the signal source is therefore negligible. The disadvantage is that the input is operating with an offset voltage, which reduces its accuracy, particularly with a single supply, as is the case in our demo circuit. In addition, the high input impedance makes it susceptible to noise.

In the demonstration non-inverting amplifier shown in the test circuit, Figure 7.6, the feedback resistors are both 10k, giving a gain of 10k/10k + 1 = 2. The test input pot RV2 is connected to provide 0–2.5 V, so the output should, in theory, be 0–5.0 V. In practice, the 324-simulation model provides a minimum output of 0.03 V and a maximum of about 4.0 V, due to the output stage limitations. An output offset (constant error) of 3 mV is also evident – if this is a potential problem, an op-amp with inherently low offset, or one with offset adjustment, can be used, or an external offset adjustment included in the circuit design.

Inverting Amplifier

The analysis is even simpler for this configuration, since the input terminals are at 0 V. The equation for the feedback current predicts that

$$V_0 = -(R_f/R_1)V_1$$

The negative sign indicates that the output is inverted, that is, it goes negative when the input is going positive, and vice versa. Unfortunately, the input impedance is inherently low, being equal to the value of R_1. A significant input current is required to or from the signal source for this configuration to work correctly. However, with symmetrical supplies, it can operate with zero offset, which reduces errors.

In the demo circuit (Figure 7.6), the inverting amplifier is operating with an offset of 1 V. The + terminal is connected to a reference voltage of 1.000 V produced by a voltage divider across the supply. It is fed to the input terminal via a 10k, which helps to equalise the input offset currents at the + and – terminals. The gain (G) is 20k/10k = 2, and the output polarity inverted. Analysis (Figure 7.7 (c)) shows that the output voltage is given by

$$V_0 = (G+1)V_r - GV_i$$

If $V_r = 1.00$ and $G = 2.00$
$V_0 = 3 - 2V_i$

Unity Gain Buffer

This is a special case of the non-inverting amplifier, where the feedback is 100%, that is, zero feedback resistance, giving a gain of 1 (Figure 7.7 (d)). The output voltage is then the same as the input voltages. So what is the point of the circuit? It is to provide current gain. The input current is very small (large input resistance at the + terminal) but the output current can be large, giving a high current gain. In practice, with standard op-amps, the output current would typically be limited to about 20 mA, but high current output IC amps are available, or a further current driver stage can be added using a discrete transistor.

Summing Amplifier

This is a development of the inverting amplifier, which has additional inputs. Only two are shown in Figure 7.7 (e), but more are possible. The output is determined by the sum of the input voltages, taking into account the input resistor weightings.

$$-V_0 = G_1V_1 + G_2V_2 + G_3V_3 + \ldots$$
$$\text{where } G_1 = R_f/R_1,\ G_2 = R_f/R_2 \ldots$$

For a summing amplifier with offset, as seen in the demo circuits, it can be shown that

$$V_0 = V_r\,(nG + 1) - G\,(V_1 + V_2 + \ldots V_n)$$

for an amplifier with identical input resistors (same gain for each input).

The demo circuit was tested as follows:

$R_f = 20\text{k}$ and $R_1 = 10\text{k}$ $\quad\therefore R_f/R_1 = 2 = G$

$V_r = 1.000$ V $\quad V_1 = 0.585$ V $\quad V_2 = 0.866$ V

Predicted output voltage

$V_{op} = (4 + 1) – 2\,(0.585 + 0.866) = 5 – 2.90 = 2.10$ V

Simulation output voltage $\quad V_{os} = 2.11$ V

Difference Amplifier

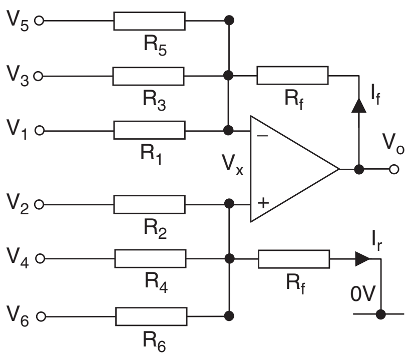
The difference amplifier (Figure 7.7 (f)) gives an output which is proportional to the difference between the input voltages. The mathematical model is again derived by analysing the current flow in the feedback path, and calculating the voltage at the + terminal from the voltage divider connected to it. This gives the relationship

$$V_0 = R_f/R_1\,(V_2V_1) = G\,(V_2 – V_1)$$

if the resistors connected to both terminals have the same values, as shown. V_2 is the input on the + terminal, V_1 on the – terminal. This circuit can be used with sensors that have a positive offset on their output, to bring the output voltages into the right range (0–3.5 V with a +5 V single supply).

Universal Amplifier

The above types of amplifier can be regarded as special cases of the universal amplifier (Figure 7.8). This has both difference and summing inputs, and can be adapted to applications where a combination of these is required.



Current sums

$$I_f = \frac{V_o - V_x}{R_f} = \frac{(V_x - V_1)}{R_1} + \frac{(V_x - V_3)}{R_3} + \frac{(V_x - V_5)}{R_5}$$

$$I_r = \frac{V_x - 0}{R_f} = \frac{(V_2 - V_x)}{R_2} + \frac{(V_4 - V_x)}{R_4} + \frac{(V_6 - V_x)}{R_6}$$

Assume $R_1 = R_2 = R_3 = R_4 = R_5 = R_6 = R_i$

Then $\boldsymbol{V_o = R_f/R_i\,(V_2 + V_4 + V_6) - (V_1 + V_3 + V_5)}$

Figure 7.8 Universal amplifier

The universal amplifier can have any number of inputs and outputs, but to make the mathematical model as simple as possible, we will set the following conditions:

- the number of inverting inputs is equal to the number of non-inverting inputs
- the input resistors (R_i) are all equal
- the feedback resistors (R_f) are equal

By summing the current at the op-amp input terminals, we can show that

$$V_0 = R_f/R_i\,(V_2 + V_4 + V_6 + \ldots) - (V_1 + V_3 + V_5 + \ldots)$$

The output voltage is given by the arithmetic sum of the input voltages multiplied by the gain, where the non-inverting (+) inputs are even numbered and the inverting (–) odd. The amplifier then behaves as a combination summing and difference amplifier, allowing positive and negative signals and offset inputs to be added as required.

Transient & Frequency Response

Real circuits have stray capacitance associated with the signal conductors and components, especially when fabricated in IC form, where planar components are formed in close proximity. This affects the response to switching and AC signals. It can be represented by a capacitor across the feedback resistor in our op-amp circuits. Such a component may often be deliberately included in an amplifier design, as it improves general stability and rejection of noise in DC amplifiers, and controls the bandwidth in AC applications (Figure 7.9 (a)).

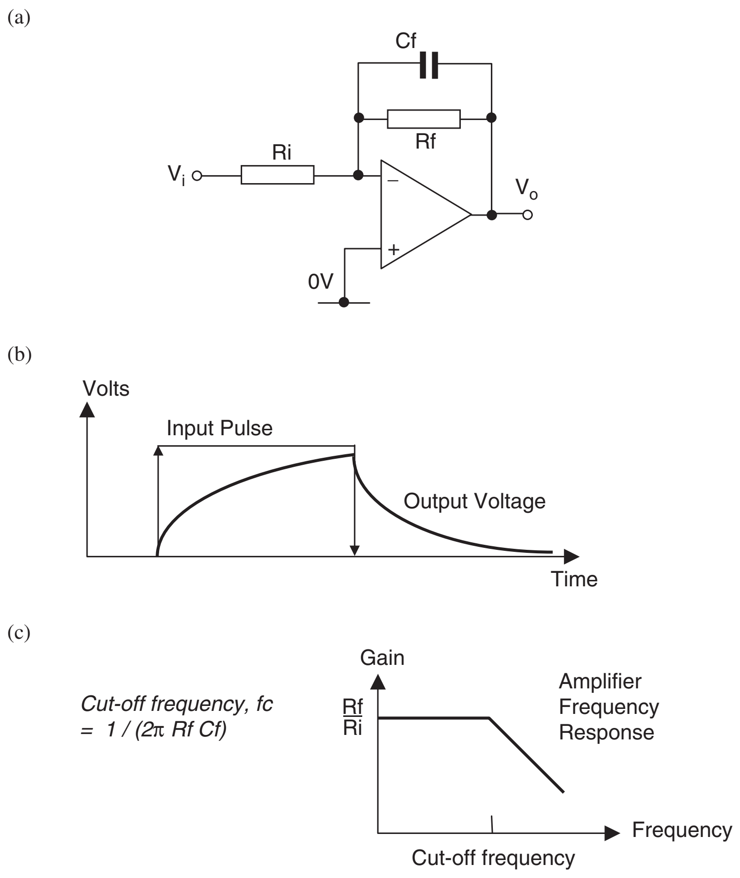


Figure 7.9 Feedback capacitance: (a) basic circuit; (b) transient response; (c) frequency response

Transient Behaviour

If the input is switched rapidly between DC values, e.g. a step input from 0 to +1 V, the circuit with capacitative feedback acts as an integrator, and a response such as that in Figure 7.9 (b) is obtained. The output rises slowly, following an exponential curve, and may not reach the final value before the input is reversed. This shows why DC switching frequencies are limited in all active digital and analogue circuits – the outputs may not reach the required levels if the switching is too fast.

This transient effect must be anticipated with switched inputs to the microcontroller ADC, especially as the ADC itself has an RC sample and hold input. If a relatively large value of capacitor is used with a large (or infinite) resistance, the curve is so extended that it appears to be a straight line. This integrator circuit can be used to generate a triangular or sawtooth waveform.

Frequency Response

So far, we have assumed that DC voltages are being used, but the amplifier analysis also applies equally to AC signals, except that at higher frequencies the amplifier circuits will be affected by limited frequency response. Any real amplifier has limited bandwidth; DC-coupled amplifiers work at 0 Hz, but an upper frequency limit always applies. Many op-amps have internal or external compensation to deliberately limit the bandwidth to a known frequency, and to improve overall stability (op-amps have a tendency to turn spontaneously into oscillators for no obvious reason!).

A first-order frequency response is shown in Figure 7.9 (c). At higher frequencies, the feedback capacitor has a low impedance (AC resistance) compared with the feedback resistor. The resistor is therefore bypassed by a lower parallel impedance, reducing the gain, *G*. The frequency at which the impedance of the resistor and capacitor are equal is called the cut-off frequency. Above this frequency, the gain rolls off at 20 dB per decade of frequency. This appears as a straight line if plotted on logarithmic axes. The cut-off frequency can be deliberately reduced by increasing the capacitance value in the feedback path. External compensation pins are sometimes provided to connect the additional capacitance.

Instrumentation Amplifier

Many sensors that need to be connected to a microcontroller analogue input have a rather small output signal. In addition, it may only be available as a differential voltage output, that is between two points with a large common mode voltage.

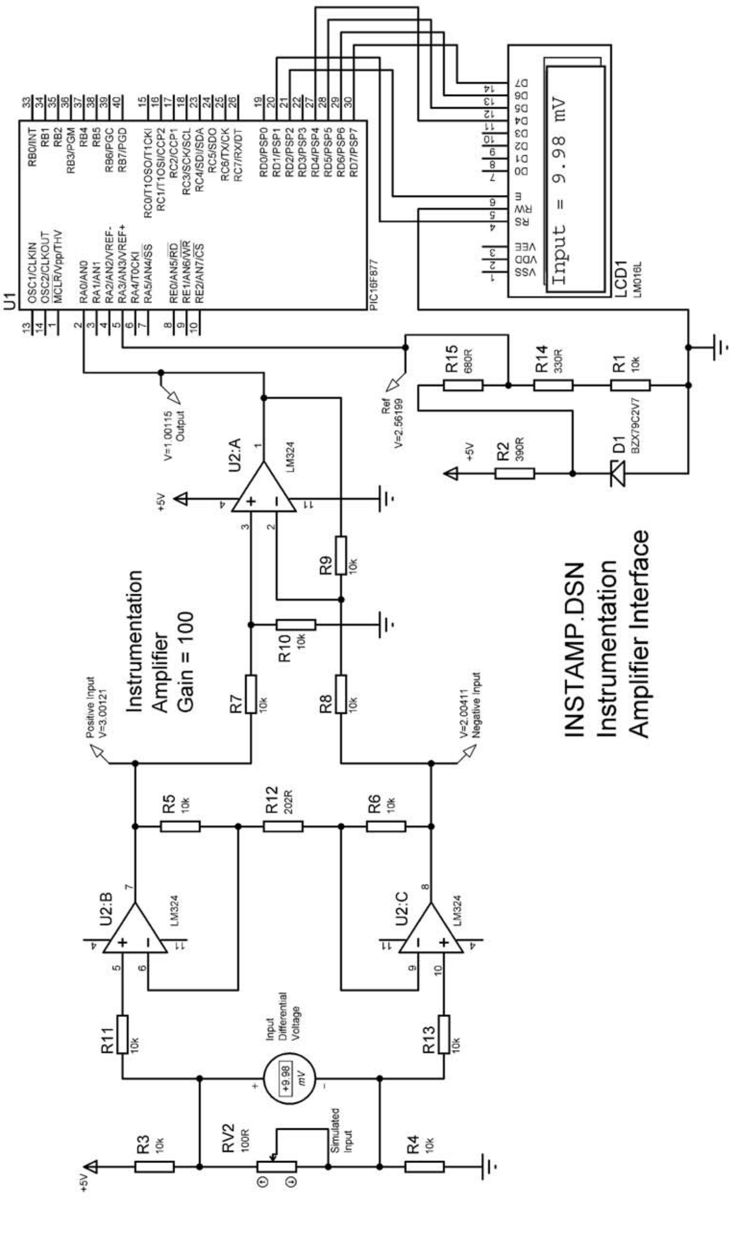


Figure 7.10 Instrumentation amplifier

Strain gauges are usually connected in this way; they might measure small changes in the shape of a mechanical part under stress, such as a strut in a crane jib. The sensor typically consists of four strain-sensitive resistors connected in a bridge arrangement, such that a change in their resistance due to stretching is output as a small differential voltage, typically in the range 0–10 mV.

A sensitive amplifier is needed, with a high gain and high input resistance, which reduces the current drawn from the sensor and hence the errors. A single-stage non-inverting amplifier has a high input resistance, but does not have differential inputs. The difference amplifier has these, but has low input resistances. In addition, if configured for a high gain, with a high value for R_f, the feedback current is small, and thus the amplifier is more susceptible to noise and offset errors.

The solution is an instrumentation amplifier, which combines the required features (Figure 7.10). It is made up of two stages: the main difference amplifier and a pair of high impedance input stages. In order to see some of the limitations, our standard single supply op-amp, LM324, has been used, but the performance can be improved by selecting a higher specification op-amp, or buying the instrumentation amplifier as a special package.

The gain of the amplifier is set by the ratio of feedback resistor chain connected between the outputs of the input stages, from the relationship

$$G = 1 + 2R_2/R_1 \qquad \text{where } R_2 = 10\text{k and } R_1 = 202R$$

$$\therefore\ G = 1 + 20000/202 = 100$$

This provides the required gain. The input maximum differential voltage is 10 mV, the output differential is 1.00 V. This is then fed to the differential output stage, which has unity gain, whose role is to provide a single ended output (i.e., measured with respect to 0 V).

Current Loop

If a DC signal is to be transmitted over a long connection, say more than 1 m, the resistance in the line will cause a volt drop, which will affect the accuracy of the received voltage. In this case it is better to represent the measurement as a current, rather than a voltage, since the current in a closed loop cannot be lost, except at the load.

If the operation of a simple inverting amplifier is considered ideal, the current in the feedback resistor must be the same as the current in the input resistor. If the input voltage and input resistance are constant, the feedback current will be constant, with the output voltage of the op-amp adjusting itself for any change in the feedback resistance value. This leads us to a general design for a constant current source, derived from a constant voltage at the input.

In Figure 7.11, a zener diode provides the constant voltage, and the current in the feedback path will then be constant (within limits), and independent of the feedback resistance value. This principle can be applied to obtain a current in this path, which is controlled by a variable input voltage from a sensor.

In Figure 7.12, a demonstration circuit is shown which will give an output change of 1.00 V for an input change of 100 mV, that is, an overall gain of 10. However, the significant feature is the current loop formed by the feedback path of the line driver. A long connection between this stage and the output differential amplifier represents a line which can have a variable resistance, depending on the length and cabling type. We need the output to be independent of the variation of this resistance, which is represented by variable 10*R* pots.

R_5 and R_6 (100R) are the input and feedback resistors in the line driver amplifier. The input stage is a simple non-inverting amplifier with a gain of 10, which feeds a voltage to the line driver, which changes by 1.00 V when the test switch is operated. The current switches between 0 and 10 mA, to give 1.00 V across the line driver feedback resistor, R_6. This is connected across the inputs of the unity gain output differential amplifier. The output of a standard op-amp is limited to about 25 mA, so the line must operate at less than this value. On the other hand, the higher the current, the better the signal-to-noise ratio is likely to be.

Since the current loop is implemented using single supply op-amps running at 5 V, the amplifiers are all offset by 1.50 V, to avoid voltage outputs near 0 V. The output switching is then between 1.50 V and 2.50 V. This is achieved to within about 1% in the simulation, the error being mainly due to variation in the individual amplifier offset conditions. The exact common offset of 1.50 V

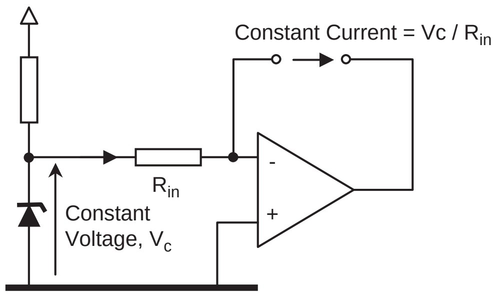


Figure 7.11 Constant current source

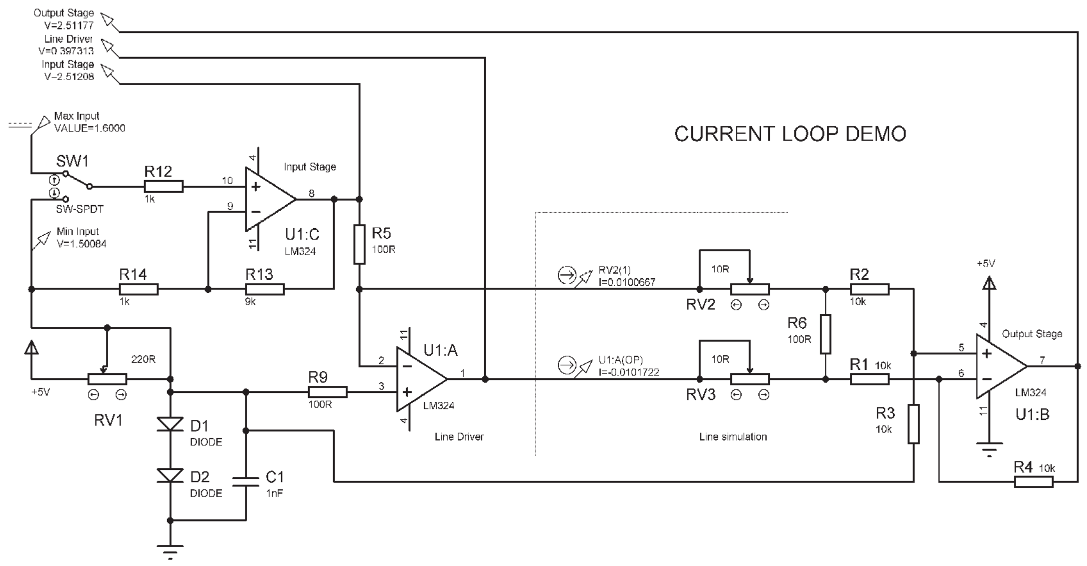


Figure 7.12 Current loop system

is derived from a stack of two diodes supplied with a current, which can be tweaked to obtain the desired volt drop. If power diodes are used, heating effects can be reduced (remember, the diode volt drop changes by 2 mV/°C). This offset will also assist in interfacing sensors which may need to go negative with respect to the reference level.

The standard current loop sensor interface operates at 4–20 mA, and is designed to provide power to the remote sensor as well as allowing it to control the current drawn from an external supply. The operating range is therefore 16 mA, convenient for converting to digital form. If zero current is detected, this will normally be interpreted as a fault condition, for example, an open circuit in the current loop.

Comparators

Comparators are a different type of op-amp circuit. Here the op-amp is used in open loop mode (no negative feedback) to compare two input voltages on the + and – terminals, and switch the output high or low depending on the relative polarity. If the + terminal is positive with respect to the – terminal, the output will go high, and vice versa.

A specific type of op-amp is normally used for this type of application, which has an open collector output. The output transistor switching circuit has to be completed by an external pull-up (load) resistor. This allows the output switching voltage to be different from the comparator supply voltage, which is useful for interfacing circuits operating at, say, +/− 15 V or with a 24 V single supply, which must be connected to a TTL MCU input or output. The switching speed can be increased by using a lower value pull-up resistor, at the cost of higher power consumption. Some MCUs have comparator inputs built in, as a simple form of analogue input.

Three types of comparator circuit are shown in Figure 7.13. The default chip type used here is the TLC339, a quad comparator.

Simple Comparator

The comparator detects whether the input is above or below the reference voltage. The circuit shown (Figure 7.13 (a)) has a reference voltage of 2.5 V applied to the - terminal. As the input changes, the output switches at this voltage. The transfer characteristic shows the effect by plotting the output against the input voltage. The reference voltage can be changed as required, giving a different switching level. The output of the comparator is connected to an LED indicator in the load circuit, which is useful, but not essential. The open collector output provides sufficient output current to drive an LED (~10 mA), without any additional driver stage.

Trigger Comparator

The output voltage in this circuit (Figure 7.13 (b)) is fed back to the + terminal to set the reference level, which changes depending on whether the output is high or low. The switching level therefore depends on the *previous* setting of the output. This gives two switching levels: the output switches at a higher voltage when increasing from low to high and at a lower voltage when decreasing from high to low. In the circuit shown, the LED circuit affects the switching level, and may be omitted. Notice that the input is applied to the – terminal, so the transfer characteristic is inverted. When identifying circuits, positive feedback indicates a comparator, or an oscillator.

The trigger circuit is often incorporated into digital signal paths as it helps to reduce noise (unwanted high frequencies). In a simple TTL gate, noise on a slowly changing input signal might cause multiple transitions at the output; with a schmitt trigger input (as it is known), once the gate has changed state, it does not change back unless there is a relatively large change in the input in the opposite direction. The PIC MCU has schmitt trigger inputs on the port input buffers for improved noise immunity.

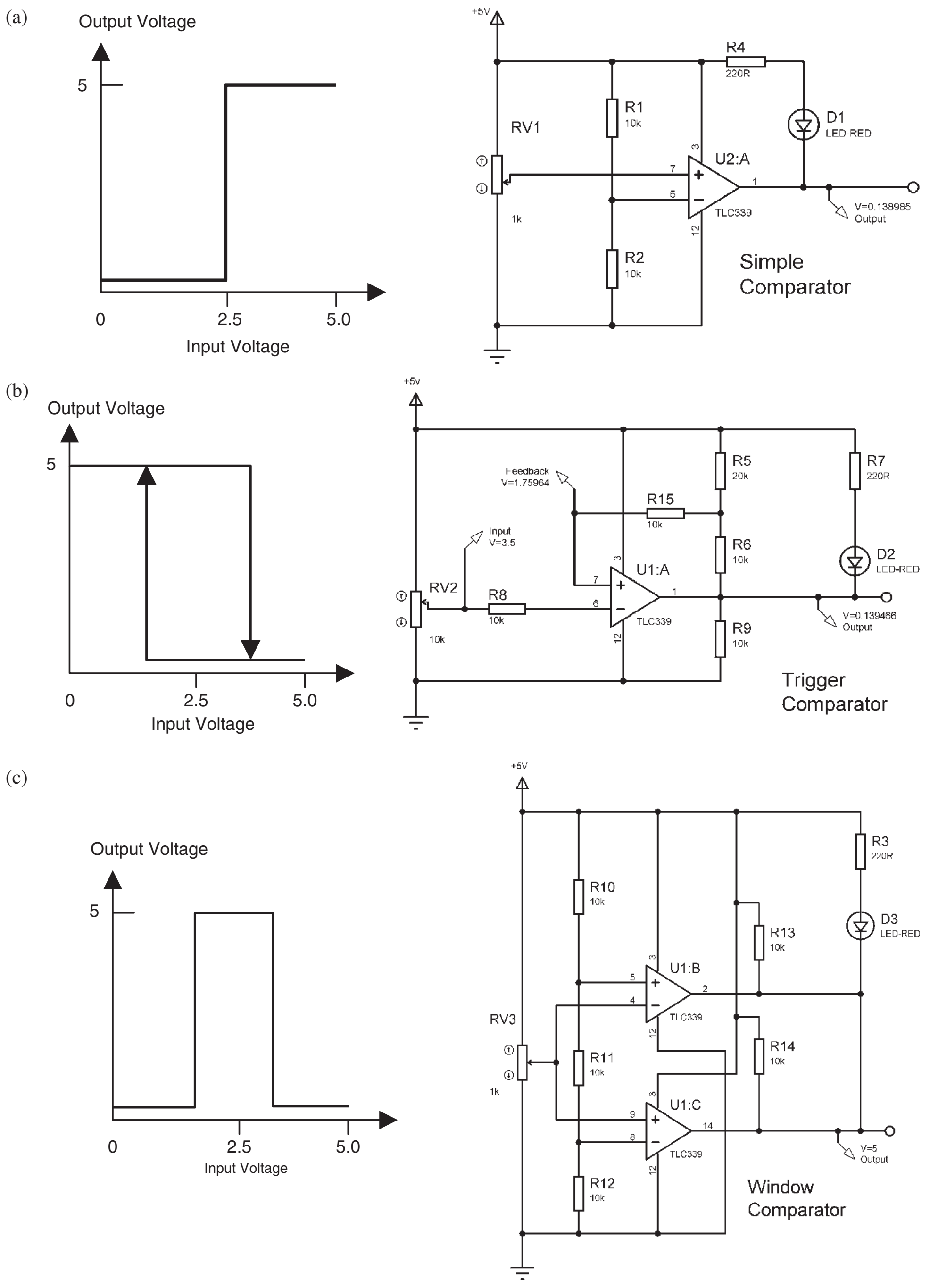


Figure 7.13 Comparators: (a) simple comparator; (b) trigger comparator; (c) window comparator

Window Comparator

In this circuit, two comparators give a range of input voltages between which the output is high, and low when outside this range (or vice versa). Comparator C output is low when the voltage is below about 1.6 V, and comparator B output is low above about 3.3 V. Between these voltages, neither is low, allowing the output to rise to 5 V. The open collectors allow this connection, while it is not allowed with the complementary output drivers in standard op-amps. The circuit is used to detect when a voltage is within or outside a given range, which could be used, for example, in a simple voltage tester giving a pass/fail output.

Op-amp Selection

There are three main types of op-amp for linear applications, plus the open collector comparator type for switching applications,

- Bipolar (e.g. LM324, LM741)
- CMOS (e.g. CA3140)
- BiFet (e.g. TL074)

The 741 is the original, standard, general purpose single op-amp in an 8-pin package. It is based on bipolar transistor technology, with internal compensation (feedback capacitance) to provide a stable, low bandwidth device for DC and audio range applications. The LM324 is a similar type of quad device, designed for single 5 V supply operation. The CMOS type uses FETs (field effect transistors), which have very low input current requirements, and therefore provide low power, high input impedance amplifiers. The BiFet type combines advantages of the bipolar and FET types in one chip. FET inputs provide high input impedance and low input currents (but not necessarily low offset voltages), while bipolar outputs are more robust (specifically, less vulnerable to high-voltage static electricity in the environment).

Op-amps are available which offer high precision, low noise, low power consumption, high bandwidth, high output current, and low input currents in various combinations. When designing analogue signal conditioning for specific applications, an op-amp with the optimum combination of features should be selected.

Analogue Output

Analogue output from microcontrollers is less commonly required than input, because many output loads can be driven by a digital signal. Relays and solenoids only need a slow switching current driver, while heaters and motors can be controlled using PWM, because the switched output current is effectively averaged by the inductive load.

The Digital to Analogue Converter (DAC) is, however, commonly found in digital signal processors, where an analogue signal is converted to digital form for processing and storage, and then back to analogue, as in a digital audio system.

DAC Types

A range of techniques are available to convert a binary output to a corresponding voltage. The typical DAC uses a ladder network of precision resistors to produce a bit-weighted output voltage. A summing amplifier can also be used, with the input resistor values in a power-of-two series: 1k, 2k, 4k, 8k, 16k, 32k, 64k and 128k for example. In all cases, an output sum voltage is produced as follows: the bit connected to the most significant bit input, if set, provides half the output voltage, the second bit a quarter, the third bit an eighth and so on.

In the general DAC shown in Figure 7.14, the output step size and maximum level are set by a reference input, as in the ADC. A reference voltage of 2.56 V, for example, would give a bit step of 0.01 V in an 8-bit DAC, since there are 256 (2^8) output levels. That is, the least significant bit will produce a change of 10 mV; this is the resolution of the converter. Some converters, such the standard DAC0808 shown in Figure 7.15, use a current reference input, and a current output, which can be converted to voltage by precision resistors. These resistors need to be at least as accurate as the DAC itself. For an 8-bit DAC, the resolution is 1 part in 256, or slightly better than 0.5% at full scale, or 1% at mid-range. A 10-bit DAC has a resolution of 1/1024, about 0.1%, at full scale and 1/512, and about 0.2%, at the mid-value. The resolution increases with the output level, since the step size is fixed.

In the schematic of the test circuit (Figure 7.15), two DACS are demonstrated, a standard 8-bit DAC0808 and a more recent device, the 12-bit MCP4921 from Microchip, which uses the SPI serial interface.

Parallel DAC

The parallel converter (PDAC) has an 8-bit digital input. The reference level must be provided as a calibrated current, derived from the supply (+5 V) via a pre-set pot, which allows the maximum output to be adjusted to 2.55 V. For greater accuracy, a stable reference voltage should be used in the current source. The PDAC is then set to operate at 10 mV/bit. It also has a current output, so that a current loop output can be easily implemented for onward signal

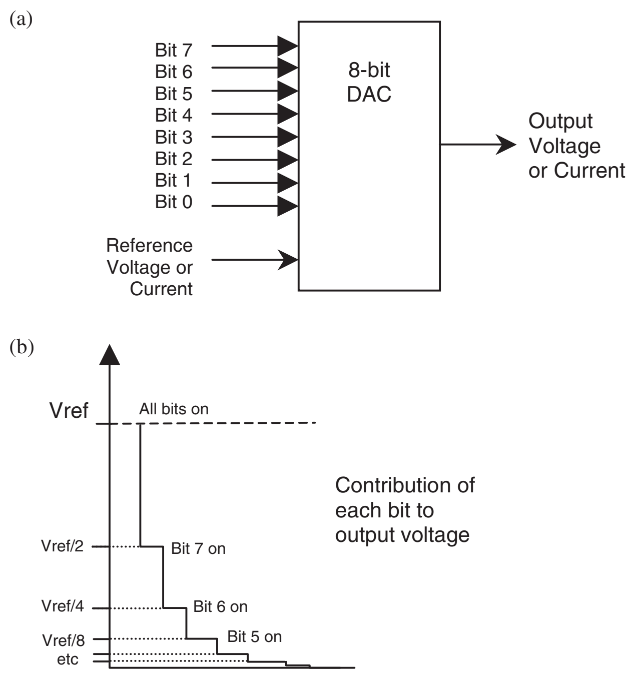


Figure 7.14 Basic digital to analogue converter: (a) general DAC hardware; (b) output voltage steps

transmission. In this case, a general purpose JFET (high impedance) input TL074 converts the output current into a voltage of 0–2.55 V. A precision resistor must be used in the feedback path if necessary. A –5 V supply allows operation down to 0 V.

The output in the test circuit can be controlled manually from the UP/DOWN push buttons, and monitored on the voltmeter. When the run button is pressed, the PDAC is driven with an incrementing output at maximum possible frequency, as determined by the MCU clock rate. Each output step takes three instruction cycles (INCF + GOTO). A sawtooth waveform is produced (Figure 7.16); if this is viewed on the oscilloscope, significant overshoot (ringing) can be seen on each step, and a large overshoot occurs on the falling edge. This overshoot could cause problems in subsequent stages of the system, so suitable filtering should always be considered on a digitally generated waveform. Here, the amplifier is damped with the 100 pF across the feedback-resistance. On the other hand, too much damping causes the waveform to lose its sharpness.

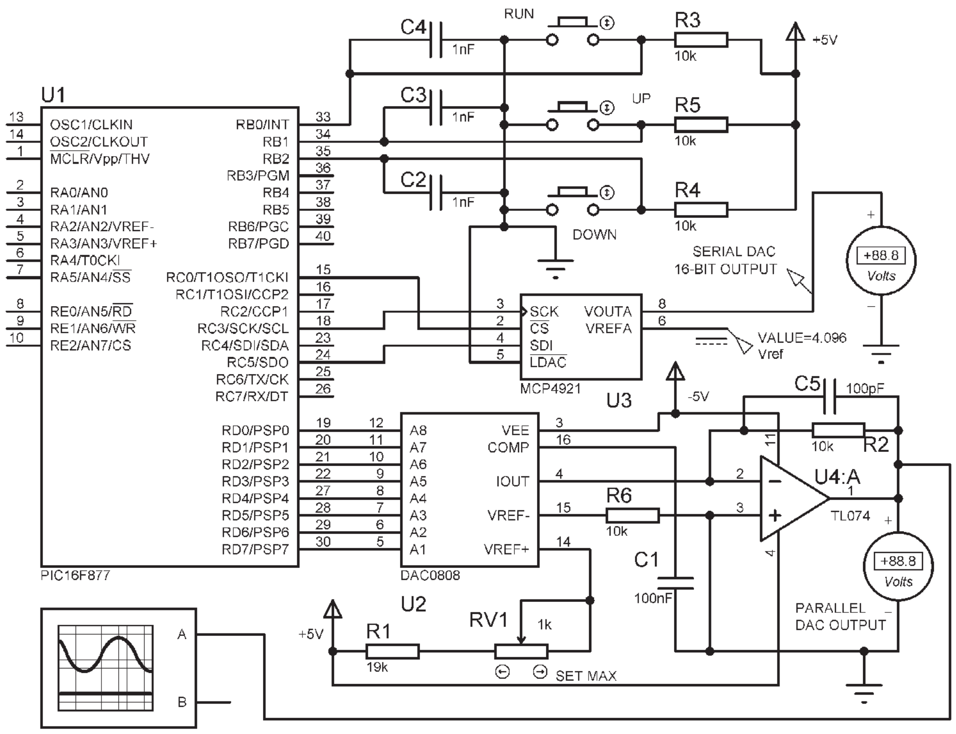


Figure 7.15 DACS schematic

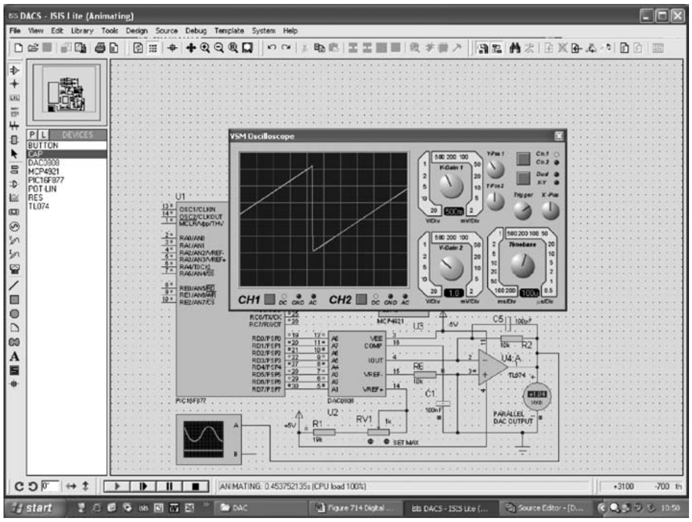

DACS - ISIS Lite (Animating)
File View Edit Library Tools Design Source Debug Template System Help
DEVICES
BUTTON
CAP
DAC0808
MCP4921
PIC16F877
POT-LIN
RES
TL074
VSM Oscilloscope
CH1
CH2
ANIMATING: 0.453752135s (CPU load 100%)
start

Figure 7.16 Screenshot of PDAC waveform

```
;************************************************************
;         DACS.ASM                                MPB 11-2-06
;
;         Test program for parallel and serial D/A Converters
;         DAC0808 & MCP4921. Proteus simulation DACS.DSN
;
;************************************************************

          PROCESSOR 16F877
          INCLUDE "P16F877.INC"
          __CONFIG 0X3731

Hibyte    EQU       020       ; SPI data high byte
Lobyte    EQU       021       ; SPI data low byte

          ORG 0               ; Load at default range
          NOP                 ; for ICD operations

; Initialise parallel and serial ports ----------------------

          BANKSEL   TRISD
          CLRF      TRISD               ; Parallel port
          BCF       TRISC,5             ; Serial data
          BCF       TRISC,3             ; Serial clock
          BCF       TRISC,0             ; Chip select
          CLRF      SSPSTAT             ; default SPI mode

          BANKSEL   PORTD
          CLRF      PORTD               ; zero PDAC
          CLRF      SSPCON              ; default SPI mode

          MOVLW     B'00111001'         ; Initial SDAC data
          MOVWF     Hibyte              ; and store
          MOVLW     B'11111111'
          MOVWF     Lobyte

; Check buttons --------------------------------------------

up        BTFSC     PORTB,1             ; Test UP button
          GOTO      down                ; and jump if off
          INCF      PORTD               ; Increment PDAC
          INCF      Hibyte              ; Increment SDAC
waitup    BTFSS     PORTB,1             ; Wait for..
          GOTO      waitup              ; button release

down      BTFSC     PORTB,2             ; Test DOWN button
          GOTO      spi                 ; and jump if off
          DECF      PORTD               ; Decrement PDAC
          DECF      Hibyte              ; Decrement SDAC
waitdo    BTFSS     PORTB,2             ; Wait for..
          GOTO      waitdo              ; button release

; Send 16-bit data to SDAC via SPI port ---------------------

spi       BSF       SSPCON,SSPEN        ; Enable SPI port

          BCF       PORTC,0             ; Enable SDAC chip
          MOVF      Hibyte,W            ; Get high data
          MOVWF     SSPBUF              ; and send it
waithi    BTFSS     PIR1,SSPIF          ; Wait for..
          GOTO      waithi              ; SPI interrupt
          BCF       PIR1,SSPIF          ; Reset interrupt

          MOVF      Lobyte,W            ; Get low data
          MOVWF     SSPBUF              ; and send it
waitlo    BTFSS     PIR1,SSPIF          ; Wait for..
          GOTO      waitlo              ; SPI interrupt
          BCF       PIR1,SSPIF          ; Reset interrupt

          BSF       PORTC,0             ; Disable SDAC chip

; Run output loop until reset -------------------------------

          BTFSC     PORTB,0             ; Test run button
          GOTO      up                  ; and repeat loop

run       INCF      PORTD               ; Increment PDAC
          GOTO      run                 ; repeat until reset

          END ;--------------------------------------------
```

Program 7.3 DACS test program source code

Other standard waveforms can be generated in a similar way. A square wave simply requires the output to be switched between maximum and minimum output values, with a controlled delay. A triangular wave is similar to the sawtooth, except that the falling edge is decremented rather than rolling over to zero. A sine wave can be generated from a program data table, which holds pre-calculated instantaneous voltage values. In fact, any arbitrary waveform can be generated in digital mode.

The test program is listed in Program 7.3. The software and initialisation required to drive the PDAC is relatively simple.

Serial DAC

Many transducers are now provided with signal input and output using a standard serial data transfer protocol, such as SPI and I^2C. The serial converter (SDAC) used here uses the SPI interface, which allows 12-bit output to be transferred on two lines (clock and data). The SPI interface is described in detail later in Chapter 9. It is easy to use, since the transfer is triggered by simply writing the data to the serial port buffer register (SSPBUF). The serial port interrupt flag is polled until it indicates that the data has been sent.

The SDAC needs 2 bytes for each data transmission. The most significant 4 bits of the first byte are used for control functions (0011). The low nibble contains the high 4 bits of the data, and the second byte the remaining 8. These are simply written one after the other to the output buffer. The chip select must then be taken high to trigger the transfer of the data to the output of the SDAC. More details are given in the device data sheet of the MCP4921.

The output voltage range is set in the usual way by a voltage reference input. The output has 2^{12} steps (4096), so a reference voltage of 4.096 V gives a conversion factor of 1 mV/bit. The resolution is 16 times better than the 8-bit PDAC. The SDAC output can also reach 0 V without a negative supply. The serial interface is inherently slower than the parallel, but fewer MCU I/O pins are needed.

SUMMARY 7

- The PIC analogue to digital converter input provides 8-bit or 10-bit conversion
- One of eight analogue inputs in the 16F877 connects to the A/D module

- The conversion result is found in FSRs ADRESH and ADRESL
- A suitable reference voltage is used to set the maximum input voltage
- Linear amplifier stages may be needed to provide input signal conditioning
- Analogue sensors need interfacing for correcting gain and offset
- A comparator converts voltage levels to a switched input
- A digital to analogue converter converts parallel or serial data into a voltage

ASSESSMENT 7 **Total** *(40)*

1 Calculate the percentage accuracy per bit of a 12-bit ADC at full scale. *(3)*

2 Explain why a 2.56 V reference voltage is convenient for an 8-bit ADC input. *(3)*

3 State the function of the CHSx bits in ADCON0. *(3)*

4 Calculate the maximum sampling frequency for a 10-bit input if 2 µs per bit is allowed and no settling time is needed. *(3)*

5 Explain the difference between left and right justified ADC results. *(3)*

6 State the gain, input resistance and output resistance of an ideal amplifier. *(3)*

7 State the device number of a single supply op-amp, and one advantage and one disadvantage of using a single supply amplifier. *(3)*

8 Calculate the gain of a simple non-inverting IC amplifier, if the input resistor is 1k0 and the feedback resistor 19k. *(3)*

9 Calculate the output voltages of (a) a summing amplifier and (b) a difference amplifier if the input voltages are 1.0 V and 0.5 V, and the gain of both is 2 (assume positive output voltages only). *(3)*

10 Describe the general effect of a capacitor across the feedback resistor in an IC amplifier stage. *(3)*

11 Design an IC amplifier stage to give an output which changes from 0 V to $+2.0$ V when the input changes from $+1.5$ V to $+1.00$ V, assuming the feedback resistor is 10k. *(5)*

12 The trigger comparator in Figure 7.10 (b) is fed with a triangular wave. Sketch the input and output on the same time axis, and show how output changes over one cycle of the input. Add some (<1 V) noise to the input and show the effect on the output. Label the drawing to indicate the benefits of the circuit. *(5)*

ASSIGNMENTS 7

7.1 Analogue Input

Modify the 8-bit conversion program so that the input measures from 0.00 to 0.64 V, by right justifying the result and processing ADRESL. When the input is 0.5 V, the display should show 0.500 V. What is the resolution of the voltage measurement. What is displayed when the input voltage is above 0.64 V, and why? Provide a program outline in suitable form.

7.2 Amplifier Test

Run the simulation of the basic amplifier interfaces. By suitable adjustment of the input voltages, record a set of values for each amplifier input and output, and demonstrate that the expressions given for the gain of each type is valid. Evaluate the accuracy of the outputs obtained in simulation mode, as a percentage. Construct the equivalent physical circuits, compare the performance with the simulated and ideal performance, and account for any discrepancies.

7.3 Summing DAC

Construct an IC summing amplifier in the circuit simulator with eight input resistors with the values 1k, 2k, 4k, 8k, 16k, 32k, 64k and 128k, and feedback resistor of 1k, using the LM324 (select a part with a simulation model attached). Connect each input to +5 V via a toggle switch, and the reference (+) input to 2.5 V derived from a voltage divider across the supply. Run the simulation and close the switches in reverse order (128k first). Record the output voltages obtained, and demonstrate that the circuit acts as a DAC. Compare its performance, ease of use and other relevant factors with the DACs as shown in Figure 7.15.

This page intentionally left blank

Part 3

Systems

This page intentionally left blank

8

Power Outputs

In this chapter, we will concentrate on power outputs. The microcontroller or microprocessor port only provides a limited amount of current: about 20 mA in the case of PIC, and even less for standard microprocessor ports. Therefore, if we want to drive an output device that needs more current than this, some kind of current amplifier or switch is needed.

Current Drivers

All solid-state current drivers and switches are derived from the semiconductor technology which is the basis of the transistor. The bipolar transistor was the first to be developed, and is still extensively used as it is robust and easy to design around. The FET (field effect transistor) was later developed alongside integrated circuits, because it generally has a higher input impedance and consumes less power in high-density circuits. In addition, the power FET has some distinct advantages over its bipolar equivalent, and is used extensively in motor control and similar applications.

Switched Bipolar Transistor Interfaces

One advantage of the bipolar junction transistor (BJT) is that there are only two basic types, NPN and PNP, so it is easier to design with. It is a current amplifier, that is, a small base current controls a larger (typically $\times$ 100) current in the collector. The emitter is the common terminal as far as current flow is concerned (Figure 8.1 (a)). In the equivalent circuit (Figure 8.1 (b)), the base behaves as a diode junction, with a forward volt drop about 0.6 V in normal operation. The base current controls a current source, which represents the collector–emitter junction. The NPN has conventional current flow out of the emitter, and the PNP into the emitter. This is indicated by the arrowhead in the transistor symbol. The NPN is illustrated in Figure 8.1.

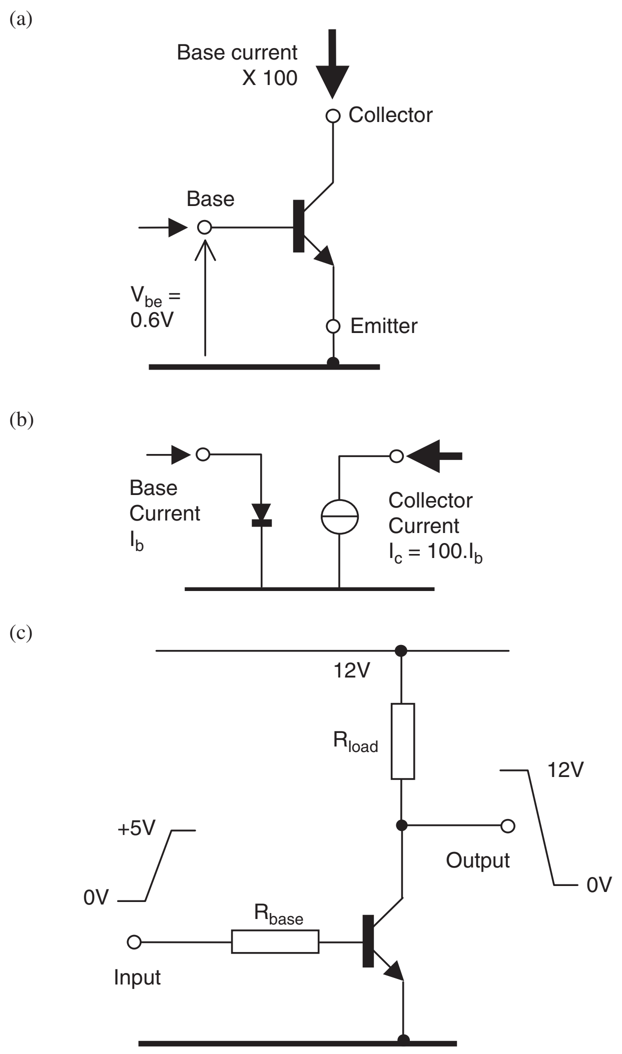


Figure 8.1 Bipolar transistor: (a) terminals; (b) equivalent circuit; (c) simple interface circuit

In the most common configuration, a signal is connected to the base of the transistor via a current limiting resistor (Figure 8.1 (c)). This then controls the larger current flow in a load connected to the collector. This is referred to as common emitter operation. We are assuming that the input to the interface is coming from the MCU output port. +5 V applied to the base resistance causes the transistor to switch on, drawing current through the load resistor, and causing the voltage at the collector to go low. The supply voltage to the transistor can be some higher value (12 V in this case), which allows more power to be dissipated in the load for a given collector current.

The circuit can be biased with a voltage divider on the base to operate as a linear amplifier, but this option is explained in detail in most introductory electronics texts, and will not be considered further here. We will focus on the switching mode of operation where the output voltage swings over its full range, and the transistor is saturated when on. In this case, the output voltage can be close to zero. Almost the full supply voltage is applied across the load, and a current flows, which depends on the load resistance value. A simple resistor load will act as a small heater, dissipating power $P = V^2/R$. Alternatively, a filament lamp will convert some of this power into light, or a motor into torque.

When the transistor is off, the output is pulled up to supply via the load resistance, and the load no longer dissipates power, as the voltage across it and the current through it are both low. The transistor dissipation is given as $P_T = V_c I_c$, where V_c and I_c are the collector voltage and current. When the transistor is off, I_c is small, and when the transistor is on, V_c is small, so that in both cases the transistor dissipates only a small amount of power. Therefore, minimal power is wasted in the transistor, and it may not need a heat sink, unless operating at a high switching frequency.

The PNP transistor operates in the inverse mode, with all current flows reversed. This may be useful with negative supplies or to provide a grounded load.

In Figure 8.2, some bipolar switching circuits are illustrated; in each case the transistor is on, and the same base and load resistors are used for comparison. In circuit (a), the basic common emitter switch is shown, using a +12 V load supply. The current gain is 118/4.3 = 27, relatively low because the TIP31 is a power transistor, which generally has a lower current gain (specified as h_{FE}). The power dissipated in the load is about 118 mA × 11.8 V = 1.4 W. The switch simulates the MCU output operating at TTL levels. The load, however, has to be connected to the positive output supply, which may not be convenient.

In circuit (b), the load has one terminal connected to ground, so when it is off, both load terminals are at 0 V, which is generally preferred. The disadvantage here is that it is not so easy to interface to a higher voltage output supply, so +5 V is used to supply the transistor. The current gain is higher (40), but

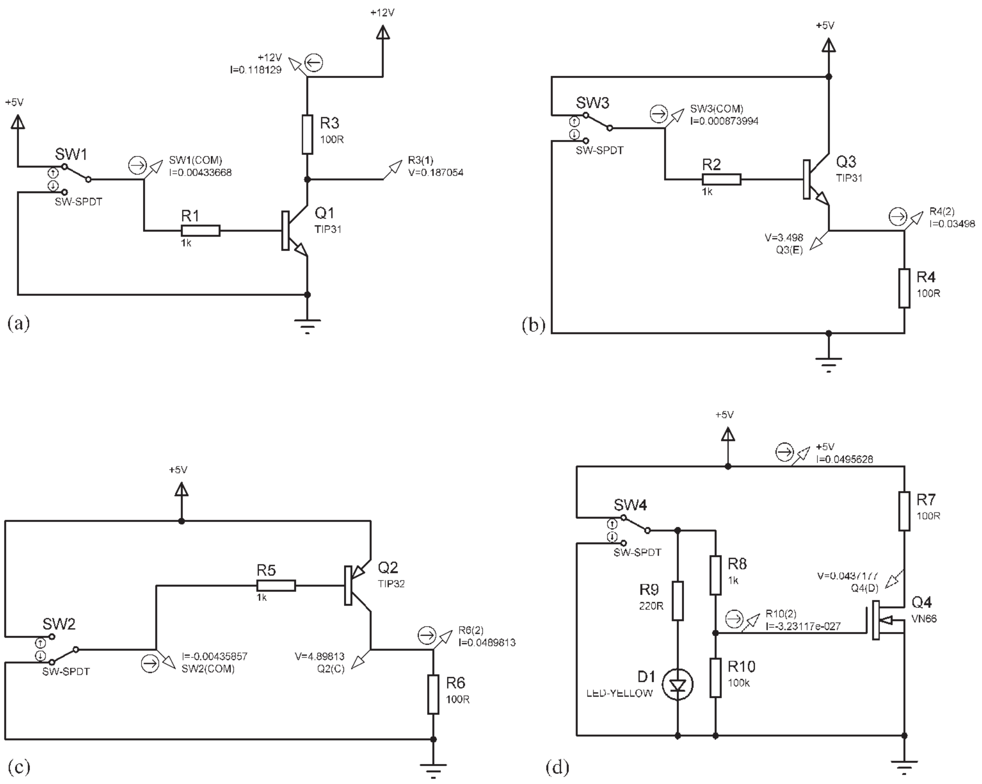


Figure 8.2 PNP and common collector: (a) common emitter switch; (b) common collector grounded load; (c) PNP CE switch grounded load; (d) FET interface

the power output is limited (122 mW). By changing to a PNP transistor in circuit (c), the common emitter configuration can be used with a grounded load. The power output is increased to 240 mW with the same supply voltage.

FET Interfaces

One advantage the FET has over the bipolar transistor is its high input impedance. There is a considerable variety of types, depending on the construction and characteristics, so we will concentrate here on one device that has the most convenient operating parameters: the VN66. This can handle up to about 1 A, and operates with its input switching between 0 V and 5 V, so it can be connected directly to digital outputs.

As can be seen in Figure 8.2 (d), the input current is extremely small, around 10^{-27} A, because it is an insulated gate (IGFET) device. The channel (load) current is controlled by the voltage at the gate (input), and negligible input current is drawn, giving almost infinite current gain. The power dissipation in the load is limited to 245 mW, because there is a significant forward resistance associated with the FET channel.

The FET therefore provides very high current gain, negligible loading on the MCU output and convenient interfacing if the appropriate device is selected. To avoid noise effects at the high impedance input, loading to ground is needed: the LED performs this function and indicates the output condition if enough current is available from the MCU output to operate it.

Relays & Motors

In control systems, the load on the output circuit of the MCU is often an electromagnetic device. This includes any actuator which uses a coil to convert electrical energy into motion, such as a solenoid, relay, loudspeaker or motor. When current is passed through a coil, the resulting magnetic field interacts with another magnet (permanent or electrically powered) or simple soft iron core. A solenoid is simply a coil containing a steel pin or yoke, which is attracted to the electromagnetic coil by the induction of an opposing magnetic pole. This motion can be used to operate a valve, a set of electrical contacts (relay) or any other mechanical device. In a motor, a set of coils interacts with permanent magnets or electromagnets to create a turning force and rotation.

Relay

Figure 8.3 shows the most common electromagnetic devices. The relay (a) consists of a coil that attracts a pivoted mild steel yoke, which in turn operates a set of changeover contacts. These are used to control an output circuit, which will usually control a high power load circuit. The relay provides complete electrical isolation, and a very high off resistance (air gap). Unfortunately, it is generally unreliable due to wear and sparking at the contacts, and is slow, because of inertia in the switch mechanism.

DC Motor

The simple DC motor (b) has a rectangular conductor, representing the armature windings, set in a magnetic field. The field may be provided by permanent magnets in small motors, or field windings in larger ones. A current is passed through the conductor, which causes a cylindrical magnetic field around it. This interacts with the planar magnetic field provided by the field magnets or windings, causing a tangential force on the rotor, which provides the motor

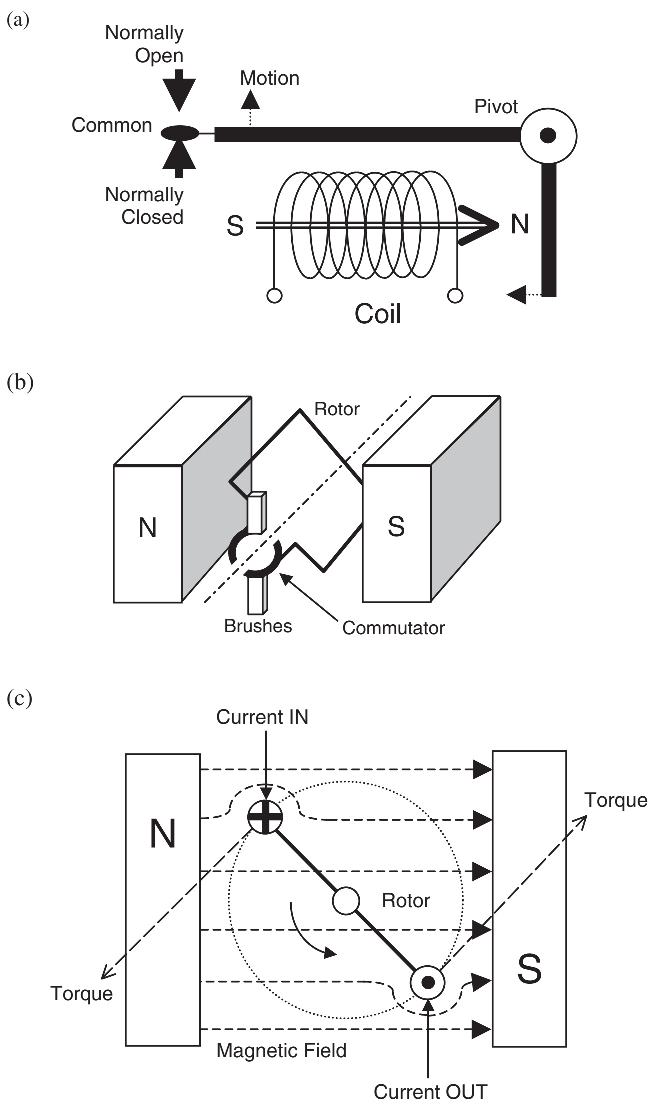


Figure 8.3 Electromagnetic devices: (a) relay operation; (b) basic motor configuration; (c) motor operating principle

torque. To allow for rotation, the current is supplied to the armature via slip rings and brushes. In order to maintain the torque in the same direction, the current has to be reversed every half revolution, so the slip ring is split to form a commutator.

A cross section is shown in (c). The current in and out of the page is represented by the cross and the dot on the respective rotor conductors. The circular rotor field caused by the rotor current is not shown (for clarity), but it is generated according to the right-hand screw rule. The distortion of the magnetic field due to the interaction with the rotor field can be seen. If you imagine the field as elastic bands, the force is generated as though the bands are trying to straighten.

Real Motors

The single turn armature described above needs further development to provide reasonable efficiency, output power and torque, principally by adding turns to the armature and rotor to increase the field strength. Small DC motors typically have a small number of armature windings with a permanent field magnet. It is a useful exercise to take apart a small, cheap, modelling DC motor and study its construction. It will typically have three armature windings and a six-segment commutator. The asymmetric windings provide more consistent torque as the rotor moves through a complete revolution.

The brushes and commutator are a weak point in the traditional motor design; mechanical wear and sparking which occurs as the current switches means that the DC motor is relatively unreliable, with limited operating life. Brushless DC motors improve on this by using a permanent magnet rotor, which eliminates the need to supply current to the armature, but these are limited in size and power. Similarly, stepper motors use a rotating magnetic field to move a passive rotor. They can be moved one step at a time, and can therefore be positioned accurately without feedback, but are complex to drive, inefficient and limited in power.

Larger motors tend to be three-phase AC motors. These use a rotating magnetic field generated by the three phases of the supply grid, giving high efficiency and output power in a compact unit, and an accurate, constant speed, usually 3000 rpm from a 50 Hz supply (1 revolution per cycle).

Power Output Interfacing

Figure 8.4 shows a selection of power output interfaces. The PIC has a simple program attached, which simply switches on each output in turn when the button is pressed.

Relay Interface

A relay can be used for either DC or AC loads, as the switch contacts will happily conduct in both directions. The relay coil is powered by a bipolar transistor,

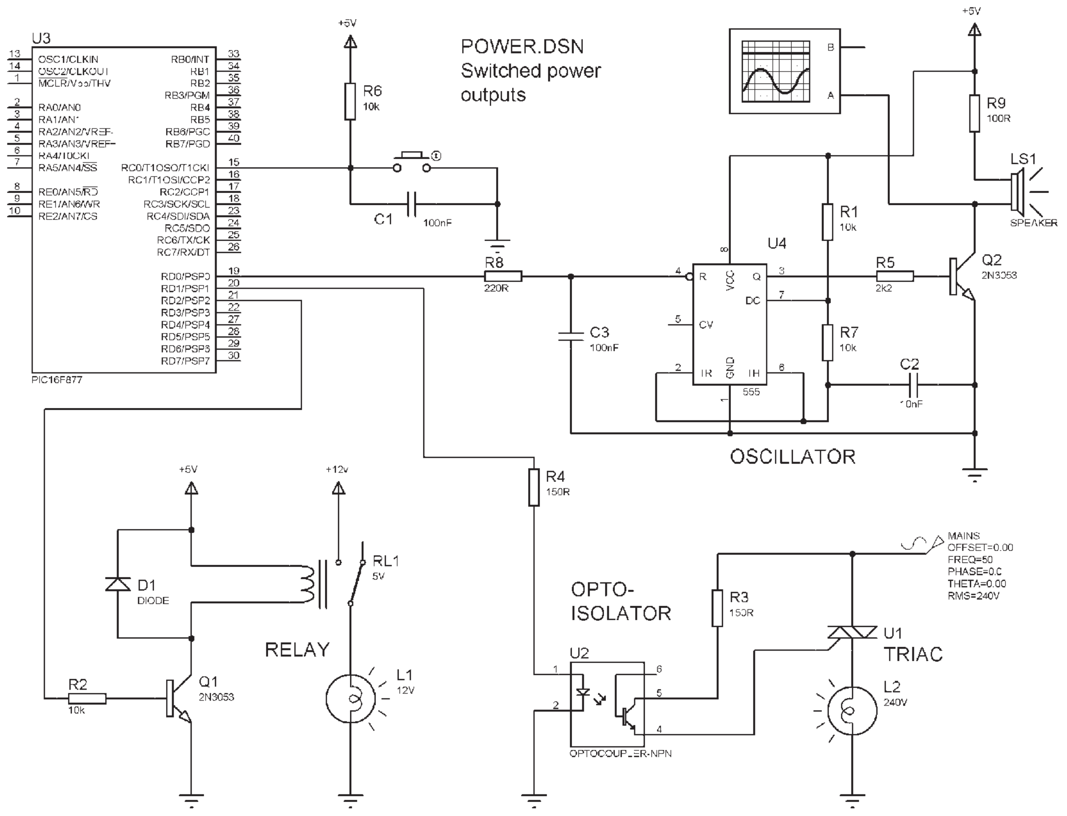


Figure 8.4 Power outputs interfaces

since the PIC cannot provide enough current. When the coil is activated, the contacts change over, completing the load circuit, which operates a lamp. Other high power loads such as heaters and motors can also be interfaced in this way, as long as simple on–off, but infrequent, switching is needed. The transistor is selected for sufficient collector current handling, and the base resistor to give plenty of base current, which is calculated from the required coil current:

$$\text{Coil current} = 40\ \text{mA} = \text{collector current}$$

$$\therefore\ \text{Base current} = 40\ \text{mA}/100 = 400\ \mu\text{A}$$

$$\therefore\ \text{Base resistor} < (5-0.6) \times 400 \times 10^{-6} = 17.6\ \text{k}\Omega \rightarrow 10\ \text{k}\Omega$$

The relay has a diode connected across the coil; this is a sensible precaution for all DC inductive loads (anything with a coil such as a motor or solenoid). When the coil is switched off, a large reverse voltage may be generated as the magnetic field collapses (this is the way the spark is generated in a car ignition). The diode protects the transistor from the back EMF by forward conduction. In normal operation, the diode is reverse biased and has no effect.

The relay is selected for the load current and voltage requirements, and the interface designed to provide the necessary coil operating current. However, it is slow, consumes a fairly large amount of power ($40 \text{ mA} \times 5 \text{ V} = 200 \text{ mW}$), and is relatively unreliable.

The relay provides low on resistance and high off resistance. However, it wastes a relatively large amount of power in the coil, is slow and unreliable due to wear on the contacts. An alternative is the solid-state relay, which is typically designed to switch AC loads from digital outputs with a solid-state device. It contains TTL buffering, isolation and triac (see below) drive in one package, with high reliability and switching speed.

Triac Interface

A relay can be used to control a DC or an AC load, as it operates as a mechanical switch. However, it has significant disadvantages, as outlined above. A solid-state switch, such as a transistor, is inherently more reliable, since it has no moving parts; but the transistor can only handle current in one direction, so is unsuitable for AC loads. The thyristor is an alternative type of solid-state switch; it has a latching mode of operation such that when switched on, it stays on, until the current falls to zero. It can therefore be pulse operated, and used to rectify AC current. By switching on at different points in the AC cycle, the average current can be controlled, allowing the power to the load to be varied. However, it only passes current in one direction, providing DC power only.

The triac is basically two thyristors connected back to back, with a common gate (trigger) input, allowing current flow in both directions. The full AC wave can then be utilised, with switching at the same point in the positive and negative half cycles of the current. A microcontroller can be used to carry out this function; the AC signal is monitored through its cycle, and the thyristor switched on at the required point in the cycle using a timer.

In Figure 8.4, a simple MCU triac interface is shown. An opto-coupler is used to isolate the control system from the high voltage load circuit. This contains an LED and phototransistor, which conducts when the light from the LED falls on its base. There is therefore no electrical connection between the two devices, and it will isolate output circuits operating at high voltage. When the MCU output is high, the opto-switch is on, and the voltage at terminal 1 of the triac is applied to the gate, turning the triac on when the voltage passes through zero. When the switch is off, the triac does not come on.

This example is manually controlled, but the output power could be controlled by monitoring the AC voltage via a feedback voltage divider and sampling it at an analogue input. An MCU timer would then be employed to control the delay between the zero crossing point in the cycle and the trigger point, where the triac is switched on each half cycle. A block diagram for this system is shown in Figure 8.5 (c).

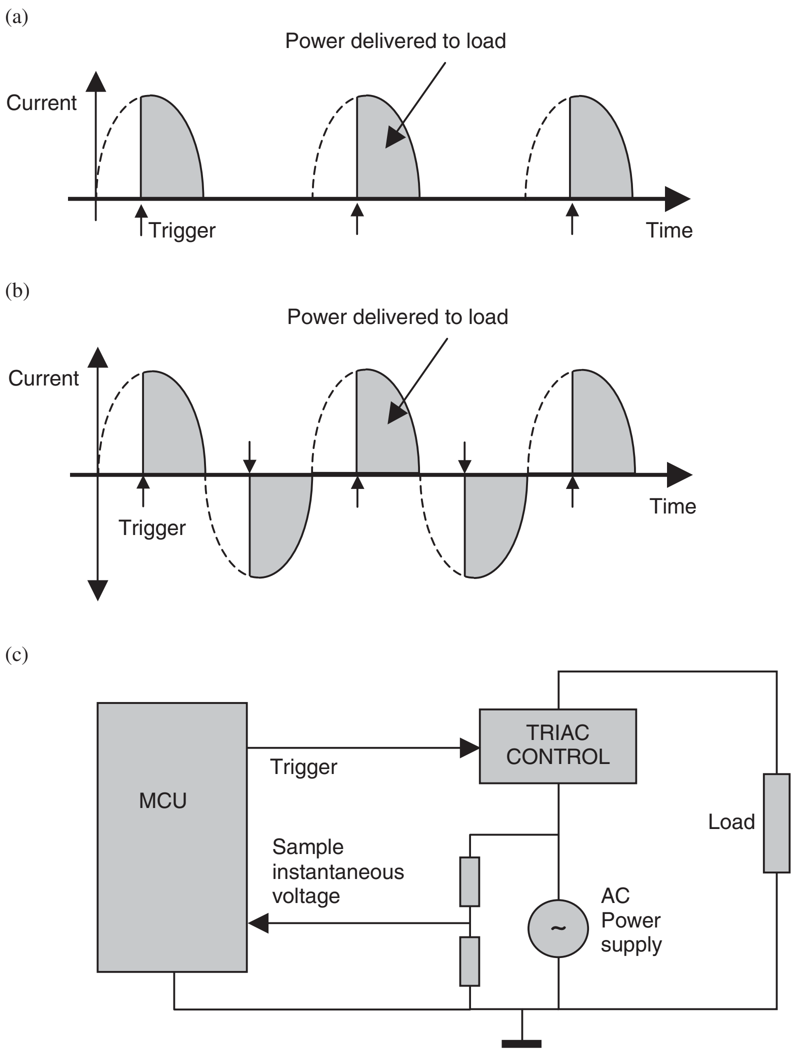


Figure 8.5 Thyristor and triac control: (a) thyristor; (b) triac; (c) MCU control

Oscillator Interface

If an output is required at a set frequency, it can be generated in a variety of ways. A software loop can set the output, delay, clear the output, delay and repeat. However, this will prevent the processor from carrying out other useful tasks in the meantime. Using a hardware timer and interrupts is one option; but if these MCU resources are required for other tasks, the oscillator function can be delegated to external hardware, so that the MCU simply switches an output to enable the oscillator.

A simple low-frequency oscillator can be implemented using a 555 Timer chip; the same chip can also be used for generating timed pulses and delays. In Figure 8.4, it drives a loudspeaker via a bipolar transistor. Input R on the chip enables the oscillator, and C2 controls the frequency.

This is an illustration of a very important design principle. A given interface can be implemented principally in hardware or software. The software implementation will use more MCU resources in terms of both the available peripheral interfaces, processor time, and programming effort. The hardware approach saves on these resources, but involves additional cost, both in hardware design effort and components for each system produced. Software, on the other hand, once written, has a negligible reproduction cost.

Motor Interfacing

As discussed above, the basic function of a motor is to convert electrical input current into output mechanical power (torque). All use electromagnetic coils to provide this conversion, and need current switches or amplifiers to operate them from an MCU.

A simple method of controlling AC motors is to use a relay as switch. Another is to use a triac to control the current, as outlined above, but in practice there are some tricky issues associated with controlling inductive loads with thyristors and triacs which require reference to specialist texts. Three-phase motors require, in simple terms, each phase to be controlled by a separate device, but simultaneously, that is, three relays or triacs operated by the same controller.

Three typical small motor interfaces are shown in Figure 8.6, a DC motor, a DC servo and a stepper motor. The motors can be operated in turn by pressing the select button. Operating parameters (speed, position, direction) can then be changed via the additional push buttons. The control program outline is given in Figure 8.7, and the source code in Program 8.1. The operation of each interface will be explained in turn.

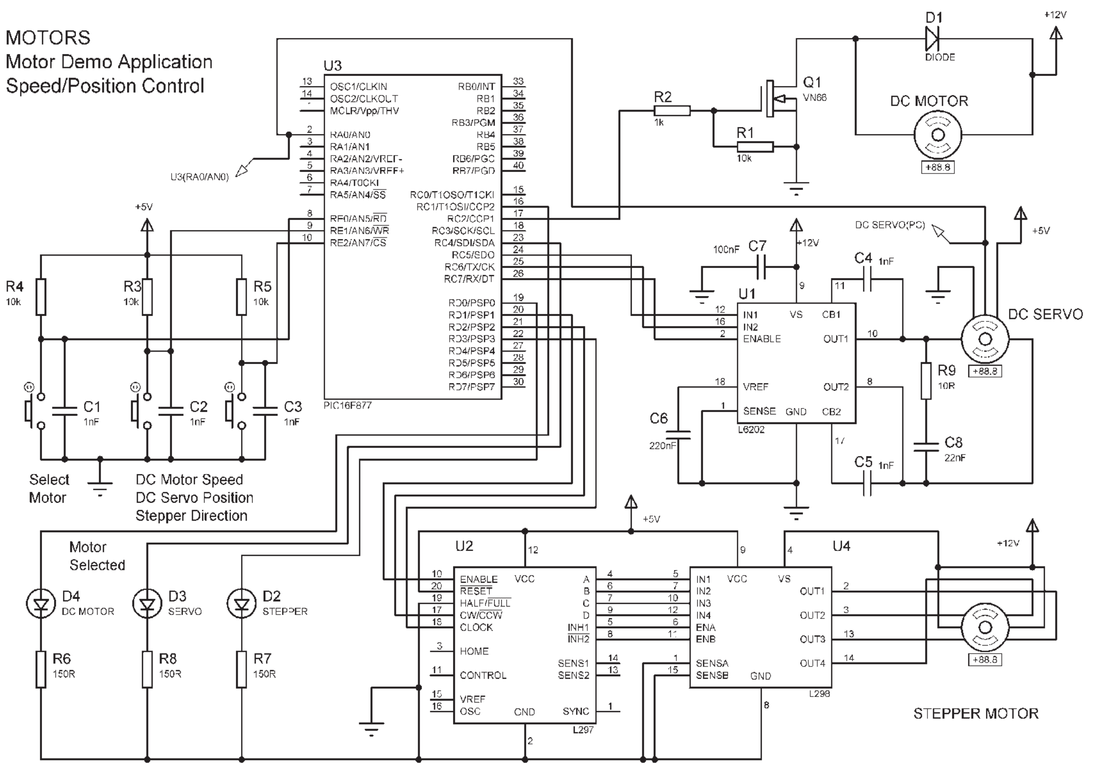


Figure 8.6 Motor interfaces schematic

PWM Speed Control

The DC motor is controlled from the PWM output of the PIC MCU (see Chapter 6), via a power FET VN66. This has an operating current of about 1 A maximum, giving a maximum motor input rating of 12 W at the operating voltage of 12 V. The motor characteristics can be set in the simulation, so a minimum motor resistance of about 10 Ω would be suitable, as the FET itself has a forward resistance of about 1 Ω.

The VN66 is a convenient device to use as it operates at TTL level gate voltages; that is, 0 V switches it off, +5 V switches it on (threshold about 1 V). It has a very high input impedance, so reliability is improved by adding shunt resistance to the gate, to improve the noise immunity. The diode across the motor is required to cut off the back EMF from the inductive load.

When the system is started and the DC motor selected, a default PWM output is generated with 50% mark/space ratio. The MSR (mark space ratio)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;        Project:                    Interfacing PICs
;        Source File Name:           MOTORS.ASM
;        Devised by:                 MPB
;        Date:                       19-8-05
;        Status:                     Working
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;        Demonstrates DC, SERVO & STEPPER MOTOR control
;        Select motor and direction using push button inputs
;        DC Motor PWM speed control - working
;        DC Servo position control - rollover not fixed
;        Stepper direction control - working
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

         PROCESSOR 16F877
;        Clock = XT 4MHz, standard fuse settings
         __CONFIG 0x3731

;        LABEL EQUATES      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

         INCLUDE "P16F877A.INC"
         ; standard register labels

;----------------------------------------------------------
; User register labels
;----------------------------------------------------------

Count1   EQU      20                 ; delay counter
Count2   EQU      21                 ; delay counter
Target   EQU      22                 ; servo target position

;----------------------------------------------------------
; PROGRAM BEGINS
;----------------------------------------------------------

         ORG      0                  ; Default start address
         NOP                         ; required for ICD mode

;----------------------------------------------------------
; Port & PWM setup

init     NOP
         BANKSEL  TRISB              ; Select control registers
         CLRF     TRISC              ; Output for dc motors
         CLRF     TRISD              ; Output for stepper
         MOVLW    B'00000010'        ; Analogue input setup code
                                     ; PortA = analogue inputs
                                     ; Vref = Vdd
         MOVWF    ADCON1             ; Port E = digital inputs
         MOVLW    D'249'             ; PWM = 4kHz
         MOVWF    PR2                ; TMR2 preload value

         BANKSEL PORTB               ; Select output registers
         CLRF     PORTC              ; Outputs off
         CLRF     PORTD              ; Outputs off
         MOVLW    B'01000001'        ; Analogue input setup code
         MOVWF    ADCON0             ; f/8, RA0, done, enable
         MOVLW    D'128'             ; intial servo position
         MOVWF    Target


;----------------------------------------------------------
; MAIN LOOP
;----------------------------------------------------------

but0     BTFSC    PORTE,0            ; wait for select button
         GOTO     but0

         MOVLW    B'00001100'        ; Select PWM mode
         MOVWF    CCP1CON            ;
         MOVLW    D'128'             ; PWM = 50%
         MOVWF    CCPR1L             ;

but1     BTFSS    PORTE,0            ; wait for button release
         GOTO     but1
         CALL     motor              ; check for speed change
         BTFSC    PORTE,0            ; wait for select button
         GOTO     but1
         MOVLW    B'00000000'        ; deselect PWM mode
         MOVWF    CCP1CON            ;
         CLRF     PORTC              ; switch off outputs

but2     BTFSS    PORTE,0            ; wait for button release
         GOTO     but2
         CALL     servo              ; move servo cw or ccw
         BTFSC    PORTE,0            ; wait for select button
         GOTO     but2
         CLRF     PORTC              ; switch off servo

but3     BTFSS    PORTE,0            ; wait for button release
         GOTO     but3
         CALL     step               ; output one step cycle
         BTFSC    PORTE,0            ; wait for select button
         GOTO     but3
         CLRF     PORTD              ; disable stepper outputs

         GOTO     but0               ; start again
```

Program 8.1 Motor control program

```
;------------------------------------------------------------
; SUBROUTINES
;------------------------------------------------------------
; Change dc motor speed by one step and wait 1ms
; to debounce and control rate of change...................

motor    BSF      PORTC,1           ; switch on motor LED

         BTFSS    PORTE,1           ; inc speed?
         INCF     CCPR1L            ; yes
         MOVLW    D'248'            ; max speed?
         SUBWF    CCPR1L,W
         BTFSS    STATUS,Z
         GOTO     lower             ; no
         DECF     CCPR1L            ; yes - dec speed

lower    BTFSS    PORTE,2           ; dec speed?
         DECFSZ   CCPR1L            ; yes - min speed?
         GOTO     done              ; no
         INCF     CCPR1L            ; yes - inc speed

done     CALL     onems             ; 1ms debounce
         RETURN

; Move servo 10 bits cw or ccw...............................

servo    BSF      PORTC,4           ; switch on servo LED
         BSF      PORTC,7           ; enable drive chip

         BTFSC    PORTE,1           ; move forward?
         GOTO     rev               ; no

wait1    BTFSS    PORTE,1           ; yes- wait for button..
         GOTO     wait1             ; ..release
         MOVLW    D'10'             ; add 10...
         ADDWF    Target            ; ..to servo target position
         BSF      PORTC,5           ; move..
         BCF      PORTC,6           ; .. forward

getfor   CALL     getADC            ; get position
         BSF      STATUS,C          ; set carry flag
         MOVF     Target,W          ; load position
         SUBWF    ADRESH            ; compare with target
         BTFSS    STATUS,C          ; far enough?
         GOTO     getfor            ; no - repeat

         BCF      PORTC,5           ; yes - stop
         MOVLW    D'250'            ; wait 250ms ..
         CALL     xms               ; .. before next step
rev      BTFSC    PORTE,2           ; move reverse?
         RETURN                     ; no
wait2    BTFSS    PORTE,2           ; yes- wait for button..
         GOTO     wait2             ; ..release
         MOVLW    D'10'             ; yes - sub 10 from...
         SUBWF    Target            ; .. servo target position
         BCF      PORTC,5           ; move ..
         BSF      PORTC,6           ; ..reverse
getrev   CALL     getADC            ; get position
         BSF      STATUS,C          ; set carry flag
         MOVF     Target,W          ; load position
         SUBWF    ADRESH            ; compare with target
         BTFSC    STATUS,C          ; far enough?
         GOTO     getrev            ; no - repeat

         BCF      PORTC,6           ; yes - stop
         MOVLW    D'250'            ; wait 250ms ..
         CALL     xms               ; .. before next step
         RETURN
; Output one cycle of stepper clock........................
step     BSF      PORTD,0           ; switch on stepper LED
         BSF      PORTD,1           ; enable stepper drive

         BTFSS    PORTE,1           ; test cw button
         BSF      PORTD,2           ; select clockwise
         BTFSS    PORTE,2           ; test ccw button
         BCF      PORTD,2           ; select counter-clockwise

         BSF      PORTD,3           ; clock high
         MOVLW    D'25'             ; load delay time
         CALL     xms
         BCF      PORTD,3           ; clock low
         MOVLW    D'25'             ; load delay time
         CALL     xms

         RETURN
; Stepper software delay ..................................

xms      MOVWF    Count2            ; receive x ms in W
down2    CALL     onems
         DECFSZ   Count2
         GOTO     down2
         RETURN

onems    MOVLW    D'249'            ; delay one millisec
         MOVWF    Count1
down1    NOP
         DECFSZ   Count1
         GOTO     down1
         RETURN
; Read ADC input and store ................................

getADC   BSF      ADCON0,GO         ; start ADC..
wait     BTFSC    ADCON0,GO         ; ..and wait for finish
         GOTO     wait

         MOVF     ADRESH,W          ; store result, high 8 bits
         RETURN
;-----------------------------------------------------------
         END                        ; of source code
```

Program 8.1 *Continued*

can then be increased and decreased using the up/down buttons. Note that the software has to check each time the MSR is modified for the maximum (FF) or minimum (00) value, to prevent rollover and rollunder of the PWM value.

DC Motor Position Control

DC motors cannot be positioned accurately without some kind of feedback; in applications such as printers and robot arms, the DC motors have feedback devices, which allow the controller to monitor the motor shaft position, speed or acceleration.

In digital control systems, this is usually achieved by using a slotted wheel and opto-sensor attached to the motor shaft. This may often be followed by a gearbox in the drive chain, for example, in robot arm where the output range of movement is less that 360°. The controller counts the pulses from the wheel to determine how far the output has moved; also, the pulse frequency can be converted to speed. In a printer, the linear position of the print head is monitored by a graduated strip attached to the traverse mechanism. The accuracy of the system can be further improved by interpolation; this means the reference strip has a sinusoidal pattern so that each cycle can be subdivided by a continuous variation in the sensor signal.

The system block diagram shown in Figure 8.7 represents a general purpose position or speed controller. The motor has a slotted or perforated wheel attached. Say there are 100 slots, then there will be 200 edges, giving a resolution of $360/200 = 1.8°$. The motor is driven via a current amplifier with a PWM signal; the speed can then be controlled, and ramped up and down to prevent the motor from overshooting the target position. The MCU may act as a slave device, receiving a position or speed command from a master controller, carrying it out, and then signalling completion of the operation.

An alternative speed control system could use a tachogenerator to measure the speed. This is a small DC generator that outputs a voltage or current in proportion to the speed of the shaft, operating in the inverse mode to a DC motor. The analogue tacho signal can then be used to control the speed. Analogue position control is even simpler, in principle. A pot is attached to the motor shaft, and provides a voltage, which represents the position. An all analogue position controller can be implemented with op-amps, which will position the output according to an analogue input signal from a pot, DAC or amplifier. The main problem is that the pot only has a range of about 300°, and may not allow continuous rotation.

A servo motor is one that incorporates a position feedback element. In Figure 8.6, the DC servo has a built-in pot, which provides a voltage representing the position, between +5 and 0 V. The motor is driven from an L6202 full bridge driver. This is an IC, which provides drive to the motor in either direction under digital control. A block diagram of the chip is shown in Figure 8.8.

```
MOTORS
      Test DC motor PWM speed, DC position step servo and
      stepper motor direction with push button inputs, using P16F877 (4MHz)

Main
     Initialise
               Port A = Analogue inputs, servo pot = RA0
               Port C = Outputs, DC motors
               Port D = Outputs, stepper motor
               Port E = Digital inputs, push buttons: Select, Up, Down
               PWM rate = 4kHz
               Servo target value = 128

      Wait for 'Select' button
     REPEAT
               Select PWM mode, 50% MSR

               REPEAT
                       CALL Motor
               UNTIL 'Select' button pressed again

               REPEAT
                       CALL Servo
               UNTIL 'Select' button pressed again

               REPEAT
                       CALL Step
               UNTIL 'Select' button pressed again

      ALWAYS

Subroutines
               Motor
                       IF 'Up' button pressed
                               Increment speed unless maximum
                       IF 'Down' button pressed
                               Decrement speed unless minimum
                       RETURN

               Servo
                       IF 'Up' button pressed
                               Add 10 to target position
                               Move forward, until target position reached
                       IF 'Down' button pressed
                               Subtract 10 from target position
                               Move reverse, until target position reached
                       RETURN

               Step
                       IF 'Up' button pressed
                               Select forward mode
                       IF 'Down' button pressed
                               Select reverse mode
                       Output one drive pulse
                       RETURN
```

Figure 8.7 Motor test program outline

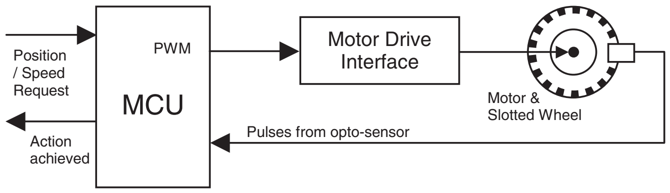


Figure 8.8 Digital position control system

The bridge circuit contains four power FETs connected such that when two are switched on together, current flows through the load. When the other pair is on, the current in the load is reversed. In a motor, the direction of rotation is reversed. The FETs are represented as simple switches. They are controlled from a simple logic circuit (see the L6202 data sheet), as summarised in the function table. Forward and reverse are selected by setting the IN1 and IN2 inputs to opposite logic states.

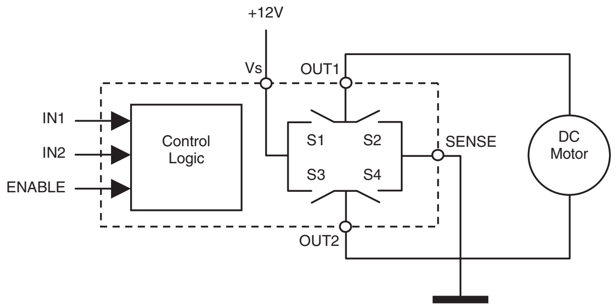
The chip operates from the motor supply voltage (+12 V) and the digital logic supply is derived from it, so no separate +5 V supply is needed. A current sensing resistor can be inserted in the 0 V connection, so that the current flow in either direction can be monitored for control purposes. Bootstrap capacitors must be fitted as shown in Figure 8.6 to ensure reliable switching of the bridge FETs. Although the FETs are protected internally with diodes, a series CR snubber network is connected across the output terminals to further protect the driver chip from current switching transients.

The test program allows the user to move the servo in steps. The required position is represented by an 8-bit number, which is initially set to the mid-value of 128. If the 'up' button is pressed, the value is increased by 10, and the servo started in the forward direction. The actual position is monitored from the servo pot voltage read in via AD0. When the input value matches the target value, the drive is stopped. The servo is moved in the reverse direction in the same way.

Stepper Motor Control

The third subcircuit in Figure 8.6 is the stepper motor interface. This also uses dedicated hardware, because a current driver chip would be needed in any case, and the stepper controller incorporates sequencing logic, which reduces the software burden. The stepper motor has a set of windings distributed around the stator, and a passive rotor, with fixed or induced magnetic poles. Incremental movement of the rotor is achieved by activating the windings in a suitable sequence.

(a)



(b)

EN	IN1	IN2	S1	S2	S3	S4	Motor
0	x	x	off	off	off	off	off
1	0	0	off	ON	off	ON	off
1	1	0	ON	off	off	ON	FORWARD
1	0	1	off	ON	ON	off	REVERSE
1	1	1	ON	off	ON	off	off

Figure 8.9 Full bridge driver: (a) block diagram; (b) function table

In Figure 8.9 (a), the stepper motor stator has 16 poles and the rotor 4 north poles which are attracted to the active pairs of rotor south poles. There are four sets of windings, A, B, C, D, connected sequence around the stator. These windings have their other ends connected to common terminals. This gives a total of six connections to the motor, with two pairs of centre-tapped windings. This allows the motor to be driven in different modes, while keeping the number of connections to a minimum. In the test circuit, the common terminals are connected to the power supply (+12 V), and the individual coil terminals driven from the sequencer (active low) (Figure 8.10).

In normal, full-step mode, the coil sets are activated in pairs (Figure 8.9 (b)) and the rotor moves half a pole per step, giving 24 steps per revolution. The step size is then 360/24 = 15°. This mode provides full torque but lower positional resolution. In half-step mode, the rotor moves by a quarter pole per step, 7.5°, providing twice as many steps per revolution, but less torque, since only one coil pair is activated at a time.

There are two chips forming the stepper drive interface. The L297 controller provides the stepping sequence on outputs A, B, C and D, and the L298 full bridge driver provides the drive current needed by the motor windings. The drive mode (full or half step) is fixed in full step by tying the step mode select input low. The active low reset is tied high. The MCU provides an enable signal, and selects the direction of rotation (clockwise (CW) or counter-clockwise

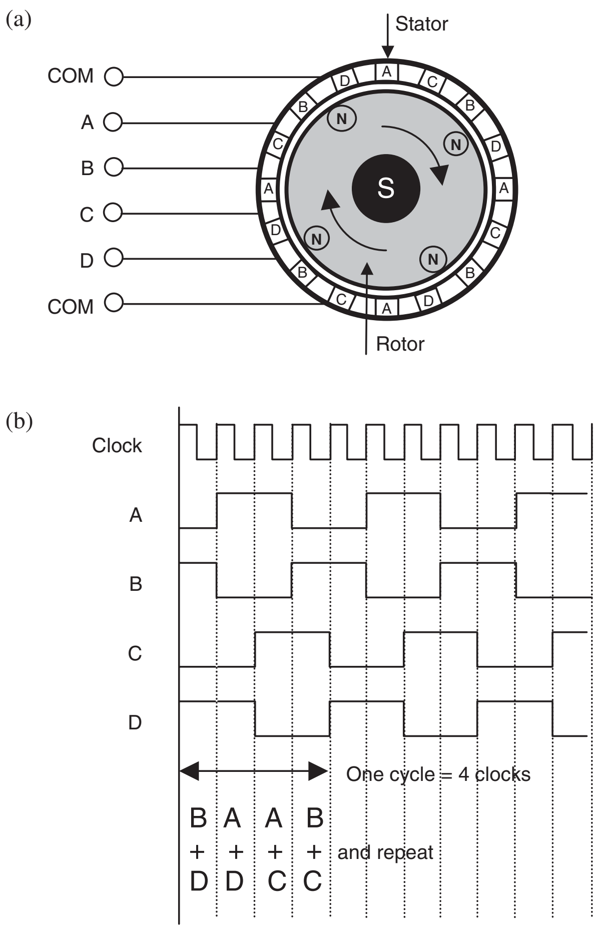


Figure 8.10 Stepper motor operation: (a) windings sequence; (b) drive sequence (normal mode)

(CCW)), and the test program outputs clock pulses at a frequency of 20 Hz, so that the stepping effect can be seen. When the stepper test is selected in the MCU program, the motor rotates CW by default, with the 'down' button changing the direction to CCW, and the 'up' button back to CW.

If the windings are left active in any position, that position can be held against a load torque. Even when powered down, stepper motors tend to have a residual torque, which holds the shaft in the last position selected, until an external torque is applied. Thus, the motor can be moved and held to a set angle, without feedback.

However, there is a lower limit to the step time required, which translates into a maximum operating frequency and speed. If starting from stationary, the speed may need to be increased gradually, until the rotor inertia gained will allow the motor to run at its maximum speed. The speed also needs to be ramped down when stopping, if correct position is to be maintained.

SUMMARY 8

- Power loads on MCU systems need a current amplifier or switch
- The bipolar transistor has current gain of 20–200 with low input resistance
- The FET provides a voltage controlled current with high input resistance
- The relay is an electromagnetic coil-operated switch
- The DC motor converts current into torque
- AC loads need a thyristor or triac to control the load current
- The speed of a DC motor can be controlled by PWM
- A DC servo provides analogue position feedback
- A shaft encoder provides digital speed and position feedback
- The stepper motor can be positioned without feedback

ASSESSMENT 8 **Total** *(40)*

1 State how an NPN and PNP resistor can be differentiated on a schematic, and a notional value for the base-emitter volt drop and the typical current gain. *(3)*

2 Describe the useful characteristics of the VN66 FET. *(3)*

3 State two advantages and one disadvantage of the relay as an interface device. *(3)*

4 Explain why the DC motor needs a commutator, and the problems this causes. *(3)*

5 Explain the main functional difference between a thyristor and triac. *(3)*

6 Explain how an oscillator can be implemented in hardware or software. *(3)*

7 Explain how PWM allows the dissipation in a power load to be controlled. *(3)*

8 Explain, using a suitable diagram, how a bridge driver allows a DC motor to drive in both directions. *(3)*

9 Sketch the waveforms used to drive a typical stepper motor, and their meaning. *(3)*

10 A stepper motor has a step size of 15°. Its maximum step rate is 100 Hz. Calculate the maximum speed in revs per second. *(3)*

11 Calculate the speed of a shaft in rpm if the MCU timer connected to a shaft encoder with 50 slots counts 200 in 100 ms. *(5)*

12 Compare the advantages and disadvantages of the DC and stepper motor for position control. *(5)*

ASSIGNMENTS 8

8.1 DC Motor Speed Control

Obtain two small DC permanent magnet motors, mount them in line and connect the shafts together using a suitable flexible coupling, as a motor and tachogenerator. Apply a variable voltage supply to one motor and note the output voltage from the other (generator). Interface the motor to an MCU with analogue input for unidirectional PWM drive. Write an application program to operate the system under closed loop control. Drive the motor at a default mid-range speed, while comparing the tacho output with a set value. Use push buttons to increment and decrement the speed, and a two-digit display showing the speed in revs per second.

8.2 Stepper Motor Characteristics

In the simulation software, select the standard stepper motor drive chip, and connect up the interactive stepper motor. Operate the stepper drive clock from a push button input and note the output sequence obtained. Draw the input clock and outputs accurately on a time axis and compare with Figure 8.9(b).

Obtain the data sheet for a stepper motor, and examine the torque/speed characteristic, and specifications for holding torque. Explain the significance of this characteristic in designing a robot arm with a stepper drive. Consider the maximum speed of operation and load handling for a single arm section, which is rotating in a vertical plane directly driven with a stepper motor at one end and a load at the other, starting from the horizontal position.

8.3 AC Power Control

Convert the block diagram for AC control (Figure 8.5(c)) into a simulation circuit. Write a control program which samples the instantaneous AC supply voltage and triggers the triac at around the peak voltage. Incorporate push buttons to increment and decrement the power delivered. Display the output current on the virtual oscilloscope.

9

Serial Communication

Serial communication only requires one signal connection, so the total number of hardware connections in a data link can be reduced to two or three, including ground. A synchronous link may have a clock signal alongside the data, for timing the transfer; an asynchronous one does not. This simplifies the wiring where there are numerous peripheral devices for the MCU to communicate with, or the connection is long distance. Within the microcontroller domain, we tend to use the simpler forms of serial communications; the three PIC 16F877 serial interfaces described here are the USART, SPI (Serial Peripheral Interface) and I^{2}C (Inter-Integrated circuit).

USART

The Universal Synchronous Asynchronous Receiver Transmitter (USART) provides the type of basic serial communication originally developed for dumb terminals to communicate with mainframe computers; it was later adopted for the COM port of the PC to interface to the mouse and other serial peripherals. When converted into a higher transmission voltage, for distance transmission, it is known as RS232. It can therefore, with additional interfacing, be used by the PIC to communicate with a PC. It is therefore helpful for us that the PC retains a standard COM port (9-pin D-type connector), even though, in general use, it has been superseded by USB. Compared with the USART, USB is fast, but complex, being more akin to a networking protocol; for this reason, it is not yet commonly provided in microcontrollers. RS232 is also the protocol used to connect to the standard PIC programmer to a host PC.

PIC USART

In the PIC 16F877, the USART is accessed through pins RB6 and RB7. It has two modes of operation, synchronous (using a separate clock signal) and asynchronous (no clock connection). The asynchronous mode is probably used more often, as other methods of synchronous transmission are available in the PIC, as we will see. In asynchronous mode, RB6 acts as a data transmit (TX) output, and RB7 as data receive input (RX) (16F877 data sheet, section 10.2). Data is usually transmitted in 8-bit words (9 is an option), with the least significant bit sent first. The receiver must sample the input at the same rate as the data is sent, so standard clock (baud) rates are used. 9600 baud is used in our example here, meaning that the bits are transmitted at about 10k bits/s. Separate transmit and receive lines are used, so it is possible for these operations to be carried out simultaneously.

In the block diagram in Figure 9.1, the PIC is connected to a PC. The PIC USART output itself operates at TTL voltages, and needs an external serial line driver to convert its output into a higher symmetrical line voltage. This is necessary because a simple baseband data signal is attenuated down a line, due to the distributed resistance and capacitance of the cabling. The standard RS232 interface operates with a higher line voltage so that the signal can be

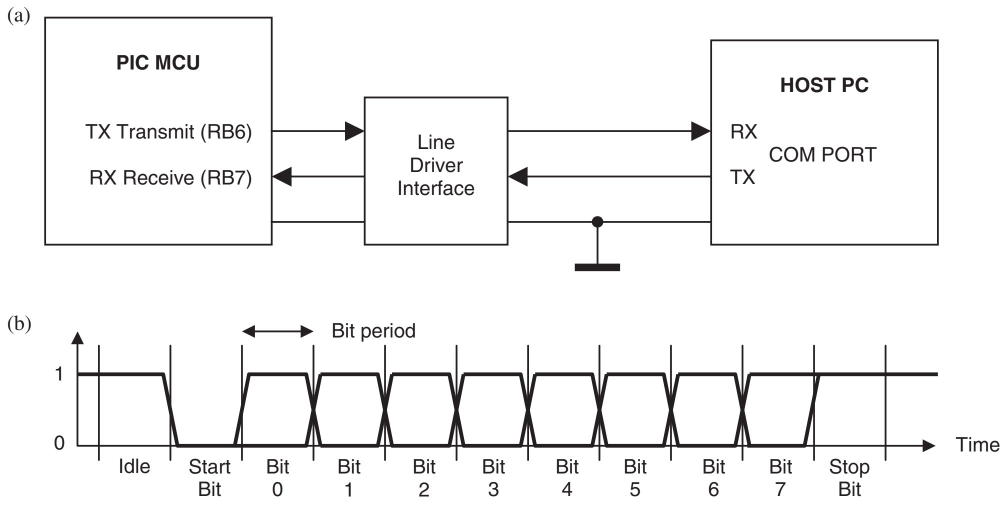


Figure 9.1 USART: (a) connections to PC; (b) signal at PIC port

transmitted further before being overcome by noise and interference. A typical link distance for RS232 is around 10 m with symmetrical voltages of up to +/−25V. +/−12 V is more typical. The signal is also inverted with respect to the TTL version as shown in Figure 9.1 (b).

We will assume initially that the MCU is transmitting. The sender and receiver have to be initialised to use the same baud rate, number of data bits (default 8) and number of stop bits (default 1). The transmit (TX) output is high when idle (external line negative). When the serial buffer register (TXREG) is written, the data is automatically sent as follows. The start of the byte transmission is signalled by the line going low for one clock period (start bit). The following 8 bits are then output from the transmit register, at intervals determined by the selected baud rate. In the diagram, the bits are shown as both high and low, to indicate that either is possible, depending on the particular data word. After the last data bit, the line is taken high by the transmitter for one clock period (stop bit), and that is the end of that transmission. The line is left high if there is no more data; another word can be transmitted after a delay, or immediately following. The protocol is thus about as simple as it is possible to get. The data is often ASCII coded, as the USART is frequently used to transmit character-based messages.

The receiver must be initialised to read in the data at the same baud rate. At 9600 baud, the bit period is about 100 μs. When the falling edge of the start bit is detected, the receiver must wait for 1.5 bit periods, then sample the line for the first data bit (LSB), read the next after a further clock cycle and so on until the set number of bits has been read in. The stop bit confirms the end of the byte, and another transmission can start. An interrupt flag is used to signal to the receiver MCU that there is data waiting. It must be read from RCREG before the next byte arrives.

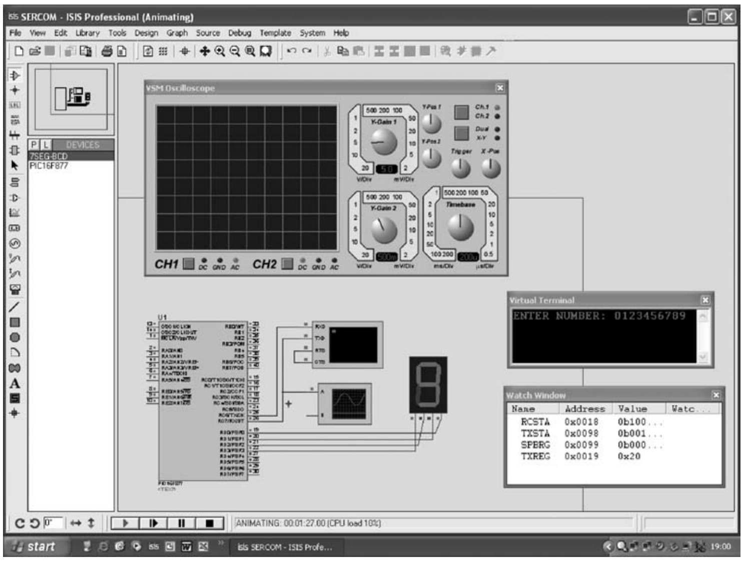
The PIC data sheet has details of the operation of the USART interface. The data is loaded into TXREG (FSR 19h) and transferred automatically to the transmit register when it is ready to send. The shift clock is derived from a baud rate generator, which uses the value in SPBRG (FSR 99h) in its counter. This counter has a post-scaler which divides the output by 16 or 64, depending on the setting of control bit BRGH, so that all the standard baud rates can be achieved using an 8-bit counter. The value to be pre-loaded into this register to obtain a given baud rate is listed in tables in the MCU data. The error associated with each counter value and post-scaler setting is also specified, so that the best option can be selected. The value 25d is used in the demo program, with BRGH = 1 giving an error of only 0.16% from the exact value for 9600 baud. However, a considerable error can be tolerated because sampling only needs to be synchronised over 10 or 11 cycles at the receiver (start + 8/9 data + stop bit). It can be seen from these tables that the error can be up to 10%, and the system should still work.

Test System

The USART test system is shown in Figure 9.2. A simulated terminal is connected to the USART port, and an oscilloscope allows the data signal to be observed. A BCD-encoded 7-segment display is attached to Port D to display the data as they are received by the MCU. It displays the decimal digit for a 4-bit pure binary input.

When the system is started, the PIC generates a user prompt for the virtual terminal, which then waits for the user to input characters at the keyboard. The terminal displays the input character and generates corresponding ASCII code in RS232 format, which is received by the PIC. The code is read from RXREG, and a BCD value is derived (subtract 30h) and output on the BCD display; only numerical characters give the correct display.

The test program is outlined in Figure 9.3, and the source code in Program 9.1. Once the USART module has been initialised, a code is transmitted and received by simply writing or reading the port buffer registers. To send, the byte is moved into TXREG, and the program then waits for the corresponding interrupt flag, TXIF, to be set. To receive, the reception-enable bit, CREN, is set, and the program waits for the interrupt bit, RCIF, to be set to indicate that a byte has been received in RCREG. This is then copied to a suitable storage location for processing. The USART module handles the received and transmit protocols transparently.


SERCOM - ISIS Professional (Animating)
File View Edit Library Tools Design Graph Source Debug Template System Help
DEVICES
7SEG-BCD
PIC16F877
VSM Oscilloscope
CH1
CH2
Y-Gain 1
Y-Gain 2
Timebase
Trigger
Virtual Terminal
Watch Window
U1
ANIMATING: 00:01:27.00 (CPU load 10%)
start
ISIS SERCOM - ISIS Profe...
19:00

```
ENTER NUMBER: 0123456789
```

```
Name      Address   Value     Watc...
  RCSTA   0x0018    0b100...
  TXSTA   0x0098    0b001...
  SPBRG   0x0099    0b000...
  TXREG   0x0019    0x20
```

Figure 9.2 USART simulation

```
SERCOM
        Program to demonstrate USART operation by
        outputting a fixed message to a simulated terminal,
        reading numerical input from it, displaying it in BCD,
        and sending it back to the terminal.

Initialise
        Port D:   BCD display outputs
        USART:    8 bits, asynchronous mode
                  9600 baud (4MHz clock)
                  Enable

Main
        Write message from ASCII table to terminal
        REPEAT
                  Read input, display and echo
        ALWAYS

Subroutines
        Write message from ASCII table to terminal
                  REPEAT
                          Get character from table
                          Output to terminal
                  UNTIL all done

        Read input, display and echo
                  Get input character
                  Convert to BCD and display
                  Echo character back to terminal
```

Figure 9.3 USART program outline

SPI

The Master Synchronous Serial Port (MSSP) module in the PIC provides two main types of communication: SPI and I^{2}C (usually pronounced I squared C). These are used for communication between processors and slower peripheral devices within a single system. SPI is simpler and faster, using a hardware addressing system. I^{2}C is more complex, with software addressing, rather like a simple network protocol, so would be used in larger multi-processor systems, and for access to serial memory and suitable transducer interfaces.

SPI is a synchronous protocol, that is, it has a separate clock signal connection to control the send and receive operations (Figure 9.4). 8-bit data is clocked in and out of the SPI shift register by a set of eight clock pulses, which are either internally generated, or detected at the clock input. No start and stop bits are necessary, and it is faster than the USART in any case, operating at up to 5 MHz if the MCU clock is 20 MHz.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;         SERCOM.ASM                             MPB 10-9-05
;...............................................................
;
;         Test RS232 communications using the
;         USART Asynchronous Transmit and Receive
;
;         The Proteus Virtual Terminal allows ASCII characters
;         to be generated from the keyboard and displayed.
;         The program outputs a fixed message to the display
;         from a table, and then displays numbers input from the
;         terminal on a BCD 7-segment LED display.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          PROCESSOR 16F877    ; define MPU
          __CONFIG 0x3731     ; XT clock (4MHz)

;         LABEL EQUATES       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          INCLUDE "P16F877.INC"         ; Standard labels

          Point     EQU       020
          Inchar    EQU       021

; Initialise ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          ORG       0                   ; Place machine code
          NOP                           ; Required for ICD mode

          BANKSEL   TRISD               ; Select bank 1
          CLRF      TRISD               ; Display outputs
          BCF       TXSTA,TX9           ; Select 8-bit tx
          BCF       TXSTA,TXEN          ; Disable transmission
          BCF       TXSTA,SYNC          ; Asynchronous mode
          BSF       TXSTA,BRGH          ; High baud rate

          MOVLW     D'25'               ; Baud rate value ..
          MOVWF     SPBRG               ; .. 9600 baud, 4MHz
          BSF       TXSTA,TXEN          ; Enable transmission

          BANKSEL   RCSTA               ; Select bank 0
          BSF       RCSTA,SPEN          ; Enable serial port


; MAIN LOOP ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


          CALL      write               ; Message on terminal
readin    CALL      read                ; Get input from terminal
          GOTO      readin              ; Keep reading until reset

; SUBROUTINES ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Write message to terminal.....................................

write     CLRF      Point               ; Table pointer = 0
next      MOVF      Point,W             ; Load table pointer
          CALL      mestab              ; Get character
          CALL      sencom              ; Output to terminal
          INCF      Point               ; Point to next
          MOVLW     D'14'               ; Number of chars + 1
          SUBWF     Point,W             ; Check pointer
          BTFSS     STATUS,Z  ; Last character done?
          GOTO      next                ; No - next
          RETURN                        ; All done
```

Program 9.1 USART serial communication

```
; Read input numbers from terminal..............................

read      BSF       RCSTA,CREN          ; Enable reception
waitin    BTFSS     PIR1,RCIF ; Character received?
          GOTO      waitin              ; no - wait

          MOVF      RCREG,W             ; get input character
          MOVWF     Inchar              ; store input character
          MOVLW     030                 ; ASCII number offset
          SUBWF     Inchar,W  ; Calculate number
          MOVWF     PORTD               ; display it
          RETURN                        ; done


; Transmit a character .........................................

sencom    MOVWF     TXREG               ; load transmit register
waitot    BTFSS     PIR1,TXIF ; sent?
          GOTO      waitot              ; no
          RETURN                        ; yes


; Table of message characters...................................

mestab    ADDWF     PCL                 ; Modify program counter
          RETLW     'E'                 ; Point = 0
          RETLW     'N'                 ; Point = 1
          RETLW     'T'                 ; Point = 2
          RETLW     'E'                 ; Point = 3
          RETLW     'R'                 ; Point = 4
          RETLW     ' '                 ; Point = 5
          RETLW     'N'                 ; Point = 6
          RETLW     'U'                 ; Point = 7
          RETLW     'M';                ; Point = 8
          RETLW     'B'                 ; Point = 9
          RETLW     'E'                 ; Point = 10
          RETLW     'R'                 ; Point = 11
          RETLW     ':'                 ; Point = 12
          RETLW     ' '                 ; Point = 13


          END       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Program 9.1 *Continued*

One processor must act as a master, generating the clock. Others act as slaves, using the master clock for sending and receiving. The SPI signals are listed below, with the 16F877 pin allocations:

- Serial Clock (SCK) (RC3)
- Serial Data In (SDI) (RC4)
- Serial Data Out (SDO) (RC5)
- Slave Select (!SS) (RA5)

The test system (Figure 9.5) consists of three processors, a master, a slave transmitter and a slave receiver. The slave transmitter has a BCD switch connected to Port D, which generates the test data. The binary code 0–9 is read in to the transmitter and sent to the master controller via the SPI link. The send is enabled via the !SS input of the slave, by an active low signal from the master, RC0. The clock is supplied by the master to shift the data out of SSPSR shift register in the slave, and shift it into SSPSR in the master at the same time. The master then retransmits the same data to the slave receiver by the

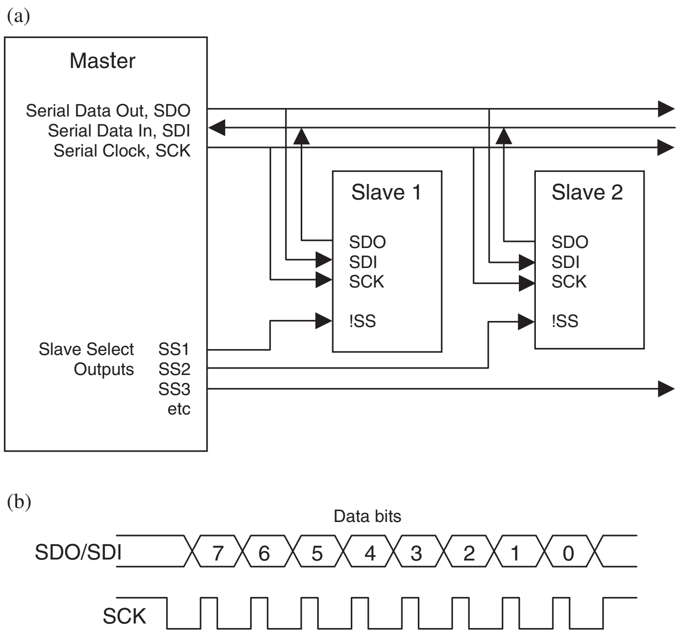


Figure 9.4 SPI operation: (a) SPI connections; (b) SPI signals

same method. The slave receiver does not need a slave select input to enable reception, as it is already initialised to expect only SPI data input.

Each chip needs its own program to operate the SPI port. The three programs are listed as Programs 9.2 (master), 9.3 (slave transmitter) and 9.4 (slave receiver). All chips run at 4 MHz, giving an SPI clock period of 1 µs. The SPI outputs (SCK and SDO) need to be set as outputs in each MCU. The operation is controlled by SFRs SSPSTAT (Synchronous Serial Port Status register, address 94h) and SSPCON (Synchronous Serial Port Control register, address 14h).

In the master program, the default-operating mode is selected by clearing all bits in both of these control registers. SSPSTAT bits mainly provide signal-timing options. The low nibble of SSPCON sets the overall mode, master or slave (0000 = master). In the slave transmitter, the bits are set to 0100 (slave mode, slave select enabled). In the slave receiver, they are set to 0101 (slave mode, slave select disabled). Bit SSPEN enables the SPI module prior to use in all three processors.

The slave transmitter initiates the data transfer by simply writing the data read in from the switches to the SSPBUF (Synchronous Serial Port Buffer). When clock pulses are input at SCK from the master, the bits in SSPBUF are shifted out on the falling edge of each pulse. The slave transmitter program waits for the

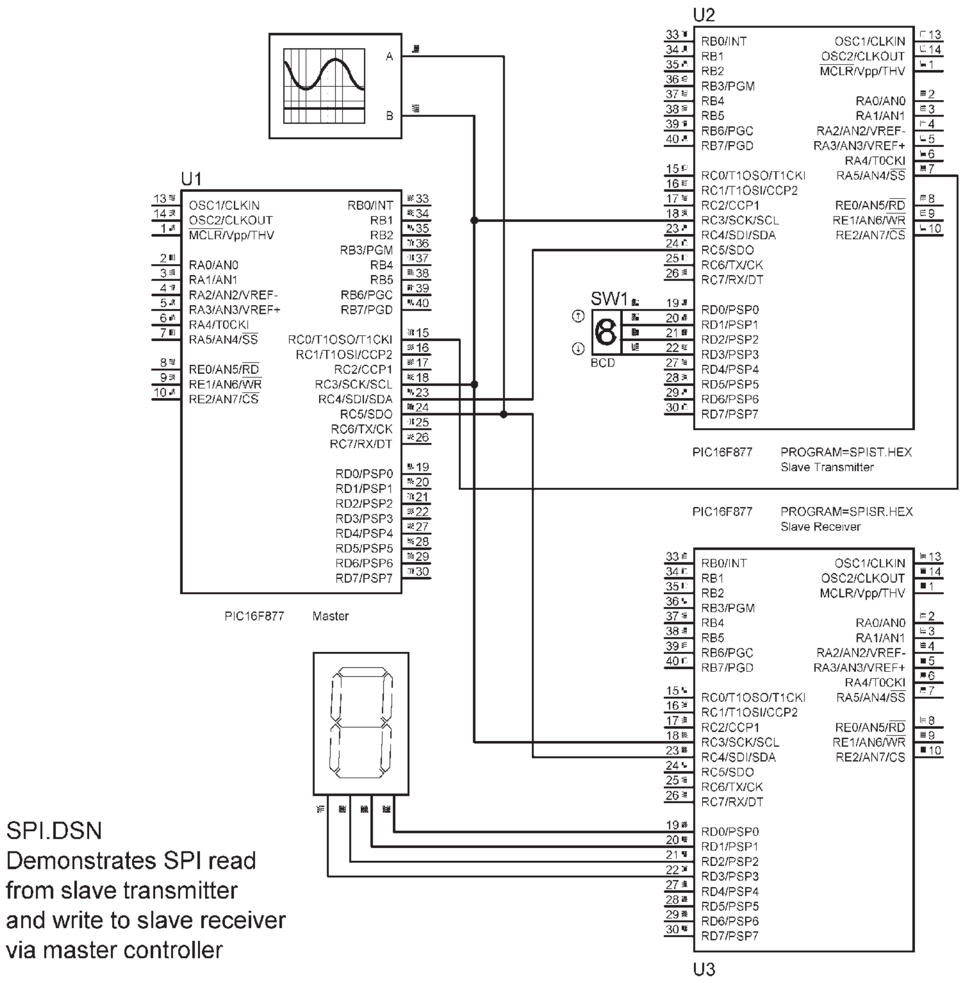


Figure 9.5 SPI Test Circuit

SSPIF (SSP Interrupt Flag) to be set to indicate that the data have been sent. In the master, the clock is started by a dummy write to the SSPBUF register. The master program then waits for the interrupt flag to indicate that the data has been received.

The test data is then rewritten to SSPBUF, which initiates the data output cycle. The master program again waits for SSPIF to indicate that the master transmission cycle is complete. This transmission is picked up by the slave receiver under control of the master clock. It simply waits for the interrupt flag to indicate that a data byte has been received, and copies it to the BCD 7-segment display, to indicate to the user a successful data cycle.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;         SPIM.ASM                      MPB       13-9-05
;...............................................................
;
;         SPI Master program
;
;         Outputs clock to slave transmitter, receives BCD data
;         and sends it to slave receiver for display
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          PROCESSOR 16F877    ; define MPU
          __CONFIG 0x3731     ; XT clock (4MHz)

;         LABEL EQUATES       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          INCLUDE "P16F877.INC"         ; Standard labels

Store     EQU       020

; Initialise ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          ORG       0                   ; Place machine code
          NOP                           ; Required for ICD mode

          BANKSEL   TRISC
          BCF       TRISC,5             ; Serial data (SDO) output
          BCF       TRISC,3             ; Serial clock (SCK) output
          BCF       TRISC,0             ; Slave select (SS) output
          CLRW      SSPSTAT             ; Default clock timing

          BANKSEL   PORTD
          BSF       PORTC,0             ; Reset slave transmitter
          CLRF      SSPCON              ; SPI master mode, 1MHz

; MAIN LOOP ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          BSF       SSPCON,SSPEN        ; Enable SPI mode
again     BCF       PORTC,0             ; Enable slave tx

          MOVWF     SSPBUF              ; Rewrite buffer to start
waitin    BTFSS     PIR1,SSPIF          ; wait for SPI interrupt
          GOTO      waitin              ; for data received

          BCF       PIR1,SSPIF          ; clear interrupt flag
          MOVF      SSPBUF,W            ; read SPI buffer
          MOVWF     Store               ; store BCD value
          BSF       PORTC,0             ; Disable slave tx
          MOVWF     SSPBUF              ; Reload SPI buffer

waits     BTFSS     PIR1,SSPIF          ; wait for SPI interrupt
          GOTO      waits               ; for data sent
          BCF       PIR1,SSPIF          ; clear interrupt flag
          GOTO      again               ; repeat main loop

          END       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Program 9.2 SPI master program

I^2C

I^2C is a more versatile system level serial data transfer method. It only needs two bus connected signals; clock (SCL) and data (SDA) lines (Figure 9.6). These allow a master controller to be connected to up to 1023 other devices. These can include other MCUs, memory devices, analogue converters and so on. The example used here is access to an external EEPROM memory, which would be used to expand the non-volatile data storage in an MCU system, and

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;         SPIST.ASM                     MPB       14-9-05
;...............................................................
;         SPI Slave Transmitter program
;         Waits for !SS and transmits switch BCD data
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          PROCESSOR 16F877    ; define MPU
          __CONFIG 0x3731     ; XT clock (4MHz)

;         LABEL EQUATES       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          INCLUDE "P16F877.INC"         ; Standard labels

; Initialise ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          ORG       0                   ; Place machine code
          NOP                           ; Required for ICD mode

          BANKSEL   TRISC
          BCF       TRISC,5             ; Serial data output
          CLRW      SSPSTAT             ; Default clock timing

          BANKSEL   PORTD
          MOVLW     B'00000100'         ; SPI slave mode with SS
          MOVWF     SSPCON              ; SPI clock = 1MHz
          BSF       SSPCON,SSPEN        ; Enable SPI mode

; MAIN LOOP ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

start     MOVF      PORTD,W             ; Read BCD switch
          MOVWF     SSPBUF              ; Write SPI buffer

wait      BTFSS     PIR1,SSPIF          ; wait for SPI interrupt
          GOTO      wait
          BCF       PIR1,SSPIF          ; clear interrupt flag
          GOTO      start               ; repeat main loop

          END       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Program 9.3 SPI slave transmit program

is used in a general purpose PIC base module described in Chapter 11. The memory used here is a Microchip 24AA128, which stores 16 kb of data (128k bits); the system simulation is shown in Figure 9.7.

It can be seen that the signal lines have to be pulled up to 5 V so that any one of the devices connected to it can control the line by pulling it down; this allows slaves to acknowledge operations initiated by the master. The transmitted byte has a start bit (low) and an 8-bit address or data byte (MSB first), terminated with an acknowledge from the slave. Each bit is accompanied by a clock pulse in the same way as SPI. Clock speeds are programmed by preloading the baud rate generator with a suitable value, giving speeds of up to 1 MHz.

Control, Address and Data Format

The system uses a software addressing system, where the external device and a location within it can be selected in the same way as a conventional address decoding system (see Chapter 11). On the chip used here, there are three hardware

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;         SPISR.ASM                     MPB       14-9-05
;...............................................................
;         SPI Slave Receiver program
;         Waits for BCD data sent from master and displays it
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          PROCESSOR 16F877              ; define MPU
          __CONFIG 0x3731               ; XT clock (4MHz)
          INCLUDE "P16F877.INC"         ; Standard labels

; Initialise ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          ORG       0                   ; Place machine code
          NOP                           ; Required for ICD mode

          BANKSEL   TRISD
          CLRF      TRISD               ; Display outputs
          CLRF      SSPSTAT             ; Default clock timing

          BANKSEL   PORTD
          MOVLW     B'00000101'         ; SPI slave, SS disabled
          MOVWF     SSPCON              ; SPI clock = 1MHz

; MAIN LOOP ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          BSF       SSPCON,SSPEN        ; Enable SPI mode
wait      BTFSS     PIR1,SSPIF          ; wait for SPI interrupt
          GOTO      wait

          MOVF      SSPBUF,W            ; get data
          MOVWF     PORTD               ; and display
          BCF       PIR1,SSPIF          ; clear interrupt flag
          GOTO      wait                ; repeat main loop

          END       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Program 9.4 SPI slave receive program

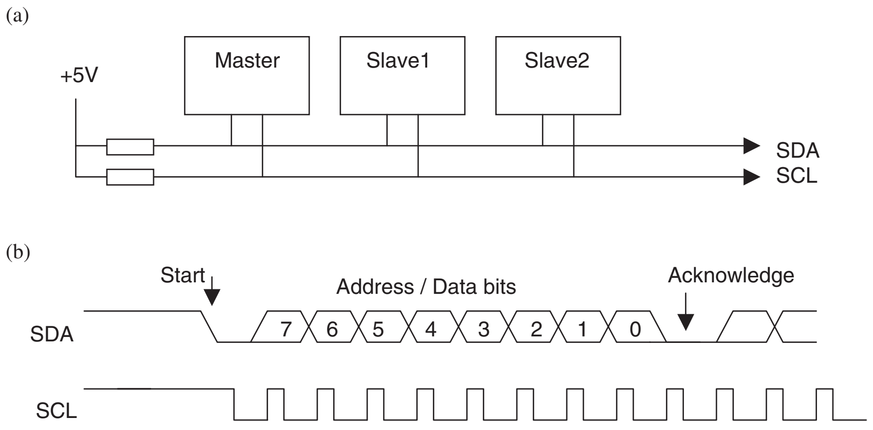
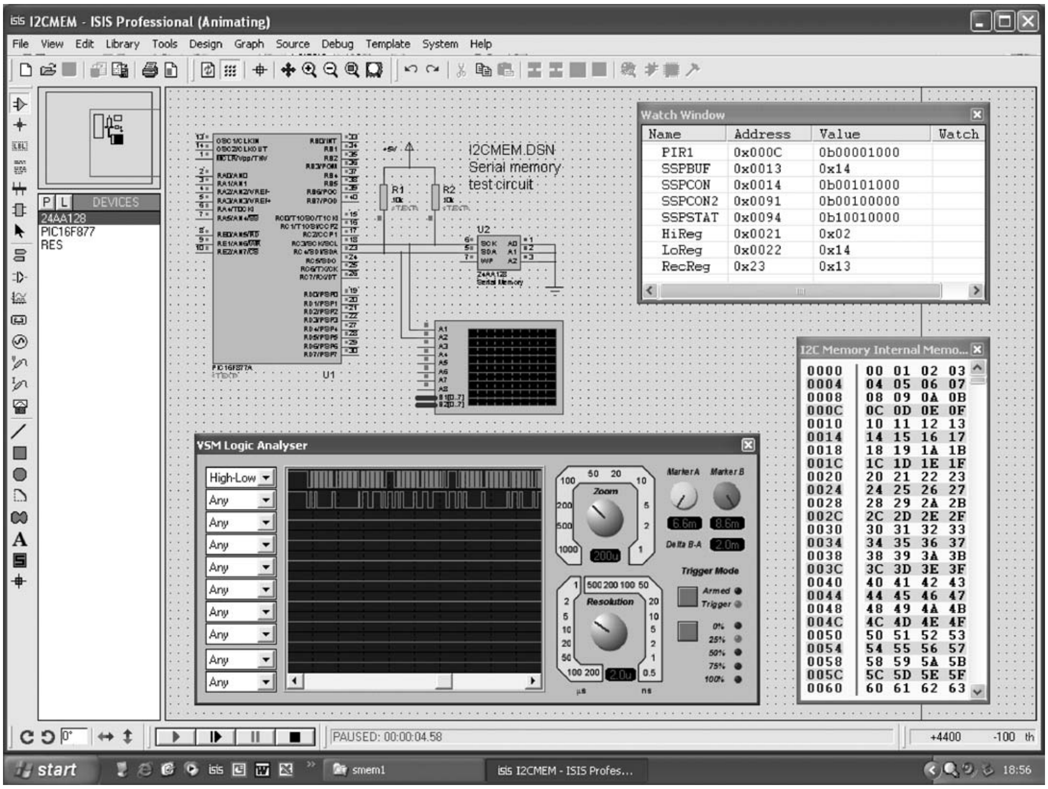


Figure 9.6 Inter-IC bus (I^2C): (a) I^2C connections; (b) I^2C signals


I2CMEM - ISIS Professional (Animating)
File View Edit Library Tools Design Graph Source Debug Template System Help
I2CMEM.DSN
Serial memory
test circuit
24AA128
PIC16F877
RES
VSM Logic Analyser
PAUSED: 00:00:04.58
start

Watch Window

Name	Address	Value	Watch
PIR1	0x000C	0b00001000	
SSPBUF	0x0013	0x14	
SSPCON	0x0014	0b00101000	
SSPCON2	0x0091	0b00100000	
SSPSTAT	0x0094	0b10010000	
HiReg	0x0021	0x02	
LoReg	0x0022	0x14	
RecReg	0x23	0x13	

I2C Memory Internal Memo...

```
0000  00 01 02 03
0004  04 05 06 07
0008  08 09 0A 0B
000C  0C 0D 0E 0F
0010  10 11 12 13
0014  14 15 16 17
0018  18 19 1A 1B
001C  1C 1D 1E 1F
0020  20 21 22 23
0024  24 25 26 27
0028  28 29 2A 2B
002C  2C 2D 2E 2F
0030  30 31 32 33
0034  34 35 36 37
0038  38 39 3A 3B
003C  3C 3D 3E 3F
0040  40 41 42 43
0044  44 45 46 47
0048  48 49 4A 4B
004C  4C 4D 4E 4F
0050  50 51 52 53
0054  54 55 56 57
0058  58 59 5A 5B
005C  5C 5D 5E 5F
0060  60 61 62 63
```

Figure 9.7 I^{2}C simulation

address selection pins, which allow it to be allocated in one of eight 3-bit addresses within the same system. Thus, a total of 16 × 8 = 128 kb could be installed. The location required within the chip is selected by 14-bit address supplied by the master controller as part of the access cycle. The chip is differentiated from other I^{2}C devices on the bus by a 4-bit code (1010) in the control word. The data sheet for the 24AA128 memory chip details the required signalling very clearly.

The format of the data blocks for write and read are shown in Table 9.1. To write a byte to memory, the control code is sent first. This alerts the memory that a write is coming, when the control code and chip select bits match its own control code and hardware address (set up on chip address inputs). The control code is 1010, the chip address is 000 and the read/not write bit is 0, to indicate a write. This chip-select byte is followed by the location address to write to, which is 14 bits for this device. The high address bits are sent first, with bits

Byte	Function	Bits								Description
Write byte										
1	Control byte	1	0	1	0	CS2	CS1	CS0	0	Control code, chip select address, WRITE
2	Address high byte	X	X	A13	A12	A11	A10	A9	A8	Memory page select
3	Address low byte	A7	A6	A5	A4	A3	A2	A1	A0	Memory location select
4	Data	D7	D6	D5	D4	D3	D2	D1	D0	Data
Read byte										
1	Control byte	1	0	1	0	CS2	CS1	CS0	0	Control code, chip select address, WRITE
2	Address high byte	X	X	A13	A12	A11	A10	A9	A8	Memory page select
3	Address low byte	A7	A6	A5	A4	A3	A2	A1	A0	Memory location select
4	Control byte	1	0	1	0	CS2	CS1	CS0	1	Control code, chip select address, READ
5	Data	D7	D6	D5	D4	D3	D2	D1	D0	Data

Table 9.1 I^{2}C sequences for random serial memory access (10-bit address)

A14 and A15 having no effect (X = don't care). The address is followed by the eight data bits to be written to the location specified.

The read sequence is 5 bytes in total. The first three are the same as for the write, where the chip is selected and location address is written to the address latches within the memory chip. A control byte to request a read operation is then sent, and the data returned from the selected location by the slave device.

Transmission Control

The clock and data lines are high when idle. The write and read sequences are initiated by a start bit sequence generated by the master controller (24AA128 data sheet, Figure 4-1). The transmission starts when the data line goes low; the clock then starts and an address or data bit output during the clock high period, which is latched into the slave receive shift register on the falling clock edge. After the eighth bit, the master (MCU) releases the data line, to allow the slave (EEPROM) to hold the line low to acknowledge the bits have been received. At the end of the next (ninth) clock high period, the slave releases the data line and the master can transmit the first bit of the next byte (Figure 4-2).

In the memory write sequence, when the acknowledge is generated after the data byte has been received, and the master stops and both lines go high. In the memory read sequence, the master stops after the address write has been sent, and restarts in order to send a read control byte. It will then read the eight data bits returned by the memory chip, but does not generate an acknowledge. The master then stops, and the lines go idle again.

The test program reads and writes every location (16384 addresses) by this method. The maximum write cycle time specified is 5 ms ($16384 \times 0.005 = 82$ s). It therefore takes a considerable time to complete this test. If the memory is being accessed sequentially, as is frequently the case, the overall access time can be reduced by using the page read and address auto-increment features of the chip, which are explained in the EEPROM data sheet.

In the test software (Program 9.5), the control operations are broken down as much as possible so that each step can be identified. The program outline (Figure 9.8) shows that the data write and read operations are carried out one

```
SMEM1
          Serial memory access using I2C serial bus

          MAIN PROGRAM ---------------------------------------------------------------------------------------

                    Initialise

                    Loop
                              Write a byte to memory
                              Read the same byte from memory
                              IF end of memory page, increment page number
                    Until end of memory


          SUBROUTINES -----------------------------------------------------------------------------------------

                    Initialise
                              SSPCON2:       Set Acknowledge flags inactive + status bits inactive
                              SSPSTAT:       Slew rate control disabled + status bits inactive
                              SSPADD:        Load with baud rate count value
                              SSPCON:        Set SSP enable bit, select I2C master mode
                              OPTION:        TMR0: internal clock, divide by 64


                    Write a byte to memory
                              Generate start condition
                              Send write control byte for memory chip
                              Send address bytes to memory chip
                              Send data byte to memory chip
                              Send stop bit to memory chip
                              Wait 10ms using TMR0


                    Read the byte from memory
                              Generate start condition
                              Send write control byte for memory chip
                              Send address bytes to memory chip
                              Send read control byte for memory chip
                              Wait until data received
                              Send acknowledge and stop bits
                              Store received data byte
```

Figure 9.8 I^{2}C program outline

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;        I2CMEM.ASM                  MPB      7-1-06
;
;        Test program for 24AA128 I2C 16k byte serial
;        memory with P16F877A (4MHz XT)
;        Demonstrates single byte write and read
;        with 10-bit address.
;
;        Write data from                      0x20
;        High address                         0x21
;        Low address                          0x22
;        Read data back to                    0x23
;
;        Version: Final
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

         PROCESSOR 16F877A
         __CONFIG 3FF1
         INCLUDE "P16F877A.INC"

; Data, address & control registers ;;;;;;;;;;;;;;;;;;;;;;

SenReg   EQU      0x20                ; Send data store
HiReg    EQU      0x21                ; High address store
LoReg    EQU      0x22                ; Low address store
RecReg   EQU      0x23                ; Receive data store
ConReg   EQU      0x24                ; Control byte store
Temp     EQU      0x25                ; Scratchpad location

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

         ORG      0                   ; Program start address

         CLRF     SenReg              ; Zeroise data
         CLRF     HiReg               ; Zeroise high address
         CLRF     LoReg               ; Zeroise low address
         GOTO     begin               ; jump to main program

; SUBROUTINES ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Wait for interrupt flag SSPIF for send/recive done ...

wint     NOP                          ; BANKSEL has address
         BANKSEL  PIR1                ; Select bank
         BCF      PIR1,SSPIF          ; reset interrupt flag
win      NOP
         BTFSS    PIR1,SSPIF          ; wait for..
         GOTO     win                 ; ..transmit done
         RETURN                       ; Done
; Send a byte -------------------------------------------
send     NOP                          ; Select..
         BANKSEL  SSPBUF              ; .. bank
         MOVWF    SSPBUF              ; Send address/data
         CALL     wint                ; Wait until sent
         RETURN                       ; Done
```

Program 9.5 I^{2}C memory test

```
; Routines to send start, control, address, data, stop ----

sencon   NOP                        ; GENERATE START BIT
         BANKSEL  PIR1              ;
         BCF      PIR1,SSPIF        ; Clear interrupt flag
         BANKSEL  SSPCON2           ; select register page
         BSF      SSPCON2,ACKSTAT   ; Set acknowledge flag
         BSF      SSPCON2,SEN       ; Generate start bit
         CALL     wint              ; wait till done

         MOVF     ConReg,W          ; SEND CONTROL BYTE
         CALL     send              ; Memory ID & address
         RETURN                     ; done
;-----------------------------------------------------------
senadd   NOP                        ; SEND ADDRESS BYTES
         BANKSEL  SSPCON            ;
         MOVF     HiReg,W           ; load address high byte
         CALL     send              ; and send
         MOVF     LoReg,W           ; load address low byte
         CALL     send              ; and send
         RETURN
;-----------------------------------------------------------
sendat   MOVF     SenReg,W          ; SEND DATA BYTE
         CALL     send              ; and send
         RETURN                     ; done
;-----------------------------------------------------------
senstop  NOP                        ; GENERATE STOP BIT
         BANKSEL  SSPCON2           ;
         BSF      SSPCON2,PEN       ; Generate stop bit
         CALL     wint              ; wait till done
         RETURN                     ; done
;-----------------------------------------------------------
senack   NOP                        ; ACKNOWLEDGE
         BANKSEL  SSPCON2           ;
         BSF      SSPCON2,ACKDT     ; Set ack. bit high
         BSF      SSPCON2,ACKEN     ; Initiate ack.sequence
         CALL     wint              ; Wait for ack. done
         RETURN                     ; done
;-----------------------------------------------------------
wait     NOP                        ; WAIT FOR WRITE DONE
         BANKSEL  TMR0              ;
         MOVLW    d'156'            ; Set starting value
         MOVWF    TMR0              ; and load into timer
         BANKSEL  INTCON            ; 64 x 156us = 10ms
         BCF      INTCON,T0IF       ; Reset timer out flag
wem      BTFSS    INTCON,T0IF       ; Wait 10ms
         GOTO     wem               ; for timeout
         BANKSEL  TMR0              ; default bank
         RETURN                     ; Byte write done....
; Initialisation sequence ---------------------------------

init     NOP                        ; INITIALISE
         BANKSEL  SSPCON2           ;
         MOVLW    b'01100000'       ; Set ACKSTAT,ACKDT bits
         MOVWF    SSPCON2           ; Reset SEN,ACK bits
         MOVLW    b'10000000'       ;
         MOVWF    SSPSTAT           ; Speed & signal levels
         MOVLW    0x13              ; Clock = 50kHz
         MOVWF    SSPADD            ; Load baud rate count-1
         BANKSEL  SSPCON            ;
         MOVLW    b'00101000'       ;
         MOVWF    SSPCON            ; Set mode & enable
         BCF      PIR1,SSPIF        ; clear interrupt flag
```

Program 9.5 *Continued*

```
; Initialise TIMER0 for write delay -----------------------

        BANKSEL  OPTION_REG         ; SETUP TIMER0
        MOVLW    B'11000101'        ; TIMER0 setup code
        MOVWF    OPTION_REG         ; Internal clock,1/64
        BANKSEL  TMR0               ; Default bank
        RETURN                      ; Done

; Write a test byte to given address ----------------------

writeb  MOVLW    0xA0               ; Control byte for WRITE
        MOVWF    ConReg             ; Load it
        CALL     sencon             ; Send control byte
        CALL     senadd             ; Send address bytes
        CALL     sendat             ; Send data byte
        CALL     senstop            ; Send stop bit
        CALL     wait               ; Wait 10ms for write
        RETURN

; Read the byte from given address -------------------------

readb   MOVLW    0xA0               ; Control byte to WRITE
        MOVWF    ConReg             ; address to memory
        CALL     sencon             ; Send control byte
        CALL     senadd             ; Send address bytes
        CALL     senstop            ; Stop

        MOVLW    0xA1               ; Control byte to READ
        MOVWF    ConReg             ; data from memory
        CALL     sencon             ; Send control byte
        BANKSEL  SSPCON2
        BSF      SSPCON2,RCEN       ; Enable receive mode
war     BTFSS    SSPSTAT,BF         ; Check ...
        GOTO     war                ; for read done
        CALL     senack             ; send NOT acknowledge
        CALL     senstop            ; send stop bit

        MOVF     SSPBUF,W           ; Read receive buffer
        MOVWF    RecReg             ; and store it
        RETURN                      ; Done

; MAIN PROGRAM   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

begin   CALL     init               ; Initialise for I2C
next    CALL     writeb             ; write the test byte
        CALL     readb              ; and read it back
        INCF     SenReg             ; next data
        INCF     LoReg              ; next location
        BTFSS    STATUS,Z           ; end of memory block?
        GOTO     next               ; no, next location
        INCF     HiReg              ; next block

        MOVF     HiReg,W            ; copy high address byte
        MOVWF    Temp               ; store it
        MOVLW    0x40               ; Last block = 3F
        SUBWF    Temp               ; Compare
        BTFSS    STATUS,Z           ; Finish if block = 40xx
        GOTO     next               ; next memory block..
        SLEEP                       ; .. unless done

        END      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Program 9.5 *Continued*

after the other on the same location, but the address resent for the read so that read and write sequences can be used separately in other programs. In real applications, a sequential read or write is more likely; this can be completed more quickly for a sequential data block, especially the read, because the memory has an automatic increment mode for the addressing, so only the first address needs to be sent.

The read and write operations use the same subroutines to generate the transmission control operations, which are

- Generate start bit
- Send control byte
- Send address bytes
- Send data byte
- Generate stop bit
- Generate acknowledge
- Wait 10 ms for write completion

Table 9.2 summarises the registers and bits used in the test program (see the master mode timing diagram in the 16F877 data sheet, Figure 9-14).

Register	Address	Bit/s	Bit name	Active	Function
SSPBUF (Data)	13h	All	---------	Data /add	SSP send/receive buffer register
SSPCON (Control)	14h	3-0	SSPMx	1000	SSP mode select bits
		5	SSPEN	1	SSP enable bit
SSPCON2 (Control)	91h	0	SEN	1	Initiate start of transmission
		2	PEN	1	Initiate stop condition
		3	RCEN	1	Receive mode-enable bit
		4	ACKEN	1	Initiate acknowledge sequence
		5	ACKDT	1	Acknowledge data bit setting
		6	ACKSTAT	0	Acknowledge received from slave
SSPSTAT (Status)	94h	0	BF	1	SSP buffer is full
		2	R/W	1	Read/write bit – transmit in progress
		6	CKE	0	I2C clock mode
SSPADD (Preload)	93h	All	---------	0x13	Baud rate count
PIR1	0Ch	3	SSPIF	1	SSP interrupt flag

Table 9.2 I^{2}C master mode, selected bits

The shift register used to send and receive the data bits is not directly accessible. The buffer register SSPBUF holds the data until the shift register is ready to send it (transmit mode), or receives it when the shift is finished in receive mode. The send operation is triggered by setting the Send Enable (SEN) bit, and the Buffer Full (BF) flag indicates that the data have been loaded. The interrupt flag (SSPIF) is automatically set to indicate start of transmission, and must be cleared in software if necessary. The Acknowledge Status (ACKSTAT) bit is cleared by an acknowledge from the slave, to indicate that the byte has been received. SSPIF is then set again to indicate the end of the byte transmission, and the buffer can then be written with the next byte.

If data is to be received by the master, the read/write bit is set in the control word, and the receive mode enabled by setting RCEN bit. The BF flag is set when the data have been received, and the buffer must be read (unloaded) before another data byte is received, or sent. Full details are provided for all I^2C transmit and receive modes in the PIC data sheet. The EEPROM data sheet explains the requirements for that particular peripheral.

It can be seen that I^2C needs relatively complex software control, while SPI needs extra connections for hardware device selection, but is faster. The USART will be used for longer inter-system connections, typically to a PC host. In due course, full network access and USB will also doubtless be integrated into standard microcontrollers.

SUMMARY 9

- The USART block provides asynchronous serial communication at low speed with a PC host and similar remote systems over a distance of a few metres using separate send and receive lines
- The SPI bus provides synchronous serial access to other on-board master or slave devices using a separate clock and data line, with hardware slave selection
- The I^2C bus provides synchronous serial access to other on-board master or slave devices using a separate clock and data line, with hardware chip selection and software slave location/register addressing

ASSESSMENT 9 Total *(40)*

1 Explain why the RS232 type protocol is described as asynchronous. *(3)*

2 Explain why the signal is sent at up to 50 V peak to peak on the RS232 line. *(3)*

3 State the minimum number of bit periods required to send an 8-bit data byte in the RS232 format. *(3)*

4 State the send and receive signal names in RS232. Explain why is it possible to send and receive simultaneously. *(3)*

5 Explain why SPI signals have a more limited transmission range than RS232. *(3)*

6 Describe the function of the signal !SS in the SPI system. How does I^2C implement the same function? *(3)*

7 How is the completion of an SPI receive sequence detected in a PIC program? *(3)*

8 Summarise why a data byte takes longer to send in the I^2C system, compared with SPI. *(3)*

9 How is the correct reception of a data byte acknowledged by an I^2C slave device? *(3)*

10 Describe how the time taken for a block write to serial memory can be reduced, compared with a single byte operation. *(3)*

11 Sketch the RS232 signal as it appears at the output of the line driver (assume +/−12 V). Label all relevant features. *(5)*

12 Explain the function of each group of bits in the I^2C control byte. *(5)*

ASSIGNMENTS 9

9.1 RS232 Output Test

Write a test program for the PIC 16F877, running at 4 MHz. Initialise for 8 data bits, 1 stop bit at 9600 baud and output the same code AAh repeatedly to the RS232 port. Test in simulation mode.

Monitor the output signal on the virtual oscilloscope, and measure the overall time per byte, and the bit period. Compare this with the value specified in the data sheet. Change the data to 81h and show that the display is correct for this data. Change the baud rate to 19200 and confirm that the display is correct. Compare the bit time with the specification. Try sending ASCII, and confirm that the codes are as shown in the ASCII table.

Transfer the application to prototype hardware and confirm that the appropriate output is obtained on an oscilloscope.

9.2 SPI and I^{2}C Debuggers

The SPI and I^{2}C debugger are found in the list of Proteus virtual instruments. They allow these signals to be monitored when the design for a serial communication system is simulated. Use the instruments to monitor the signals in the demonstration systems. Write a full report on the interpretation of the displays obtained, and how to use these devices to aid the development of serial systems.

9.3 Serial Memory System

Modify the serial memory test program to carry out a page write and read (refer to the serial memory data sheet). Estimate the time saved compared with the byte write and read process.

Expand the I^{2}C memory system to 128k, and modify the test program to ensure that all devices can be successfully written and read. Estimate the time required to test every location, and modify the program to test a sample of locations in each chip to indicate its correct function within a reasonable timescale, say 10 s.

10

Sensor Interfacing

A wide variety of sensors are used in digital control systems and interfacing them requires a good understanding of linear amplifier design and signal conditioning techniques. Connection to an MCU is simplified if the sensor itself contains built-in signal conditioning, such that the output is linear, reasonably large, conveniently scaled and pre-calibrated. The LM35 temperature sensor is a good example, giving an output of 10 mV/°C, starting at 0°C = 0 mV, but it needs an ADC input. Increasingly, sensors are incorporating all the necessary signal processing and serial data outputs, so that interface design is simplified or eliminated. For example, Microchip supplies a range of temperature sensors with I^2C output.

Sensors

A sensor is an essential device that responds to some environmental variable and converts it into electrical output. This signal may then need to be conditioned (filtered, amplified, attenuated, converted) to allow the MCU to receive the input in a usable form. Digital sensors may provide a direct input at TTL levels, while some analogue inputs might need a high-performance amplifier or complex digital processing.

Digital Sensors

The simplest form of digital sensor is a switch. A manually operated push button or toggle switch may only need a pull-up resistor, or possibly debouncing via a parallel capacitor, hardware latch or software process, as outlined in Chapter 4. A micro-switch can be attached to a mechanical system so that it detects the position of, say, the guard on a machine tool. The machine controller can then be programmed not to start until the guard is closed. The micro-switch may often have an extended operating lever to make it more sensitive.

One disadvantage of the switch or relay contact is that physical wear causes unreliable operation. This problem can be overcome by using a switch which has no moving parts, or is specially designed for reliability. A reed switch is enclosed in a vacuum and operated by the proximity of a magnet to the sprung contacts, which are gold-plated to reduce corrosion effects. The vacuum prevents burning at the contacts due to high-voltage discharge as they open or bounce with an inductive load.

Opto-electrical devices have no moving parts, and are therefore inherently more reliable. An LED and phototransistor are connected in separate circuits, with the transistor operating as a light-activated switch. The opto-isolator includes both in a single package, providing electrical isolation between control and load circuits, which may operate at high voltage. The signal to noise ratio may also be improved, and the digital signal thus ‘cleaned up’. The same devices are used in an opto-detector where the light beam is interrupted by a moving object, grating or perforated wheel; this arrangement can be used to monitor position, speed or acceleration. Light transmission or reflection may be used, depending on measurement. The reflective type can be used as a simple proximity sensor, while position detection often uses a transmissive system.

The inkjet printer provides a good example of a position system. A plastic strip with a fine grating is used to provide position feedback for the print head. The simple periodic grating can be made more precise by grading the light transmission sinusoidally over a cycle, allowing calculation of fractions of a cycle (interpolation). Axis position in machine tools can be controlled down to about 1 μm by this means. If a pair of gratings is used, offset by 90°, the direction of travel can be detected by the phase relationship. To establish absolute position, a reference position is needed from which relative motion can be calculated. For example, a robot arm may need to be started from a known ‘home’ position. Alternatively, a Gray code can be used on the opto-disk; each sector has a unique combination of light and dark bands, so that the absolute position of the stationary shaft can be detected by a set of sensors, one for each band. The pattern is a modified binary code which only changes 1 bit at a time, to prevent incorrect data being sampled on the sector boundaries (Figure 10.1).

Some sensors have a built-in data processing so that an MCU compatible signal is produced; for example, the measured variable may be converted

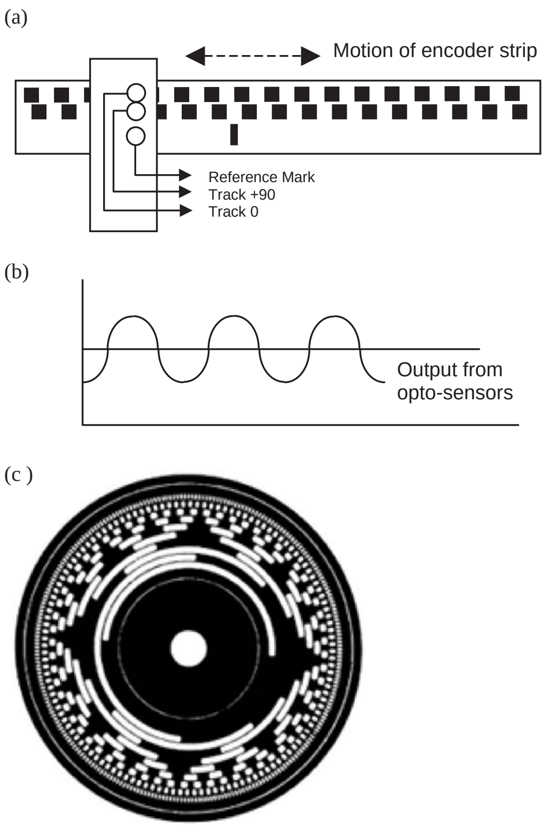


Figure 10.1 Incremental encoder: (a) linear encoder; (b) sinusoidal output; (c) Gray code opto-disk (10 bit)

into a periodic TTL signal. This can be fed into a digital input, and the frequency determined in software by using a timer/counter to measure the number of pulses in unit time, or the period. Analogue to digital converter chips are available which convert the measured voltage into frequency, or transmit the binary form of the measurement in a standard serial format, such as I^2C.

Analogue Sensors

Analogue sensors produce a variable output, which may be voltage, resistance or current. In microcontroller systems, they are usually converted into a voltage in a range suitable for an input comparator (high/low detection) or analogue to digital conversion. Suitable signal conditioning may be needed using amplifiers, filters and so on, to produce a clean signal, controlling noise, drift, interference and so on, with the required output range.

Sensors have certain characteristics which should be specified in the data sheet:

- Sensitivity
- Offset
- Range
- Linearity
- Error
- Accuracy
- Resolution
- Stability
- Reference level
- Transfer function
- Interdependence

The meaning of some of these is illustrated in Figure 10.2.

SENSITIVITY

The ideal sensor characteristic is shown in the characteristic $y = m_1x$. The sensor has a large change in its output for a small change in its input; that is, it has high

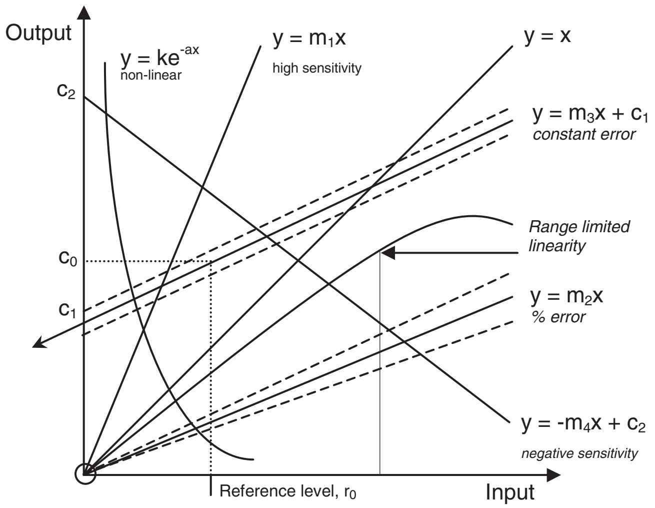


Figure 10.2 Sensor characteristics

sensitivity. The output could be fed directly into the analogue input of the MCU. The line also goes through the origin, meaning no offset adjustment is required – a linear pot would give this result. If the sensor has low sensitivity ($y = m_2x$), an amplifier may be needed to bring the output up to the required level.

OFFSET

Unfortunately, many sensors have considerable offset in their output. This means, that over range for which they are useful, the lowest output has a large positive constant added ($y = m_3x + c$). This has to be subtracted in the amplifier interface to bring the output back into the required range, where maximum resolution can be obtained. The same can be achieved in software, but this is likely to result in a loss of resolution. Temperature sensors tend to behave in this way, as their characteristic often has its origin at absolute zero (−273°C). The sensor may have offset and negative sensitivity, such as the silicon diode temperature characteristic ($y = -m_4x + c_2$). In this case, an inverting amplifier with offset is needed.

LINEARITY

The ideal characteristic is a perfect straight line, so that the output is exactly proportional to the input. This linearity then has to be maintained through the signal conditioning and conversion processes. Metal temperature sensors tend to deviate from linearity at higher temperatures, as their melting point is approached, which limits the useful range. The deviation from linearity is usually expressed as a maximum percentage error over the specified range, but care must be taken to establish whether this is a constant over the range, or a proportion of the output level. These two cases are illustrated by the dotted lines in Figure 10.2, indicating the possible error due to non-linearity and other factors.

REFERENCE LEVEL

If the sensitivity is specified, we still need to know a pair of reference values to place the characteristic. In a temperature sensing resistor (TSR), this may be given as the reference resistance at 25°C (e.g. 1 kΩ). The sensitivity may then be quoted as the resistance ratio – the proportional change over 100°C. For a TSR, this is typically 1.37. This means that at 125°C, the resistance of the 1 kΩ sensor will be 1.37 kΩ.

TRANSFER FUNCTION

Linear sensors are easier to interface for absolute measurement purposes, but some that are non-linear may have other advantages. The thermistor, for example, has a negative exponential characteristic, but it has high sensitivity, so is often used to detect whether a temperature is outside an acceptable range. If the sensor is to be used for measurement, the transfer function must be known precisely in order to design the interface to produce the correct output.

ERROR

Many factors may contribute to sensor error: limitations in linearity, accuracy, resolution, stability and so on. Accuracy is evaluated by comparison with a standard. A temperature of 25°C is only meaningful if Celsius is an agreed scale, in this case based on the freezing and boiling points of water. Resolution is the degree of precision in the measurement: 25.00°C (+/−0.005) is a more accurate measurement that 25°C (+/−0.5). However, this precision must be justified by the overall precision of the measurement system. Poor stability may appear as drift, a change in the sensor output over time. This may be caused by short-term heating effects when the circuit is first switched on, or the sensor performance may deteriorate over the long term, and the measurement become inaccurate. Recalibration of accurate measurement systems is often required at specified intervals, by comparing the output with one that is known to be correct. Interdependence in the sensor may also be significant; for example, the output of a humidity sensor may change with temperature, so this incidental variable must be controlled so that the required output is not affected.

Sensor Types

There is an enormous range of specialist sensors developed for specific applications in the engineering field. Some of the more commonly used sensors will be outlined here. Table 10.1 shows some basic position sensing devices, Table 10.2 different temperature sensors and Table 10.3 light, humidity and strain measurement techniques.

Position

POTENTIOMETER

A potentiometer can be used as a simple position sensor. The voltage output represents the angular setting of the shaft. It has limited range (about 300°) and is subject to noise and unreliability due to wear between the wiper contact and the track. There are therefore a range of more reliable position transducers, which tend to be more expensive.

LVDT

A linear variable differential transformer (LVDT) uses electromagnetic coils to detect the position of a mild steel rod which forms a mobile core. The input coils are driven by an AC signal, and the rod position controls the amount of flux linked to the output coil, giving a variable peak–to-peak output. It needs a high-frequency AC-supply, and is relatively complex to construct, but reliable and accurate.

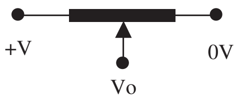
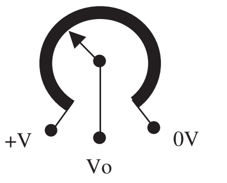
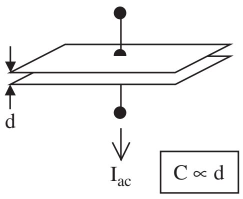
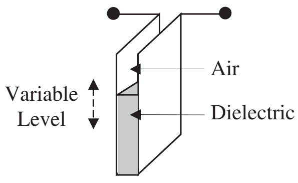
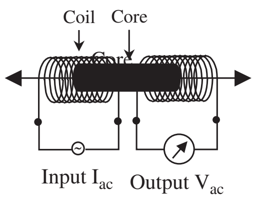
Transducer	Description	Applications	Evaluation
	**Linear potentiometer** Resistive track with adjustable wiper position. DC supply across track gives a variable voltage at the wiper representing absolute linear position.	*Linear position sensing* Faders and multi-turn, pre-sets, medium-scale linear displacement.	Physical wear causes unreliability, but cheap and simple.
	**Rotary potentiometer** Rotary version senses absolute shaft position as voltage or resistance (connect one end an wiper together to form two variable resistance). Log scaling also available.	*Rotary position sensing* Manual pots and pre-sets, any shaft with a range of movement less than 300 degrees. May be used with float for liquid level sensing.	Physical wear causes unreliability, but cheap and simple. Wire wound are more robust, but may have limited resolution.
	**Capacitor plate separation** Capacitance is proportional to plate separation (d is normally small) Small change in d gives a large change in C. Requires a high frequency drive signal to detect changes in reactance.	*Linear position sensing* Sensitive transducer for small changes in position. Plate overlap can also varied, although change may be less linear due to edge effects.	No physical contact, so more reliable. Needs more complex drive and interfacing.
	**Capacitor dielectric** Capacitance depends on dielectric material, effectively producing two capacitors in parallel whose values add. Requires a high frequency drive signal to detect changes in reactance.	*Level or position sensing* The dielectric may be any insulating material, liquid or powder. A solid dielectric can detect linear motion as its position is varied.	No physical contact, so more reliable Needs more complex drive and interfacing involving AC to DC conversion Simple to construct.
	**Magnetic flux** The flux linkage, therefore the output voltage varies with the position of the ferrite core Alternatively, the measured inductance of a single coil will increase as the ferrite is inserted further. A permanent magnet may be used to create a pulse of current as it moves past a coil.	*Position/motion sensing* Magnetic circuits can be used in various ways to detect position, motion, or vibration. Linear voltage differential transformer, electric guitar pick-up, rev counter (magnet on shaft +stationary coil). No physical contact required.	Versatile sensor, pulse detector is simple, but flux linkage types may need more complex drive and detector Involving AC to DC conversion.

Table 10.1 Position sensors

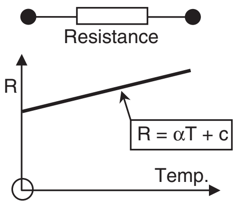
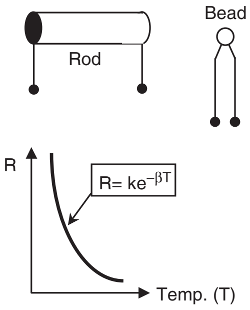
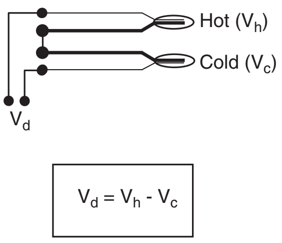
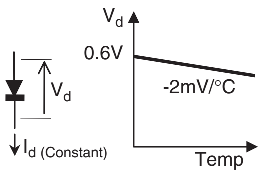
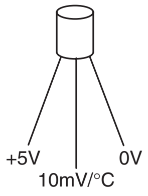
Transducer	Description	Applications	Evaluation
	**Metal resistance temperature sensor** A metal film or solid sensor has the linear characteristic shown (within limits). The offset must be compensated in the amplifier interface. The sensitivity is typically of the order of 4 Ω/°C.	*Temperature measurement* Measurement over the range −50°C to 600°C. The Self-heating may be significant, as reasonable current is needed to reduce noise.	Metal resistance sensors operate over a wide range of temperatures, but may suffer from non-linearity at outside a limited range Sensitivity low, but inexpensive and large range.
	**Thermistor** The thermistor is a solid semiconductor whose resistance falls rapidly with temperature increase. Following a negative exponential curve. The rod is large and design for high current use, while the bead is small and responds rapidly to temperature change over the range above room temperature.	*High temperature sensing* It is typically used in applications such as detecting overheating in system components such as transformers and motors, triggering load shedding or shutdown, up to about 150°C.	The main advantage is high sensitivity, that is, a large change in resistance over a relatively small temperature range. However, it is non-linear, making it difficult to obtain an absolute temper-ature measurement. Therefore, most useful for limit sensing.
	**Thermocouple** This is based on the junction of two dissimilar metals, e.g. iron and copper, generating a small voltage, as in a battery. The large offset voltage from each junction is cancelled out by connecting the measuring junction (hot) and another (cold) thermocouple in opposite polarity. Only the voltage difference due to the temperature difference then appears at the terminals.	*High temperature measurement* As the sensor is all metal, high temperatures can be measured. An interface with a high gain (instrumentation) amplifier is needed. The interface is usually provided in the form of a self-contained controller, with cold junction temperature control and curve compensation.	The interface is complex, requiring cold junction temperature control and a high-gain amplifier. This is worthwhile because the output is accurate over a wide range of temperatures.
 V_d 0.6V V_d -2mV/°C $I_{d\ (Constant)}$ Temp	**Silicon diode** The volt drop across a forward biased silicon diode p–n junction depends on temperature, dropping by about 2 mV/°C. A constant current is needed, as the volt drop also depends on this.	*Temperature sensing* A simple signal diode can be used An interface amplifier will be needed giving a gain of about −10 (inverting), with offset adjust. In addition, a constant current source should used to supply the diode.	This can be used as a cheap and simple temperature sensor. Probably best used for level detection, but is surprisingly accurate if used in a carefully designed circuit.
	**Integrated Temp sensor** This is based on silicon junction temperature sensing. An amplifier is built in, giving a calibrated output of typically 10 mV/°C, over the range of −50 to + 150°C. The accuracy is around +/− 0.5°C.	*Temperature measurement* General purpose low temperature sensing with reasonable accuracy. Can be operated from +5 V, so is easy to intergrate into digital systems.	This is a versatile sensor, and the first choice for a low cost, low temperature MCU-based system. It is easy to interface, does not need calibrating and is inexpensive. Response may be slow due to size.

Table 10.2 Temperature sensors

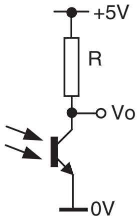
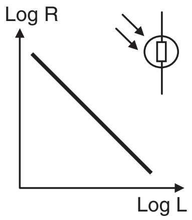
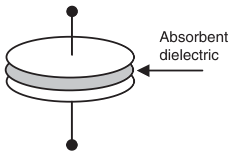
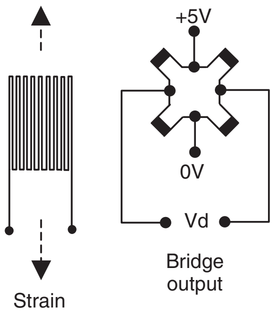
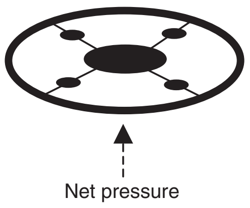
Transducer	Description	Applications	Evaluation
	**Phototransistor** The phototransistor has no base connection, but it is exposed to light by transparent encapsulation. The base current is generated by light energy absorbed by the charge carriers. With a load resistance, the collector voltage varies with base current in the usual way.	*Light sensing* The transistor provides inherent gain (about 100) making the device quite sensitive. It is incorporated in opto -couplers and detectors, which usually use infra-red light from an LED, which reduces interference from visible light sources.	A high sensitivity detector, but difficult to obtain a calibrated output. It is therefore more frequently used in digital systems for isolation and position/speed measurement using a counter.
	**Light-dependent resistor** The LDR uses a CdS (cadmium disulphide) cell which is sensitive to visible light over a wide range from dark to bright sunlight. If the light input (lux) and resistance are plotted on decade scales, a straight line is obtained.	*Light measurement* The LDR is the standard cell used in light meters and cameras, since photographic exposure is also calculated on a log scale A coarse level voltage can be obtained with a simple series resistance e.g. dark, overcast, sun.	The CdS cell provides an accurate output over a wide range, but interfacing for a calibrated output via an MCU requires conversion of the log scale, either via an accurate log amplifier or in software.
	**Humidity** A capacitor with an absorbent dielectric can vary in capacitance value depending on the humidity of the surrounding air.	*Humidity measurement* Environmental monitoring is the general area of applications, either for weather recording, product testing or production control.	Plain sensors requiring an HF AC signal to drive the detection system available, or devices with integrated signal conditioning are simpler to interface.
	**Strain gauge** This is simply a folded conductor mounted on a flexible sheet whose resistance increases as it is stretched. It is frequently used in groups of four where the pairs on opposite sides of the bridge are mounted on the same side of a component under extension, and the other pair on the opposite side which is under compression, so that the differential voltage is maximised	*Stress, strain, position measurement* Typically used to measure the deformation in a mechanical component under load (e.g. crane jib) for safety monitoring purposes. Can also be used to measure motion at the end of fixed beam to measure force applied or weight A high gain, differential (instrumentation) amplifier is needed.	Relatively simple and reliable method of monitoring small mechanical defor-mations. The high gain amplifier is susceptible to noise and interference, and may need careful circuit design to obtain a stable output.
	**Pressure** If a set of strain gauges are mounted on both sides of a diaphragm as shown, they will respond to deformation as a result of a differential pressure. The output voltages from each pair can be added to give a measurement.	*Differential pressure measurement* For measurement relative to atmosphere one side of the gauge will be exposed to atmosphere, the other to higher-pressure air or gas. If a vacuum is used on one side, absolute pressure may be gauged.	Piezoresistive sensors, accurately trimmed during manufacture, and integrated amplifier provide accurate output over selected ranges.

Table 10.3 Other sensors

CAPACITOR

The capacitor principle provides opportunities to measure distance and level. If considered as a pair of flat plates, separated by an air gap, a small change in the gap will give a large change in the capacitance, since they are inversely proportional; if the gap is doubled, the capacitance is halved. If an insulator is partially inserted, the capacitance also changes. This can make a simple but effective level sensor for insulating materials such as oil, powder and granules. A pair of vertical plates is all that is required. However, actually measuring resulting small changes in capacitance is not so straightforward. A high-frequency sensing signal may need to be converted into clean direct voltage for input to a digital controller.

ULTRASONIC

Ultrasonic ranging is another technique for distance measurement. The speed of sound travelling over a few metres and reflecting from a solid object gives the kind of delay, in milliseconds, which is suitable for measurement by a hardware timer in a microcontroller. A short burst of high-frequency sound (e.g. 40 kHz) is transmitted, and should be finished by the time the reflection returns, avoiding the signals being confused by the receiver.

Speed

DIGITAL

The speed or position of a DC motor cannot be controlled accurately without feedback. Digital feedback from the incremental encoder described above is the most common method in processor systems, since the output from the opto-detector is easily converted into a TTL signal. The position relative to a known start position is calculated by counting the encoder pulses, and the speed can then readily be determined from the pulse frequency. This can be used to control the dynamic behaviour of the motor, by accelerating and decelerating to provide optimum speed, accuracy and output power.

ANALOGUE

For analogue feedback of speed, a tachogenerator can be used; this is essentially a permanent magnet DC motor run as a generator. An output voltage is generated which is proportional to the speed of rotation. The voltage induced in the armature is proportional to the velocity at which the windings cut across the field. This is illustrated by the diagrams of the DC motor in Chapter 8. If the tacho is attached to the output shaft of a motor controlled using PWM, the tacho voltage can be converted by the MCU and used to modify the PWM output to the motor, giving closed loop speed control. Alternatively, an incremental encoder can be used, and the motor output controlled such that a set input frequency is obtained from the encoder.

Temperature

Temperature is another commonly required measurement, and there is variety of temperature sensors available for different applications and temperature ranges. If measurement or control is needed in the range of around room temperature, an integrated sensor and amplifier such as the LM35 is a versatile device which is easy to interface. It produces a calibrated output of 10 mV/°C, starting at 0°C with an output of 0 mV, that is, no offset. This can be fed directly into the PIC analogue input if the full range of −50°C to +150°C is used. This will give a sensor output range of 2.00 V, or 0.00 V – 1.00 V over the range 0–100°C. For smaller ranges, an amplifier might be advisable, to make full use of the resolution of the ADC input. For example, to measure 0–50°C:

$$\textit{Temp range} = 50°C$$

$$\textit{Input range used} = 0-2.56\ V\ (8\text{-}\textit{bit conversion}, V_{REF} = 2.56\ V)$$

$$\textit{Let maximum} = 2.56 \times 20 = 51.2°C$$

$$\textit{Then conversion factor} = 2.56/5.12 = 50\ mV/°C$$

$$\textit{Output of sensor} = 10\ mV/°C$$

$$\textit{Gain of amplifier required} = 50\ mV/10\ mV = 5.0$$

A non-inverting amplifier with a gain of 5 will be included in the circuit (see Chapter 7). Note that if a single supply amplifier is used, the sensor will only go down to about +2°C.

DIODE

The forward volt drop of a silicon diode junction is usually estimated as 0.6 V. However, this depends on the junction temperature; the voltage falls by 2 mV/°C as the temperature rises, as the charge carriers gain thermal energy, and need less electrical energy to cross the junction. The temperature sensitivity is quite consistent, so the simple signal diode can be used as a cheap and cheerful alternative to the specialist sensors, especially if a simple high/low operation only is needed. A constant current source is advisable, since the forward volt drop also depends on the current.

METALS

Metals have a reasonably linear temperature coefficient of resistance over limited ranges. Metal film resistors are produced which operate up to about 150°C, with platinum sensors working up to 600°C. The temperature coefficient is typically around 3–4000 ppm (parts per million), which is equivalent to 0.3%/°C. If the resistance at the reference temperature is, say, 1 kΩ, the resistance change over 100°C would be 300–400 Ω. A constant current is needed to convert the resistance change into a linear voltage change. If a 1 kΩ temperature-sensing resistor is supplied with a constant 1 mA, the voltage at the reference temperature, 25°C, would be 1.00 V, and the change at 125°C would be 370 mV, taking it to 1.37 V. An accuracy of around 3% may be expected.

THERMOCOUPLE

Higher temperatures may be measured using a thermocouple. This is simply a junction of two dissimilar metals, which produces a battery effect, producing a small EMF. The voltage is proportional to temperature, but has a large offset, since it depends on absolute temperature. This is compensated for by a cold junction, connected in series, with the opposite polarity, and maintained at a known lower temperature (say 0°C). The difference of voltage is then due to the temperature difference between the cold and hot junctions.

THERMISTOR

Thermistors are made from a single piece of semiconductor material, where the charge carrier mobility, therefore the resistance, depends on temperature. The response is exponential, giving a relatively large change for a small change in temperature, and a particularly high sensitivity. Unfortunately, it is non-linear, so is difficult to convert for precise measurement purposes. The thermistor therefore tends to be used as a safety sensor, to detect if a component such as a motor or transformer is overheating. The bead type could be used with a comparator to provide warning of overheating in a microcontroller output load.

Strain

The strain gauge is simple in principle. A temperature-stable alloy conductor is folded onto a flexible substrate which lengthens when the gauge is stretched (strained). The resistance increases as the conductor becomes longer and thinner. This can be used to measure small changes in the shape of mechanical components, and hence the forces exerted upon them. They are used to measure the behaviour of, for example, bridges and cranes, under load, often for safety purposes. The strain gauge can measure displacement by the same means.

The change in the resistance is rather small, maybe less than 1%. This sits on top of an unstrained resistance of typically 120 Ω. To detect the change, while eliminating the fixed resistance, four gauges are connected in a bridge arrangement and a differential voltage is measured. The gauges are fixed to opposite sides of the mechanical component, such that opposing pairs are in compression and tension. This provides maximum differential voltage for a given strain. All the gauges are subject to the same temperature, eliminating this incidental effect on the metal conductors. A constant voltage is supplied through the bridge, and the difference voltage fed to a high gain, high input impedance amplifier. The instrumentation amplifier described in Chapter 7 is a good choice. Care must be taken in arranging the input connections, as the gauges will be highly susceptible to interference. The amplifier should be placed as near as possible to the gauges, and connected with screened leads, and plenty of signal decoupling. The output must then be scaled to suit the MCU ADC input.

Pressure can be measured using an array of strain gauges attached to a diaphragm, which is subjected to the differential pressure, and the displacement measured. Measurement with respect to atmosphere is more straightforward, with absolute pressure requiring a controlled reference. Laser-trimmed piezoresistive gauge elements are used in low-cost miniature pressure sensors.

Humidity

There are various methods of measuring humidity, which is the proportion of water vapour in air, quoted as a percentage. The electrical properties of an absorbent material change with humidity, and the variation in conductivity or capacitance, can be measured. Low-cost sensors tend to give a small variation in capacitance, measured in a few picofarads, so a high-frequency activation signal and sensitive amplifier are needed.

Light

There are numerous sensors for measuring light intensity: phototransistor, photodiode, light-dependent resistor (LDR, or cadmium disulphide cell), photovoltaic cell and so on. The phototransistor is commonly used in digital applications, in opto-isolators, proximity detectors, wireless data links and slotted wheel detectors. It has built-in gain, so is more sensitive than the photodiode. Infra-red (IR) light tends to be used to minimise interference from visible light sources, such as fluorescent lights, which nevertheless, can still be a problem. The LDR is more likely to be used for visible light, as its response is linear (when plotted log R vs. log L) over a wide range, and it has a high sensitivity in the visible frequencies. The CdS cell is widely used in photographic light measurement, for these reasons. Conversion into a linear scale is difficult, because of the wide range of light intensity levels between dark and sunlight.

Amplifier Design

In order to design the interface between a sensor and the MCU, we need to specify the performance of a linear amplifier which will translate the output of the sensor into a suitable input for the MCU, which is in the right range for the amplifier to handle and allows a convenient conversion factor to be used in the ADC. For example, our standard temperature sensor with built-in signal conditioning produces an output of 10 mV per degree C, with an output range of -55 to $+150$°C. Let us assume it is to be interfaced to a PIC MCU to measure between 10 and 35°C.

Gain

At 10°C, the input will be $10 \times 10 = 100$ mV. At 35°C, it will be $35 \times 10 = 350$ mV. The gain is calculated as the required change in the output divided by the range of the input. Now if we are using a single supply op-amp package, such as the LM324, the output range is strictly limited. The output only goes down to about 50 mV and up to about 3.5 V. So if we assume an output swing of 2.5 V is available, the gain required $= 2.5/(0.35–0.1) = 10$.

Offset

As in this case, sensors often have a positive offset, so the output range has to be shifted down. With a gain of 10, the output at the lowest temperature will be 100 mV $\times 10 = 1.00$ V, and at the highest 350 mV $\times 10 = 3.50$ V. This can be shifted down by 1.00 V, so that the output range will be 0–2.50 V. This also allows us to use a reference voltage of 2.56 V, just above the required maximum, which gives a convenient 8-bit conversion factor (10 mV/bit). Including 0 V in the output means that we will lose the lowest degree or two from the range. If this is not acceptable, the offset can be adjusted to suit, so that the output ranges from 0.5 to 3.0 V. In this case, the negative reference voltage for the ADC should be modified to 0.5 V, and the positive reference to 3.06 V. If necessary, corrections can also be made in software.

Frequency Response

Since we are measuring direct voltages, it is sensible to restrict the frequency response of the amplifier to low frequencies, as any instability in operation tends to produce high-frequency oscillation and incorrect DC readings. A moderate value of capacitance across the feedback network will reduce the frequency response by reducing the feedback impedance at high frequency. However, too high a value will slow down the response time, so this needs to be considered if transient behaviour is a significant consideration.

Calibration

A circuit which meets these requirements is shown in Figure 10.3. It is based on a non-inverting amplifier configuration, with the offset added as a positive voltage at the reference input. The temperature sensor input is represented by three selected levels, corresponding to the minimum (100 mV), mid-range (225 mV) and maximum (350 mV) output voltage. The gain is notionally 10, but is adjusted by the reference input resistance. The offset input is notionally 100 mV, but is also adjustable.

Calibration of the amplifier normally consists of adjusting the gain and the offset at the minimum and maximum output levels, assuming that it is linear in between these values. However, there is a problem with the single supply case – the minimum output is not reached because the amplifier cannot reach the supply rail voltage (0.000 V). In the simulation, the minimum reached is about 80 mV. The output gives a resolution of 100 mV/°C, so readings from 10 to 11°C will be affected, leaving an operating range of 11–35°C. This will be accepted. If unacceptable, the operating range can be modified by adjusting the offset input voltage and recalibrating.

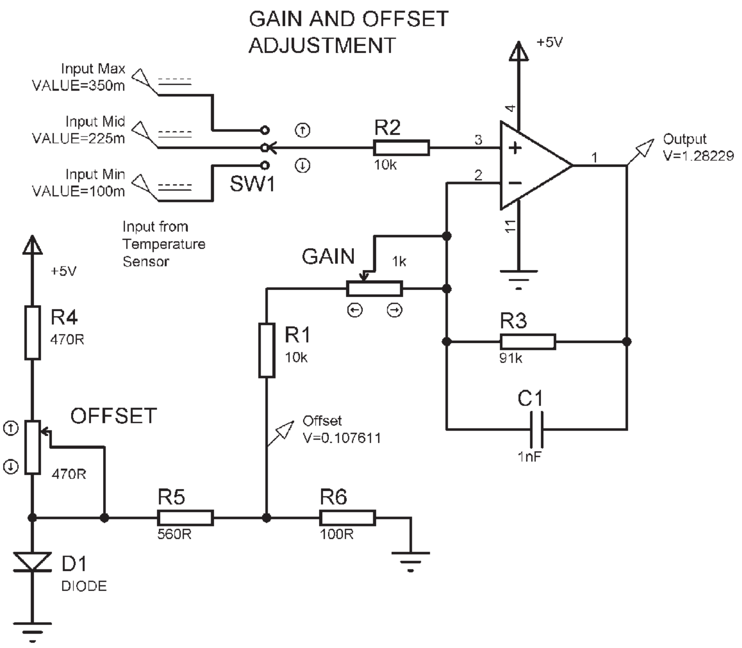


Figure 10.3 Gain and offset adjustment

Therefore, in this circuit, we will calibrate the amplifier by adjusting the offset to approximately the correct value at the mid-output level (1.280 V), and then adjusting the gain to give the right output at the maximum level (2.500). These steps are then repeated until the reading is correct at the mid and max values. This is usually necessary because the gain and offset interact, that is, adjusting one affects the other. In practice, multi-turn pre-set pots (typically 10 turns) are often used to give greater sensitivity or range to the adjustment.

In this circuit, a relatively small offset voltage is required, and it is obtained by taking the forward volt drop of a standard signal diode (about 0.7 V) and dividing it down to around 100 mV. A fine adjustment of this is then obtained by 'squeezing' the diode voltage via its current supply. A diode current of about 10 mA is used, dissipating about $10 \times 0.7 = 7$ mW in the diode. It is possible that self-heating in the diode could cause some temperature instability. If necessary, a more stable reference circuit design should be used, or, at the very least, the circuit temperature should be allowed to reach a steady state before the calibration procedure is attempted.

The accuracy of the sensor is quoted as $+/-0.5$°C, and the interface needs to match this. The output changes by 100 mV/°C, so 0.5°C $=$ 50 mV. At mid-range, 22.5°C, the output is 1.28 V, and the allowed range is 1.23–1.33 V. The accuracy of the amplifier should in fact be better than $+/-10$ mV, and this is more than adequate. The ADC will be working at 2.56 V/256 $=$ 10 mV/bit, the same resolution.

Weather Station

To illustrate sensor interfacing, a weather station measuring temperature, light, pressure and humidity will be designed. These variables will be sampled at an interval of 5 minutes (12/hour) and data stored for a period of up to 10 days. The specification is detailed in Table 10.4.

The system will be based on a general purpose module using the PIC 16F877, an LCD and a serial memory, details of which will be provided in the next chapter. It has a 12-button keypad, 16×2 line backlit display and a 16 kb serial memory (Figure 10.4). Each variable will occupy eight characters on screen in run mode. If sampled at 8-bit resolution, one sample for each sensor $=$ 1 byte of data. Over 10 days, the system will store $10 \times 24 \times 12 \times 4 =$ 11520 bytes of data. The user should be able to reset, run and read back data manually. Optionally, an RS232 link to host computer will allow the data to be downloaded for further analysis and long-term storage.

Input	Range	Display (Max 8 Chars)
Temperature	−25°C to +75°C	`Temp:XX (7)`
Light	Dark, dusk, dull, cloud, sun	`XXXXX (5)`
Pressure	850–1100 millibar	`mbar:XXXX (8)`
Humidity	0–100% Relative humidity	`RH:XXX% (7)`
Precision	8-bits	`<1% @ mid range`
Storage	10 days @ 12 samples/h	`11520 bytes`
User	Run, recall data	`Essential`
interface	Display/set date and time	`Desirable`
Host interface	Data download to database and spreadsheet	`Desirable`

Table 10.4 Weather station specification

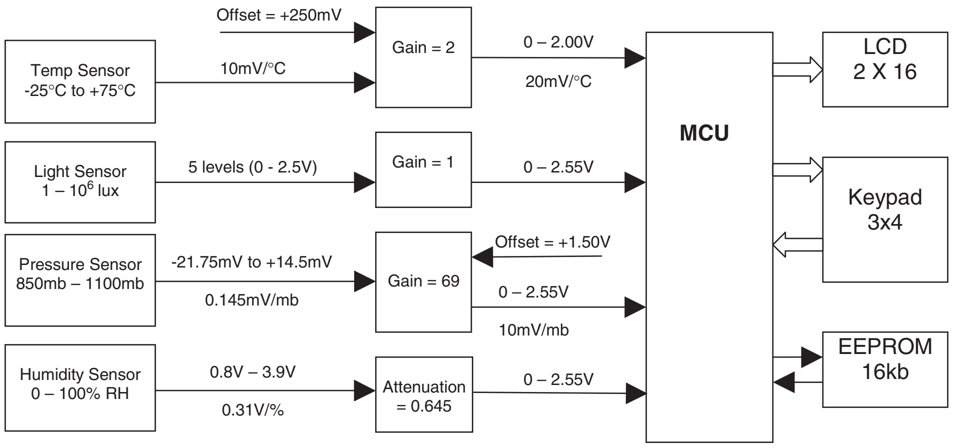
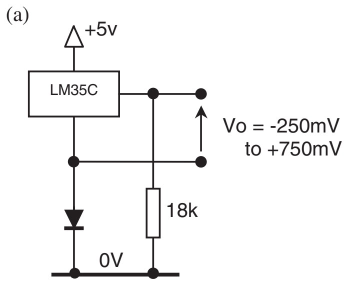


Figure 10.4 Block diagram of weather station

The ADC inputs will be connected to this module via a 10-way ribbon cable, with the analogue interfaces built on a separate board. A sensor was selected for each weather variable, primarily based on the range required, ease of interfacing and low cost. An analogue interface was then developed to provide the gain and offset required for each. Signal filtering was not considered in detail, but the possibility of controlling high-frequency interference and noise always needs to borne in mind. Typically, some low-pass filtering or decoupling may be included in the interface as a pre-caution when conditioning DC signals. This may be in the form of a simple first-order CR network in the input, and an integrating capacitance connected across the feedback resistance in the amplifier stage. The maximum source resistance allowed at the PIC ADC input is 10 kΩ; a low-pass filter with a 1 kΩ series resistance and 100 nF decoupling capacitor will give a cut-off frequency of around 2 kHz.

Temperature Sensor Input

The default choice for this sensor is the LM35 type. The performance is adequate for this application, and it is possible to connect it direct to the ADC input. In this case, the LM35C is used which allows negative temperatures to be measured. To provide these as a positive voltage with single supply, the sensor negative supply is connected to ground via a diode to lift the zero degrees output to around 0.7 V. This allows the actual output voltage to go below the zero level while remaining positive with respect to supply 0 V (Figure 10.5).



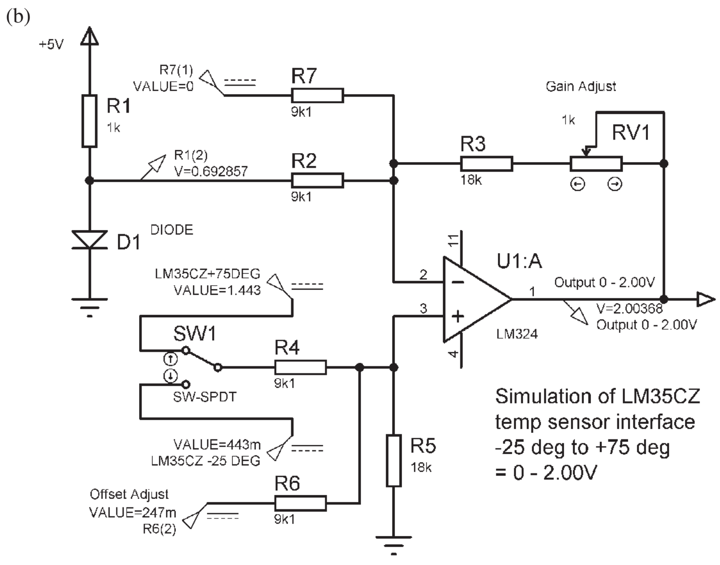


Figure 10.5 Temperature sensor interface: (a) sensor connections; (b) interface simulation

The interface uses a differential amplifier with two positive and two negative inputs, based on the universal amplifier described previously. The number of positive and negative inputs must always be equal to conform to this model. A reference diode provides a negative input to balance the positive offset on the sensor input. The input from the sensor is simulated by a switch which provides the maximum and minimum voltage which would be seen at the input. A further positive input provides the offset at the amplifier output to give 0.00–2.00V corresponding to the input range of 100°C. The overall sensitivity is 20 mV/°C. A further negative input of 0 V is needed to match the offset input. The preset feedback resistance is adjusted for a gain of 2.00.

The circuit provides the following arithmetic sums at each end of the range (−25°C and +75°C).

$$2.000 \text{ x } (443 + 247 - 693 - 0) = -6 \text{ mV} \qquad @ -25°C$$

$$2.000 \text{ x } (1443 + 247 - 693 - 0) = -6 \text{ mV} \qquad @ +75°C$$

The 6 mV at the output (3 mV at the input) is the offset of the amplifier, which is allowed for in the external offset adjust (250–3 = 247 mV). Notice that in the simulation there is a residual offset at 2.000 V output, but this is less than 5 mV, which is acceptable (<0.5% at full scale). The reference diode current may need to be adjusted in the real hardware by changing its 1k current feed resistor to a value that gives the same current as that provided by the sensor to its offset diode.

When converted with a 2.56 V reference, the temperature range will be represented by binary numbers equivalent to 0–200, with 50 representing 0°C. This scaling offset can be corrected in software, prior to display. Remember that the single supply amplifier output will not go all the way to zero, so the actual range starts at about −23°C. In normal circumstances, this is acceptable, as this temperature is rarely experienced in temperate climates.

Light Sensor Input

The light sensor input is designed around the standard NORP12 cadmium disulphide-LDR. It has a spectral response which is similar to the human eye, and is sensitive to a wide range of values of light intensity and is relatively easy to interface. Its resistance is inversely proportional to light intensity, as shown in Figure 10.6 (b).

The light level is divided into five decades, from <1 lux (dark) to >10000 lux (direct sun), so the output levels are similarly divided. When the LDR is connected in series with a 4k7 resistor across the 5 V supply, a set of voltages is obtained which vary from 2.5 V (high resistance, dark) to 0 V (low resistance, light). In the simulated interface, these values are represented by switched parallel resistances, with the sensor voltage simply buffered by a unity gain amplifier (Figure 10.6 (c)). The software can then compare the input with any chosen set of limits, which correspond to the required light levels. The actual reading will be stored for further analysis.

Pressure Sensor Interface

Measurement of barometric pressure is not particularly straightforward, since pressure measurement in usually made relative to atmosphere (1 bar = 1000 mb). For example, it is straightforward to measure a low-pressure air supply for a pneumatic system operating at 5 bar. One side of the gauge diaphragm is

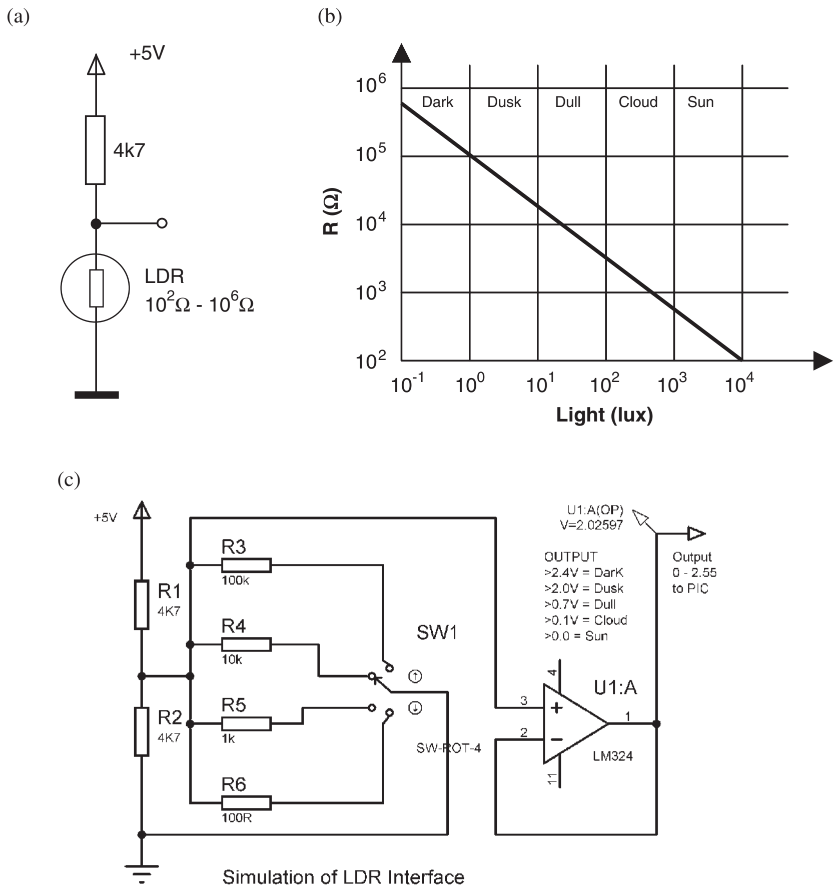


Figure 10.6 Light sensor interface: (a) sensor connection; (b) LDR characteristic; (c) interface simulation

exposed to atmosphere, while the pressurised system is connected to the other side. Small deviations from atmosphere caused by meteorological variation are more difficult to measure accurately.

It is suggested here that one side of the gauge is connected to a closed tube representing 1 atmosphere, while the other is exposed to the varying meteorological pressure. Careful calibration will be required, with temperature compensation for its effect on the fixed volume of air. This temperature measurement is available from sensor input described above.

Low-cost pressure sensors use a strain gauge bridge made up of laser-trimmed piezo-resistive elements in a compact, robust package. A pressure in the range of 850–1106 mbar is proposed (range $= 256$ mbar), allowing an 8-bit conversion at 1 bit per mbar. Standard atmospheric pressure will then occur at a reading of 150.

The gauges investigated are rated in psi (pounds per square inch). 1 psi $= 69$ mbar, so the range required is $256/69 = 3.71$ psi. A gauge is available which measures up to 5 psi with a 10 V supply. If the supply is $+5$ and 0 V, the output will be $+/-2.5$ psi, with a sensitivity of 5 mV/psi and offset of 2.5 V. This is equivalent to $5/69 = 0.0725$ mV/mbar. The range will then be 256×0.0725mV $= 18.56$ mV. The amplifier gain required is therefore 2.56 V/18.56 mV $= 138$.

The output offset at 1000 mbar input will be 1.50 V. The low end will be curtailed by the output of the single supply amp not quite being reaching zero, but as this will be an extreme event, this is acceptable. The input and output voltages are then as follows:

$$\begin{aligned}
\textit{Input}\quad & V_{inzero} = 0 \text{ mV} \\
& V_{inmin} = 0.0725 \times -140 = -10.15 \text{ mV } (<140 \text{ not used}) \\
& V_{inmax} = 0.0725 \times 100 = 7.25 \text{ mV}
\end{aligned}$$

$$\begin{aligned}
\textit{Output}\quad & V_{outzero} = 1.50 \text{ V} \\
& V_{outmin} = 1.50\text{–}1.40 = 0.10 \text{ V} \\
& V_{outmax} = 1.50 + 1.00 = 2.50 \text{ V}
\end{aligned}$$

If the standard instrumentation amplifier is used (Chapter 7), gain $G = 1 + 2R_2/R_1$, where R_1 and R_2 are the values in the input stage.

$$\therefore \quad R_2/R_1 = (G-1)/2 = (138-1)/2 = 68.5$$

$$\text{If } R_1 = 1\text{k}, R_2 = 68\text{k} + 470\text{R}$$

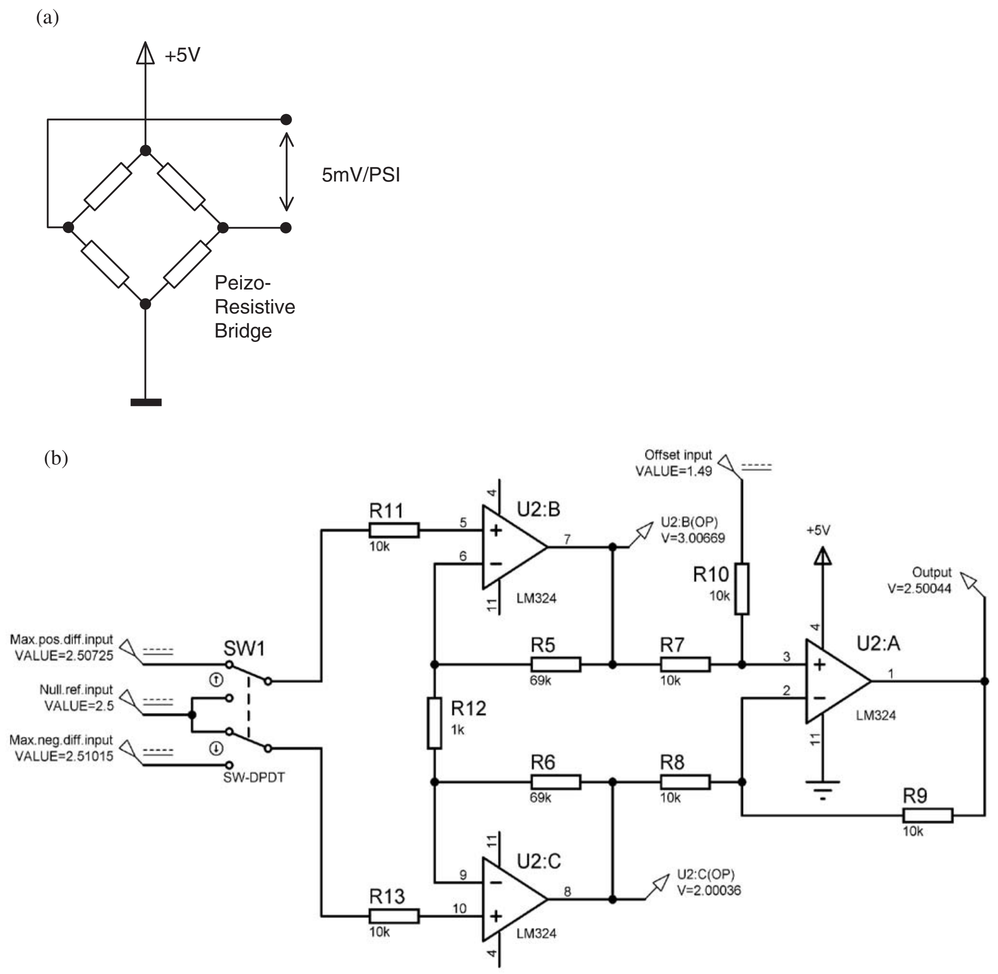


Figure 10.7 Pressure sensor interface: (a) sensor connections; (b) interface simulation

The offset voltage (+1.50 V) will be input at the non-inverting reference input. This can also include some adjustable element to compensate for the amplifier input offset. Figure 10.7 (b) shows the circuit simulation operating with the maximum input.

Humidity Sensor Interface

The humidity sensor selected has integrated signal conditioning so that an output between 0.8 and 3.9 V is produced, representing a change in relative humidity of 0–100%. A simple buffered attenuator is used to shift the signal range for input to the ADC. The output of 0 V from the single supply amplifier cannot be obtained, so the output is shifted up to the range 0.5–2.50 V, giving 20 mV/%. This offset must be removed in software, by subtracting 50_{10} from the 8-bit binary input.

$$\begin{aligned} &\textit{Input range} &&= 3.9-0.8 = 3.1\text{ V} \\ &\textit{Output range} &&= 2.50-0.50 = 2.00 \\ &\therefore \textit{Required gain} &&= 2.00/3.1 = 0.645 \text{ (attenuation)} \\ &\rightarrow \textit{Use unity gain + output attenuator} && \\ &\textit{Output max} &&= 3.9 \times 0.645 = 2.516 \\ &\textit{Output min} &&= 0.8 \times 0.645 = 0.516 \end{aligned}$$

The small residual offset is easier to eliminate in software, by adjusting the offset correction factor, and subtracting 52 instead of 50. This allows preferred values to be used in the attenuator, reducing component cost. The input and output buffering of the attenuator network simply reduces any error due to loading effects. However, the sensor is only specified about 4% accurate normally, so this may not be absolutely necessary. The sensor can be supplied with individual calibration data if a more accurate output is needed. Figure 10.8 shows the simulated interface operating at 100% humidity.

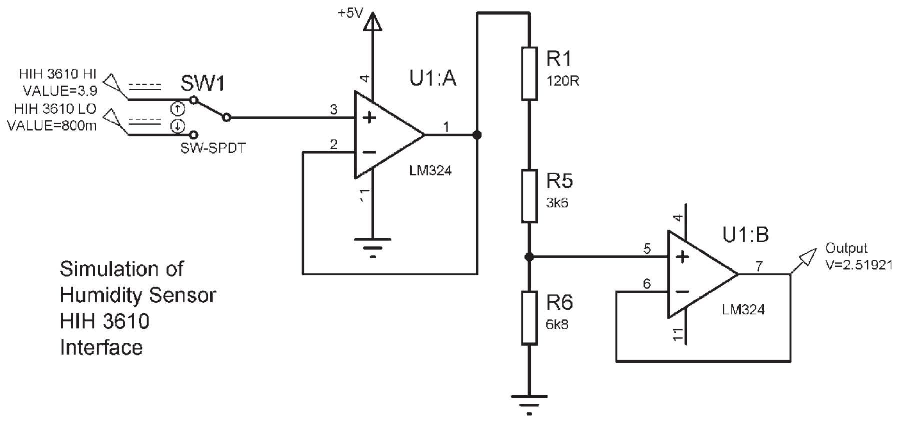


Figure 10.8 Humidity sensor interface

SUMMARY 10

- Digital sensors include switches, opto-detectors and incremental encoders
- Analogue sensors produce a variation in voltage, current or resistance
- Their main characteristics are sensitivity, range, offset, accuracy and error
- Inputs include position, speed, temperature, pressure, light, strain, humidity
- Sensors are resistive, capacitative, inductive, semi-conductor or voltaic
- Interface signal conditioning adjusts gain, offset and frequency response

ASSESSMENT 10 **Total** *(40)*

1 Describe how the reliability of a mechanical switch can be improved. *(3)*

2 Explain the meaning of interpolation in position measurement. *(3)*

3 Define the term sensitivity as applied to a sensor. *(3)*

4 Explain the difference between the terms accuracy and precision. *(3)*

5 State three sensors for measuring temperature, and the materials that each is made from. *(3)*

6 Explain why strain gauges are normally connected as a bridge circuit. *(3)*

7 State the gain required to obtain 50 mV/°C from an LM35 temperature sensor. *(3)*

8 Sketch a typical linear transfer characteristic and use it to illustrate the effect of varying the gain and offset of the output. *(3)*

9 Explain why the instrumentation amplifier configuration is suitable for interfacing a strain gauge bridge. *(3)*

10 From the LDR characteristic shown, state the resistance in kΩ of the LDR at 1.0 lux illumination. *(3)*

11 Describe an analogue and a digital method to measure the angular position of a shaft, and suggest an advantage of each type of sensor. *(5)*

12 Describe an analogue and a digital method to measure the angular speed of a shaft, and suggest an advantage of each type of sensor. *(5)*

ASSIGNMENTS 10

10.1 Rain Gauge Design

Investigate and design a system for measuring rainfall. The cumulative rain for each day should be displayed continuously. At midnight the total should be logged and the gauge should be reset. Do not design the controller itself, but specify its requirements to operate the gauge. Compare alternative sensors for the gauge and identify the advantages and disadvantages of each option.

10.2 Interface Design

The forward voltage drop across a silicon signal diode, used as temperature sensor, falls by 2 mV/°C. The diode current is adjusted so that voltage is 650 mV at 25°C. Design an interface that will produce an output of 0–2.50 V representing diode temperatures of 0–50°C. Test your design in simulation mode and comment on any limitations or deviation of the circuit from ideal performance.

10.3 Sensor Comparison

Obtain the specification for three types of temperature sensor: a metal film temperature-sensing resistor, a thermocouple and thermistor. Construct a chart showing the sensitivity (if linear), range and total possible error at mid-range. Investigate and establish a mathematical representation of the transfer function for each. From the function, predict the sensor output at minimum, maximum and mid-range temperature. Suggest at least one appropriate application for each sensor.

This page intentionally left blank

11

System Design

Now that we have studied a range of system components, we can put them together to form some typical MCU-based systems. A base module will be designed which will be used for a range of different measurement and control applications, a parallel memory expansion scheme outlined, and the range of PIC microcontrollers and other processor families reviewed.

Base System

The base system can be used as a general purpose PIC board, and as the basis for applications using the subsystems and interfaces described in the previous chapters. A PIC 16F877 is provided with a keypad, alphanumeric display and serial flash memory, with an RS232 serial link for connecting a PC host. The additional components are included to run the hardware version, and provide basic interfacing facilities: clock circuit, ICD interface, ADC reference voltage, ADC test input, I/O signal connector, LED indicator and buzzer.

The block diagram in Figure 11.1 shows these features. At this point, it might be useful to review the use of block diagrams in embedded system design:

- The main elements are shown in block form and labelled accordingly
- These are connected by arrow segments indicating the nature of the signal and the principal direction of information flow
- Serial and parallel data is represented by single and block arrows, respectively

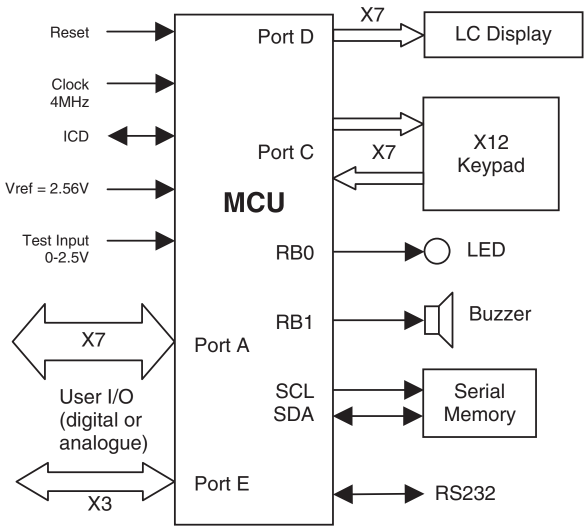


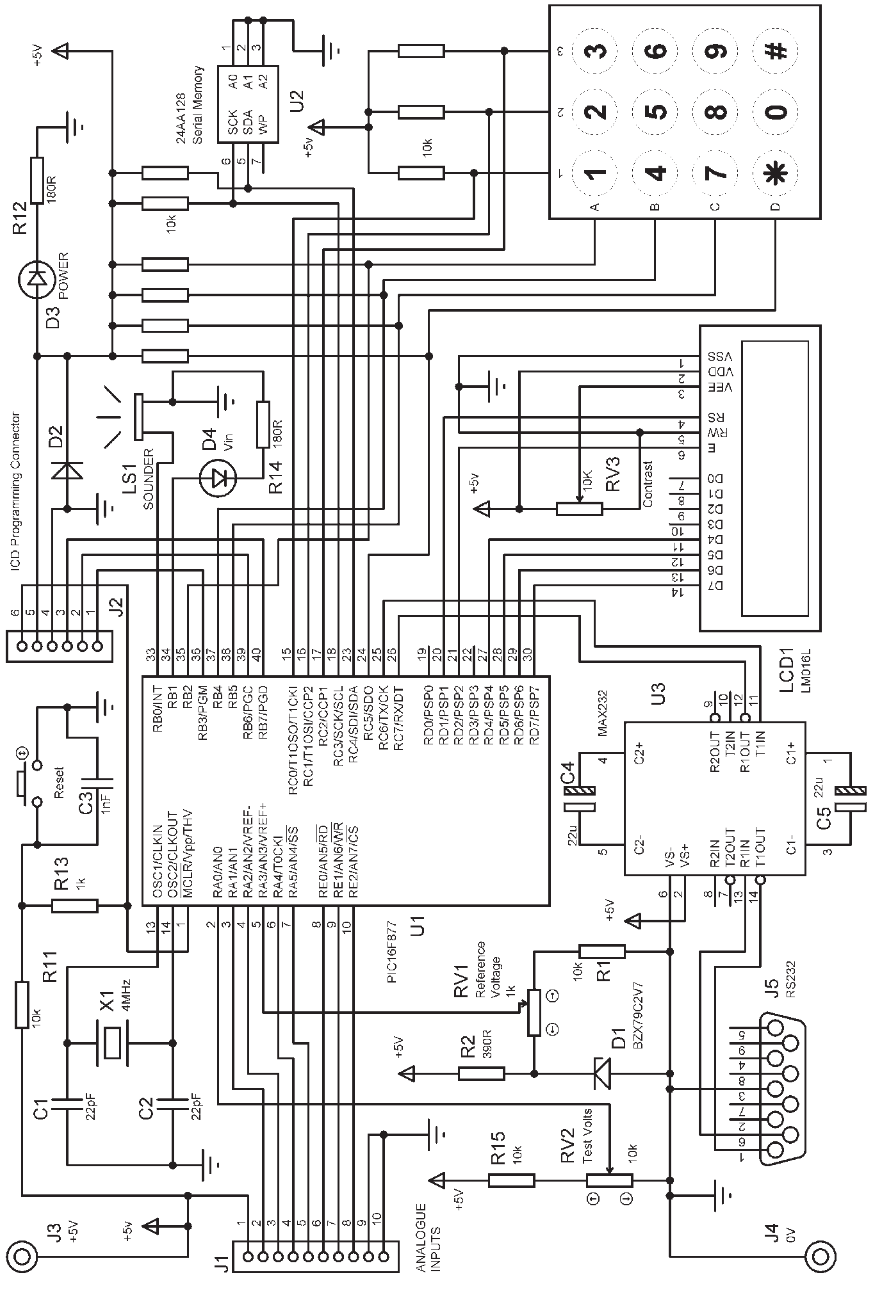
Figure 11.1 Base module block diagram

- If the signal is not digital, it should be labelled accordingly, specifying voltage levels and signal type, with signal diagram if necessary
- The block diagram allows the I/O requirements to be identified, and the most suitable microcontroller selected
- The block diagram is then expanded into circuit schematic

Once the application specification has been converted into a block diagram and a suitable MCU provisionally selected, a circuit schematic can be started. Proteus™ schematic capture provides drawing objects for the whole range of commonly used components. Simulation models are provided with selected devices, and some are interactive on screen to facilitate circuit testing. This is particularly useful for interface components such as the keyboard and LCD in the base board design. Devices are selected from the library of parts; if a listed part does not have a simulation model attached, an equivalent can be selected. Devices can also be created by the user.

Base Board Hardware

The base board schematic is shown in Figure 11.2. The circuit is built around the 16F877, with ports A and E brought out to an in-line connector for the external interface circuits. Port D is allocated to the LCD with Port C interfacing with the serial memory and PC host via serial ports. Port B programming pins are brought out to the ICD connector, and remaining Port B and C pins used for the keypad, an LED indicator and buzzer.



ICD Programming Connector
J2
D2
D3 POWER
R12 180R
+5V
24AA128 Serial Memory
U2
SCK SDA WP
A0 A1 A2
10k
LS1 SOUNDER
D4 Vin
R14 180R
LCD1 LM016L
VSS VDD VEE RS RW E D0 D1 D2 D3 D4 D5 D6 D7
RV3 10K Contrast
U3 MAX232
C4 22u
C5 22u
C2+ C2- C1+ C1- VS- VS+
R2OUT T2IN R1OUT T1IN R2IN T2OUT R1IN T1OUT
U1 PIC16F877
OSC1/CLKIN OSC2/CLKOUT MCLR/Vpp/THV
RA0/AN0 RA1/AN1 RA2/AN2/VREF- RA3/AN3/VREF+ RA4/T0CKI RA5/AN4/SS
RE0/AN5/RD RE1/AN6/WR RE2/AN7/CS
RB0/INT RB1 RB2 RB3/PGM RB4 RB5 RB6/PGC RB7/PGD
RC0/T1OSO/T1CKI RC1/T1OSI/CCP2 RC2/CCP1 RC3/SCK/SCL RC4/SDI/SDA RC5/SDO RC6/TX/CK RC7/RX/DT
RD0/PSP0 RD1/PSP1 RD2/PSP2 RD3/PSP3 RD4/PSP4 RD5/PSP5 RD6/PSP6 RD7/PSP7
Reset
C3 1nF
R13 1k
R11 10k
X1 4MHz
C1 22pF
C2 22pF
J3 +5V
J1 ANALOGUE INPUTS
RV1 Reference Voltage 1k
R1 10k
R2 390R
D1 BZX79C2V7
R15 10k
RV2 Test Volts 10k
J4 0V
J5 RS232

Figure 11.2 Base module schematic

RESET

A manual reset is included, so that programs can be restarted when the board is running independently. If the program appears to be malfunctioning, a hardware reset is usually the first remedy.

CLOCK

A standard crystal circuit is used, running at 4 MHz. This gives a 1 μs instruction execution time, which is convenient for analysing program timing. The crystal needs to be physically near the MCU, so that additional track capacitance does not prevent the crystal from oscillating, or affect the resonant frequency. The maximum frequency possible is 20 MHz, giving a 200 ns instruction cycle, or 5 million instruction cycles per second. For maximum speed, the crystal must be replaced with a HS (High Speed) type. Note, however, that the power dissipation increases with frequency, so the supply needs to be adequate. In addition, signals at higher frequencies tend to radiate more easily, so clock interference affecting other signals is more likely. If the MCU is used with other ICs in circuit, it is standard practice to decouple the supply near to each chip with a small ceramic capacitor (e.g. 10 pF). This helps to prevent the clock signal getting into the IC on the supply, and causing a malfunction. The longer the board tracks are, the more likely this type of problem is to occur.

ICD

The ICD connections are brought out to a connector which will match the connector on the ICD programmer module, which is connected in turn to the host PC, to provide program downloading and final debugging in hardware. At this stage, any final timing or interfacing issues which only appear in the real hardware can be resolved. The PIC development system must be used for program downloading, so the program, which has been tested by simulation must be transferred to MPLAB. The debugging tools in MPLAB, which are more extensive than in Proteus, may sometimes be called into use. However, the assembler (MPASM) is the same, so the object code (PROGNAME.HEX) will be the same when reassembled in MPLAB.

INPUT/OUTPUT

Ports A and E are attached to a connector for external circuits, allowing analogue input or digital I/O on seven pins. RA3 is used for an external reference voltage which is shown as 2.56 V, assuming 8-bit conversion will be used. If 10-bit conversion is required, a 4.096 V reference can be substituted, as shown in Chapter 7. A test voltage is connected to AN0, and when the test program is running, the value will be displayed on the LCD (0.00–2.50 V). An LED and buzzer are also provided on spare pins at Port B to provide some status indications. In the test program, the LED indicates if the input voltage is over 50%, and the buzzer provides some audible feedback when a button is pressed on the keypad.

KEYPAD

The keypad interface is detailed in Chapter 4. The 12-button keypad is connected in the usual way to Port C. In the test program, the outputs to the rows (ABCD) and the inputs from the columns (123) are initially all high. The rows are taken low in turn and the columns tested for 0. When a button is detected, the corresponding ASCII code is returned and processed.

LCD

The operation of the 16×2 character LCD is detailed in Chapter 4. It is connected in 4-bit mode, that is, ASCII codes are fed to it in two stages, high nibble then low nibble; for this reason, the data inputs are connected to the high bits of Port D. The low bits provide the control lines RS (Register Select) and E (Enable). The RW (Read/Write) line is connected low for writing only − it is not necessary to use the LCD handshaking, which would require a change in data direction, and make the software more complex. In outline, the LCD operates as follows: a control byte is presented at the data inputs and RS set low to select command mode. A pulse on E then latches the high nibble, with the low nibble following in the same way. The command mode is used for operations such as resetting the cursor position to the first character, or clearing the display. The ASCII character codes are loaded by taking RS high for data mode, and latching the code in two stages as above. The display can be initialised to auto-increment the cursor to the next space on the same line when a character is added to the line, but needs a specific command to go to the second line.

SERIAL MEMORY

The memory chip locations are accessed via the I^{2}C serial interface (RC3, RC4), as detailed in Chapter 9. Data is transferred in 1 byte packets on SDA, preceded by addressing bytes to select the chip and the location. SCK provides a clock pulse with each bit to latch it into the destination device. The hardware address pins are connected low to assign the default address 0. WP (Write Protect) allows the chip write to be disabled to prevent accidental overwriting of important data, but it is not connected here.

PC INTERFACE

The RS232 port is connected to a 9-pin D-type connector via a standard MAX232 chip. This converts the signal level between a higher, symmetrical voltage of about 18 V (+/−9 V) for communication with the PC host and TTL levels for the MCU. The line voltage is generated by an internal charge pump from the single 5 V supply, using the externally fitted capacitors. The hardware handshaking lines (RTS, CTS) are not implemented. The RS232 interface is described in Chapter 9.

POWER SUPPLY

The +5 V power supply must have the following characteristics:

- Accurate voltage
- Sufficient current
- Low noise & ripple

The PIC 16F877 is specified to draw less than 2 mA at 4 MHz. The LCD module may draw up to 10 mA, and any interfacing circuits must be included in the power supply current budget. A standard 1 A linear regulator chip should be sufficient in most cases, as any high power loads will normally run from the unregulated supply. A 5 V regulator circuit can be added to the circuit if necessary, but an external plug-top regulated supply and coaxial input could be more convenient. These typically supply at least 500 mA, while a bench supply will provide at least 1 A. Standard IC regulators will provide +5 V +/−0.25 V, with low noise and ripple.

Base Board Test Program

A test program which exercises all parts of the hardware, while being as simple as possible, is always useful. If the hardware can be proved to function correctly, the software development can then be undertaken with confidence. The base board test program reads the analogue input, indicates if it is over 1.28 V, displays it and stores the 8-bit voltage code in the serial memory. The second part reads the keypad and displays the key, with audible feedback (Program 11.1).

The serial memory access routine, the display driver routine and the BCD conversion routine are allocated reserved GPR ranges in the register label equates. These routines, as well as the analogue port read routine, are included as separate source code files at the end of the main source listing. This allows these routines to be re-used in future programs, ideally without modification. Information about the way the routine is used (register requirements, parameter passing and so on) is included in the header to make this as straightforward as possible. The keypad scanning routine was modified to use a mix of Ports B and C lines.

The directive DT has been used here to create the data table of ASCII codes required for the display of fixed messages. It generates a sequence of RETLW instructions for each code, which is accessed in the usual way by modifying the program counter with ADDWF PCL (ensure that there is no page boundary in the table, or it will not work correctly!). The table is terminated with a zero, which can be detected by the output routine to terminate the message.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;         Project:            Interfacing PICs
;         Source File Name:   BASE1.ASM
;         Devised by:         MPB
;         Date:               31-1-06
;         Status:             Finished
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;         Program to exercise the 16F877 BASE module
;         with 8-bit analogue input, LCD, phone keypad
;         and serial memory
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          PROCESSOR 16F877
;         Clock = XT 4MHz, standard fuse settings
          __CONFIG 0x3731

; LABEL EQUATES     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          INCLUDE "P16F877A.INC" ; standard labels

; User register label allocation ;;;;;;;;;;;;;;;;;;;;;;;;;

; GPR 20 - 2A       local variables
; GPR 30 - 32       keyin subroutine
; GPR 60 - 65       SERMEM serial memory driver
; GPR 70 - 75       LCDIS display driver
; GPR 77 - 7A       CONDEC BCD conversion routine

LCDport EQU         08        ; assign LCD to Port D
LCDdirc EQU         88        ; data direction register

Temp      EQU       20        ; temp store
Tabin     EQU       21        ; Table pointer

; Keypad registers

Cont      EQU       30        ; Delay counter
Key       EQU       31        ; Input key
Test      EQU       32        ; Key check

;----------------------------------------------------------
; MAIN PROGRAM
;----------------------------------------------------------

          ORG       0         ; Default start address
          NOP                 ; required for ICD mode

; Port & display setup -----------------------------------
          BANKSEL   TRISA               ; Select bank 1
          MOVLW     B'11001000'         ; Port B code for
          MOVWF     TRISB               ; keypad row outputs
          MOVLW     B'10010111'         ; Port C code for
          MOVWF     TRISC               ; rows and columns
          CLRF      TRISD               ; Display port
          BANKSEL   PORTA               ; Select bank 0
          CLRF      PORTD               ; Clear display
          CLRF      HiReg               ; memory page 0
          CLRF      LoReg               ; first location
          CALL      inimem              ; init. serial memory
          CALL      inid                ; Initialise display
;----------------------------------------------------------
; MAIN LOOP
;----------------------------------------------------------
start     CLRW                          ; Select AN0 input
          CALL      adin                ; read analogue input
          CALL      condec              ; convert to decimal
          CALL      putdec              ; display input
          CALL      store               ; store in memory

          CALL      putkey              ; Fixed message
          CALL      keyin               ; scan phone keypad
          CALL      send                ; display key
          GOTO      start               ; and again
```

Program 11.1 Base module test program

```
;------------------------------------------------------------
; SUBROUTINES
;-----------------------------------------------------------
; Routine to scan 3x4 phone key pad
; Returns ASCII code in W
; Output rows: RB2,RB4,RB5,RC5
; Input cols: RC0,RC1,RC2
;-----------------------------------------------------------
keyin     NOP
          BANKSEL   TRISC
          MOVLW     B'10010111'         ; Port C code for
          MOVWF     TRISC               ; rows and columns
          BANKSEL   PORTC

          BSF       PORTB,2             ; Set
          BSF       PORTB,4             ; rows
          BSF       PORTB,5             ; high
          BSF       PORTC,5             ; initially

          BSF       Cont,0              ; Counter not zero
          CLRF      Test                ; No key
; Scan keyboard -------------------------------------------

again     CLRW                          ; No key yet
          BCF       PORTB,2             ; Row 1
          NOP                           ; wait
          NOP

          BTFSS     PORTC,0             ; key pressed?
          MOVLW     '1'                 ; yes - load ASCII
          BTFSS     PORTC,1             ; next
          MOVLW     '2'                 ; etc
          BTFSS     PORTC,2             ;
          MOVLW     '3'                 ;
          BSF       PORTB,2             ; deselect row
; ---------------------------------------------------------
          BCF       PORTB,4             ; second row
          BTFSS     PORTC,0
          MOVLW     '4'
          BTFSS     PORTC,1
          MOVLW     '5'
          BTFSS     PORTC,2
          MOVLW     '6'
          BSF       PORTB,4
; ---------------------------------------------------------
          BCF       PORTB,5             ; third row
          BTFSS     PORTC,0
          MOVLW     '7'
          BTFSS     PORTC,1
          MOVLW     '8'
          BTFSS     PORTC,2
          MOVLW     '9'
          BSF       PORTB,5
; ---------------------------------------------------------
          BCF       PORTC,5             ; fourth row
          BTFSS     PORTC,0
          MOVLW     '*'
          BTFSS     PORTC,1
          MOVLW     '0'
          BTFSS     PORTC,2
          MOVLW     '#'
          BSF       PORTC,5

; Test key ------------------------------------------------

          MOVWF     Test                ; get code
          MOVF      Test,F              ; test it
          BTFSS     STATUS,Z            ; if code found
          GOTO      once                ; beep once

          MOVF      Key,W               ; load key code and
          RETURN                        ; if no key
; Check if beep done --------------------------------------

once      MOVF      Cont,F              ; beep already done?
          BTFSC     STATUS,Z
          GOTO      again               ; yes - scan again

          MOVF      Test,W              ; store key
          MOVWF     Key
; Beep ----------------------------------------------------

beep      MOVLW     10                  ; 10 cycles
          MOVWF     Cont

buzz      BSF       PORTB,0             ; one beep cycle
          CALL      onems               ; 2ms
          BCF       PORTB,0
          CALL      onems
          DECFSZ    Cont                ; last cycle?
          GOTO      buzz                ; no

          GOTO      again               ; yes

; End of keypad routine ------------------------------------
```

Program 11.1 *Continued*

```
; -----------------------------------------------------------
; Display input test voltage on top line of LCD
;------------------------------------------------------------

putdec    BCF       Select,RS           ; set display command
mode
          MOVLW     080                 ; code to home cursor
          CALL      send                ; output it to display
          BSF       Select,RS           ; and restore data mode

; Convert digits to ASCII ----------------------------------

          MOVLW     030                 ; load ASCII offset
          ADDWF     Huns                ; convert hundreds to
ASCII
          ADDWF     Tens                ; convert tens to ASCII
          ADDWF     Ones                ; convert ones to ASCII

; Display voltage on line 1 -------------------------------

          CALL      volmes              ; Display text on line 1

          MOVF      Huns,W              ; load hundreds code
          CALL      send                ; and send to display
          MOVLW     '.'                 ; load point code
          CALL      send                ; and output
          MOVF      Tens,W              ; load tens code
          CALL      send                ; and output
          MOVF      Ones,W              ; load ones code
          CALL      send                ; and output
          MOVLW     ' '                 ; load space code
          CALL      send                ; and output
          MOVLW     'V'                 ; load volts code
          CALL      send                ; and output

          RETURN                        ; done

; Store voltage in serial memory -------------------------

store     BSF       SSPCON,SSPEN        ; Enable memory port
          MOVF      ADRESH,W            ; Get voltage code
          MOVWF     SenReg              ; Load it to write
          CALL      writmem             ; Write it to memory
          INCF      LoReg               ; Next location
          BCF       SSPCON,SSPEN        ; Disable memory port
          RETURN                        ; done

;----------------------------------------------------------
; Display key input on bottom line of LCD
;----------------------------------------------------------

putkey    BCF       Select,RS           ; set display command
mode
          MOVLW     0C0                 ; code to home cursor
          CALL      send                ; output it to display
          BSF       Select,RS           ; and restore data mode
          CALL      keymes
          RETURN                        ; done

;----------------------------------------------------------
; Display fixed messages
;----------------------------------------------------------
volmes    CLRF      Tabin               ; Zero table pointer
next1     MOVF      Tabin,W             ; Load table pointer
          CALL      mess1               ; Get next character
          MOVWF     Temp                ; Test data...
          MOVF      Temp,F              ; ..for zero
          BTFSC     STATUS,Z            ; Last letter done?
          RETURN                        ; yes - next block
          CALL      send                ; no - display it
          INCF      Tabin               ; Point to next letter
          GOTO      next1               ; and get it

; ---------------------------------------------------------
keymes    CLRF      Tabin               ; Zero table pointer
next2     MOVF      Tabin,W             ; Load table pointer
          CALL      mess2               ; Get next character
          MOVWF     Temp                ; Test data...
          MOVF      Temp,F              ; ..for zero
          BTFSC     STATUS,Z            ; Last letter done?
          RETURN                        ; yes - next block
          CALL      send                ; no - display it
          INCF      Tabin               ; Point to next letter
          GOTO      next2               ; and get it
;----------------------------------------------------------
; Text strings for fixed messages
;----------------------------------------------------------
mess1     ADDWF     PCL                 ; Set table pointer
          DT        "Volts = ",0        ; Text for display

mess2     ADDWF     PCL                 ; Set table pointer
          DT        "Key = ",0          ; Text for display
;-------------------------------------------------------------
```

Program 11.1 *Continued*

```
;----------------------------------------------------------
; INCLUDED ROUTINES
;----------------------------------------------------------
; LCD DRIVER
;         Contains routines:
;         init:     Initialises display
;         onems:    1 ms delay
;         xms:      X ms delay
;                   Receives X in W
;         send:     sends a character to display
;                   Receives: Control code in W (Select,RS=0)
;                               ASCII character code in W (RS=1)
;
          INCLUDE   "LCDIS.INC"
;
;----------------------------------------------------------
; Convert 8 bits to 3 digit decimal
;
;         Receives 8-bits in W
;         Returns BCD diits in 'huns','tens','ones'
;
          INCLUDE   "CONDEC.INC";
;
;----------------------------------------------------------
; Read selected analogue input
;
;         Receives channel number in W
;         Returns 8-bit input in W
;
          INCLUDE   "ADIN.INC"
;
;----------------------------------------------------------
; SERIAL MEMORY DRIVER
;         Write high address into 'HiReg' 00-3F
;         Write low address into 'LoReg' 00-FF
;         Load data send into 'SenReg'
;         Read data received from 'RecReg'
;
;         To initialise call 'inimem'
;         To write call 'writmem'
;         To read call 'readmem'
;
          INCLUDE "SERMEM.INC"
;
;----------------------------------------------------------
          END                           ; of source code
;----------------------------------------------------------
```

Program 11.1 *Continued*

The include routines must be stored in the same application folder with the main source code, or the full file path to the folder must be given in the include statement. For relatively small files, it is more convenient to copy them into each application folder, as is the case here for the standard register label file 'P16F877.INC'. These files were created by modifying the demonstration program for each interface into the form of a subroutine. This entails deleting the initialisation which is common with the main program, and using a suitable label at the start (same as the include file name), and finishing with a RETURN. This is a simple way to start building a library of utilities for the base hardware.

As can be seen, the software design philosophy is to make the main program as concise as possible, so that ultimately it consists of a sequence of subroutine calls. This makes the program easier to understand and debug. The subroutines in the main program are mainly concerned with operating the display, while the included routines are specific to particular interfaces. These are not printed here, but are similar to the stand-alone demo programs, and can be inspected in the actual source code files provided.

Memory System

A conventional microprocessor system contains separate CPU and memory chips. A similar arrangement can be used if we need extra memory in a PIC system and there is no shortage of I/O pins. Parallel memory is inherently faster than the serial memory as seen in Chapter 9, because the data is transferred 8 bits at a time. A system schematic is shown in Figure 11.3.

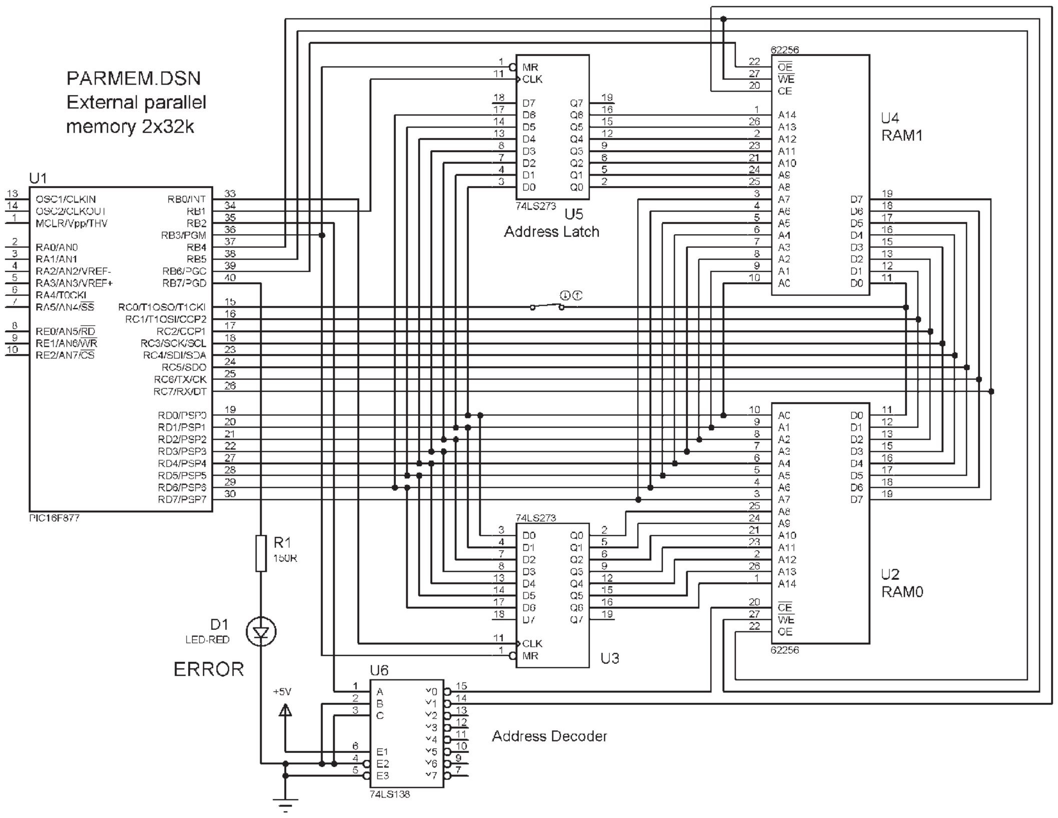


Figure 11.3 Parallel memory system

Memory System Hardware

A pair of conventional 62256 32k RAM chips are used to expand the memory to 64k bytes. Port C in the PIC 16F877 is used as a data bus, and Port D as an address bus. In order to reduce the number of I/O pins needed for external memory addressing, an address latch is used to store the high byte of the 15-bit address (D7 unused). The address is output in two stages; the high byte is latched, selecting the high address block; the low byte is then output direct to the memory chips low address bits to select the location. If a block read is required, the high address can remain unchanged while a page of 256 bytes is accessed (low address 00-FF). The next page can then be selected if required. By incrementing the high byte (page select) from 00 to 7F, the whole RAM range can be accessed in each chip. Each chip is selected individually via an address decoder, but the pairs of bytes from corresponding locations can be put together and processed as 16-bit data within the MCU.

Each RAM chip has eight data I/O pins (D0−D7) and 15 address pins (A0−A14). This means that each location contains 8 bits, and there are $2^{15} = 32768$ locations. An address code is fed in, and data for that address read or written via the data pins. To select the chip, the Chip Enable (!CE) pin is taken low. To write a location, an address code is supplied, data presented at D0−D7, and the Write Enable (!WE) is pulsed low. To read data, the Output Enable (!OE) is set active (low), along with the chip enable, and the data from the address can then be read back.

The high address byte is temporarily stored in a ‘273 latch (8-bit register), which is operated by a master reset and clock. The 7-bit high address is presented at the inputs, and the clock pulsed high to load the latch. The MCU then outputs the low address direct to the memory low address pins, and the combined address selects the location. In the test program, all addresses are accessed in turn by incrementing the low address from 00 to FF for each high address (memory page select). The memory can be organised as 64k × 8 bytes or 32k × 16-bit words.

Memory System Software

The test program (Program 11.2) writes a traditional checkerboard pattern to the memory chips, placing the codes 01010101 (55h) and 10101010 (AAh) in successive locations. Adjacent memory cells are therefore all set to opposite voltage values, and any interaction between them, for example, due to charge leakage, is more likely to show up. The memory is written and read, the data retrieved, and compared with the correct value. If the write and read values do not agree, an error LED is lit. A switch has been placed in the data line D0 so that the error detection system can be tested in the simulation. When the switch is open, data 0 will be written to all D0 bits (open circuit data input), so all the least significant bits of the test data 55h will be incorrect, with the value 54h read back (Figures 11.4 and 11.5).

The system operates in a similar way as a conventional processor, with address decoding hardware to organise the memory access. The address decoder

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;         PARMEM.ASM                    MPB       6-9-05
;...............................................................
;
;         Parallel memory system
;         Status: Complete
;
;         PIC 16F877 operates with expansion memory RAM
;         = 2 x 62256 32kb
;         Control bits = Port B
;         Data bus = Port C
;         Address Bus = Port D
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          PROCESSOR 16F877    ; define MPU
          __CONFIG 0x3731     ; XT clock

;         LABEL EQUATES       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          INCLUDE "P16F877.INC"         ; Standard labels

ConReg    EQU       06        ; Port B = Control Register
DatReg    EQU       07        ; Port C = Data Register
AddReg    EQU       08        ; Port D = Address Register

HiAdd     EQU       20        ; High address store

CLK0      EQU       0         ; RAM0 address buffer clock
CLK1      EQU       1         ; RAM1 address buffer clock
SelRAM    EQU       2         ; RAM select bit
ResHi     EQU       3         ; High address reset bit
WriteEn   EQU       4         ; Write enable bit
OutEn0    EQU       5         ; Output enable bit RAM0
OutEn1    EQU       6         ; Output enable bit RAM1
LED       EQU       7         ; Memory error indicator


; Initialise ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          ORG       0                   ; Place machine code
          NOP                           ; Required for ICD mode

          BANKSEL   TRISB               ; Select bank 1
          CLRF      TRISB               ; Control output bits
          CLRF      TRISC               ; Data bus initially output
          CLRF      TRISD               ; Address bus output

          BANKSEL   AddReg              ; Select bank 0
          CLRF      DatReg              ; Clear outputs initially
          CLRF      AddReg              ; Clear outputs initially

          BCF       ConReg,CLK0         ; RAM0 address buffer clock
          BCF       ConReg,CLK1         ; RAM1 address buffer clock
          BCF       ConReg,SelRAM       ; Select RAM0 initially
          BCF       ConReg,ResHi        ; Reset high address latches
          BSF       ConReg,OutEn0       ; Disable output enable RAM0
          BSF       ConReg,OutEn1       ; Disable output enable RAM1
          BSF       ConReg,WriteEn      ; Disable write enable bit
          BCF       ConReg,LED          ; Switch off error indicator


; MAIN LOOP ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

start     CALL      write               ; test write to memory
          CALL      read                ; test read from memory
          SLEEP                         ; shut down
```

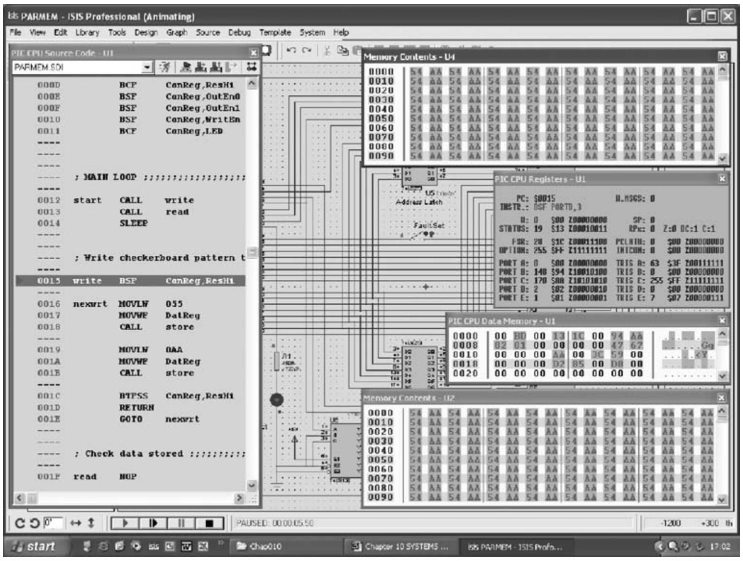
Program 11.2 Parallel memory program source code

```
; SUBROUTINES ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Write checkerboard pattern to both RAMs ;;;;;;;;;;;;;;;;;;;;;

write     BSF       ConReg,ResHi        ; Enable address latches

nexwrt    MOVLW     055                 ; checkerboard test data
          MOVWF     DatReg              ; output on data bus
          CALL      store               ; and write to RAM

          MOVLW     0AA                 ; checkerboard test data
          MOVWF     DatReg              ; output on data bus
          CALL      store               ; and write to RAM

          BTFSS     ConReg,ResHi        ; all done?
          RETURN                        ; yes - quit
          GOTO      nexwrt              ; no - next byte pair
; Check data stored ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

read      NOP                           ; required for label
          BANKSEL   TRISC               ; select bank 1
          MOVLW     0FF                 ; all inputs..
          MOVWF     TRISC               ; ..at Port C

          BANKSEL   ConReg              ; select default bank 0
          BSF       ConReg,ResHi        ; Enable address latches

          BCF       ConReg,SelRAM       ; select RAM0
          BCF       ConReg,OutEn0       ; set RAM0 for output
          CALL      nexred              ; check data in RAM0

          BSF       ConReg,SelRAM       ; select RAM1
          BCF       ConReg,OutEn1       ; set RAM1 for output
          CALL      nexred              ; check data in RAM1

          RETURN                        ; all done

; Load test data and check data ...............................

nexred    MOVLW     055                 ; load even data byte
          CALL      test                ; check data
          MOVLW     0AA                 ; load odd data byte
          CALL      test                ; check data

          BTFSS     ConReg,ResHi        ; all done?
          RETURN                        ; yes - quit
          GOTO      nexred              ; no - next byte pair
; Write data to RAM ...........................................
store     BCF       ConReg,SelRAM       ; Select RAM0
          BCF       ConReg,WritEn       ; negative pulse ..
          BSF       ConReg,WritEn       ; ..on write enable

          BSF       ConReg,SelRAM       ; Select RAM1
          BCF       ConReg,WritEn       ; negative pulse ..
          BSF       ConReg,WritEn       ; ..on write enable
          INCF      AddReg              ; next address
          BTFSC     STATUS,Z            ; last address?
          CALL      inchi               ; yes-inc. high address
          RETURN                        ; no-next byte
; Test memory data ............................................
test      MOVF      DatReg,F            ; read data
          SUBWF     DatReg,W            ; compare data
          BTFSS     STATUS,Z            ; same?
          BSF       ConReg,LED          ; no - switch on LED

          INCF      AddReg              ; yes - next address
          BTFSC     STATUS,Z            ; last address in block?
          CALL      inchi               ; yes-inc. high address
          RETURN                        ; no - continue
; Select next block of RAM ....................................
inchi     INCF      HiAdd               ; next block
          BTFSC     STATUS,Z            ; all done?
          GOTO      alldon              ; yes

          MOVF      HiAdd,W             ; no - load high address
          MOVWF     AddReg              ; output it

          BSF       ConReg,CLK0         ; clock it into latches
          BSF       ConReg,CLK1
          BCF       ConReg,CLK0
          BCF       ConReg,CLK1
          CLRF      AddReg              ; reset low address
          RETURN                        ; block done

alldon    BCF       ConReg,ResHi        ; reset address latches
          RETURN                        ; all blocks done
          END       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Program 11.2 *Continued*

```
PARMEM
          Parallel memory test program
          Writes test data to both RAM chips simultaneously
          Checks test data in RAM0, then RAM1

Initialisation
          Control Register = Port B = outputs
                    Bit 0 = RAM0 address buffer clock = 0
                    Bit 1 = RAM1 address buffer clock = 0
                    Bit 2 = RAM chip select bit = 0
                    Bit 3 = RAM address latch reset = 0
                    Bit 4 = RAM !Write enable = 1
                    Bit 5 = RAM0 !Output enable = 1
                    Bit 6 = RAM1 !Output enable = 1
                    Bit 7 = Error indicator LED = 0
          Data Register = Port C = outputs = 00
          Address Register = Port D = outputs = 00

Main
          Write checkerboard pattern to both RAMs
          Check data stored
          Sleep

Subroutines

          Write checkerboard pattern to both RAMs
          REPEAT
                    Load data 55h
                    Write it to current memory location address
                    Load data AAh
                    Write it to current memory location address
          UNTIL all locations done

                    Write it to current memory location address
                              Write data to RAM0
                              Write data to RAM1
                              Increment low address
                              IF last address, Select next page

          Check data stored
                    Set data port for input
                    Select RAM0 for output
                    Check data in RAM
                    Select RAM1 for output
                    Check data in RAM

                    Check data in RAM
                              REPEAT
                                        Load 55h
                                        Compare with stored byte (even)
                                        Load AAh
                                        Compare with stored byte (odd)
                              UNTIL all done

                              Compare with stored byte
                                        Compare bytes at current address
                                        IF different, switch on error LED
                                        Increment low address
                                        IF end of page, Select next page

                                        Select next page
                                                  Increment page select
                                                  IF last page done
                                                            reset high address latch
                                                            quit
                                                  Latch high address
                                                  Reset low address
```

Figure 11.4 Parallel memory program outline



```
0000D            BCF      ConReg,ResH1
0000E            BSF      ConReg,OutEn0
0000F            BSF      ConReg,OutEn1
0010             BSF      ConReg,WritEn
0011             BCF      ConReg,LED
----
----
----
----   ; MAIN LOOP ;;;;;;;;;;;;;;;;;
----
0012   start     CALL     write
0013             CALL     read
0014             SLEEP
----
----
----   ; Write checkerboard pattern t
----
0015   write     BSF      ConReg,ResH1
----
0016   nexwrt    MOVLW    055
0017             MOVWF    DatReg
0018             CALL     store
----
0019             MOVLW    0AA
001A             MOVWF    DatReg
001B             CALL     store
----
001C             BTFSS    ConReg,ResH1
001D             RETURN
001E             GOTO     nexwrt
----
----
----   ; Check data stored ;;;;;;;;;;
----
001F   read      NOP
```

```
Memory Contents - U4
0000 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
0010 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
0020 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
0030 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
0040 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
0050 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
0060 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
0070 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
0080 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
0090 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
```

```
PIC CPU Registers - U1
      PC: $0015                 W.MSGS: 0
  INSTR.: BSF PORTD,3

       W: 0    $00 %00000000        SP: 0
  STATUS: 19   $13 %00010011       RPx: 0   Z:0 DC:1 C:1

     FSR: 28   $1C %00011100   PCLATH: 0   $00 %00000000
  OPTION: 255  $FF %11111111   INTCON: 0   $00 %00000000

  PORT A: 0    $00 %00000000   TRIS A: 63  $3F %00111111
  PORT B: 148  $94 %10010100   TRIS B: 0   $00 %00000000
  PORT C: 170  $AA %10101010   TRIS C: 255 $FF %11111111
  PORT D: 2    $02 %00000010   TRIS D: 0   $00 %00000000
  PORT E: 1    $01 %00000001   TRIS E: 7   $07 %00000111
```

```
PIC CPU Data Memory - U1
0000 | 00 BD 00 13 | 1C 00 94 AA |
0008 | 02 01 00 00 | 00 00 47 67 | ......Gg
0010 | 00 00 00 AA | 00 3C 59 00 | .....<Y.
0018 | 00 00 00 D2 | 85 00 D8 00 |
0020 | 00 00 00 00 | 00 00 00 00 | ........
```

```
Memory Contents - U2
0000 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
0010 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
0020 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
0030 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
0040 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
0050 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
0060 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
0070 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
0080 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
0090 | 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA 54 AA
```

Figure 11.5 Memory test simulation screen

chip has three inputs CBA, which receive a binary select code from the processor. The corresponding output is taken low − for example, if binary 6 is input (110), output Y6 is selected (low), while all the others stay high. This decoder can generate 8-chip select signals, and if attached to the high address lines of a processor, enable the memory chips in different ranges of addresses. In our system here, only the least significant input (A) and two outputs (Y0, Y1) are used, giving a minimal system. The additional address decoder outputs could be used to control extra memory chips attached to the same set of address and data lines.

The bus system operation depends on the presence of tri-state buffers at the output of the RAM chips. These can be switched to allow data input (!CE & !WE = low), data output (!CE & !OE = low) or disabled (!CE & !OE = high). In the disabled state, the outputs of the RAM are effectively disconnected from the data bus. Only one RAM chip should be enabled at a time, otherwise there will be contention on the bus − different data bytes present at the same time, causing a data error.

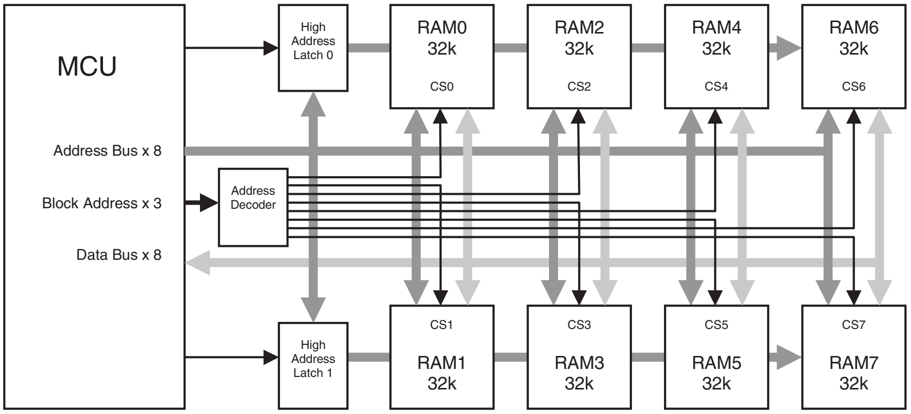
Extended Memory System

If this system was extended using six more RAM chips, there would be a total of 32k × 8 bytes = 256k. A 3-bit input would be required into the address decoder (Port E could be used) to extend the chip selection system.

The high address (page select) would still be 7 bits, and the location select 8 bits, giving a total address width of 18 bits. The address decoder chip also has some enable inputs, which disable all outputs when active − this can be used to extend the addressing system further.

A memory map has been constructed for this extended memory design (Figure 11.6 (b)). It contains a total of 256k locations, divided into 8 blocks of 32k (one chip), each containing 128 pages of 256 bytes. In an extended system, consideration must be given to the amount of current required by each input connected to the busses. The high current output of the PIC is useful here, but the standard digital outputs of the address latches have a more limited drive

(a)



(b)

Block 32k	Start Address	End Address
0	00000	07FFF
1	08000	0FFFF
2	10000	17FFF
3	18000	1FFFF
4	20000	27FFF
5	25000	2FFFF
6	30000	37FFF
7	38000	3FFFF

(c)

	Block Add (E)			Page Address (Port D latched)							Location Address (Port D)							
Bits	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
Allocation	RE2	RE1	RE0	RD6	RD5	RD4	RD3	RD2	RD1	RD0	RD7	RD6	RD5	RD4	RD3	RD2	RD1	RD0

Figure 11.6 256k Extended memory system: (a) block diagram; (b) memory map; (c) address bit allocation

capability. If necessary, current drivers should be added at the latch outputs in the extended system.

Other PIC Chips

The PIC 16F877 is used throughout this book for simplicity − in real applications, a chip should be chosen from the PIC range which most closely meets the design requirements, both in terms of the absolute number of I/O pins and the special interfaces available. In addition, the program memory size must be sufficient for the application, and the clock speed, EEPROM space and so on, taken into account. The range is constantly expanding; new chips with new features and different combinations of existing features are constantly added.

The 16F877 can be used as a reference point when comparing the features of the other chips that are available. The main criteria are

- Number of I/O pins
- Program memory size
- Peripheral set
- Data memory size
- Instruction set features

A small sample has been selected from each series for comparison from the current manufacturers catalogue, found at www.microchip.com.

PIC 16FXXX Mid-range Series

The PIC 16FXXX chips all have the same 14-bit instruction set and run at 20 MHz (maximum clock rate). A selection of currently available devices is listed in Table 11.1 (a).

We can see that the complexity seems to increase with the type number, including the number of I/O pins, memory size and range of peripherals. Most of those listed have been added since the 16F84 and the 16F877 were introduced, and many include an internal oscillator, which can be used if precise timing is not required. This saves on I/O pins, and means that all pins except two (for the power supply) can be allocated to I/O devices. The '84A is obsolete for commercial applications, where chips are bought in bulk, and so the price is now relatively high. It is still used in education as a training device.

PIC Small MCUs

These include the 10FXXX and 12FXXX devices, of which a few examples are listed in Table 11.1 (b) for comparison with the mid-range types. A simplified 12-bit instruction set is used in these chips. The smallest has only 6 pins, so in surface mount form they are among the smallest microcontrollers available. Obviously, these only have limited features, only an internal oscillator and no analogue inputs. Some 8-pin devices can offer analogue inputs, EEPROM and serial interfaces, in which case, the 14-bit instruction set is used.

PIC Power MCUs

The high power (in processing terms and in power consumption) PIC chips are designated 18FXXXX (Table 11.1 (c)). These have a more extensive 16-bit instruction set, run at 40 MHz, and are generally optimised for programming in ‘C’. All these examples also have a hardware multiplier to speed up arithmetic operations, and an internal oscillator option which runs at 8 MHz. The lowest number has been selected in each range, to show the significance of the numbering. The number following the ‘F’ refers to the number of pins: 1 = 18, 2 = 28, 4 = 40, 6 = 64, 8 = 80. A full range of peripherals is available, and the largest in the group has 32k of program memory, a total of 69 I/O pins, 16 ADC inputs and 5 hardware timers. If even more power is needed, Microchip produces a range of digital signal processors.

The relative cost quoted is the guide price in dollars at the time of writing. Actual prices depend on the volume of demand, and variation between competing suppliers, but the relative cost should remain a useful guide over time. As it happens, the 12F675 has a guide price of $1.00 at the time of writing, providing a useful reference point.

Only flash memory devices have been listed, as these are most likely to be purchased for experimental work, application prototyping and small-scale production. A corresponding range of OTP (one-time programmable) devices is available which can be sourced at lower cost, for medium-scale production. For higher production volumes, the chips can be ordered pre-programmed from the manufacturer, either in OTP form or mask programmed. In the latter case, the programme is built in during the final stages of the chip manufacture, and is used to produce large numbers of MCUs at minimum cost per chip, for applications where the code will probably not need updating during the product lifetime.

(a) PIC mid-range 16FXXXX 8-bit flash MCUs (max clock = 20 MHz, 14-bit instructions)

MCU	# Pins	# Instructions	# RAM	# EEPROM	# Total I/O	ADC Channels	Timers (× bits)	CCP Modules	PWM Modules	Other Interfaces	Int Osc (MHz)	Relative Cost
16F505	14	1024	72	–	12	–	1×8	–	–	–	4	0.56
16F506	14	1024	67	–	12	3	1×8	–	–	–	8	fp
16F627A	18	1024	224	128	16	–	2×8, 1×16	1	–	–	4	1.19
16F628A	18	2048	224	128	16	–	2×8, 1×16	1	–	USART	4	1.29
16F687	20	2048	128	256	18	12	1×8, 1×16	–	–	USART, 2×I^{2}C	8	1.14
16F690	20	4096	256	256	18	12	2×8, 1×16	1	–	USART, 2×I^{2}C,SPI	8	1.34
16F777	40	8192	368	0	36	14	2×8, 1×16	1	3	USART, 2×I^{2}C,SPI	8	3.46
16F818	18	1024	128	128	16	5	2×8, 1×16	1	1	2×I^{2}C,SPI	8	1.37
16F84A	18	1024	68	64	13	–	1×8	–	–	–	–	2.71
16F877A	40	8192	368	256	33	8	2×8, 1×16	2	2	USART, 2×I^{2}C,SPI	–	3.71
16F88	18	4096	368	256	16	7	2×8, 1×16	1	1	USART, 2×I^{2}C,SPI	8	1.93
16F946	64	8196	336	256	53	8	2×8, 1×16	2	–	USART, 2×I^{2}C,SPI	8	2.38

(b) PIC small flash MCUs

MCU	# Pins	# Instructions	# RAM	# EEPROM	# Total I/O	ADC Channels	Analogue Comparators	Timers (× bits)	Instruction Length	Int Osc (MHz)	Relative Cost
10F200	6	256	16	–	4	–	–	1×8	12 bits	4	0.43
10F222	6	512	23	–	4	–	–	1×8	12 bits	8	fp
12F508	8	512	25	–	6	–	–	1×8	12 bits	8	0.49
12F675	8	1024	64	128	6	4	1	1×8, 1×16	14 bits	4	1.00

fp = future product at the time of writing

(c) PIC power flash MCUs (max clock 40 MHz, 16-bit instructions)

MCU	# Pins	# Instructions	# RAM	# EEPROM	# Total I/O	ADC Channels	Timers (× bits)	Hardware Multiplier	Interfaces	Int Osc (MHz)	Relative Cost
18F1220	18	2048	256	256	16	7	1×8, 3×16	8×8	USART	8	2.20
18F2220	28	2048	512	256	25	10	1×8, 3×16	8×8	USART, 1^{2}C, SPI	8	3.53
18F4220	40	2048	512	256	36	13	1×8, 3×16	8×8	USART, 1^{2}C, SPI	8	3.90
18F6310	64	4096	768	–	50	12	2×8, 3×16	8×8	USART, 1^{2}C, SPI	8	3.84
18F8310	80	4096	768	–	70	12	2×8, 3×16	8×8	USART, 1^{2}C, SPI	8	4.29
18F8680	80	32768	3328	1024	69	16	2×8, 3×16	8×8	USART, 1^{2}C, SPI	8	6.98

Table 11.1 Selected PIC microcontrollers

System Design

When designing a microcontroller application, we normally start with a specification of the functions the system is intended to perform. The appropriate chip should then be provisionally selected. Any given design team is likely to have a preferred choice for the type of controller, since they will have experience and development tools to support that type already. Alternative types will probably be considered only if the default range cannot provide the features required, or there is some other reason to change, such as designing for a customer who uses a different range and is tooled up for products based on this type.

Specification

Here, our default choice is the PIC. We have to identify the features required for the MCU, it's interfacing and select any sensors, transducers and communication links needed.

Here is a typical specification:

A control system is required for a refrigeration unit which will maintain the temperature within an insulated closed space, such as a temperature-controlled shipping container, at a selected temperature between 1°C and 9°C. The controller will connect to the refrigeration unit via a suitable relay, which switches the compressor on and off. The temperature will be controlled to within +/− 0.5°C, and settable using up/down push buttons. It must be displayed on a self-illuminating display, which is readable from 2 m. When the unit is switched on, the previous temperature setting must be used. If the temperature deviates from the set temperature by more than 2°C, or any other significant fault occurs, an alarm must sound within the unit, and remotely (e.g. in the lorry cab) with a flashing light. The design must be highly reliable, robust, moisture proof and low maintenance. It will be powered from the vehicle 12 V DC supply.

Design Outline

The first step in the development process is to draw a block diagram, so that the system requirements can be visualised (Figure 11.7). This also allows design requirements to be incorporated at an early stage.

For reliable operation, it is suggested that a set of four temperature sensors are installed at the four corners of the storage space. In normal operations, an average of these will displayed. If a single sensor goes faulty, we will assume that its reading will go out of range. A 'sensor faulty' alarm can be generated, say a short beep and flashing indicator. As long as the other

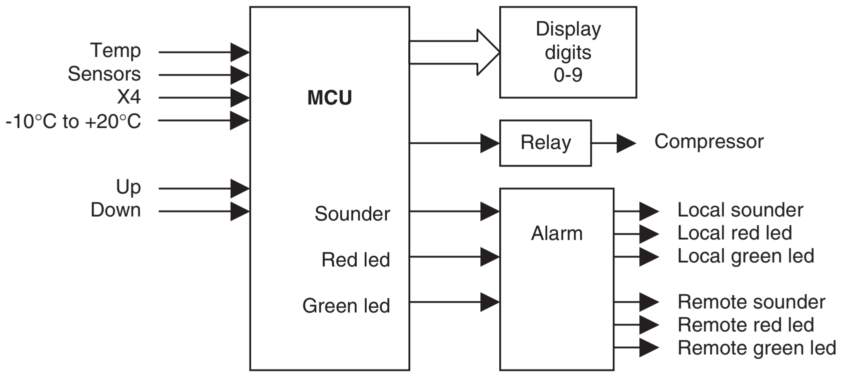


Figure 11.7 Block diagram of refrigeration controller

three sensors agree within 2°C, the controller will continue to operate, taking an average of these three only and ignoring the faulty sensor. If more than one sensor goes out of range, the temperature too high (high frequency) or too low (low frequency) will be sounded and indicated. These alarm conditions can be checked and demonstrated by simply unplugging the temperature sensors.

Component Selection

The MCU needs four 8-bit analogue inputs. At mid-range, 8 bits will give a resolution of about 1%, which is more than adequate. A total of 10 digital I/O pins are needed. Program memory of 1k will be assumed initially, but this will be reviewed when the code is complete. An accurate clock is not needed, so an internal oscillator will be used, reducing the component count and improving reliability. The PIC 16F818 seems to fit the bill, with 16 I/O pins in total, including 5 analogue inputs. Remember, we will need an extra analogue input for the reference voltage. It has 1k program memory, and EEPROM for storing the previous set temperature. The hardware timers will be useful for generating the timed outputs. If we run out of program memory, the 16F819 (2k) can be substituted at slightly higher cost. Considering the cost of failure of the unit, this will not be significant. Both chips have an 8 MHz internal oscillator, and ICD programming and debugging.

Good-quality push buttons with moisture proof housings will be selected. The relay will be the default control interface for the compressor. This is likely to be an independent diesel unit, so will have its own control unit, whose interfacing requirements must be known. The display can be a single 7-segment

LED type, which is cheap, simple to interface and self-illuminating. A larger than standard size can be used for good visibility, and these are not expensive. Since different frequencies will be used in the alarm, loudspeakers will be used, with the drive signals generated in software. Red LEDs for the alarm will be used, with a high brightness LED in the remote monitor unit. A green power LED indicating normal operation will also be incorporated into both the main unit and the remote alarm unit.

The temperature sensors will be housed in aluminium boxes, bonded to the face of one side, for good thermal contact. The LM35 covers the range with sufficient accuracy (just), but an alternative could be sought which has a smaller range, giving greater resolution, for example −10°C to +20°C. A local amplifier is suggested which provides a 0−20 mA current output, representing temperatures −4°C to +16°C, with 4 mA representing 0°C. A current-driven link is more reliable in harsh environments, and avoids the effect of any volt drop over the length of the sensor connectors, which could be several meters. Screened screw connectors will be used at both ends of the connecting cables for electrical and mechanical robustness. The regulated 5 V supply from the main unit will be provided to the remote sensors, with the supply 0 V connected via the cable screen. The signal 0 V will be separated and screened, to minimise the possibility of interference and false alarms due to the compressor switching currents or the vehicle ignition system. The same connectors and cabling can be used for the remote alarm unit, since it also needs three signal wires, and aluminium boxes can be used for all units. The system overall design is visualised in Figure 11.8.

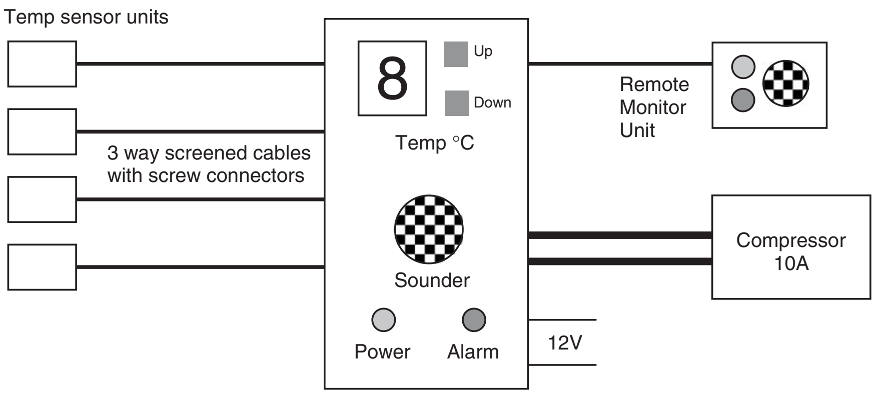


Figure 11.8 Refrigeration control system

Circuit Design and Firmware

The circuit will not be designed in detail, as most of the relevant features have been illustrated previously; this task will be assigned to the reader!

We are assuming a single supply of +5 V, derived from the 12 V vehicle battery, to which it must be permanently connected. A regulator providing sufficient current must be selected, and there will be a relatively high dissipation in this component since the volt drop across it will be 12–5 = 7 V. At a supply current of 1 A, 7 W will be dissipated, so a heat sink may be needed. A power budget should be calculated from the consumption of the main components when the circuit has been designed. The vehicle system can supply plenty of power, but a back-up battery could be considered in case the supply is disconnected e.g., the container is separated from the tractor unit.

The analogue conversion registers must be set up as required. Only 8-bit conversion will be needed, and a reference voltage of 2.56 V is recommended, since the input range is 20°C. If the amplifier output is 0.00–2.00 V, the low end will be below the alarm limit, and therefore does not need to be accurate. 0.40 V will represent 0°C, and 2.00 V 16°C, using single supply amplifiers as previously discussed. If 4 mA input current represents 0°C, a resistor load on the input of 100R (use a 1% resistor) will give the right scaling.

The display can use a program look-up table for the digit display codes 0−9. If the temperature goes to low, ‘L’ could be displayed as well as the alarm operating. Similarly, ‘H’ could be displayed if too high. The alarm sounds should use the hardware timers to generate suitable frequencies on the outputs, and the delay times for the flashing LED warnings.

The arithmetic processing should be straightforward, as only single digit numbers are in use. The temperature will be read in as an 8-bit number in the range 0−200. This should be divided by 10 (see Chapter 5) to obtain a number in the range 0−20; subtract 4 to calculate the temperature in the range −4 to +16. It will be easier to average at this stage, but more accurate with the original 8-bit data. Similarly, checking for a faulty sensor is easier at this stage. An open circuit sensor connection will give zero input, while the other sensors should read approximately the same value, so they can all be compared by subtraction, and a sensor failure indicated if the difference between any two is greater than, say, 3. If a sensor is mounted near the doors of the container, opening the doors should be detected by this alarm, which may be viewed as a useful additional feature! The average reading will be used to compare with the set temperature.

The set temperature should be displayed when the up or down button is pressed, and incremented by 1°C for each press. When released, the display should revert to the measured temperature. The set temperature should be

```
COLD1
        Refrigeration controller:
        Averages input from four temperature sensors
        checks for faulty sensor, averages and switches
        a relay output to the compressor

Initialise
        Analogue inputs (5)
                4 channels + Vref
        Digital Inputs (2)
                Up, Down
        Digital Outputs (8)
                Compressor
                Display (4)
                Power, Alarm LEDs
                Alarm Sounder

        Analogue control
        Timers
        Recall stored SetTemp

Main
        REPEAT
                Read inputs
                Check for faulty sensor
                        IF fault, set alarm
                Average inputs
                Check temperature
                        IF too high or too low, set alarm

                Check buttons
                        IF 'up' pressed
                                Increment SetTemp & store
                        IF 'down' pressed
                                Decrement SetTemp & store
                Display temperature

                Switch compressor on/off
        ALWAYS
```

Figure 11.9 Refrigeration controller program outline

stored in EEPROM when the button is released. This figure should be recalled during program initialisation.

The program outline is shown in Figure 11.9.

Other MCU Families

The PIC is our default choice of MCU type here, but if a given application demands it, the whole range of available devices from all manufacturers must be considered. The families of devices currently supported by Proteus are listed below, which gives an indication of the most popular types at the current time, at least in small embedded systems

- 8051
- ARM
- AVR
- HC11

Some conventional CPUs and supporting devices are also present, such as the Motorola 68000 and Zilog Z80, mainly for historical reasons. The main suppliers and their offerings are outlined below.

Intel/Philips

The 8051 type was the standard microcontroller for many years, originally developed by Intel in the 1980s alongside the 8085/6 range of PC processors which dominated the business computer market. The microcontroller shares the same assembly language with the Intel CPU range. More recently, the product range has been supplied by Philips and others. The basic 80C51 (C=CMOS) had 4k of mask or OTP program ROM, 128 bytes of RAM, four 8-bit ports, three 16-bit timers and serial UART. For application prototyping, an 8051 with EPROM program memory could be used. This memory type requires erasing by ultra-violet light, so is mounted on the chip behind a transparent window, making this type of chip easy to identify. It can then be reprogrammed, but this process is much less convenient than using flash ROM, the current technology for re-programmable MCUs.

ARM

In 1985, the UK Acorn Computer Group pioneered the development of the RISC (reduced instruction set computer) processor. This was based on analysis of program execution in CISC (complex instruction set computers), such as the 68000 and Intel CPUs, which showed that most of the time was spent executing a relatively small number of the most common instructions, such as moving data. It was decided that a CPU with a smaller instruction set, with the more complex operations made up from this reduced set as required, would be more efficient and faster. This proved to be the case, and a new branch of the microprocessor tribe was created, of which the PIC MCU is one family. In the US, Sun Microsystems developed the SPARC RISC CPU, which powered the ground-breaking high-performance Sun workstation. ARM technology is now licensed to manufacturers around the world. ARM processors are designed for the power CPU and MCU market, and only a limited number of their range are currently modelled in Proteus, but this selection will doubtless be expanded in due course.

Atmel

Atmel AVR is the most similar range to the PIC, in that it concentrates on 8-bit MCUs, and includes miniature devices. The smallest AVR MCU is currently the ATtiny11, with 1k flash program memory, 32 bytes of RAM, 6 I/O pins and a single analogue comparator in an 8-pin package. It has a more extensive instruction set than the PIC, based on the standard 8051 assembler instructions, and avoids the file register paging which is so inconvenient in the PIC. The AVR range has become popular as the next step up from the PIC, but currently Proteus support for AVR devices is limited to the 8051 flash derivatives, which it includes in its range. Currently, AVR MCUs are available in the following categories: Automotive, CAN (controller area network), LCD, lighting and battery controllers, and a 'mega' range which extends up to a 256k program memory device running at 16 MHz. In addition, AVR also produces high-performance MCUs based on the ARM processor core.

Motorola/Freescale

Motorola has always been a major player in the microprocessor field. Its most successful product may have been the 68000 CPU, which was the first popular 16-bit microprocessor, which was used in several different home computers, including the first Apple Mac, in the 1980s. The company has also always been prominent in embedded applications, producing its own mobile phones and similar products for many years. The HC11 type microcontroller is an 8-bit MCU based on the 68000 architecture and instruction set. That is, a complex instruction set that has multiple addressing modes. As it is only available with masked ROM program memory, it cannot be recommended for student projects and small-scale development work. Furthermore, it is now categorised as a legacy product, that is, in production to support existing products, and not recommended for new designs. Motorola MCUs now tend to be used in mass-produced, high-end products using 16-bit and 32-bit processors. Since 2004, Motorola embedded system components have been supplied by a spin-off company, Freescale. It currently claims world leadership in automotive and communications embedded applications, and number two spot in microcontrollers overall.

ST Microelectronics

Originally a French/Italian electronics company, SGS-Thomson, ST Microelectronics is well established in the automotive market, which accounts for a significant part of the growth in microcontroller applications. ST entered the low cost, flash program memory market relatively late, but offers a full range and a free C compiler (limited memory), so should be considered in any comparison of microcontroller suppliers.

SUMMARY 11

- The base module can be used as the basis for a range of applications
- It has a PIC 16F877, keypad, display and serial memory
- The parallel memory system provides up to 256k of conventional RAM
- The specification determines the choice of MCU in an embedded project

ASSESSMENT 11 — Total *(40)*

1	How are parallel, serial and analogue signals shown in a block diagram?	*(3)*
2	State three problems associated with a high-speed clock.	*(3)*
3	State three characteristics that a DC power supply must have.	*(3)*
4	State the function of an address decoder in a processor system.	*(3)*
5	Calculate the number of locations in a memory chip with 20 address pins. State the memory size.	*(3)*
6	Why would a program written for a 16F877 not work in a 10FXXXX chip?	*(3)*
7	Explain why an OTP chip is unsuitable for development work.	*(3)*
8	Identify an early Intel MCU, and describe its relationship with the standard PC.	*(3)*
9	Explain briefly how the superior performance of a RISC processor is achieved.	*(3)*
10	Identify a CPU used in the first generation of 16-bit home computers.	*(3)*
11	State the main criteria for selecting an MCU for a given application.	*(5)*
12	List 5 of the most significant global MCU manufacturers in 2006, other than Microchip.	*(5)*

ASSIGNMENTS 11

11.1 Weather Station

Design a remote weather station which can measure and display temperature, humidity, light and pressure. Specify the accuracy of each measurement. The data will be up-loaded to a PC for analysis and long-term storage every week. The data should not be lost if the power fails. Use the base module described in this chapter, and the interfaces described in Chapter 7.

11.2 Fridge Controller

Complete the implementation of the refrigeration controller specified in this chapter. Produce a schematic and demonstrate the simulation of the control program implemented in stages:

- Temperature control at default value
- Temperature display of default value
- Set temperature and display
- Sensor averaging and fault detection

Select the most appropriate PIC MCU for the final design and a costed parts list.

11.3 Multiprocessor Systems

Investigate the parallel serial port in the PIC 16F877, and show how it could be used for passing data between two PIC MCUs in a dual processor system. Compare this with SPI and I2C as multiprocessor communications systems, in terms of speed, flexibility and ease of hardware and software design.

ANSWERS TO ASSESSMENT QUESTIONS

Assessment 1

1 Processor, memory and I/O

2 A microcontroller has processor, memory and I/O on one chip, while the microprocessor needs separate memory and I/O chips to form a working system.

3 Output address from program counter on the address bus, select memory location containing instruction code and copy it back to the instruction register via the data bus.

4 Flash ROM can be electrically re-written many times, but is non-volatile.

5 The data has to be converted to serial form in a shift register and transmitted one bit at a time on a single line, while parallel data is transferred 8 (or more) bits at a time.

6 Ports A & E default to analogue input.

7 8k = 8192 = 8192 × 8 = 64536 bits

8 Place the chip in a programmer unit and open the application program in MPLAB. Assemble it to create the hex file. Select the programmer type and download.

9 From data sheet Table 13-2: Instruction code = 00 0000 1000 1100.
Therefore op-code = 0000001, register operand = 000 1100.

10 From data sheet Table 12-1: CP off = 11, ICD on = 0, BOD = 1, PWRT = 0, WDT = 0. Code = 11 0111 0111 0011 = 3773H.

11 A subroutine is a programmed jump (CALL) and return; the return address is stored automatically on the stack, so that when the routine has been completed, a RETURN instruction causes the return address to be replaced in the program counter, taking the execution point back to the instruction following the call.

The interrupt is an asynchronous external event which forces a jump to program address 004, from where an interrupt service routine is executed. This is terminated with RETFIE, return from interrupt, to take the execution point back to the original position. The stack is used in the same way as in the subroutine to store the return address.

12 A bit test is used to determine whether the next instruction is skipped, or not (BTFSS, BTFSC). This is usually followed by a GOTO or CALL, to change the program sequence. If this instruction is skipped, program execution continues on the original path. Often, the zero flag is tested to control a branch. The zero flag test is combined with a decrement or increment in DECFSZ and INCFSZ to provide counting loops and similar sequences.

Assessment 2

1 The program can be run, single stepped and paused in the actual target hardware, allowing hardware and timing faults to be identified as well a the usual syntax and logical errors; also the chip does not need to be removed from the application hardware once fitted, preventing possible damage.

2 001011, 10001100

3 END, EQU, PROCESSOR

4 GOTO, SLEEP; program will run through blank locations and repeat.

5 Address, register

6 Clock type, power up timer, watchdog timer

7 Assigns a label to a register

8 Program jumps to subroutine code, executes and returns; macro code is inserted each time by the assembler. Program is shorter, but slower, with the subroutine, and longer, but faster with the macro.

9 Standard header file contains a standard set of labels for registers and bits

10 END indicates end of source code to assembler

11 Start/end
Process/sequence
Input/output
Branch/selection
Subroutine/procedure/function

12
```
Main
          DO
                    If Reset pressed
                              Switch off LEDs
                    DO
                              Increment LEDs
                              Load Count
                              DO
                                        Decrement Count
                              WHILE Count not zero
                    WHILE run pressed
          ALWAYS
```

Assessment 3

1 Draw the schematic, attach the source code and assemble, and test by simulation.

2 The clock settings for simulation are set in the MCU component properties dialogue.

3 Clock = 10MHz, clock cycle time = 1/10 μs = 100ns, instruction time = 400ns.

4 Assembler: syntax errors. Simulation: logical errors.

5 *Step into* and then through a subroutine; *Step over* a subroutine, which is run at full speed, and continue to step after return; *Step out* of subroutine at full speed, then stop on return and resume stepping.

6 The program can be stopped at a particular point and the system status inspected; the program otherwise executes at full speed.

7 It is a digital multi-channel display which captures data at a known sampling rate from a group of data lines when triggered by a pre-set input combination.

8 Select simulation graph mode, draw a graph window, add signal probes to the circuit, drag these onto the graph, run and stop the simulation and hit the space bar to display the digital signals.

9 A netlist is a file which records the component connections in a circuit, which is used to generate a circuit layout.

10 Program can be tested in the final hardware, interacting with real components at relatively low cost.

11 The conventional process is to build prototype hardware, download the program to the MCU and test it in circuit. Simulation allows the design to be tested and debugged before building hardware. The schematic can then be converted into a netlist and a layout to produce the final PCB without prototyping.

12 Voltmeter – dc or ac volts. Oscilloscope – displays analogue signals at a range of frequencies. Logic Analyser – multiple digital signals displayed on the same time axis. The simulation graph can be expanded full screen for detailed analysis and printed.

Assessment 4

1 If the switch is connected between the input and 0V, the pull-up resistor ensures that the input is high when the switch is open.

2 Capacitor, software delay, timer delay

3 Hardware timers allow timing operations to proceed simultaneously with other program processes, giving a more efficient use of the processor.

4 The timer pre-scaler is a digital frequency divider which reduces the frequency of the input clock by a factor of 2, 4, 8 etc, which increases the timer range by the same factor.

5 The segments must be illuminated in the correct combination to display digits 0, 1, 2 etc. The data table provides the required binary output code for each digit displayed.

6 The BCD display has an internal hardware decoder so that it displays the digit corresponding to the input binary code (0 – 9).

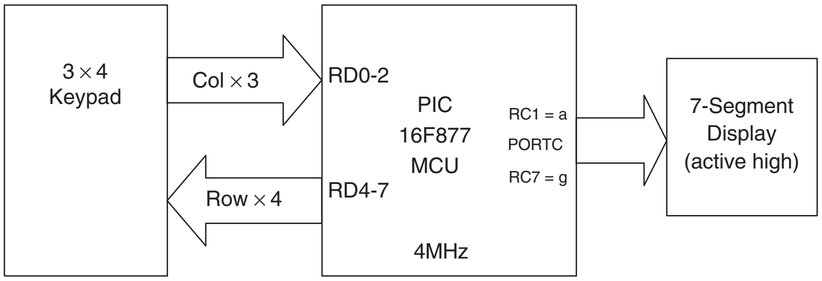
7 The rows are connected to MCU outputs and set high. The columns are connected to inputs, and pulled high. Each output is taken low in turn. If a key is pressed, a low input is detected on that column, identifying the key.

8 The LCD can operate with 4-bit input, receiving 8-bit control and data codes in 2 nibbles. An enable input strobes the data in, and a register select input indicates if the input code is a command or display data.

9 The LCD receives 8-bit command codes and ASCII character codes. RS is the register select input which directs these codes into the right register. The codes are loaded when the E input is pulsed.

10 23 h

11



12 The port can only be written with all 8 bits. If the high four bits are connected to the data inputs on the LCD, two of the low bits can be used for RS and E. When the data is written, the control bits must be modified individually after the data has been output. The data write must not cause unwanted command signal outputs.

Assessment 5

1 Approx. 18×10^{18}

2 41 h, 7Ah, 23 h

3 128 + 16 + 2 + 1 = 147d

4 Divide by 2: 617r0, 308r1, 154r0, 77r0, 38r1, 19r0, 9r1, 4r1, 2r0, 1r0, 0r1
Remainders in reverse order gives result: 10011010010

5 0011 1111 1011 0000 b, 16 304 d

6 Sign bit 1, Exponent 8, Mantissa 23

7 1001 × 0101 = 0101 + 0101000 = 101101

8 9 × 5 = 45 = 1 + 4 + 8 + 32

9 145 − 23 = 122 − 23 = 99 − 23 = 76 − 23 = 53 − 23 = 30 − 23 = 7
Answer = 6 remainder 7

10 99d = 1100011. 2s comp. = 0011100 + 1 = 001 1101 = 1Dh

11 Declare registers: Num1, Num2 (numbers) ResLo, ResHi
(results)

```
Clear ResLo and ResHi
Loop
    Add Num1 to ResLo
    If Carry set, increment ResHi
    Decrement Num2
Until Num2 = 0
```

12 Declare registers: Num1, Num2

```
Load integer into Num1
Complement Num1
Increment Num1
Add Num1 to Num2
Result in Num2
```

Assessment 6

1 The key code is obtained by taking the row low and checking the column input. If it is low, the ASCII code is loaded and the scan quit.

2 The code to operate the LCD only needs to be written once, saved and included into new programs as required, saving time and effort.

3 The negative result is detected when the carry flag is cleared. A minus sign character is displayed, and the inverse 2s complement of the result calculated and displayed.

4 In capture mode, a free-running timer value is captured and stored when a hardware input changes.

5 After setup, the program just waits for the compare mode interrupt from Timer 1, and the output is generated entirely within the interrupt service routine.

6 2710h is equal to 10000d. The timer is clocked at 1MHz, so the compare interrupt is generated after 10ms, giving the period of the output.

7 To restore the value of the compare value to zero when the decrement button has taken it negative. This prevents roll-under of the value.

8 Division can be carried out by repeat subtraction. The carry flag is set before the process to detect if the remainder has gone negative. When this happens, the result is corrected and stored.

9 In compare mode, a preset value is stored and continuously compared with a free running timer register. When they match after the fixed time, the timer interrupt flag triggers the required process.

10 PIR1, CCP1IF.

11 The number must be broken down into hundreds, tens and units by division. This can be achieved by repeat subtraction of 100, and 10, from the original value. The subtraction is controlled by monitoring the carry flag. When it is cleared, the result and remainder are corrected. The last remainder is the units value. 30h must then be added to the digit values to convert to ASCII (eg ASCII for 1 is 31h). These codes can then be sent to a suitable display, in the correct order.

12 A similar process is used to the above. This time, the maximum number obtained will be 65535, so the value is first divided by 10000, then 1000, then 100, then 10. The division result gives the corresponding denary digit, while the remainder is the units digit. The maximum result in each case is 9; each BCD value can be converted to ASCII and displayed.

Assessment 7

1 12-bit ADC gives $2^{12} = 4096$ steps. 100/4096 = 0.024% per step.

2 The full-scale input is divided into $2^8 = 256$ steps for conversion to binary. With a 2.56V reference, this converts into exactly 2.56 V / 256 = 10 mV per step.

3 Three bits are set up to select 1 of 8 input channels AN0 – AN7.

4 $2 \times 10 = 20$ µs conversion time gives maximum frequency of 1/20 MHz = 50 kHz.

5 If the 10-bit result is left justified, the high 8 bits of the ADC result are placed in the ADRESH register, with the low 2 in the high bits of ADRESL. If right justified, the low 8 bits are placed in ADRESL, and the high bits in the low 2 bits of ADRESH.

6 Gain and input resistance are infinite, output resistance is zero.

7 LM324 - common single supply (5 V) can be used – restricted output swing, may not reach zero.

8 $G = 19/1 + 1 = 20$

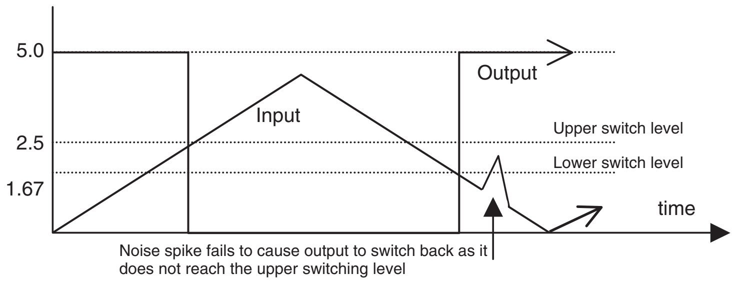
9 $Vs = 2(1.0 + 0.5) = 3.0$ V; $Vd = 2(1.0 - 0.5) = 1.0$ V

10 The capacitor slows down the output transient response, and reduces the cut off point in the frequency response.

11 Output polarity inverted.
Generate simultaneous equations from formula for inverting amp with offset:

$2.0 = (G + 1)Vr - 1.0G$	If $G = 4$
$0.0 = (G + 1)Vr - 1.5G$	$Vr = 6/5 = 1.2$ V
Subtract	and
$2 = (1 - 1.5).G$	$Ri = \frac{10k}{4} = 2k5$
$G = -4$	

12



Assessment 8

1 Emitter arrow-head shows direction of current, NPN out from base, PNP in.

Vbe ~ 0.6 V. Typical current gain = 100.

2 It is a voltage controlled current source, with high input impedance. Zero and +5 V applied at the gate will switch it off and on.

3 Relay contacts have a low on resistance and high off resistance, but the operating coil consumes significant power.

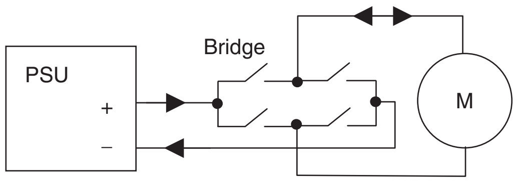
4 The DC motor needs a commutator to reverse the armature current on each half revolution, so that the torque is developed in one direction only.

5 The thyristor switches direct current only, while the triac switches alternating current.

6 The software option can be implemented by the MCU toggling an output with a delay. Alternatively, a separate hardware oscillator based on the 555 timer chip can be switched on an off by the MCU.

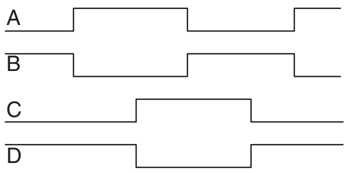
7 Pulse Width Modulation uses a pulse waveform to control a current switch connected to the load. If the ON time increases as a percentage of the overall period, the average current in the load, and hence the power dissipated, increases.

8 Bridge driver:



The switches in the bridge (FETs) are turned on in pairs to allow the current to flow in either direction in the motor.

9 The stepper motor has four sets of coils which are activated in pairs, to create a rotating magnetic field which operates the rotor.



10 360/15 = 24 steps/rev
Speed = 100 steps/sec → 100/24 = 4.04 revs/s

11 200 slots/100 ms → 2000 slots/s → 2000/50 = 40 revs/s → 40×60 = 2400 rpm.

12 The DC motor drive is simpler in construction, more efficient, and higher speeds and torque are possible, but it needs a feedback system for position control, and a gearbox for low speeds. The stepper can positioned without feedback, and holds its position, but is less inefficient and is complex to drive.

Assessment 9

1 No separate clock is sent with the data signal.

2 To increase the signal to noise ratio, and the distance sent, by increasing the signal amplitude.

3 10 (8 data bits + start + stop)

4 TX (TXD), RX (RXD); there are separate send and receive lines.

5 Line attenuation and noise limits the distance in proportion to the sending amplitude. SPI signals are sent at TTL levels (5 V) only, while RS232 uses amplitude up to 50 V p-p.

6 Slave select is a hardware input to an SPI device which enables slave transmission, generated by the master controller. I^{2}C uses software addressing, where the required device and location are selected by an address sent on the serial data line.

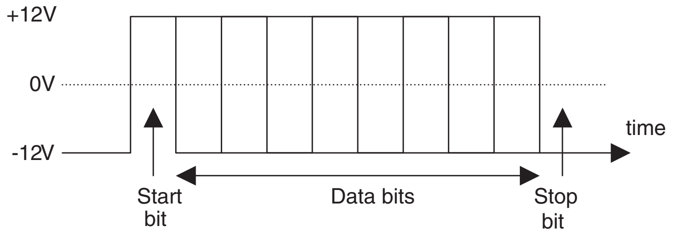
7 SSPIF (synchronous serial interface interrupt flag) is set.

8 In I^{2}C, a control code and address must be sent before the data, making up to 5 bytes in all, plus control bits. In SPI, only data bits are sent as the slave device is selected in hardware (slave select).

9 It holds the SDA line low for a bit cycle, which is detected by the master.

10 Only the start address is sent, and the memory automatically increments its internal address pointer to the next location to fetch the next byte.

11



12	Bits 7-4:	Slave device select code (1 of 16)
	Bits 4-1:	Hardware chip select address (1 of 8)
	Bit 0:	Read/!Write bit

Assessment 10

1 Gold plated contacts, operation in a vacuum or inert gas (reed switch), debouncing/snubbing with parallel capacitance/diodes, to reduce discharge and effect of back emf with inductive loads.

2 If a position sensing grating has a graduated transmission or reflectance (eg sinusoidal) when used with an optical sensor, intermediate positions can be calculated within each grid cycle if the sensor provides a suitable analogue output.

3 The rate of change of the output divided by the rate of change of the input, corresponding to the gradient of the characteristic.

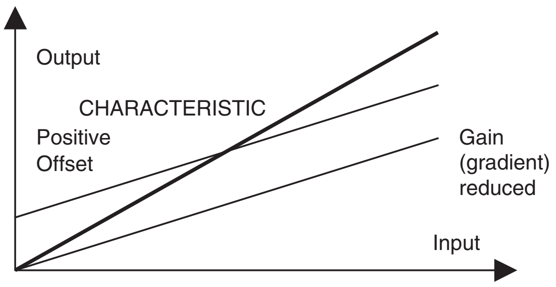
4 Accuracy is the extent to which a measurement is consistent with the agreed standard, precision is the smallest output change detectable; both may be expressed as a percentage.

5 Any 3 of: temperature sensing resistor (metal film), semiconductor junction (p-type and n-type silicon), thermocouple (dissimilar metals), thermistor (semiconductor), resistance (platinum).

6 Strain gauges are connected as a bridge circuit to provide a differential output which eliminates the large offset voltage when operated with a single supply, to maximise the output amplitude and to provide inherent temperature compensation.

7 $50/10 = 5$

8



9 The instrumentation amplifier is a differential configuration, which eliminates offset in the source voltage, has a high input impedance suitable for the high source impedance of the strain gauge bridge, and has a high gain suited to the low sensitivity of the bridge.

10 100 kΩ

11 A potentiometer can be used to measure the angular position of a shaft, and is simple, inexpensive and reasonably accurate. The digital method uses an incremental encoder, where pulses are counted as the shaft moves from a home position; this is easy to interface to an MCU, and is reliable.

12 Angular speed can be measured by a tachogenerator, which produces a voltage or current in proportion to the speed of its input shaft; speed can then be measured via an analogue input. The incremental encoder is used for speed measurement by measuring the frequency of the pulses, and is reliable and easier to interface as it does not need an analogue input.

Assessment 11

1 Parallel – block arrow, serial – single arrow, analogue – single arrow with labelling and optional representation of waveform.

2 High frequency interference with other components, high power dissipation, unreliable transmission down long connections.

3 Stable voltage, sufficient current, low noise

4 Selects an individual device to have access to a shared set of bus lines.

5 $2^{20} = 1048576$ locations $\rightarrow$ 1Mb assuming 8-bit locations.

6 The instruction set is not the same, and has a different instruction length.

7 One-time programmable chips cannot be re-programmed with a new version of the code.

8 The Intel 8051 MCU was developed from the 8085 CPU, and uses the same instruction set as the Intel CPUs used in PCs.

9 It has a simplified instruction set and structure, and high clock rate, for faster program execution.

10 Motorola 68000 CPU

11 Number of I/O pins, program memory size, peripherals available, data memory, instruction set, developer expertise, cost.

12 ARM, Atmel, Motorola/Freescale, ST Microelectronics, Philips

Index & Abbreviations