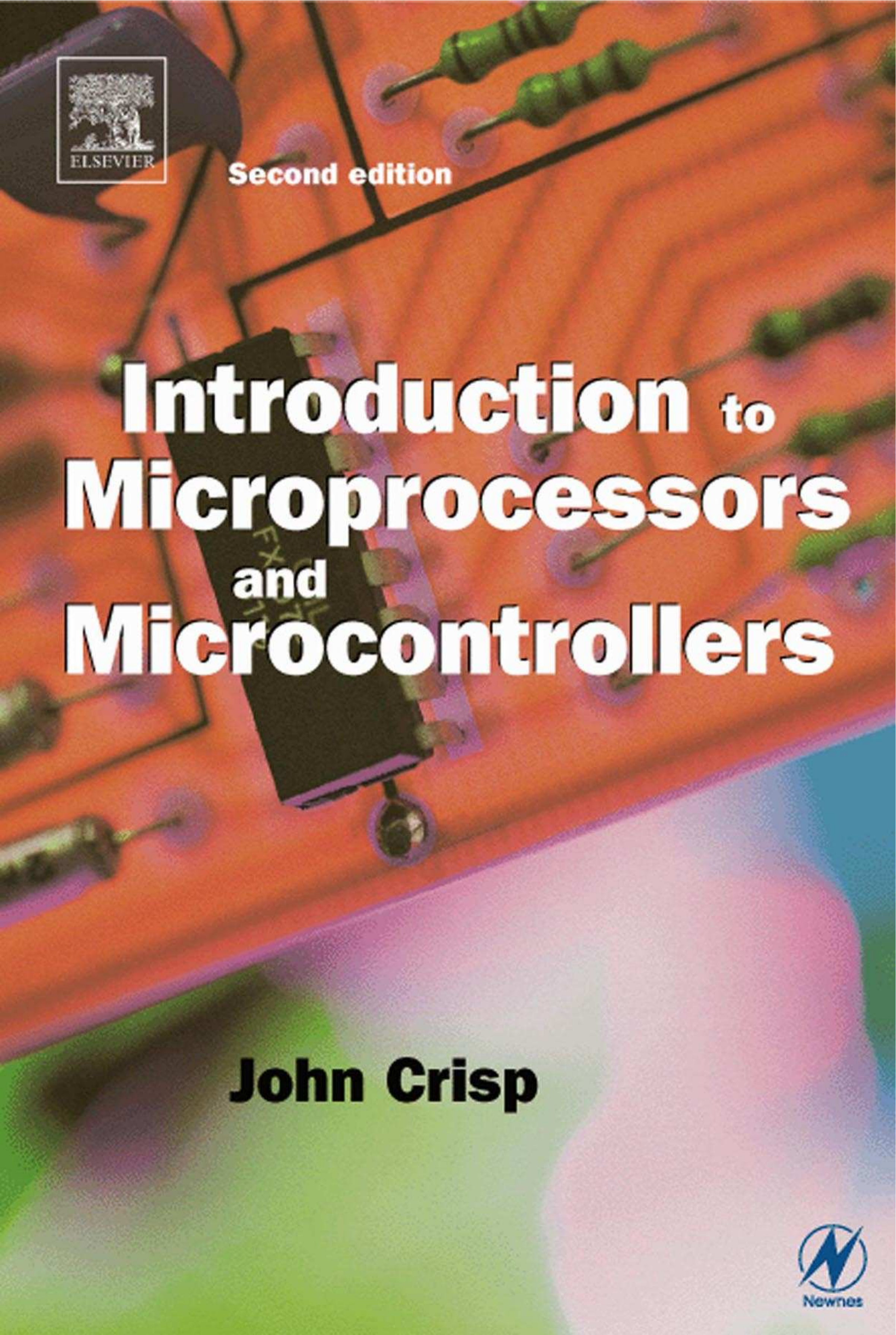


Second edition

Introduction to Microprocessors and Microcontrollers

John Crisp

Introduction to Microprocessors and Microcontrollers

This Page Intentionally Left Blank

Introduction to Microprocessors and Microcontrollers

Second edition

John Crisp

ELSEVIER

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

Newnes is an imprint of Elsevier

Newnes

Newnes
An imprint of Elsevier
Linacre House, Jordan Hill, Oxford OX2 8DP
200 Wheeler Road, Burlington MA 01803

First published 1998 as *Introduction to Microprocessors*
Reprinted 2000, 2001
Second Edition 2004

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

ISBN 0 7506 5989 0

Composition by Genesis Typesetting Limited, Rochester, Kent
Printed and bound in Great Britain

Contents

Preface

The first edition of this book started with the words: 'A modern society could no longer function without the microprocessor.'

This is certainly still true but it is even truer if we include the microcontroller. While the microprocessor is at the heart of our computers, with a great deal of publicity, the microcontroller is quietly running the rest of our world. They share our homes, our vehicles and our workplace, and sing to us from our greetings cards.

They are our constant, unseen companions and billions are being installed every year with little or no publicity.

The purpose of this book is to give a worry-free introduction to microprocessors and microcontrollers. It starts at the beginning and does not assume any previous knowledge of microprocessors or microcontrollers and, in gentle steps, introduces the knowledge necessary to take those vital first steps into the world of the micro.

John Crisp

This Page Intentionally Left Blank

1

Basic microprocessor systems

The microprocessor was born

In 1971 two companies, both in the USA, introduced the world to its future by producing microprocessors. They were a young company called Intel and their rival, Texas Instruments.

The microprocessor and its offspring, the microcontroller, were destined to infiltrate every country, every means of production, and almost every home in the world. There is now hardly a person on the planet that does not own or know of something that is dependent on one of these devices. Yet curiously, so few people can give any sort of answer to the simple question ‘What is a microprocessor?’ This, and ‘How does it work?’ form two of the starting points for this book.

Let’s start by looking at a system

The word ‘system’ is used to describe any organization or device that includes three features.

A system must have at least one input, one output and must do something, i.e. it must contain a process. Often there are many inputs and outputs. Some of the outputs are required and some are waste products. To a greater or lesser extent, all processes generate some waste heat. Figure 1.1 shows these requirements.

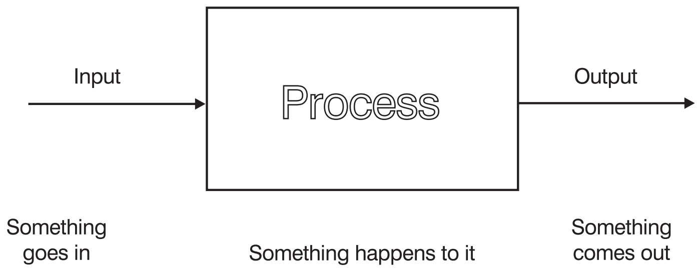


Figure 1.1
The essential requirements of a system

A wide range of different devices meets these simple requirements. For example, a motor car will usually require fuel, water for cooling purposes and a battery to start the engine and provide for the lights and instruments. Its process it to burn the fuel and extract the energy to provide transportation for people and goods. The outputs are the wanted movement and the unwanted pollutants such as gases, heat, water vapour and noise.

The motor car contains other systems within it. In Figure 1.2, we added electricity as a required input to start the engine and provide the lights

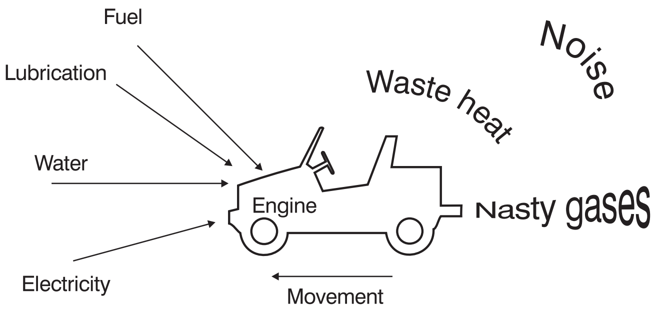


Figure 1.2
An everyday system

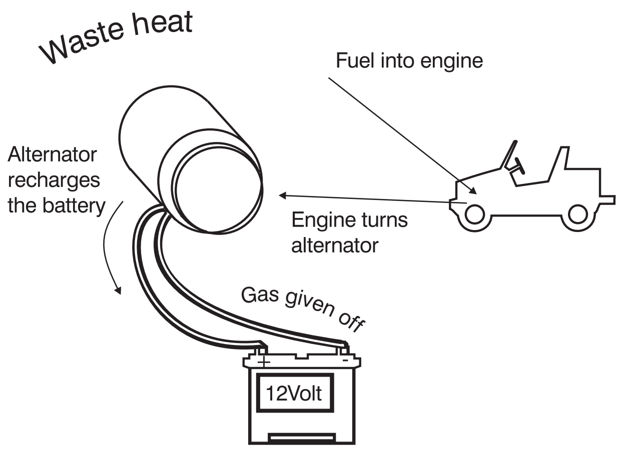


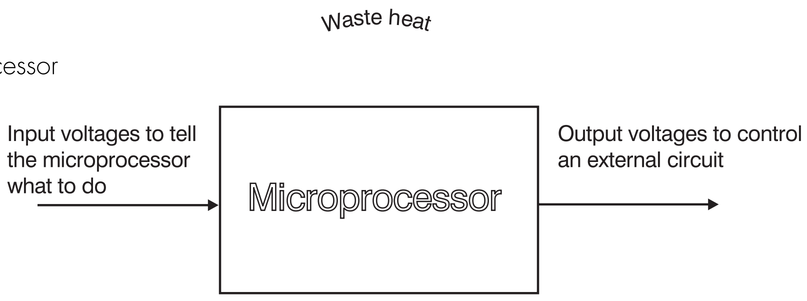
Figure 1.3
Recharging the battery

and the instruments but thereafter the battery is recharged by the engine. There must, then, be an electrical system at work, as in Figure 1.3, so it is quite possible for systems to have smaller systems inside or embedded within them. In a similar way, a motor car is just a part of the transport system.

A microprocessor system

Like any other system, a microprocessor has inputs, outputs and a process as shown in Figure 1.4. The inputs and outputs of a microprocessor are a series of voltages that can be used to control external devices. The process involves analysing the input voltages and using them to 'decide' on the required output voltages. The decision is based on previously entered instructions that are followed quite blindly, sensible or not.

Figure 1.4
The microprocessor system


Waste heat
Input voltages to tell the microprocessor what to do
Microprocessor
Output voltages to control an external circuit

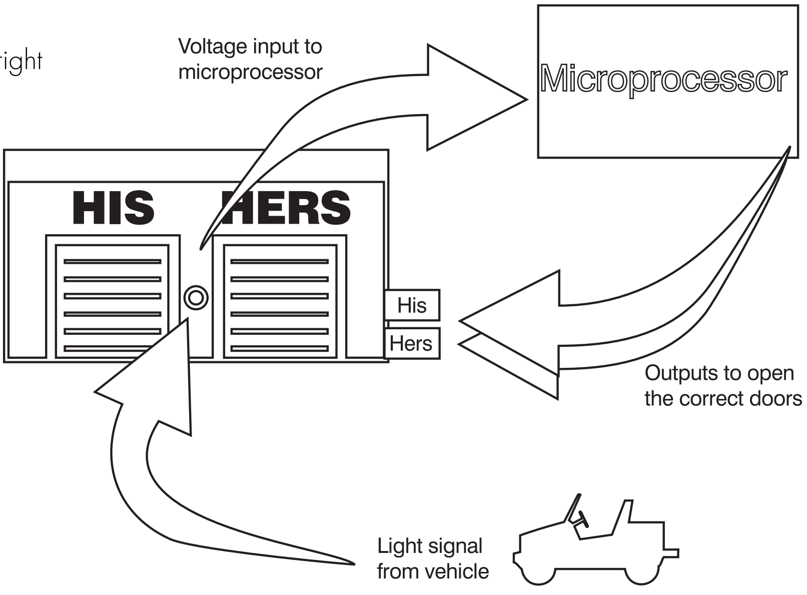
His and hers garage door opener

Here is a little task that a simple microprocessor can solve for us. When the woman arrives in her car, a light signal is flashed at the sensor and only her garage door opens. When the man arrives home, his car flashes a light signal at the same sensor but this time his garage door opens but hers remains closed.

The cars are sending a different sequence of light flashes to the light sensor. The light sensor converts the incoming light to electrical voltage pulses that are recognized by the microprocessor. The output voltage now operates the electrical motor attached to the appropriate door. The overall scheme is shown in Figure 1.5.

In the unlikely event of it being needed, a modern microprocessor would find it an easy task to increase the number of cars and garages to include every car and every garage that has ever been manufactured. Connecting all the wires, however, would be an altogether different problem!

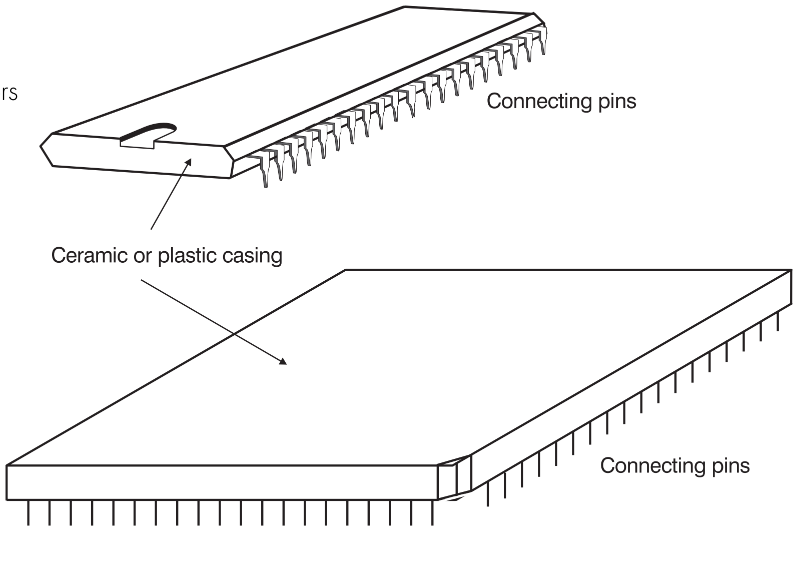
Figure 1.5
Opening the right garage door


Voltage input to microprocessor
Microprocessor
HIS
HERS
His
Hers
Outputs to open the correct doors
Light signal from vehicle

The physical appearance of a microprocessor

A microprocessor is a very small electronic circuit typically ½ inch (12 mm) across. It is easily damaged by moisture or abrasion so to offer it some protection it is encapsulated in plastic or ceramic. To provide electrical connections directly to the circuit would be impractical owing to the size and consequent fragility, so connecting pins are moulded into the case and the microprocessor then plugs into a socket

Figure 1.6
Typical microprocessors



on the main circuit board. The size, shape and number of pins on the microprocessor depend on the amount of data that it is designed to handle. The trend, as in many fields, is forever upward. Typical microprocessors are shown in Figure 1.6.

Terminology

Integrated circuits

An electronic circuit fabricated out of a solid block of semiconductor material. This design of circuit, often called a solid state circuit, allows for very complex circuits to be constructed in a small volume. An integrated circuit is also called a 'chip'.

Microprocessor (μp)

This is the device that you buy: just an integrated circuit as in Figure 1.6. On its own, without a surrounding circuit and applied voltages it is quite useless. It will just lie on your workbench staring back at you.

Microprocessor-based system

This is any system that contains a microprocessor, and does not necessarily have anything to do with computing. In fact, despite all the hype, computers use only a small proportion of all the microprocessors manufactured. Our garage door opening system is a microprocessor-based system or is sometimes called a microprocessor-controlled system.

Microcomputer

The particular microprocessor-based systems that happen to be used as a computer are called microcomputers. The additional circuits required for a computer can be built into the same integrated circuit giving rise to a single chip microcomputer.

Microcontroller

This is a complete microprocessor-based control system built onto a single chip. It is small and convenient but doesn't do anything that could not be done with a microprocessor and a few additional components. We'll have a detailed look at these in a later chapter.

MPU and CPU

An MPU is a MicroProcessor Unit or microprocessor. A CPU is a Central Processing Unit. This is the central 'brain' of a computer and can be (usually is) made from one or more microprocessors. The IBM design for the 'Blue Gene' supercomputer includes a million processors!

Remember:

MPU is the thing
CPU is the job.

Micro

The word micro is used in electronics and in science generally, to mean 'one-millionth' or 1×10^{-6}. It has also entered general language to mean something very small like a very small processor or microprocessor. It has also become an abbreviation for microprocessor, microcomputer, microprocessor-based system or a micro controller – indeed almost anything that has 'micro' in its name. In the scientific sense, the word micro is represented by the Greek letter μ (mu). It was only a small step for microprocessor to become abbreviated to μP.

Some confusion can arise unless we make sure that everyone concerned is referring to the same thing.

Quiz time 1

In each case, choose the best option.

1 A microprocessor:

(a) requires fuel, water and electricity.
(b) is abbreviated to μc.
(c) is often encapsulated in plastic.
(d) is never used in a CPU but can be used in an MPU.

2 A system must include:

(a) an input, an output and a process.
(b) something to do with a form of transport.
(c) a microprocessor.
(d) fuel, water and electricity.

3 All systems generate:

(a) movement.
(b) chips.
(c) waste heat.
(d) waste gases.

4 An MPU:

(a) is the same as a μP.
(b) can be made from more than one Central Processing Unit.
(c) is a small, single chip computer.
(d) is an abbreviation for Main Processing Unit.

5 Integrated circuits are *not*:

(a) called chips.
(b) used to construct a microprocessor-based system.
(c) solid state circuits.
(d) an essential part of an engine.

2

Binary – the way micros count

Unlike us, microprocessors have not grown up with the idea that 10 is a convenient number of digits to use. We have taken it so much for granted that we have even used the word digit to mean both a finger and a number.

Microprocessors and other digital circuits use only two digits – 0 and 1 – but why? Ideally, we would like our microprocessors to do everything at infinite speed and never make a mistake. Error free or high speed – which would you feel is the more important?

It's your choice but I would go for error free every time, particularly when driving my car with its engine management computer or when coming in to land in a fly-by-wire aircraft. I think most people would agree.

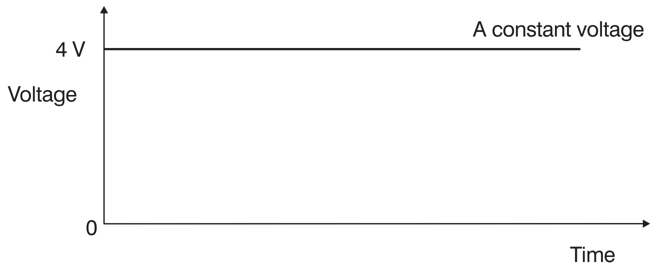
So let's start by having a look at one effect of persuading microprocessors to count in our way.

The noise problem

If the input of a microprocessor is held at a constant voltage, say 4 V, this would appear as in Figure 2.1.

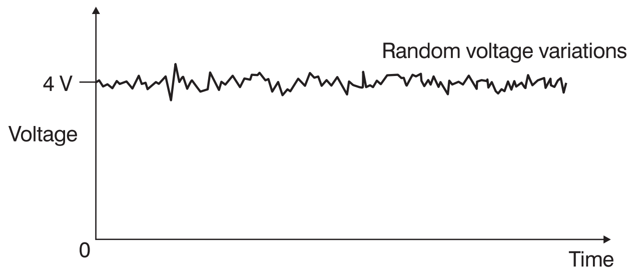
If we try to do this in practice, then careful measurements would show that the voltage is not of constant value but is continuously wandering

Figure 2.1
A constant voltage



above and below the mean level. These random fluctuations are called electrical noise and degrade the performance of every electronic circuit. We can take steps to reduce the effects but preventing it altogether is, so far, totally impossible. We can see the effect by disconnecting the antenna of our television. The noise causes random speckles on the screen which we call snow. The same effect causes an audible hiss from the loudspeaker. The effect of noise is shown in Figure 2.2.

Figure 2.2
A 'noisy' voltage



Most microprocessors use a power supply of 5 V or 3.3 V. To keep the arithmetic easy, we will assume a 5 V system.

If we are going to persuade the microprocessor to count from 0 to 9, as we do, using voltages available on a 5 V supply would give 0.5 V per digit:

0	=	0 V
1	=	0.5 V
2	=	1 V
3	=	1.5 V
4	=	2 V
5	=	2.5 V
6	=	3 V
7	=	3.5 V
8	=	4 V
9	=	4.5 V

If we were to instruct our microprocessor to perform the task 4 + 4 = 8, by pressing the '4' key we could generate a 2 V signal which is then remembered by the microprocessor. The + key would tell it to add and pressing the '4' key again would then generate another 2 V signal.

So, inside the microprocessor we would see it add the 2 V and then another 2 V and, hence, get a total of 4 V. The microprocessor could then use the list shown to convert the total voltage to the required numerical result of 8. This simple addition is shown in Figure 2.3.

This seemed to work nicely – but we ignored the effect of noise. Figure 2.4 shows what could happen. The exact voltage memorized by the microprocessor would be a matter of chance. The first time we pressed

Figure 2.3
It works! 4 + 4 does equal 8

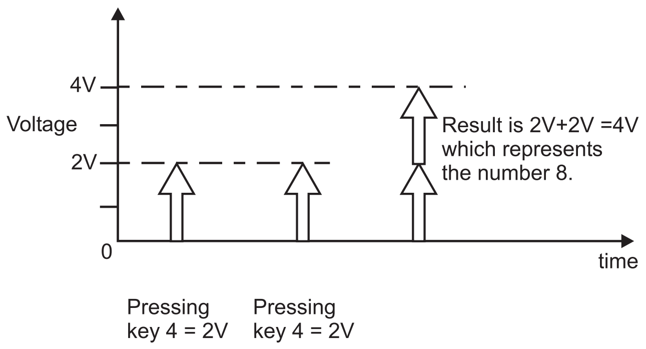
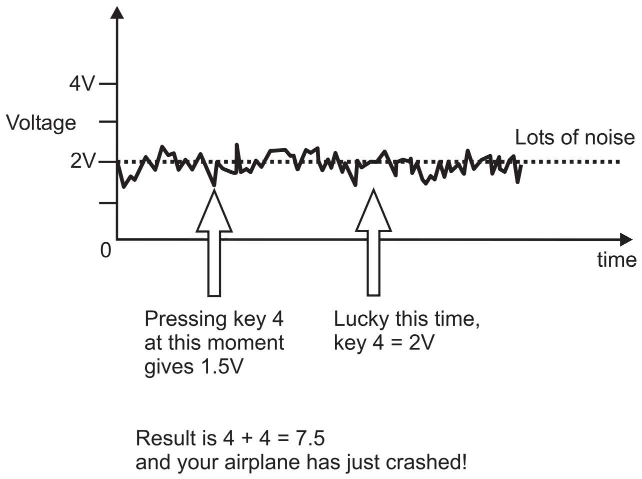


Figure 2.4
Noise can cause problems


4V
Voltage
2V
0
time
Lots of noise
Pressing key 4 at this moment gives 1.5V
Lucky this time, key 4 = 2V
Result is 4 + 4 = 7.5 and your airplane has just crashed!

key 4, the voltage just happened to be at 1.5 V but the second time we were luckier and the voltage was at the correct value of 2 V.

Inside the microprocessor:

$$1.5\,\text{V} + 2\,\text{V} = 3.5\,\text{V}$$

and using the table, the 3.5 V is then converted to the number 7. So our microprocessor reckons that 4 + 4 = 7.5!

Since the noise is random, it is possible, of course, to get a final result that is too low, too high or even correct.

A complete cure for electrical noise

Sorry, just dreaming. There isn't one. The small particle-like components of electricity, called electrons, vibrate in a random fashion powered by the surrounding heat energy. In conductors, electrons are very mobile and carry a type of electrical charge that we have termed negative. The resulting negative charge is balanced out by an equal number of fixed particles called protons, which carry a positive charge (see Figure 2.5).

The overall effect of the electron mobility is similar to the random surges that occur in a large crowd of people jostling around waiting to enter the stadium for the Big Match. If, at a particular time, there happens to be more electrons or negative charges moving towards the

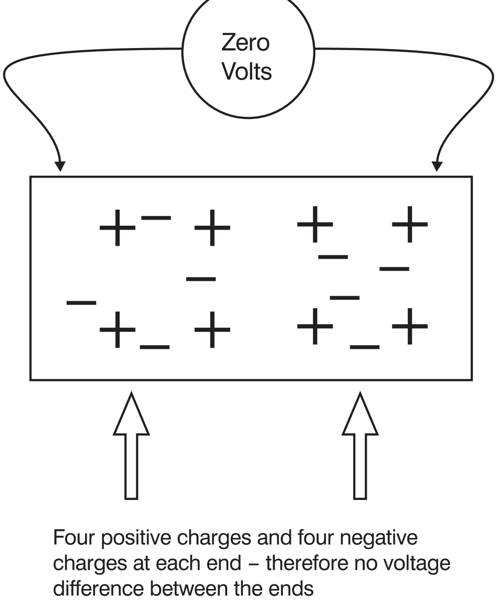
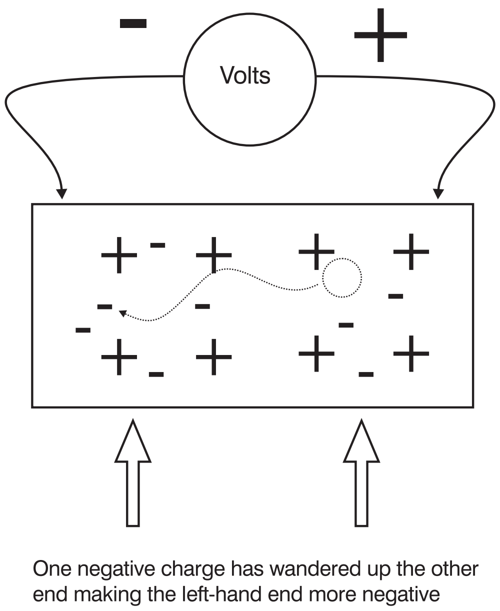

Zero Volts
Four positive charges and four negative charges at each end – therefore no voltage difference between the ends

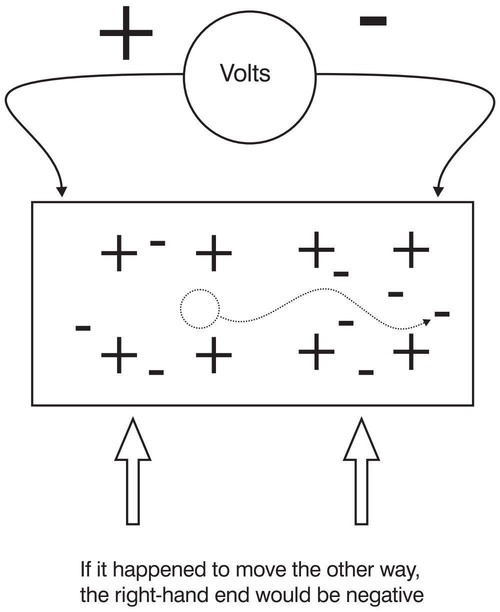
Figure 2.5
Equal charges result in no overall voltage

Figure 2.6
A random voltage has been generated


-
+
Volts
One negative charge has wandered up the other end making the left-hand end more negative

left-hand end of a piece of material then that end would become more negative, as shown in Figure 2.6. A moment later, the opposite result may occur and the end would become more positive (Figure 2.7). These effects give rise to small random voltages in any conductor, as we have seen.

Figure 2.7
The opposite effect is equally likely


+
-
Volts
If it happened to move the other way, the right-hand end would be negative

Thermal noise

The higher the temperature, the more mobile the electrons, the greater the random voltages and the more electrical noise is present.

A solution:

High temperature = high noise

so:

Low temperature = low noise.

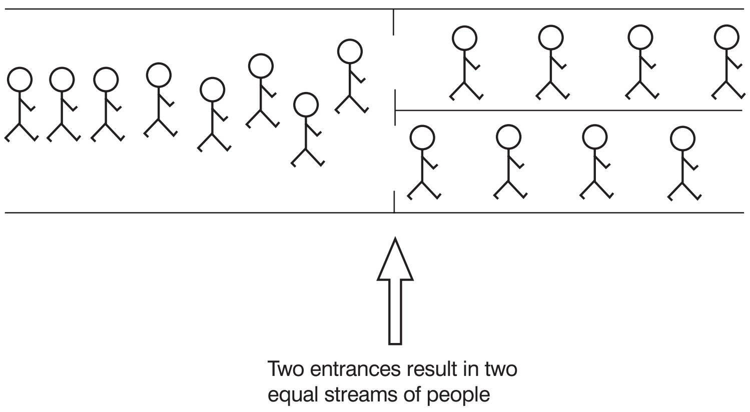
Put the whole system into a very cold environment by dropping it in liquid nitrogen (about −200°C) or taking it into space where the 'shade' temperature is about −269°C. The cold of space has created very pleasant low noise conditions for the circuits in space like the Hubble telescope. On Earth most microprocessors operate at room temperature. It would be inconvenient, not to mention expensive, to surround all our microprocessor circuits by liquid nitrogen. And even if we did, there is another problem queuing up to take its place.

Partition noise

Let's return to the Big Match. Two doors finally open and the fans pour through the turnstiles. Now we may expect an equal number of people to pass through the two entrances as shown in Figure 2.8 but in reality this will not happen. Someone will have trouble finding their ticket; friends will wait for each other; cash will be offered instead of a ticket; someone will try to get back out through the gate to reach another section of the stadium. As we can imagine, the streams of people may be equal over an hour but second by second random fluctuations will occur.

Electrons don't lose their tickets but random effects like temperature, voltage and interactions between adjacent electrons have a very similar effect.

Figure 2.8
The fans enter the stadium



A single current of, say, 1 A can be split into two currents of 0.5 A when measured over the long-term, but when examined carefully, each will contain random fluctuations. This type of electrical noise is called partition noise or partition effect. The overall effect is similar to the thermal noise and, between them, would cause too much noise and hence would rule out the use of a 10-digit system.

How much noise can we put up with?

The 10-finger system that we use is called a 'denary' or 'decimal' system. We have seen that a 5 V supply would accommodate a 10-digit counting system if each digit was separated by 0.5 V or, using the more modern choice of 3.3 V, the digits would be separated by only 0.33 V.

Question: Using a 5 V supply and a denary system, what is the highest noise voltage that can be tolerated?

Answer: Each digit is separated by only 5 V/10 = 0.5 V.

The number 6 for example would have a value of 3 V and the number 7 would be represented by 3.5 V. If the noise voltage were to increase the 3 V to over 3.25 V, the number is likely to be misread as 7. The highest acceptable noise level would therefore be 0.25 V. This is not very high and errors would be common. If we used a supply voltage of 3.3 V, the situation would get even worse.

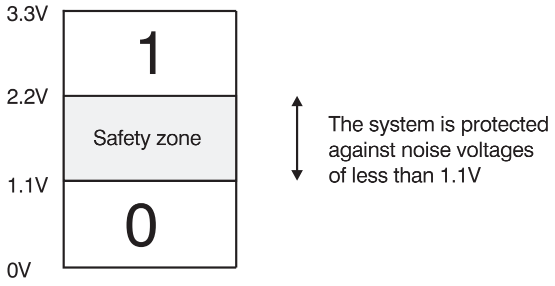
So why don't we just increase the operating voltage to say, 10 V, or 100 V? The higher the supply voltage the less likely it is that electrical noise would be a problem. This is true but the effect of increasing the supply would be to require thicker insulation and would increase the physical size of the microprocessor and reduce its speed. More about this in Chapter 11.

Using just two digits

If we reduce the number of digits then a wider voltage range can be used for each value and the errors due to noise are likely to occur less often.

We have chosen to use only two digits, 0 and 1, to provide the maximum degree of reliability. A further improvement is to provide a safety zone between each voltage. Instead of taking our supply voltage of 3.3 V and simply using the lower half to represent the digit 0 and the top half for 1, we allocate only the lower third to 0 and the upper third to 1 as shown in Figure 2.9. This means that the noise level will have to be at least 1.1 V (one-third of 3.3 V) to push a level 0 digit up to the minimum value for a level 1.

Figure 2.9
A better choice of voltages


3.3V
1
2.2V
Safety zone
1.1V
0
0V
The system is protected against noise voltages of less than 1.1V

The voltages chosen to represent the digits 0 and 1

How do we count?

Normally, we count in the system we call 'denary'. We start with 0 then go to 1 then to a new symbol that we write as 2 and call 'two'. This continues until we run out of symbols. So far, it looks like this:

0
1
2
3
4
5
6
7
8
9

At this point we have used all the symbols once and, to show this, we put a '1' to the left of the numbers as we re-use them. This gives us:

10
11
12
13
14

. . . and so on up to 19 when we put a 2 on the left-hand side and start again 20, 21, 22 etc.

When we reach 99, we again add a '1' on the left-hand side and put the other digits back to zero to give 100. After we reach 999, we go to 1000 and so on.

Counting is not easy. We often take it for granted but if we think back to our early days at school, it took the teacher over a year before we were happy and reasonably competent. So counting is more difficult than microprocessors – you've mastered the difficult part already!

The basic basis of bases

The base of a number system is the number of different symbols used in it. In the case of the denary system, we use 10 different symbols, 0 . . . 9, other numbers, like 28 657, are simply combinations of the 10 basic symbols.

Since the denary system uses 10 digits, the system is said to have a base of 10. The base is therefore just the technical word for the number of digits used in any counting system.

Counting with only two figures

We can count using any base that we like. In the denary or decimal system, we used a base of 10 but we have seen that microprocessors use a base of 2 – just the two digits 0 and 1. This is called the binary system.

We usually abbreviate the words BInary digiT to bit.

Counting follows the same pattern as we have seen in the denary system: we use up the digits then start again.

Let's give it a try. Start by listing all the digits:

0
1

and that's it!

We now put a '1' in the next column and start again:

10
11

It is convenient at this stage to keep the number of binary columns the same and so we add a 0 at the start of the first two digits. These extra zeros do not alter the value at all. For example, the denary number 25 is not affected by writing it as 025 or 0025 or even 000 000 000 000 025.

The binary and decimal equivalents are:

Binary	Denary
00	0
01	1
10	2
11	3

We do the same again – put a '1' in the next column and repeat the pattern to give:

Binary	Denary
100	4
101	5
110	6
111	7

and once more:

Binary	Denary
1000	8
1001	9
1010	10
1011	11

Confusion and the cure

Here is a number: 1000. But what number is it? Is it a thousand in denary or is it eight written in binary?

I don't know. I could take a guess but the difference between flying an aircraft at eight feet and a thousand feet is a serious matter. The only way to be certain is to say so at the time. This is done by showing the base of the number system being used to make the meaning quite clear. The base of the number system is shown as a subscript after the number.

If the 1000 were a binary number, it is written as 1000_2 and if it were a denary number it would be shown as 1000_{10}.

It would be easy to advise that the base of the number system in use is always shown against every number but this would be totally unrealistic. No one is going to write a base after their telephone number or a price in a shop. Use a base when it would be useful to avoid confusion, such as by writing statements like 1000 = 8 (a thousand = eight???). Write it as $1000_2 = 8_{10}$ and make life a little easier.

Converting denary to binary

Of course, if someone were to ask us for the binary equivalent of nine we could just start from zero and count up until we reach nine. This is a boring way to do it and with larger numbers like $1\,000\,000_{10}$ it would be very tedious indeed. Here is a better way. The method will be explained using the conversion of 52_{10} to binary as an example.

A worked example

Convert 52_{10} to binary

Step 1: Write down the number to be converted

```
52
```

Step 2: Divide it by 2 (because 2 is the base of the binary system), write the whole number part of the answer underneath and the remainder 0 or 1 alongside

```
52
26  0
```

Step 3: Divide the answer (26) by 2 and record the remainder (0) as before

```
52
26  0
13  0
```

Step 4: Divide the 13 by 2 and write down the answer (6) and the remainder (1)

```
52
26  0
13  0
 6  1
```

Step 5: 2 into 6 goes 3 remainder 0

```
52
26  0
13  0
 6  1
 3  0
```

Step 6: Dividing 3 gives an answer of 1 and a remainder of 1

```
52
26  0
13  0
 6  1
 3  0
 1  1
```

Step 7: Finally, dividing the 1 by 2 will give 0 and a remainder of 1

```
52
26  0
13  0
 6  1
 3  0
 1  1
 0  1
```

Step 8: We cannot go any further with the divisions because all the answers will be zero from now on. The binary number now appears in the remainder column. To get the answer read the remainder column from the bottom UPWARDS

$$\begin{array}{rrl} 52 & & \\ 26 & 0 & \quad = 110100_2 \\ 13 & 0 & \uparrow \\ 6 & 1 & | \\ 3 & 0 & | \\ 1 & 1 & \\ 0 & 1 & \end{array}$$

Method

1 Divide the denary number by 2 – write the whole number result underneath and the remainder in a column to the right.

2 Repeat the process until the number is reduced to zero.

3 The binary number is found by reading the remainder column from the bottom upwards.

Another example

Here is one for you to try. If you get stuck, the solution is given below.

Convert 2187_{10} to a binary number

$$\begin{array}{rrl} 2187 & & \\ 1093 & 1 & = 100010001011_2 \\ 546 & 1 & \qquad\quad \uparrow \\ 273 & 0 & \qquad\quad | \\ 136 & 1 & \qquad\quad | \\ 68 & 0 & \qquad\quad | \\ 34 & 0 & \qquad\quad | \\ 17 & 0 & \\ 8 & 1 & \\ 4 & 0 & \\ 2 & 0 & \\ 1 & 0 & \\ 0 & 1 & \end{array}$$

Doing it by calculator: Many scientific calculators can do the conversion of denary to binary for us. Unfortunately, they are limited to quite low numbers by the number of digits able to be seen on the screen.

To do a conversion, we need:

1 A scientific calculator that can handle different number bases.
2 The instruction booklet.

3 About half an hour to spare – or a week if you have lost the instructions.

The exact method varies but on my elderly Casio it goes something like this:

> To tell the calculator that the answer has to be in binary I have to press mode mode 3 then the 'binary' key.
>
> It now has to be told that the input number is decimal. This is the exciting key sequence logic logic logic 1 now just put in our number 52 and press the = key and out will pop the answer 110100.

Converting binary to denary

If we look at a denary number like 8328, we see that it contains two eights. Now these two figures look identical however closely we examine them, but we know that they are different. The 8 on the right-hand end is really 8 but the other one is actually 8000 because it is in the thousands column.

The real value of a digit is dependent on two things: the digit used and the column in which it is placed.

In the denary system, the columns, starting from the right, are units, tens, hundreds, thousands etc. Rather than use these words, we could express them in powers of ten. A thousand is $10 \times 10 \times 10 = 10^3$ and in a similar way, a hundred is 10^2, ten is 10^1 and a unit is 10^0. Each column simply increases the power applied to the base of the number system.

Columns in a binary world also use the base raised to increasing powers as we move across the columns towards the left.

So we have:

$2^3 \quad 2^2 \quad 2^1 \quad 2^0$

The denary equivalent can be found by multiplying out the powers of two. So 2^3 is $2 \times 2 \times 2 = 8$ and $2^2 = 4$, $2^1 = 2$ and finally $2^0 = 1$. Starting from the right-hand side, the column values would be 1, 2, 4, 8 etc. Let's use this to convert the binary number 1001 into denary.

Method

Step 1: Write down the values of the columns

```
8 4 2 1
```

Step 2: Write the binary number underneath

```
8 4 2 1
1 0 0 1
```

Step 3: Evaluate the values of the columns

$$8 \times 1 = 8$$
$$4 \times 0 = 0$$
$$2 \times 0 = 0$$
$$1 \times 1 = 1$$

Step 4: Add up the values

$$8 + 1 = 9$$

As we have seen, all the columns containing a binary 0 can be ignored because they always come out to 0 so a quicker way is to simply add up all the column values where the binary digit is 1.

Method

1. Write down the column values for the binary system using the same number of columns as are shown in the binary number.
2. Enter the binary number, one bit under each column heading.
3. Add the values of each column where a 1 appears in the binary number.

Calculator note: This is much the same as we saw the previous conversion. To tell the calculator that the answer has to be in decimal I have to press mode mode 3 then the 'decimal' key.

It now has to be told that the input number is binary. This is done by the key sequence logic logic logic 3 now just put in our binary number 1001 and press the = key and out will pop the answer 9.

Another example

Once again, here is one for you to try. If you have problems, the answer follows.

Convert 101100101_2 to a denary number

Step 1: Write down the column values by starting with a 1 on the right-hand side then just keep doubling as necessary

2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
256	128	64	32	16	8	4	2	1

Step 2: Enter the binary number under the column headings

256	128	64	32	16	8	4	2	1
1	0	1	1	0	0	1	0	1

Step 3: Add up all the column values where the binary digit is 1

$$256 + 64 + 32 + 4 + 1 = 357$$

So, $101100101_2 = 357_{10}$ or just 357 since denary can be assumed in this case.

Bits, bytes and other things

All the information entering or leaving a microprocessor is in the form of a binary signal, a voltage switching between the two bit levels 0 and 1.

Bits are passed through the microprocessor at very high speed and in large numbers and we find it easier to group them together.

Nibble

A group of four bits handled as a single lump. It is half a byte.

Byte

A byte is simply a collection of 8 bits. Whether they are ones or zeros or what their purpose is does not matter.

Word

A number of bits can be collected together to form a 'word'. Unlike a byte, a word does not have a fixed number of bits in it. The length of the word or the number of bits in the word depends on the microprocessor being used.

If the microprocessor accepts binary data in groups of 32 at a time then the word in this context would include 32 bits. If a different microprocessor used data in smaller handfuls, say 16 at a time, then the word would have a value of 16 bits. The word is unusual in this context in as much as its size or length will vary according to the situations in which it is discussed. The most likely values are 8, 16, 32 and 64 bits but no value is excluded.

Long word

In some microprocessors where a word is taken to mean say 16 bits, a long word would mean a group of twice the normal length, in this case 32 bits.

Kilobyte (Kb or KB or kbyte)

A kilobyte is 1024 or 2^{10} bytes. In normal use, kilo means 1000 so a kilovolt or kV is exactly 1000 volts. In the binary system, the nearest column value to 1000 is 1024 since $2^9 = 512$ and $2^{10} = 1024$.

The difference between 1000 and 1024 is fairly slight when we have only 1 or 2 Kb and the difference is easily ignored. However, as the numbers increase, so does the difference. The actual number of bytes in 42 Kb is actually 43 008 bytes (42×1024).

The move in the computing world to use an upper case K to mean 1024 rather than k for meaning 1000 is trying to address this problem.

Unfortunately, even the upper or lower case b is not standardized so tread warily and look for clues to discover which value is being used. If in doubt use 1024 if it is to do with microprocessors or computers.

Bits often help to confuse the situation even further. 1000 bits is a kilobit or kb. Sometimes 1024 bits is a Kb. One way to solve the bit/byte problem is to use kbit (or Kbit) and kbyte (or Kbyte).

Megabyte (MB or Mb)

This is a kilokilobyte or 1024 × 1024 bytes. Numerically this is 2^{20} or 1 048 576 bytes. Be careful not to confuse this with mega as in megavolts (MV) which is exactly one million (10^6).

Gigabyte (Gb)

This is 1024 megabytes which is 2^{30} or 1 073 741 824 bytes. In general engineering, giga means one thousand million (10^9).

Terabyte (TB or Tb)

Terabyte is a megamegabyte or 2^{40} or 1 099 511 600 000 bytes (Tera = 10^{12}).

Petabyte (PB or Pb)

This is a thousand (or 1024) times larger than the Terabyte so it is 10^{15} in round numbers or 2^{40} which is pretty big. If you are really interested, you can multiply it out yourself by multiplying the TB figure by 1024.

Quiz time 2

In each case, choose the best option.

1 Typical operating voltages of microprocessors are:

(a) 0 V and 1 V.
(b) 3.3 V and 5 V.
(c) 220 V
(d) 1024 V.

2 The most mobile electrical charge is called:

(a) a proton and has a positive charge.
(b) a voltage and is always at one end of a conductor.
(c) an electron and has a negative charge.
(d) an electron and has a positive charge.

3 The denary number 600 is equivalent to the binary number:

(a) 1001011000.
(b) 011000000000.
(c) 1101001.
(d) 1010110000.

4 When converted to a denary number, the binary number 110101110:

(a) will end with a 0.
(b) must be greater than 256 but less than 512.
(c) will have a base of 2.
(d) will equal 656.

5 A byte:

(a) is either 1024 or 1000 bits.
(b) is simply a collection of 16 bits.
(c) can vary in length according to the microprocessor used.
(d) can have the same number of bits as a word.

3

Hexadecimal – the way we communicate with micros

The only problem with binary

The only problem with binary is that we find it so difficult and make too many errors. There is little point in designing microprocessors to handle binary numbers at high speed and with almost 100% accuracy if we are going to make loads of mistakes putting the numbers in and reading the answers.

From our point of view, binary has two drawbacks: the numbers are too long and secondly they are too tedious. If we have streams and streams of ones and zeros we get bored, we lose our place and do sections twice and miss bits out.

The speed of light in m/s can be written in denary as 299792459_{10} or in binary as $10001110111100111100001001011_2$. Try writing these numbers on a sheet of paper and we can be sure that the denary number will be found infinitely easier to handle. Incidentally, this binary number is less than half the length that a modern microprocessor can handle several millions of times a second with (almost) total accuracy.

In trying to make a denary number even easier, we tend to split it up into groups and would write or read it as 299 792 459. In this way, we are dealing with bite-sized portions and the 10 different digits ensure that there is enough variety to keep us interested.

We can perform a similar trick with binary and split the number into groups of four bits starting from the right-hand end as we do with denary numbers.

```
1   0001   1101   1110   0111   1000   0100   1011
```

Already it looks more digestible.

Now, if we take a group of four bits, the lowest possible value is 0000_2 and the highest is 1111_2. If these binary numbers are converted to denary, the possibilities range from 0 to 15.

Hexadecimal, or 'hex' to its friends

Counting from 0 to 15 will mean 16 different digits and so has a base of 16. What the digits look like really doesn't matter. Nevertheless, we may as well make it as simple as possible.

The first 10 are easy, we can just use 0123456789 as in denary. For the last six we have decided to use the first six letters of the alphabet: ABCDEF or abcdef.

The hex system starts as:

Hex	Denary
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

When we run out of digits, we just put a 1 in the second column and reset the first column to zero just as we always do.

So the count will continue:

10	16
11	17
12	18
13	19
14	20
15	21
16	22
17	23
18	24
19	25
1A	26
1B	27
1C	28
1D	29
1E	30
1F	31
20	32

. . . and so on.

It takes a moment or two to get used to the idea of having numbers that include letters but it soon passes. We must be careful to include the base whenever necessary to avoid confusion. The base is usually written as H, though h or 16 would still be acceptable.

'One eight' in hex is equal to twenty-four in denary. Notice how I avoided quoting the hex number as eighteen. Eighteen is a denary number and does not exist in hex. If you read it in this manner it reinforces the fact that it is not a denary value.

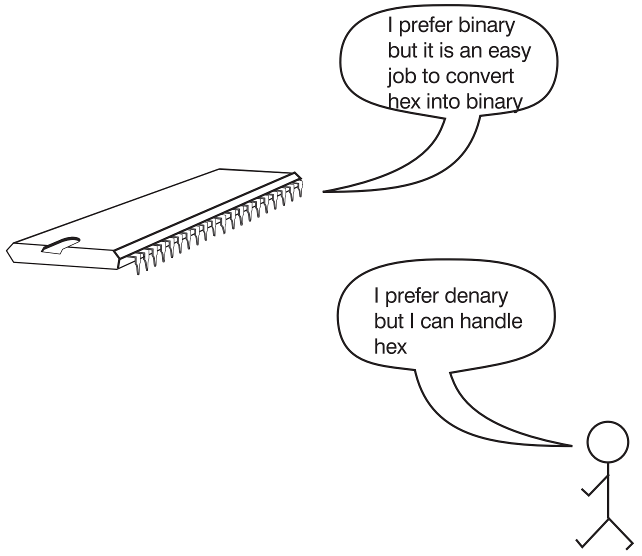
Here are the main options in order of popularity:

$16H = 24_{10}$
$16_H = 24_{10}$
$16h = 24_{10}$
$16_h = 24_{10}$
$16_{16} = 24_{10}$

The advantages of hex

1 It is very compact. Using a base of 16 means that the number of digits used to represent a given number is usually fewer than in binary or denary.
2 It is easy to convert between hex and binary and fairly easy to go between hex and denary. Remember that the microprocessor only works in binary, all the conversions between hex and binary are carried out in other circuits (Figure 3.1).

Figure 3.1
Hex is a good compromise


I prefer binary but it is an easy job to convert hex into binary
I prefer denary but I can handle hex

Converting denary to hex

The process follows the same pattern as we saw in the denary to binary conversion.

Method

1 Write down the denary number.

2 Divide it by 16_{10}, put the whole number part of the answer underneath and the remainder in the column to the right.

3 Keep going until the number being divided reaches zero.

4 Read the answer from the bottom to top of the remainders column.

REMEMBER TO WRITE THE REMAINDERS IN HEX.

A worked example

Convert the denary number 23 823 to hex

1 Write down the number to be converted

23 823

(OK so far).

2 Divide by 16. You will need a calculator. The answer is 1488.9375. The 1488 can be placed under the number being converted

```
23 823
  1488
```

but there is the problem of the decimal part. It is 0.9375 and this is actually 0.9375 of 16. Multiply 0.9375 by 16 and the result is 15. Remember that this 15 needs to be written as a hex number – in this case F. When completed, this step looks like:

```
23 823
  1488   F
```

3 Repeat the process by dividing the 1488 by 16 to give 93.0 There is no remainder so we can just enter the result as 93 with a zero in the remainder column.

```
23 823
  1488   F
    93   0
```

4 And once again, 93 divided by 16 is 5.8125. We enter the 5 under the 93 and then multiply the 0.8125 by 16 to give 13 or D in hex

```
23 823
  1488   F
    93   0
     5   D
```

5 This one is easy. Divide the 5 by 16 to get 0.3125. The answer has now reached zero and $0.3125 \times 16 = 5$. Enter the values in the normal columns to give:

```
23 823
  1488   F = 5D0F
    93      0 ↑
     5      D |
     0      5
```

6 Read the hex number from the bottom upwards: 5D0FH (remember that the 'H' just means a hex number).

Example

Convert $44\,256_{10}$ into hex

```
44 256
  2766   0     = ACE0H
   172   E ↑
    10   C |
     0   A
```

A further example

Convert $540\,709_{10}$ to hex

```
540 709
 33 794   5     = 84025H
   2112   2  ↑
    132   0  |
      8   4
      0   8
```

So $540\,709_{10}$ = 84025H but, especially when the hex number does not contain any letters, be careful to include the base of the numbers otherwise life can become really confusing.

Converting hex to denary

To do this, we can use a similar method to the one we used to change binary to denary.

Example

Convert A40E5H to denary

1 Each column increases by 16 times as we move towards the right-hand side so the column values are:

16^4	16^3	16^2	16^1	16^0
65536	4096	256	16	1

2 Simply enter the hex number using the columns

65536	4096	256	16	1
A	4	0	E	5

3 Use your calculator to find the denary value of each column

65536	4096	256	16	1
A	4	0	E	5
655360	16384	0	224	5

The left-hand column has a hex value of 10_{10} (A = 10) so the column value is 65536 × 10 = 655360. The next column is 4 × 4096 = 16384. The next column value is zero (256 × 0). The fourth column has a total value of 16 × 14 = 224 (E = 14). The last column is easy. It is just 1 × 5 = 5 no calculator needed!

4 Add up all the denary values:

$$655\,360 + 16\,384 + 0 + 224 + 5 = 671\,973_{10}$$

Method

1 Write down the column values using a calculator. Starting on with 16^0 (=1) on the right-hand side and increasing by 16 times in each column towards the left.

2 Enter the hex numbers in the appropriate column, converting them into denary numbers as necessary. This means, for example, that we should write 10 to replace an 'A' in the original number.

3 Multiply these denary numbers by the number at the column header to provide a column total.

4 Add all the column totals to obtain the denary equivalent.

Another example

Convert 4BF0H to denary

16^3	16^2	16^1	16^0	column values
4096	256	16	1	column values
4	11	15	0	hex values
16 384	2816	240	0	denary column totals

Total = 16 384 + 2816 + 240 + 0 = $19\,440_{10}$

Converting binary to hex

This is very easy. Four binary bits can have minimum and maximum values of 0000_2 up to 1111_2. Converting this into denary by putting in the column headers of: 8, 4, 2 and 1 results in a minimum value of 0 and a maximum value of 15_{10}. Doesn't this fit into hex perfectly!

This means that any group of four bits can be translated directly into a single hex digit. Just put 8, 4, 2 and 1 over the group of bits and add up the values wherever a 1 appears in the binary group.

Example

Convert 100000010101011_2 to hex

Step 1 Starting from the right-hand end, chop the binary number into groups of four.

100/ 0000/ 1010/ 1011/

Step 2 Treat each group of four bits as a separate entity. The right-hand group is 1011 so this will convert to:

8	4	2	1	column headers
1	0	1	1	binary number
8	0	2	1	column values

The total will then be 8 + 0 + 2 + 1 = 11_{10} or in hex, B.

The right-hand side binary group can now be replaced by the hex value B.

```
100/  0000/  1010/  1011/
                      B
```

Step 3 The second group can be treated in the same manner. The bits are 1010 and by comparing them with the 8, 4, 2, 1 header values this means the total value is (8 $\times$ 1) + (4 $\times$ 0) + (2 $\times$ 1) + (1 $\times$ 1) = 8 + 0 + 2 + 0 = 10_{10} or in hex, A.

We have now completed two of the groups.

```
100/  0000/  1010/  1011/
               A      B
```

Step 4 The next group consists of all zeros so we can go straight to an answer of zero. The result so far will be:

```
100/  0000/  1010/  1011/
        0      A      B
```

Step 5 The last group is incomplete so only the column headings of 4, 2, and 1 are used. In this case, the 4 is counted but the 2 and the 1 are ignored because of the zeros. This gives a final result of:

```
100/  0000/  1010/  1011/
 4      0      A      B
```

So, 100000010101011_2 = 40ABH.

Having chopped up the binary number into groups of four the process is the same regardless of the length of the number. Always remember to start chopping from the right-hand side.

Example

Convert the number 1100011111001_2 to hex

Split it into groups of four starting from the right-hand side

```
1/  1000/  1111/  1001/
```

Add column headers of 8 4 2 1 to each group

1	8421	8421	8421	column headings
1/	1000/	1111/	1001	binary number
1	8	8421	81	column values
1	8	15	9	group value in denary

Now just convert group values to hex as necessary. In this example only the second group 15, will need changing to F.

Final result is 1100011111001_2 = 18F9H.

Converting hex to binary

This is just the reverse of the last process. Simply take each hex number and express it as a four bit binary number.

As we saw in the last section, a four-bit number has column header values of 8, 4, 2 and 1, so conversion is just a matter of using these values to build up the required value. All columns used are given a value of 1 in binary and all unused columns are left as zero.

When you are converting small numbers like 3H we must remember to add zeros on the left-hand side to make sure that each hex digit becomes a group of four bits.

Imagine that we would like to convert 5H to binary. Looking at the column header values of 8, 4, 2 and 1, how can we make the value 5? The answer is to add a 4 and a 1. Taking each column in turn: we do not need to use an 8 so the first column is a 0. We do want a 4 so this is selected by putting a 1 in this column, no 2 so make this 0 and finally put a 1 in the last column to select the value of 1. The 5H is converted to 0101_2. All values between 0 and FH are converted in a similar way.

Example

Convert 2F6CH to binary

Step 1 Write the whole hex number out with enough space to be able to put the binary figures underneath

```
  2     F     6     C
```

Step 2 Put the column header values below each hex digit

```
  2     F     6     C
8421  8421  8421  8421
```

Step 3 The hex C is 12_{10} that can be made of 8 + 4 so we put a binary 1 in the 8 and the 4 columns. The four-bit number is now 1100_2

```
  2     F     6     C
8421  8421  8421  8421
                  1100
```

Step 4 Now do the same for the next column. The hex number is 6, which is made of 4 + 2, which are the middle two columns. This will result in the binary group 0110_2

```
  2     F     6     C
8421  8421  8421  8421
            0110  1100
```

Step 5 Since 8 + 4 + 2 + 1 = 15, the hex F will become 1111_2

```
   2     F     6     C
8421  8421  8421  8421
      1111  0110  1100
```

Step 6 Finally, the last digit is 2 and since this corresponds to the value of the second column it will be written as 0010_2

```
   2     F     6     C
8421  8421  8421  8421
0010  1111  0110  1100
```

The final result is 2F6CH = 0010111101101100_2.

But do we include the two leading zeros? There are two answers, 'yes' and 'no' but that's not very helpful. We need to ask another question: why did we do the conversion? were we doing math or microprocessors? If we were working on a microprocessor system then the resulting 16 bits would represent 16 voltages being carried on 16 wires. As the numbers change, all the wires must be able to switch between 0 V and 3.3 V for binary levels 0 and 1. This means, of course, that all 16 wires must present so we should include the binary levels on all of them.

If the conversion was purely mathematical, then since leading (left-hand end) zeros have no mathematical value there is no point in including them in the answer.

Method

1 Write down the hex number but make it well spaced.

2 Using the column header values of 8, 4, 2 and 1, convert each hex number to a four bit binary number.

3 Add leading zeros to ensure that every hex digit is represented by four bits.

Example

Convert 1E08BH to binary

Step 1

```
   1     E     0     8     B
8421  8421  8421  8421  8421
```

Step 2

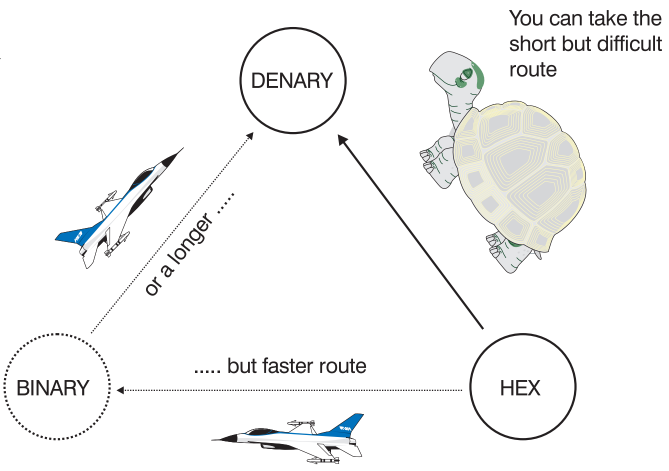
```
0001  1110  0000  1000  1011
```

So, 1E08BH = 00011110000010001011_2.

Using stepping stones

It is fairly easy to convert binary to hex and hex to binary. I find it much easier to multiply and divide by 2 rather than by 16, so when faced with changing hex into denary and denary into hex I often change them into binary first. It is a longer route but at least I can do it without my calculator (see Figure 3.2).

Figure 3.2
A longer route may prove easier



Obsolete octal – probably not worth reading

Octal is another number system which has no advantages over hex but is still met from time to time. Only a brief look will be offered here just to make sure that we have at least mentioned it.

In hex, we used binary bits in groups of four because 1111_2 adds up to 15 which is the value of the highest digit (F) in hex. In octal, we use groups of three bits. The highest value is now 111_2 which is 7. Octal therefore has eight digits and counts from 0 to 7. The count proceeds:

0
1
2
3
4
5
6
7

There is no 8th digit so reset the count to 0 and put a 1 in the next column.

10
11
12
13
14
15
16
17

Now go straight to 20

20
etc.

No letters are involved and it is often not recognized as octal until we realize that none of the numbers involve the digits 8 or 9.

Conversions follow the same patterns as we have seen for hex.

Octal to denary: the column heading values are 8^4, 8^3, 8^2, 8^1, 8^0.

Denary to octal: divide by 8 and write down the remainder then read remainders from the bottom upwards. Use the subscript 8 to indicate an octal number, e.g. $64_{10} = 100_8$.

Octal to binary: write each octal digit down as a *three* digit binary group.

Binary to octal: start from the right-hand side and chop the binary numbers into groups of three, then evaluate each group.

I think that is enough for octal. It's (fairly) unlikely you will meet it again so we can say 'goodbye Octal'.

Quiz time 3

In each case, choose the best option.

1 Which of these represents the largest number?

(a) 1000_8
(b) 1000_{10}
(c) 1000_2
(d) 1000H

2 The number CD02H is equal to:

(a) 52482_{10}
(b) 54228_{10}
(c) 56322_{10}
(d) 52842_{10}

3 The base of a number system is:

(a) always the same as the highest digit used in the system.
(b) usually +5 or +3.3.
(c) equal to the number of different digits used in the system.
(d) one less than the highest single digit number in the system.

4 Which of these numbers is the same as 10110111010_2:

(a) 1646_{10}
(b) $5BA_{16}$
(c) AB5H
(d) B72h

5 The number of digits in a denary number is often:

(a) more than the number of digits in the equivalent binary number.
(b) less than or equal to the number of digits in the equivalent hex number.
(c) more than the number of digits in the equivalent hex number.
(d) more than the number of digits in the equivalent decimal number.

4

How micros calculate

How the microprocessor handles numbers (and letters)

In the last chapter, we saw how numbers could be represented in binary and hex forms. Whether we think of a number as hex or binary or indeed denary, inside the microprocessor it is only binary. The whole concept of hex is just to make life easier for us.

We may sit at a keyboard and enter a hex (or denary) number but the first job of any microprocessor-based system is to convert it to binary. All the arithmetic is done in binary and its last job is to convert it back to hex (or denary) just to keep us smiling.

There was a time when we had to enter binary and get raw binary answers but thankfully, those times have gone. Everything was definitely NOT better in the 'good old days'.

The form binary numbers take inside of the microprocessor depends on the system design and the work of the software programmers. We will take a look at the alternatives, starting with negative numbers.

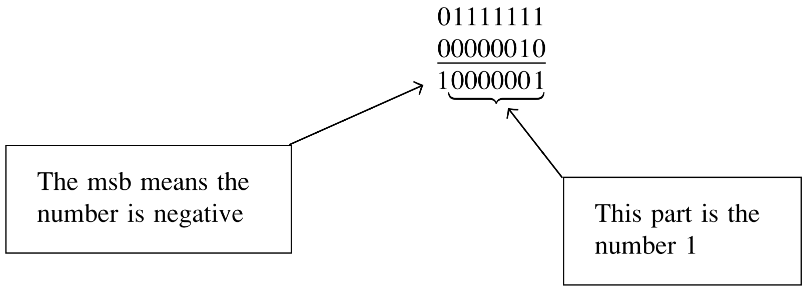
In real life, it is easy, we just put a – symbol in front of the number and it is negative so +4 becomes –4. Easy, but we don't have any way of putting a minus sign inside the microprocessor. We have tried several ways round the problem.

Signed magnitude numbers

The first attempt seemed easy but it was false optimism. All we had to do was to use the first bit (msb) of the number to indicate the sign 1 = minus, 0 = plus.

This had two drawbacks.

1 It used up one of the bits so an 8-bit word could now only hold seven bits to represent numbers and one bit to say 'plus' or 'minus'. The seven bits can now only count up to 1111111_2 = 127 whereas the eight bits should count to 255.
2 If we added two binary numbers like +127 and +2, we would get:


01111111
00000010
10000001
The msb means the number is negative
This part is the number 1

The msb (most significant bit) of 1 means it is a minus number and the actual number is 0000001 = 1. So the final result of +127 + 2 is not 129 but minus 1.

When we use a microprocessor to handle arithmetic with these problems, we can ensure that the microprocessor can recognize this type of accidental negative number. We can arrange for the microprocessor to compensate for it but it is rather complicated and slow.

Luckily, a better system came along which has stood the test of time, having been used for many years.

Complementary numbers

This has two significant advantages:

1 It allows the full number of bits to be used for a number so an 8-bit word can count from 0 to 11111111_2 or 255.
2 It is easy to implement with addition and subtraction using substantially the same circuitry.

So, how do we manage to use all eight bits for numbers yet still be able to designate a number positive or negative?

That's clever. We will start by looking at positive numbers first because it is so easy. All positive numbers from 0 to 255 are the same as we get by simply converting denary to binary numbers. So that's done.

Addition

Example

Add 01011010 + 00011011.

The steps are just the same as in 'normal' denary arithmetic.

Step 1 Lay them out and start from the lsb (least significant bit) or right-hand bit

$$\begin{array}{ccccccccc} 0&1&0&1&1&0&1&0&+\\ 0&0&0&1&1&0&1&1& \\ \hline \end{array}$$

Add the right-hand column and we have 0 + 1 = 1.

So we have

$$\begin{array}{ccccccccc} 0&1&0&1&1&0&1&0&+\\ 0&0&0&1&1&0&1&1& \\ \hline &&&&&&&1& \\ \hline \end{array}$$

Step 2 Next we add the two 1s in the next column. This results in 2, or 10 in binary. Put the 0 in the answer box and carry the 1 forward to the next column

$$\begin{array}{ccccccccc} 0&1&0&1&1&0&1&0&+\\ 0&0&0&1&1&0&1&1& \\ \hline &&&&&&0&1& \\ \hline &&&&&1&&& \end{array}$$

Step 3 The next column is easy 0 + 0 + 1 = 1

$$\begin{array}{ccccccccc} 0&1&0&1&1&0&1&0&+\\ 0&0&0&1&1&0&1&1& \\ \hline &&&&&1&0&1& \\ \hline &&&&&1&&& \end{array}$$

Step 4 The next line is like the second column, 1 + 1 = 10. This is written as an answer of 0 and the 1 is carried forward to the next column

$$\begin{array}{ccccccccc} 0&1&0&1&1&0&1&0&+\\ 0&0&0&1&1&0&1&1& \\ \hline &&&&0&1&0&1& \\ \hline &&&&1&&1&& \end{array}$$

Step 5 We now have 1 in each row and a 1 carried forward so the next column is 1 + 1 + 1 = 3 or 11 in binary. This is an answer of 1 and a 1 carried forward to the next column

$$\begin{array}{ccccccccc} 0&1&0&1&1&0&1&0&+\\ 0&0&0&1&1&0&1&1& \\ \hline &&&1&0&1&0&1& \\ \hline &&1&1&&1&&& \end{array}$$

Step 6 The next column is 0 + 0 + 1 = 1, and the next is 1 + 0 = 1 and the final bit or msb is 0 + 0 = 0, so we can complete the sum

```
0 1 0 1 1 0 1 0 +
0 0 0 1 1 0 1 1
---------------
0 1 1 1 0 1 0 1 ──── | 90 + 27 = 117 |
---------------
    1 1   1
```

Subtraction

Here is a question to think about: What number could we add to 50 to give an answer of 27? In mathematical terms this would be written as $50 + x = 27$.

What number could x represent? Surely, anything we add to 50 must make the number larger unless it is a negative number like –23:

$50 + (-23) = 27$

The amazing thing is that there is a number that can have the same effect as a negative number, even though it has no minus sign in front of it. It is called a 'two's complement' number.

Our sum now becomes:

50 + (the two's complement of 23) = 27

This magic number is the *two's complement* of 23 and finding it is very simple.

How to find the two's complement of any binary number

Invert each bit, then add 1 to the answer

All we have to do is to take the number we want to subtract (in its binary form) and invert each bit so every one becomes a zero and each zero becomes a one. Note: technically the result of this inversion is called the 'one's complement' of 23. The mechanics of doing it will be discussed in the next chapter but it is very simple and the facility is built into all microprocessors at virtually zero cost.

Converting the 23 into a binary number gives the result of 00010111_2 (using eight bits). Then invert each bit to give the number 11101000_2 then add 1. The resulting number is then referred to as the 'two's complement' of 23.

```
1 1 1 0 1 0 0 0 ──── | The one's complement |
              1 +
---------------
1 1 1 0 1 0 0 1 ──── | The two's complement |
---------------
```

In this example, we used 8-bit numbers but the arithmetic would be exactly the same with 16 bits or indeed 32 or 64 bits or any other number.

Doing the sum

We now simply add the 50 and the two's complement of 23:

50 + (the two's complement of 23) = 27

$$\begin{array}{r l}
1\ 1\ 1\ 0\ 1\ 0\ 0\ 1 & \text{———} \boxed{50} \\
\underline{0\ 0\ 1\ 1\ 0\ 0\ 1\ 0}\ + & \text{———} \boxed{\text{The two's complement of 23}} \\
\underline{1\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1} & \text{———} \boxed{\text{Answer is 27}} \\
1\ &
\end{array}$$

The answer is 100011011.

Count the bits. There are nine! We have had a carry in the last column that has created a ninth column. Inside the microprocessor, there is only space for eight bits so the ninth one is not used. If we were to ask the microprocessor for the answer to this addition, it would only give us the 8-bit answer: 00011011_2 or in denary, 27. We've done it! We've got the right answer!

It was quite a struggle so let's make a quick summary of what we did.

1 Convert both numbers to binary.
2 Find the two's complement of the number you are taking away.
3 Add the two numbers.
4 Delete the msb of the answer.

Done.

A few reminders

1 Only find the two's complement of the number you are taking away – NOT both numbers.
2 If you have done the arithmetic correctly, the answer will always have an extra column to be deleted.
3 If the numbers do not have the same number of bits, add leading zeros as necessary as a first job. Don't leave until later. Both of the numbers must have the same number of bits. They can be 8-bit numbers as we used, or 16, or 32 or anything else so long as they are equal.

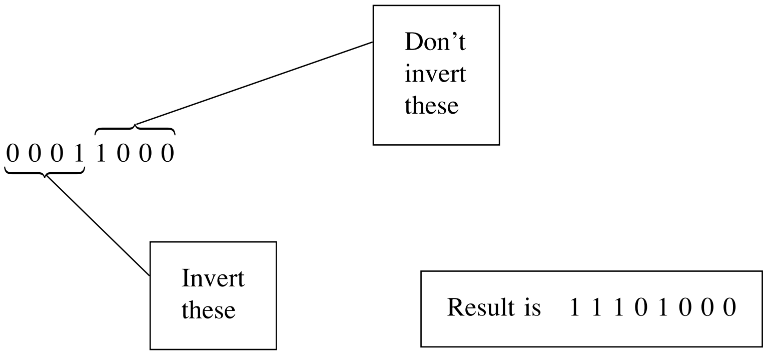
A quick way to find the two's complement of a binary number

Start from the left-hand end and invert each bit until you come to the last figure 1. Don't invert this figure and don't invert anything after it.

Example 1

What is -24_{10} expressed as an 8-bit two's complement binary number?

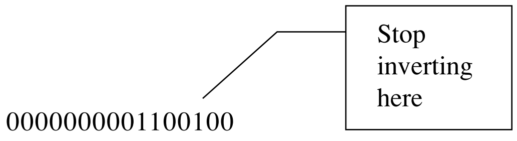
1 Change the 24_{10} into binary. This will be 11000.
2 Add leading zeros to make it an 8-bit number. This is now 00011000.
3 Now start inverting each bit, working from the left until we come to the last figure '1'. Don't invert it, and don't invert the three zeros that follow it.



Example 2

What is -100_{10} expressed as a 16-bit two's complement binary number?

1 Convert the 100_{10} into binary. This gives 1100100_2.
2 Add nine leading zeros to make the result the 16-bit number 0000000001100100.
3 Now, using the quick method, find the two's complement:



The result is 1111 1111 1001 1100

Example 3

Find the value of $1011\ 0111_2 - 00\ 1011_2$ using two's complement addition.

1 The second number has only six bits so add two zeros on the left-hand end to give 1011 0111 – 0000 1011.
2 Invert each bit in the number to be subtracted to find the one's complement. This changes the 00001011 to 11110100.
3 Add 1 to give the two's complement: 11110100 + 1 = 11110101 (or do it the quick way).

4 Add the first number to the two's complement of the second number:

$$\begin{array}{cccccccccc}
 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & \\
 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & + \\
\hline
1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & \\
\hline
1 & 1 & 1 & 1 & & 1 & 1 & 1 & &
\end{array}$$

5 The result so far is 110101100 which includes that extra carry so we cross off the msb to give the final answer of 10101100_2.

Floating point numbers

Eight-bit numbers are limited to a maximum value of 11111111_2 or 255_{10}. So, 0 – 255 means a total of 256 different numbers. Not very many. 32-bit numbers can manage about 4¼ billion. This is quite enough for everyday work, though Bill Gates' bank manager may still find it limiting. The problem is that scientific studies involve extremely large numbers as found in astronomy and very small distances as in nuclear physics.

So how do we cater for these? We could wait around for a 128-bit microprocessor, and then wait for a 256-bit microprocessor and so on. No, really, the favourite option is to have a look at alternative ways of handling a wide range of numbers. Rather than write a number like 100 we could write it as 1×10^2. Written this way it indicates that the number is 1 followed by two zeros and so a billion would be written as 1×10^9. In a similar way, 0.001 is a 1 preceded by two zeros would be written as 1×10^{-3} and a billionth, 0.000000001, would be 1×10^{-9}. The negative power of ten is one greater than the number of zeros. By using floating point numbers, we can easily go up to 1×10^{99} or down to 1×10^{-99} without greatly increasing the number of digits.

Fancy names

Normalizing

Changing a number from the everyday version like 275 to 2.75×10^2 is called normalizing the number. The first number always starts with a single digit between 1 and 9 followed by a power of ten.

In binary we do the same thing except the decimal point is now called a binary point and the first number is always 1 followed by a power of two as necessary.

Three examples

1 Using the same figure of 275, this could be converted to 100010011 in binary. This number is normalized to 1.00010011×2^8.

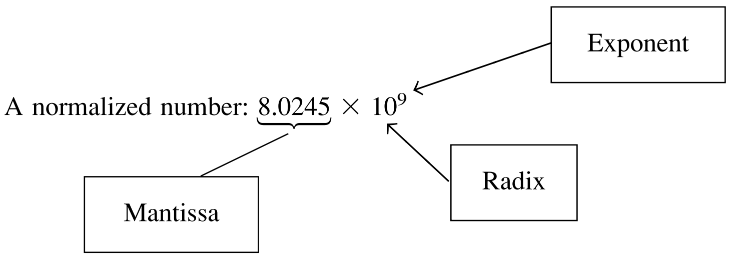
2 A number like 0.0001001_2 will have its binary point moved four places to the *right* to put the binary point just after the first figure 1 so the normalized number can be written as 1.001×2^{-4}.

3 The number 1.101_2 is already normalized so the binary point does not need to be moved so, in a formal way, it would be written as 1.101×2^0.

A useless fact
Anything with a power of zero is equal to 1. So $2^0 = 1$, $10^0 = 1$. It is tempting but total nonsense to use this fact to argue that since $2^0 = 1$ and $10^0 = 1$ then 2 must equal 10!

Terminology

There are some more fancy names given to the parts of the number to make them really scary.



The exponent is the power of ten, in this example, 9. The mantissa, or magnitude, is the number, in this case 8.0245. The radix is the base of the number system being used, 2 for binary, 16 for hex, 10 for decimal.

Storing floating point numbers

In a microprocessor, the floating point is a binary number. Now, in the case of a binary number, the mantissa always starts with 1 followed by the binary point. For example, a five digit binary mantissa would be between 1.0000 and 1.1111.

Since all mantissas in a binary system start with the number 1 and the binary point, we can save storage space by missing them out and just assuming their presence. The range above would now be stored as 0000 to 1111.

It is usual to use a 32-bit storage area for a floating point number. How these 32 bits are organized is not standardized so be careful to check before making any assumptions. Within those 32 bits, we have to include the exponent and the mantissa which can both be positive or negative. One of the more popular methods is outlined below.

	Exponent	Mantissa	S
Bit	31 24	23 1	0

Bit 0 is used to hold the sign-bit for the mantissa using the normal convention of 0 = positive and 1 = negative.

Bits 1–23 hold the mantissa in normal binary.

Bits 24–31 hold the exponent. The eight digits are used to represent numbers from –127 to +128 using either two's complement numbers or excess-127 notation.

We have already met two's complement numbers earlier in this chapter so we will look at excess-127 notation now.

Excess-127 notation

This is very simple, despite its impressive name. To find the exponent just add 127 to its value then convert the result to binary. This addition will ensure that all exponents have values between 0 and 255, i.e. all positive values.

Example

If the exponent is –35 then we add 127 to give the result 92, which we can then convert to binary (01011100).

When the value is to be taken out of storage and converted back to a binary number, the above process is reversed by subtracting the 127 from the exponent.

Size, accuracy and speed

The mantissa can go as high as $1.1111\ 1111\ 1111\ 1111\ 1111\ 111_2$. To the right of the binary point the decimal equivalents are values of 1.5 + 0.25 + 0.125 + 0.0625 etc. Adding these up gives a total that is virtually 2 – but not quite. The larger the number of bits in the mantissa, the more accuracy we can expect in the result. The exponent has eight bits so it can range from –127 to +128 giving a maximum number of 1×2^{128} which is approximately 3.4×10^{38}. The accuracy is limited by the number of bits that can be stored in the mantissa, which in this case is 23 bits.

If we want to keep to a total of 32 bits, then we have a trade-off to consider. Any increase in the size of the exponent, to give us larger numbers, must be matched by reducing the number of bits in the mantissa that would have the effect of reducing the accuracy. Floating point operations per second (FLOPS) is one of the choices for measuring speed.

IBM are building (2002) a new super computer employing a million microprocessors. The Blue Gene project will result in a computer running at a speed of over a thousand million million operations per second (1 petaflop). This is a thousand times faster that the Intel 1998 world speed record or about two million times faster than the current top-of-the-range desktop computers.

Single and double precision

If we need more accuracy, an alternative method is to increase the number of bits that can be used to store the number from 32 (single-precision) to 64 (double-precision). If this extra storage space is devoted to increasing the mantissa bits, then the accuracy is increased significantly.

Binary coded decimal (BCD)

Binary coded decimal numbers are very simple. Each decimal digit is converted to binary and written as a 4-bit or 8-bit binary number. The number 5 would be written as 0101_2 or 00000101_2. So far, this is the same as 'ordinary' binary but the change occurs when we have more digits.

Consider the number 25_{10}. In regular binary this would convert to 11001_2. Alternatively, we could convert each digit separately to 4-bit or 8-bit numbers:

$$2 = 0010_2 \text{ or } 0000\ 0010_2$$
$$5 = 0101_2 \text{ or } 0000\ 0101_2$$

Putting these together, 25_{10} could be written using the 4-bit numbers as $0010\ 0101_2$. This uses one byte and is called Packed BCD.

Alternatively, we could use the 8-bit formats and express 25_{10} as $0000\ 0010\ 0000\ 0101_2$ and would now use two bytes. This is called Unpacked BCD.

There are two disadvantages. Firstly, many numbers are of increased length after converting to BCD, particularly so if we use unpacked BCD or the numbers are very large like 25×10^{75}. In addition, arithmetic is much more difficult although, generally, microprocessors do have the ability to handle them.

The advantage becomes apparent when the microprocessor is controlling an external device like digits on displays at a filling station or accepting inputs from a keyboard. The coding is simple and does not involve the conversion of the numbers to binary.

Overall
Arithmetic → use binary
Inputting and outputting numbers → use BCD

Quiz time 4

In each case, choose the best option.

1 The number -35_{10}, when expressed as an 8-bit binary number in two's complement form, is:

(a) 00100011.
(b) 1111011101.
(c) 11011101.
(d) 00110101.

2 The number 7_{10} converted to an unpacked BCD format would be written as:

(a) 1110 0000.
(b) 7H.
(c) 0000 0111.
(d) 0111.

3 The signed magnitude number 11001100_2 is equivalent to:

(a) -76_{10}.
(b) 204_{10}.
(c) CCH.
(d) 1212_{10}.

4 In the number 0.5×10^{24} the number:

(a) 10 is the mantissa.
(b) 24 is the exponent.
(c) 0 is the sign bit.
(d) 5 is the radix.

5 A signed magnitude number that has a figure:

(a) zero as the msb is a negative number.
(b) one as the lsb is a negative number.
(c) one as the msb is a negative number.
(d) zero as the lsb is a negative number.

5

An introduction to logic gates and their uses

Opening and closing gates

In the last chapter the binary values zero and one are represented by two different voltages. Binary zero is a voltage close to 0 V and binary one by a voltage close to +5 V (some logic circuits use other voltage levels but this is a popular value and will serve as an example).

A gate is a simple electronic circuit that has a single output voltage that corresponds to one of the two binary values. These gates are often referred to as 'logic gates' and the output voltages as 'logic 0' or 'logic 1' instead of binary 0 and 1. The distinction is just in the name. If you were to ask a mathematician or a computer programmer, they will refer to the outputs as binary values but an electronics engineer will call them logic levels. It really doesn't matter.

What decides the output voltage?

We connect one or more voltages to the input of the gate. These input voltages are either logic 0 or logic 1 levels. The logic gate looks at the input voltages and 'decides', depending on its design, what voltage to produce at the output of the circuit.

There are only four basic designs of gate. They are called the NOT gate, the AND gate, the OR gate and the XOR gate. Notice how we use capital letters for the names of the gates otherwise we can finish up with some almost indecipherable sentences. Not not or and not and or not . . .

A little reminder before we start. Logic gates are clever little chaps but they are not magic. Just like any other electronic circuit, they need power supplies to make them work. Now, because all gates and microprocessors need power supplies, we tend to assume that everyone knows that. You will notice that power supplies are not shown in any of the diagrams in this chapter but that doesn't mean that they are not there!

We will explore these gates now, starting from the simplest.

The NOT gate

It has only one input and performs a very simple function. It simply reverses the binary value. If we put a logic 1 into it, we get a logic 0 at the output. Similarly, a 0 at the input gives a 1 at the output. On a diagram, we represent a NOT gate by a symbol as shown in Figure 5.1.

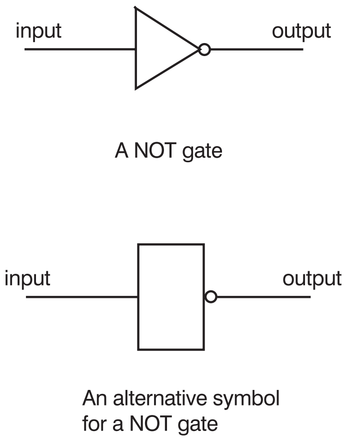


Figure 5.1
Symbols for a NOT gate

A truth table

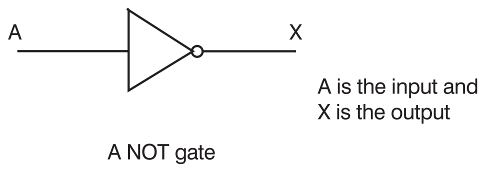
This is an alternative to the wordy description of how a gate works. It simply lists all the possible inputs to the gate together with the corresponding outputs. The truth table for a NOT gate is really easy. There are only two possible inputs: 0 and 1 as we can see in Figure 5.2.

So, how is it used in the microprocessor?

The truth table only shows what happens to a single bit but in the microprocessor we may want to use a NOT gate to invert a hex number like A4H. In this case the hex number is first converted to an 8-bit binary number. This process is not performed by the

Figure 5.2
The truth table for a NOT gate

Symbol



Truth table

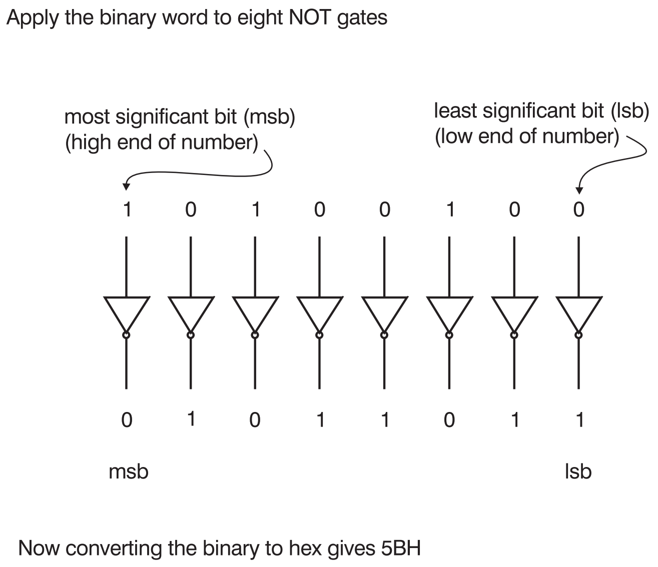
input A	output X
0	1
1	0

The output is just the reverse of the input

microprocessor but by other external circuits. By the time it reaches the microprocessor it has been converted to the binary equivalent of 10100100_2.

The NOT gate has only one input so, to handle an 8-bit binary word, we will need eight NOT gates. Now it becomes much easier. Each NOT gate inverts just one of the bits and all the outputs are grouped together to form a new hex number. See how it works in Figure 5.3. The result was the hex number 5BH. This is curious. If we add A4H to this result of 5BH we get FFH or all ‘ones’ in binary, 11111111_2. There was nothing special about the number A4H. It happens with any pair of numbers generated by NOT gates. Why is this? Figure 5.3 gives a clue.

Figure 5.3
Inverting a hex number

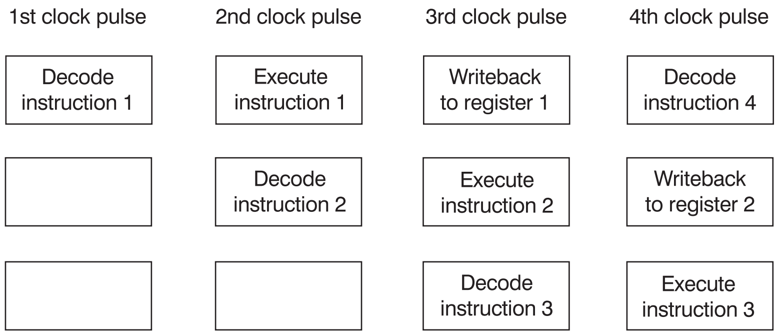

Apply the binary word to eight NOT gates
most significant bit (msb) (high end of number)
least significant bit (lsb) (low end of number)
1 0 1 0 0 1 0 0
0 1 0 1 1 0 1 1
msb
lsb
Now converting the binary to hex gives 5BH

An AND

like someone reading the back of your newspaper as you are reading the front – except that registers don't find it irritating. 'Port', by the way, is just a fancy electronic word meaning 'connection'. Transistors, generally, have three wires going to them and so are described as three-port devices.

The integer unit handles all instructions like integer arithmetic bit manipulation and transferring data to and from the external memory and is organized into a three-stage pipeline. In Figure 13.3, the second clock pulse executes the first instruction. The next clock pulse executes the second instruction and the last clock pulse executes the third. We have achieved the target of one clock pulse per clock pulse. And in the fourth clock pulse, we can see the next instruction just arriving to be decoded immediately after the first write-back.

Figure 13.3 Integer unit pipeline



Floating-point unit

This has a further 32 registers but in this case, they are 64-bits wide and to fill a register with a single clock pulse, there is an internal 64-bit bus connecting it with the cache. The pipeline is five stage: prefetch, buffer, decode, execution and write-back.

Memory buffer

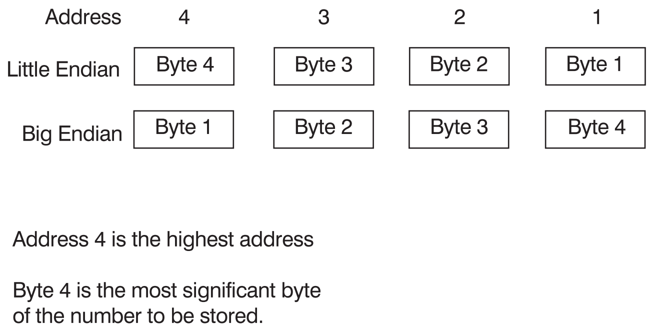
This acts as a buffer for the external memory. The buffers include two reads and three writes, each up to 32 bytes. It is also used in writing-back to the cache.

Big and little endians

The main memory is divided into locations each having its own address. Each location can hold a single byte of information. If we wanted to store a 32-bit number, then we would have to utilize four consecutive locations.

Imagine that we wished to store the 32-bit number $00000000\ 01010101\ 00010001\ 11111111_2$ and we had addresses 24646603H, 24646602H, 24646601H and 24646600H available. Little-endian format would store the most significant byte in the highest memory address so, in our example, the data 00000000 would go into address 24646603H. This is used by Intel microprocessors. Big-endian, which Motorola uses, works the other way around. The most significant byte is put in the lowest memory address so, in our example, the data 00000000 would go into address 24646600H. These are shown in Figure 13.4. All the PowerPC microprocessors are switchable to enable little or big-endian to be used.

Figure 13.4
Big and little endians


Address
4
3
2
1
Little Endian
Byte 4
Byte 3
Byte 2
Byte 1
Big Endian
Byte 1
Byte 2
Byte 3
Byte 4
Address 4 is the highest address
Byte 4 is the most significant byte of the number to be stored.

PowerPC 970

A large number of PowerPCs have continued to power the Apple-Mac and IBM desktops and, in addition, support both the UNIX and Linux operating systems.

The latest offering is the 970 with its 52 million transistors started life as a 1.8 GHz device and has now progressed to 2.0 GHz. This may appear slow but it has compensating attributes such as its 900 MHz bus as opposed to the 533 MHz bus of the Pentium 4.

It is a 64-bit micro so it handles data in 64-bit chunks but remains compatible with earlier 32-bit designs. It has two level 1 caches, one for instructions at 64 kB and a data cache of 32 kB, which are somewhat larger that the Intel product but both companies use a level 2 cache of 512 kB.

As memory size is continuing to increase with each design, the size of memory that can be directly accessed increases with the move to 64-bit processing. The Pentium 4 can access 40 GB of memory, which seems excessively large at the moment but there was a time when 4 MB was something to wonder at. The PowerPC 970 can handle memory of Star Trek proportions measured in terabytes (thousands of Gigs).

Table 13.1 Cache sizes

	L1 Instruction	*L1 Data*	*L2 cache*
PowerPC 970	64 kB	32 kB	512 kB
Pentium 4	It's a secret	8 kB	512 kB

For maximum microprocessor speed we need a high clock speed combined with the maximum use being made of every part of the microprocessor. The early 8-bit microprocessors would accept the first instruction and it would pass through the microprocessor being decoded, then acted upon, then having the results stored before it considered the next instruction. This meant that each bit of the micro was doing nothing for much of the time.

Modern micros load many instructions at the same time and split up the tasks so that as many as possible can be carried out at the same time to have the minimum time wastage.

As with the Pentium 4, the PPC970 makes use of level 1 caches that, as is now common, are split into an Instruction cache and a Data cache. There is also a level 2 cache and an external level 3 cache.

Loading the instructions

The instructions pour down from the Instruction cache at a maximum rate of eight per cycle, though five is a more likely overall figure. But this is still fast.

The PP970 uses a very long pipeline and can be handling up to 200 instructions simultaneously. The price of such a long pipeline is that we must be careful to ensure that it is filled with the most useful instructions and hence we need to back it up with very effective branch prediction techniques.

Branch prediction

To obtain the maximum possible speed, the PP970 has devoted a great deal of resources into its branch prediction. As the instructions are loaded, the branch prediction circuitry scans the incoming instruction looking for branch instructions. Every time we meet a branch instruction that offers a choice of outcome the branch will have to be accepted or rejected.

The 970 has two branch prediction methods. The first is very similar to that used in the Pentium 4 and, to over simplify the situation, it follows the same sort of reasoning as we often adopt in everyday life. If it usually happens, it is most likely to happen again. The 970 keeps a record of the previous 16384 branches in its BHT (Branch History Table) to see how often each choice was made and then this information is further sorted by a prediction program before it comes to a final decision.

The second method involves a similar sized table called a Global Predictor. This method also comes up with a final go/no go for the branch but it decides by generating an 11-bit vector that stores the actual execution path taken by the previous eleven fetch groups leading up to the branch.

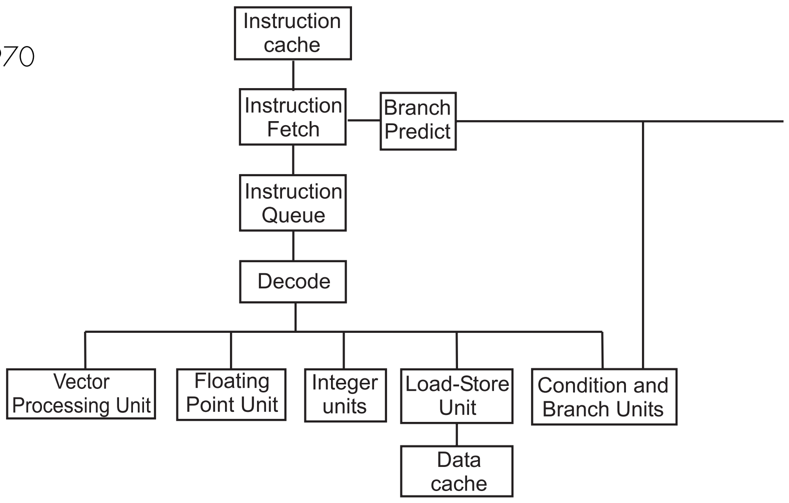
So there are two independent mechanisms that make a decision as to whether the branch should be taken. If they disagree, we need a referee. This job is performed by a 'Selector Table' that stores the success rate for each of the two previous methods for each particular branch. It then makes the final decision – and it is said (by IBM) to be very successful, which it probably is.

Handling the instructions

Having combined the incoming instruction stream from the Instruction cache with the information from the Branch predict, the instruction are queued and passed to the Decode, Crack and Group Formation Unit.

At this stage, in order to keep the instruction handling speed at a maximum, this unit takes the instruction codes from the Instruction cache, decodes them and cracks them into their component parts called Internal Operations (IOPs). These very small but simple tasks are passed out to specialized units like the five blocks shown along the bottom of Figure 13.5.

Figure 13.5
The PowerPC 970



The IOPs are executed in whatever order that will result in the fastest throughput and to reduce the complexity of keeping track of the execution of each and every one, they are organized in groups of five and then the groups are tracked.

Of the final row shown, there are the arithmetically based block that handle the vectors, floating point and integer calculations, the load-store that handles the transfer of data to the memory via the second level cache and finally the feedback path for the branch prediction information.

The PC market place

The PowerPC may not be in our PC but it may well be in our car. The Ford Motor Company has elected to use the PowerPC as first choice for their engine management computer into the next century.

Quiz time 13

In each case, choose the best option.

1 The maximum number of instructions that the PowerPC 970 can be dealing simultaneously is:

(a) 200.
(b) 3.
(c) 16384.
(d) 128.

2 Write-back:

(a) reverses the order of the bits of data.
(b) is used to double-check the accuracy of data before use.
(c) is only used in the little-endian system.
(d) stores results in the cache rather than in the external memory.

3 The PowerPC 970 has an internal bus running at a frequency of:

(a) 64 bits/s although it can run at 32 bits/s.
(b) 512 kB/s.
(c) 900 MHz.
(d) 533 MHz.

4 A register that can be accessed by two circuits at the same time is referred to as:

(a) a second-level cache.
(b) dual-ported.

(c) a buffer.
(d) a three-ported device.

5 Big endian format:

(a) stores the low byte in the highest address.
(b) stores the high byte in the highest address.
(c) is used in all microprocessors.
(d) is used in a cache but never in the main memory.

14

The Athlon XP

This is AMD's competitor to the Pentium and is concentrating the mind of both companys and greatly benefiting the rest of us.

Competition concentrates the mind as well as improving things for the customers.

AMD has been creeping up on Intel for several years and finally the Athlon's 37 million transistors are giving the Pentium a serious problem. It is usually cheaper and, in many tests, faster. The thought behind the Athlon is not to compete in terms of clock speed but to go for real speed by doing more work for each clock cycle. Even so, the Athlon XP is now competing head-to-head on speed, having matched the Pentium at 2.8 GHz using the same 0.13 micron technology though with a different internal design and ensuring (of course) that the two microprocessors are not pin-for-pin compatible. The Athlon includes a similar system of protection against thermal overload as in the Pentium.

An outline of the Athlon XP is shown in Figure 14.1.

Caches

For maximum speed the caches are on-chip. This eliminates the travel-time delay as the data is moved.

From the external memory and the surrounding hardware, the incoming information from the system bus is fed into a 64 kB

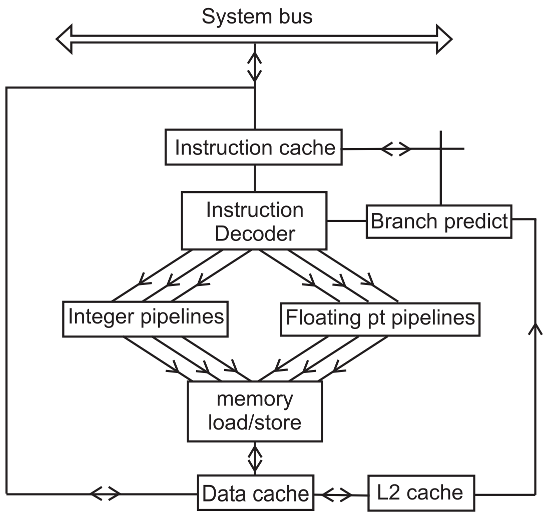


Figure 14.1
The Athlon XP processor

instruction cache and a separate 64 kB data cache. The data cache feeds data into the L2 cache, which is somewhat larger at 256 kB and has techniques to ensure that the L2 cache does not duplicate any of the information stored in the data cache and hence we effectively have a 384 kB local high speed storage area.

Branch predict

As with all current microprocessors, great care is taken to guess the likely result in each branch instruction. Such instructions produce, usually, two alternative routes for the program. They answer questions like 'is the result zero?' and the answer will determine what happens next. If we always wait until the question is answered and only then do we load the instructions for the next bit of the program there is much wasted time as we saw in the earlier microprocessor designs. If we guess correctly, we can already pre-load the next part of the program and get started on it. The branch prediction circuitry does the guessing. If it gets it wrong, the old data is ditched and replaced.

Hardware data prefetch

This is a further form of prediction similar in which the incoming instructions are monitored and, as they are still arriving, the data that will be needed is guessed at, and loaded into the data cache so the Athlon loads data before it knows that it will be needed. As with the branch prediction, incorrect data has to be overwritten but on balance, it speeds up the data flow.

Instruction decoders

To make full use of its slower clock speed, the Athlon has three instruction decoders that can run independently. Each of these can handle three operations per clock cycle giving an overall throughput of nine operations per clock cycle, which is still significantly greater than the six operations per clock cycle of the Pentium.

Pipelines and instructions

The Athlon has three independent integer pipelines and also three similar floating-point pipelines whereas the Pentium has four pipelines for integers but only two for floating points.

The three floating-point execution units simultaneously handle:

(a) store and load functions
(b) add functions
(c) multiply functions such as all the Intel MMX (multimedia extensions) instructions plus AMCs own SIMD (single instruction multiple data) instructions to provide full support SSE (streaming SIMD extension) and more lifelike 3D imaging and graphics – AMD's name for these new instructions is '3D NOW!' technology. (MMX is an Intel trademark; 3D NOW! is an AMD trademark.)

The state of the competition

The Pentium had a 'rapid execution engine' which had two ALUs (arithmetic and logic units) for the integer instructions, each clocked at twice the core processor speed running a front side bus at 533 MHz whereas the Athlon XP had only a 333 MHz FSB. This continues the pattern of the Pentium claiming the headline figure for speed. However, on balance, the Athlon is, by most tests, slightly faster than the Pentium.

An update . . .

That was written yesterday. This morning came the news that Intel has burst through the 3 GHz barrier (just) with a 3.06 GHz device. This, they say, includes hyper-threading, a technique that involves splitting a program into units that can be ran simultaneously. It allows the micro to run multiple applications at the same time, with the processor appearing to be two processors. Such multitasking is available in Windows XP and Linux and probably all their successors. So where does this leave the future, are we going to go for greater and greater speeds, or will we develop multi-tasking so we effectively have greater and greater numbers of micros sharing the work? I have a feeling that task sharing will be the answer.

It seems likely that Intel is now back out in front.

Exciting times ahead . . .

Another update . . .

Almost immediately, Athlon has replied with what appears to be another significant step forward – 64-bit computing.

The microprocessor which as yet has been living with the codename 'Hammer' will be sold as the more user-friendly name of 'AMD Athlon 64' and will be available in mid-2003 and will join the PowerPC 970 in the '64' club. It will be able to run 64-bit, 32-bit and 16-bit applications without any speed penalty and so avoid the cost of buying new software.

The only technical information that is included in the initial announcement is a new bus system using 'hypertransport' technology which AMD claims to increase throughput by 50% over existing designs. Intel will have something to say about that claim, I expect. The clock speed of the first batch will be little different from the XP, around the 2.8 GHz, but the design will provide more scope for development and will be able to run programs at a higher speed.

Really exciting times ahead . . . over 3 GHz clock speeds, 64-bit computing and multiple instructions being carried out simultaneously. Sounds good.

The desktop speed Olympics is shared between the PowerPC 970, Pentium 4 and the Athlon 64 whereas the computer market is dominated by the IBM clones leaving just a minor role for the PowerPC 970 in the Apple-Mac. As we saw earlier, the result of any speed test does depend on the nature of the test. Having said that, and at the risk of irritating the fans of each, in the race for the overall speed freak the Athlon 64, when it is available, will appear to be the winner with the other two virtually shoulder to shoulder a pace behind. But it depends on the test chosen and we know that any speed king will be dethroned so very quickly.

A (very) approximate comparison based on the currently available information is shown in Table 14.1.

Table 14.1

	PowerPC 970	*Pentium 4*	*Athlon 64*
Clock speed	2 GHz	2.8 GHz	2 GHz
Bus speed	900 MHz	533 MHz	533 MHz
Bits	64	32	64
Process size	0.13/ 0.09 microns	0.13 microns	0.13 microns
Op systems	OSX IBM linux	Windows	Windows
Comparative speed	1988	1984	2372
Max memory	Terabytes	40 GB	Terabytes

Quiz time 14

In each case, choose the best option.

1 Compared with the Pentium 4, the Athlon XP design has:

(a) faster FSB, running at 533 MHz.
(b) the same speed of FSB.
(c) slower FSB, running at 333 MHz.
(d) faster FSB, running at 2.8 GHz.

2 As the Pentium 4 and the Athlon XP are both using 0.13 micron technology:

(a) it does NOT imply any other similarities between the designs.
(b) they will both run at the same clock speed.
(c) they will have the same number of pins.
(d) the cache sizes are equal.

3 The three floating point execution units in the Athlon XP simultaneously handle store and load, multiply functions and:

(a) SIMD functions.
(b) add functions.
(c) divide functions.
(d) 3D NOW! functions.

4 When Branch prediction is correct it:

(a) increases the overall speed of running the program.
(b) increases the length of the pipeline.
(c) decreases the clock speed.
(d) prevents overheating of the microprocessor.

5 The Athlon Instruction cache has a capacity of:

(a) 256 kB.
(b) 32 bits.
(c) 64 kB.
(d) 384 MB.

15

Microcontrollers and how to use them

Getting ready for takeoff

In the 1960s, electronics started to awake from its slumber that had used thermionic valve technology that was recognizably similar to circuits that had been built for thirty years. The pace of progress was gentle. The first semiconductor material was developed and the transistor came into use in just a few years. The photographic process used to design and produce the transistor quickly led to simple integrated circuits and the microprocessor.

The start of the microcontroller

No sooner had the microprocessor and the associated memories arrived in 1971 than it became obvious that the microprocessor was always accompanied by other circuits, like input/output devices, memory and timing circuits so it would be a good move to combine them into a single device.

We had a choice – we could keep everything general and universal and call it a microprocessor or design it for a single purpose and call it a microcontroller.

The multipurpose devices went into computers and even here we had a choice. Computers were either 'microcomputers' where price was a significant feature and these microprocessors had some built-in ROM and RAM. Soon, however, speed became the main feature as the prices began to fall and we could afford to equip our homes with computing power which equalled many offices of just a few years previously and the microprocessors became expensive and fast. Speed headlines drive the publicity machines as home and office computers became faster and faster. They sold in their millions.

Meanwhile the single-purpose devices, really the descendents of the early microcomputers, were developed further and made really cheaply, sold by the billion and were never mentioned. They power the pocket calculator, video recorders, cameras, microwaves, washing machines and greetings cards that play music – in fact almost anything vaguely electronic.

Just a thought

The microcontrollers outnumber the population of the world many times and as mentioned earlier we are likely to be sharing our homes with, possibly, fifty of them. They are in every essential industry – food production, transport, communications, research, weaponry, power generation, medicine, heating and air conditioning – there is little that we rely on that does not use a microcontroller. If they learn to communicate independently of us, they may develop their own agenda. Now there's a thought.

Most microcontrollers are similar

Once we have learned to drive, most vehicles are easily recognized as being very similar. We are happy with the general idea and can concentrate on the minor differences. Microcontrollers are much the same. Having already become familiar with the basic building blocks of the simple microprocessor in Chapter 8, we can move very easily into the microcontroller. It is not surprising then to find that all microcontrollers are basically very similar.

To give an overall impression of the range of microcontrollers available we are going to look at three popular ranges. The first is the 8051, probably the most widely used microcontroller, over twenty years old and continuously developed by many different companies and showing no signs of fading away. The next is from the AVR family produced by the Atmel Corporation, one of the leaders in this field. From this range we look at the AT90S/LS2343 one that is small, modern and RISC. The final one will be explored in Chapter 16.

The 8051

Probably the transition between the microcomputer to the microcontroller occurred with the Intel 8048 as we saw in Chapter 11. The 8048 added on-chip RAM, ROM and a timer so it could be used as a single purpose device such as controlling a keyboard – it was, in fact, a microcontroller.

With the experience gained by using this, it became apparent that there was a significant market for a microcontroller.

In 1980, Intel launched the 8051 which, twenty-three years later, is alive and well. In fact very well indeed. It is probably the most popular microcontroller ever. It is made by about 44 suppliers.

These suppliers have often added some extra features to make versions or 'variants' as they are called particularly suitable for specific jobs. There are at least 92 variants all compatible with the original code. Even within variants, there are a series of options that lifts the total number of members of the 8051 family to several hundred.

Numbering

The device numbering is not very obvious as many microcontrollers are available from several different suppliers with their own product code. They then produce a group of basically similar devices with minor changes like different operating voltages or differing amount of RAM and ROM on-board memory – these groups are referred to as 'families'.

The family is given a name which often has little connection with the product codes. For example, Intel's 8051 family has the family name of MCS51. This contains the 803X, 805X, 875X and the low power versions bXC45X and the 8XCX52. As usual the X refers to any figure or letter in that position.

The situation is further confused (or possibly simplified) by referring to all of them as 'the' 8051.

The block diagram of the 8051

The block diagram shown in Figure 15.1 is the family portrait of the 8051 family. There are some features that differ between the family

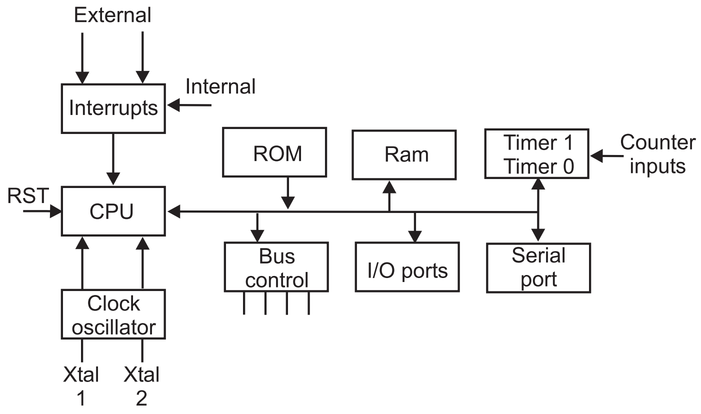


Figure 15.1 8051 block diagram

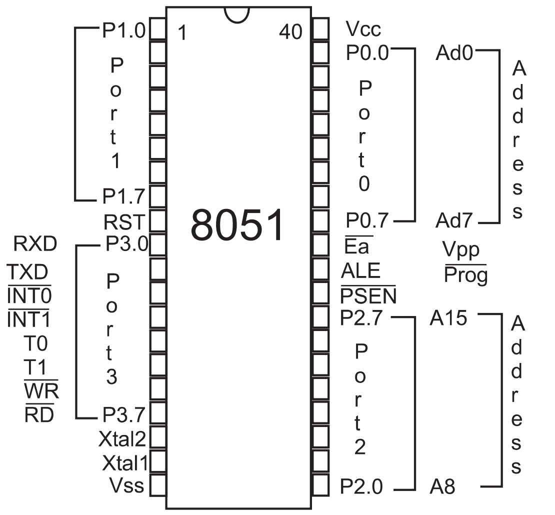
members, principally the memory configuration – some versions have less memory and some have none at all. However, we will look at the operation of our 'middle-of-the-road' version and worry about the individual differences later.

The 8051 pinout

As in all micro and digital chips, a line over a pin designation indicates that it is active low or, put more simply, to use this feature we need to apply zero volts.

The pinout shown in Figure 15.2 looks, at first glance to be rather complicated due to the dual use of many pins. This is a common feature of microcontrollers as a method of reducing the number of pins to be used. The more pins, the more expensive and the larger the device. This is bad news for a device often destined to be embedded within another circuit.

Figure 15.2
Pinout of the 8051



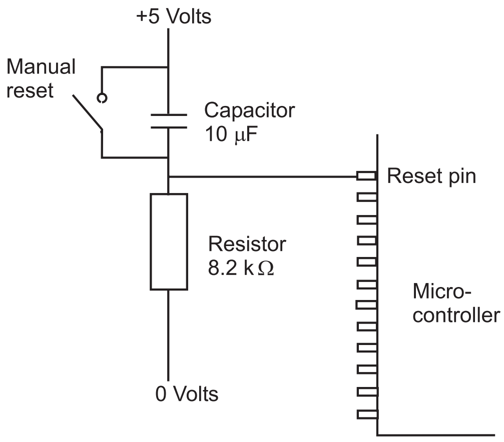
There are variants available that provide the increased number of pins so that there is a separate pin for each function.

Reset

Regardless of what program is being run, we must always be able to gain control of the microcontroller just as we must with a microprocessor. The procedure is just the same. The microcontroller has a reset pin which, in the 8051, is taken from the bus control block and, in normal operation must be held to zero volts. When a positive voltage over 2.0 V is applied to it the microcontroller immediately returns to its startup memory location, which in this case is 0000H. We can arrange this to occur automatically when the power is switched on but we should also provide a reset switch to gain control of the system at any time without removing the power. This is the 'reset' switch which we use when our computer locks up and ignores us.

When changing microcontrollers, remember to check the polarity of the reset voltage. Compare this circuit in Figure 15.3 with the one shown in Figure 8.7.

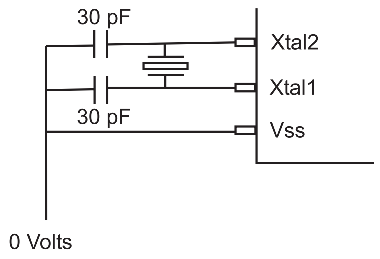
Figure 15.3
The reset switch



Clock input

As we have seen in Chapter 7, we are going to need a clock signal. The original design of the 8051 called for a 12 MHz crystal though later versions can run at 33 MHz, 40 MHz or even 44 MHz. As an alternative, we can use a ceramic resonator or an external signal. The clock input is shown in Figure 15.4 using a crystal. To use an external signal throw away the crystal and the capacitors then apply the external signal to the Xtal1 pin and leave Xtal2 disconnected.

Figure 15.4
The clock



Ports

We have four ports numbered 0 to 3. In the standard 8051 ports 0, 2 and 3 are dual purpose and port 1 is just an input/output port. In some variants, all ports are dual purpose.

With regard to the memory, we can use the on-chip memory, which keeps the circuit as simple as possible and makes the embedding of the device at its most convenient.

The on-chip memory may be masked ROM or an EPROM which we met in Chapter 6. To use external memory obviously increases the chip count but can allow up to 64 kB of memory, which is the standard size for 8-bit microprocessors.

The external memory is accessed by taking the 'external access' (EA) pin to a logic zero. To switch the external ROM on, the output enable (OE) pin of the external memory is taken low by the program store enable (PSEN) pin of the 8051. In a similar way, an external RAM is accessed for reading and writing by the read (RD) and write (WR) pins applying a logic zero voltage to the output enable (OE) and read/write (R/W) pins respectively.

Interrupts

The 8051 has a total of five interrupt signals. Two of these can be externally generated and three are of internal origin. Interrupts are discussed more generally in the chapter dealing with interfacing but basically, when an interrupt occurs, the program which is running at the time is interrupted and another code sequence is ran, called an interrupt service routine (ISR). When the ISR is compete, the microcontroller returns to its original program and continues as if the interruption has not occurred. There is a different ISR for each interrupt which must be pre-loaded in specific memory addresses, all ready to go if needed.

What if two interrupts occur at the same time?

The interrupts are checked continuously in what we call the polling order. Starting from the top, the order in the 8051 is:

External Interrupt 0
Timer 0
External Interrupt 1
Timer 1
Serial port

In addition, each interrupt can be given two levels of priority so if two interrupts occur, the one with the high priority will be handled first. If two have the same priority, it is decided by the polling order.

Timer/counters

A timer or counter is a series of bistables or flip-flops that change state once for every input signal, thus one of these circuits would divide the input frequency by a factor of two. If this signal is then fed into the next bistable, the output is $\frac{1}{4}$ of the original frequency. The next circuit would have an output of 1/8, then 1/16 and so on.

There are two timers, T0 and T1. These can be programmed to divide by 256, 8192 or 65536 and will generate an interrupt signal upon completion that can be detected by the software. One of the modes allows the timer/counter in 8-bit (divide by 256) mode to reload and start counting again each time continuously.

The input signal being counted can originate from an external circuit so it counts the number of incoming pulses, or it can use an internal signal which is actually 1/12 of the clock frequency in use. As mentioned above, it generates an interrupt signal when it reaches its maximum value. We can preload the timer with a number to start counting from. This will allow the interrupt to be generated after any required number of events, or time interval.

Serial port

Since the microcontroller normally handles data eight bits at a time, it is operating in parallel but two receive or transmit serial data we have perform serial/parallel conversions. This is always achieved by using a shift register working under the control of a clock signal. The working of a shift register is described in Chapter 17.

The serial port is able to transmit and receive data, at the same time, this is referred to as a full duplex system. As the name suggests, the bits of data are moved one after the other in a continuous stream. Pin 10 is called RXD or receive data and pin 11 is the TXD, transmit data but, as is often the case, things are not quite that simple.

It can operate in three ways, or modes numbered 0, 1, 2 and 3.

Mode 0

This is the case that spoils the simple RXD, TXD as, in this mode only, the TXD pin is actually used as a clock signal and the RXD is used to receive or transmit data. The clock frequency is fixed at 1/12 of the onboard oscillator frequency and this, of course, determines the speed at which the data is transferred via an 8-bit shift register.

Mode 1

This mode also sends data in 8-bit lumps but its frequency is adjustable and operates as an 8-bit UART (universal asynchronous receiver transmitter – see more about this in Chapter 17). The 8 bits are increased to 10 bits buy adding a logic 0 to indicate the start of the group and a logic 1 to mark the end of the group. This is the normal format used in RS232 transmissions. Unfortunately, the output voltages do not comply with the RS232 standards so an external chip must be added to do the voltage conversion. Some suitable chips are discussed in Chapter 17.

Mode 2

This is very similar to Mode 1 except the number of bits transmitted is increased from 10 bits to 11. The extra bit can be used as a parity bit which is used to check for transmission errors, The pattern is Start bit (0), 8 data bits, parity bit and stop bit (1). (Have a look at Chapter 17 again.) The transmission rate can be 1/32 or 1/64 of the onboard oscillator frequency.

Mode 3

This is an 11-bit transmission with a programmable baud rate. The baud rate is near enough the same as the transmission rate measured in bits per second.

Watchdog timer

When a microcontroller is embedded in equipment it may find itself used in areas where electrical interference is a problem. This, or a software problem can cause the microcontroller to lock up by getting into an endless loop. The watchdog timer will reset the microcontroller after a period of time, about 20 milliseconds, unless it is told not to. Stuck in a software loop, it no longer generates this ‘don’t reset me’ signal and so the microcontroller is reset and escapes from the loop.

The watchdog timer is only fitted into some of the 8051 variants but is available as a stand-alone chip but is commonplace in newer designs.

When we want to leave the microcontroller in a continuous loop, we have a choice of ensuring that the loop contains the required software code or disabling the watchdog timer before the loop is started.

AVR 8-bit RISC microcontrollers

Taking the AT90S/LS2343 as an example, we can see a really basic microcontroller with minimal complexity yet having many useful features that make it inexpensive, small and comparatively fast. The RISC design and the width of the registers allows the vast majority of instructions to be executed in a single clock cycle and all the others, apart from five, are completed in two cycles.

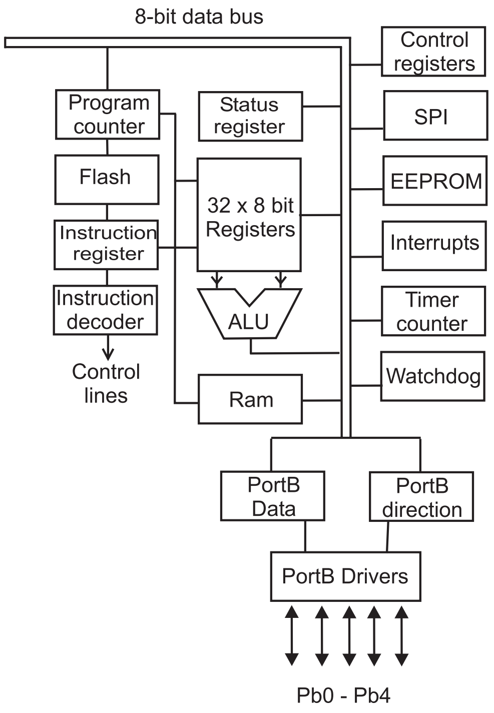


Figure 15.5
AVRAT90S/LS2343

In Figure 15.5, we have a block diagram. It is quite a relief to see that mostly it is quite familiar after looking at the earlier 8-bit microprocessors and the 8051 considered in previous chapters. Already we see that most devices are a combination of one or two innovative ideas added to a standard mix.

Inputs and outputs

One notable feature of the block diagram, we see only five input/outputs shown as PortB 0 to 4 so we have only five connections for data which is a small number until we look at the sister version, the AT90S/LS2323 which has only three – PortB 0 to 2! Yet they are called 8-bit devices and in previous 8-bit micros we have grown to expect at least one and sometimes two 8-bit input/output connections. The answer is just that the '8-bit' description refers to the internal data bus width.

The PortB pins can be used to send data in either direction, they can be used as inputs or outputs. As is common with other microprocessors and microcontrollers, the direction of data movement through each of the PortB lines is individually controlled by a data-direction register. Loading a 'one' into the data-direction register will make the corresponding PortB line into an output, on the other hand a ‘zero’, of course, will change it into an input.

Memories

This microcontroller has three memories, or four if we count the general purpose registers. The loading of the memory storage areas employs a serial transfer of data and is achieved by the SPI (serial programming interface).

One ROM storage area is achieved by a Flash memory organized as 1k × 16. The program instructions are all either 16 or 32 bits wide and can be cleared and reprogrammed whilst remaining in circuit. It can be cleared and reprogrammed at least 1000 times. The contents of the Flash memory cannot be changed by the program being executed by the CPU and so is free from accidental corruption.

The data is held in an EEPROM, which again can be cleared and reprogrammed electronically without removal from the device. It only holds 128 bytes of memory but is able to go through at least 100 000 cycles.

Data corruption can occur in the EEPROM if the supply voltage is reduced too far but this effect can be avoided by any one of the following three methods. The first we have already mentioned – use the Flash memory for critical data. The other two methods are ways to detect the reduction in voltage and immediately put the microcontroller into a safe condition. This is often referred to as ‘brown out protection’. An external circuit detects the falling voltage and applies a low voltage to the reset pin which effectively switches the chip off until the supply voltage recovers. The alternative is to put the microcontroller into a power-down sleep mode which is a power saving mode which has the effect of preventing any decoding or execution of any instructions – which, of course, precludes any ‘writes’ to the EEPROM.

It also has 128 bytes of SRAM (Static RAM) for the temporary storage of data and 32 8-bit general purpose registers that can be connected two at a time with the ALU (arithmetic and logic unit) which is the heart of the ‘brain’ within the microcontroller.

Clock

The AT90S/LS2343 has an internal RC oscillator which runs at 1 MHz, 4 MHz or 10 MHz depending on the version in use. It is one of the few of the micro devices that does not make use of an external crystal although it can use an external clock pulse. This external clock pulse only requires a single pin and hence we have an extra pin to use as an output.

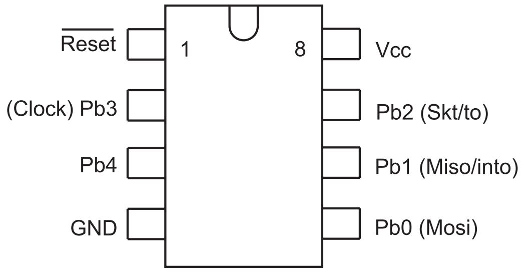
Interrupts

There are only three interrupts. The first, and highest priority, is the reset which is activated by a low voltage applied to pin 1, a power-on reset or a signal from the watchdog. The next is an external interrupt request as described in a moment. Lastly, an overflow from the timer/counter circuit.

Pinout and package

These are shown in Figure 15.6 and we can see that it is available as an 8-pin DIL (dual in line) package which has two lines of pins and also the surface mount version, plastic gull wing SOIC (small outline IC).

Figure 15.6
AT90S/LS2343 pinout



Pin 1 – Active low reset. Must go low for at least 50 ns.
Pin 2 – External clock signal input or PortB 3
Pin 3 – PortB 4. All lines can sink 20 mA and therefore are able to power LEDs directly. Sinking means that the LED or other load is connected between the positive Vcc supply and a low voltage output at the port.
Pin 4 – Ground.
Pin 5 – PortB 0 or MOSI. In serial programming mode, MOSI is the serial data input.
Pin 6 – PortB 1 or MISO/INT0. In serial programming mode, MISO is the serial data output. This pin can also act as the external interrupt described in the previous paragraph.
Pin 7 – PortB 2 or SCK/T0. In serial programming mode, SCK is the serial clock input. This pin can also provide the timer/counter0 clock input.
Pin 8 – Vcc. The LS version requires a positive supply voltage that remains in the range 2.7–6.0 V.

Sleep modes

When the microcontroller is not being used, it can switch off some of its circuitry to save power. Sleep modes are employed in all modern microcontrollers and make an enormous difference to the overall life of an intermittently used device. This enables sealed units in toys and greetings cards to remain active for months or years.

Idle-mode

To see the benefits, this microcontroller has a normal operating current drain of 2.4 mA but when switched to the 'idle' mode, the current falls to 0.5 mA. It does this by stopping the CPU activity but allows the timer/counter, watchdog and interrupts to remain operational. This is about an 80% power reduction but we can do a lot better than that otherwise my musical socks would have stopped long ago.

Power-down mode

In this mode, only the external interrupt and watchdog (if enabled) continue to work and current falls to less than 1 microamp, which is a really impressive reduction in power. The microcontroller can be aroused from its sleep only by one of the following: an external reset, the watchdog (if enabled) or INTO external interrupt.

The PIC16F84A

This is another modern RISC development and has many features that are similar to the AVR that we have just looked at. The AT90S/LS2343 was chosen as representative of the very small and basic micro-controllers found embedded in many products. This PIC16F84A example from Microchip Technology is a mid-range device which is larger and more capable than the AVR.

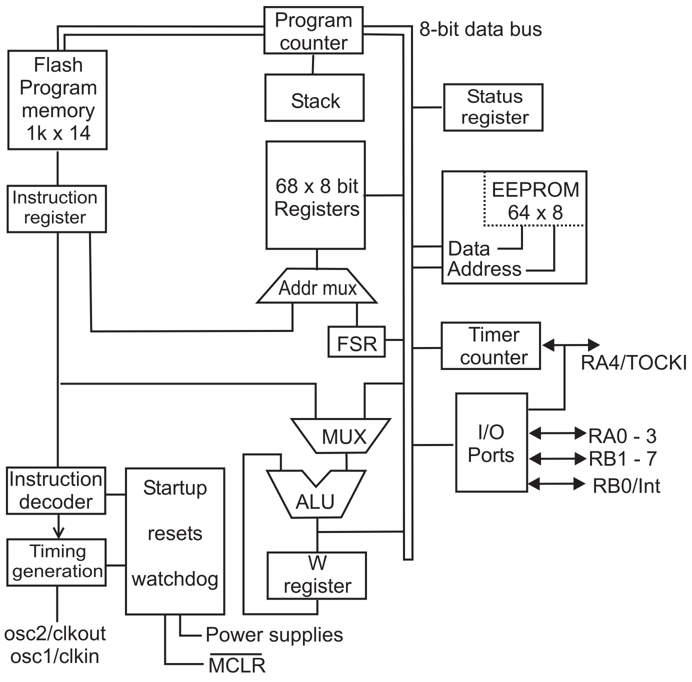
The PIC series ranges from a really simple 8-pin, 4 MHz micro-controller on a level with the AT90S/LS2343 that we have just considered up to a 40-pin 25 MHz device. As mentioned the PIC16F84A is a mid-size version that has 18 pins and runs up to 20 MHz.

The PICs are designed for easy use and are becoming increasingly popular as the first step into the world of microcontrollers. Microchip Technology provides a PICSTART™ PIC development system that provides, at a very reasonable price, an assembler, compiler, EPROM and EEPROM programmer, all the hardware manuals and even a sample PIC to play with. It should be mentioned that other companies have similar systems compatible with the PIC series and for other microcontrollers like the AVR and 8051 series.

It has proved to be such an easy, off the shelf, starting point that to many people 'PIC' is not only their first choice but is becoming used as a generic term for any microcontroller.

The general layout of the PIC16F84A is shown in Figure 15.7.

Figure 15.7
PIC16F84A



Supply voltage

The DC supply voltage must remain in the range 2.0–5.5 V for it to work happily though similar versions such as the PIC16F84 can run between 2.0 and 6.0 V.

Sleep mode

To save the current drain, a software instruction can put the PIC into a sleep mode. The supply current is very dependent on the clock frequency and is normally between 1 and 20 mA and when put to sleep the drain is reduced to approximately 1 μA.

To obtain the lowest possible sleep current we should hold all I/O pins at V_{DD} or V_{SS} and disable any external clock and hold the master clear pin in a logic high state – not in the reset state. The purpose of all this is to prevent any voltages from floating up and down. If it did so, it would switch and the technology used results in very low currents drawn except at the moment of switching during which it causes a really high spike of current so the higher the frequency, the more often this spike occurs – hence the increased current.

The microprocessor will wake if: a reset occurs by a logic low voltage being applied to the MCLR pin, a wakeup pulse arrives from the watchdog unit (if it is enabled) or an 'EEPROM write complete' signal.

Memory

As usual, we have two blocks of memory. One is the program memory and the other is for data. For maximum speed, they each have a separate bus connection so that both memories can be accessed during a single clock cycle.

Program memory

The program memory is situated in the flash memory which is organized as 1028 × 14. All instructions in the PIC16 series use 14-bit instructions.

The reset vector points to address 0000H and the interrupt vector is 0004H so address locations 0005H to 03FFH are available for us to hold our programs.

Data memory

The data area is subdivided into two areas, the FSR (file select register) and the GP (general purpose registers) as shown in Figure 15.8.

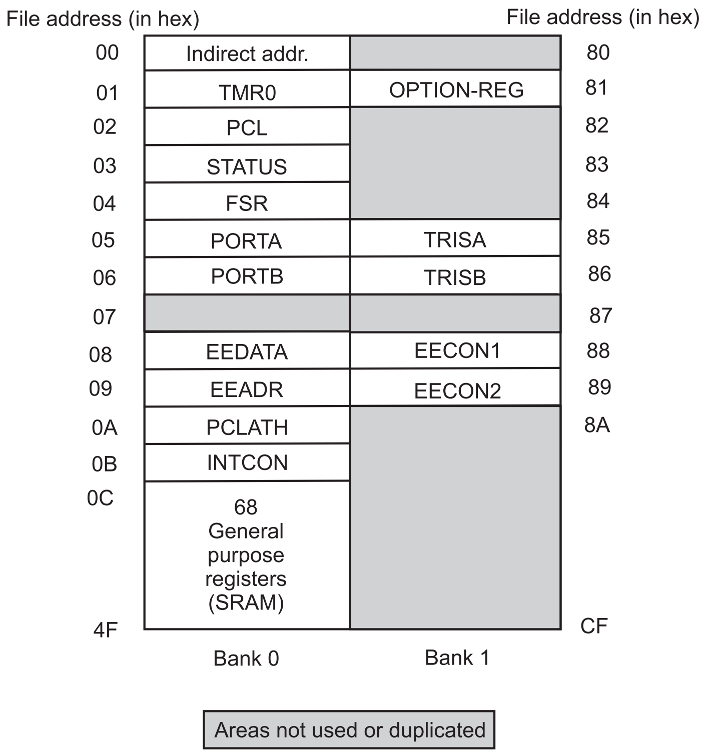


Figure 15.8
Register file map

The SFR (special function register)

This register controls the operation of the CPU and involves such things as the input and output ports, EEPROM address and data, timer, program counter and that sort of housekeeping.

All the register files are 8-bits wide and are arranged in two banks called bank 0 and bank 1. We have to instruct the microcontroller as to which bank is to be used and this is done by using special instructions to access some of the page 1 registers. Those accessible are indicated in Figure 15.8. Microchip Technology are planning to remove the choice of using the OPTION and the two TRIS registers and suggest that the STATUS register is used instead. This does not affect the use with this chip but it will ensure that upgrading in the future will not require any modifications to the software.

I/O (input/output) ports

All outputs can source or sink 25 mA and can therefore power significant external circuits without further power amplifiers being required. Sinking means that the load is connected between the positive Vcc supply and a low voltage output at the port and sourcing is connecting the load between a positive output on the pin to the ground.

PortA and TRISA registers

PortA is a 5-bit wide bi-directional port, each line being individually controlled so some of the lines can be inputs whilst the others are outputs. The choice of input or output is made by loading a 0 (output) or a 1 (input) into the appropriate bit of the data direction register TRISA.

In common with other devices, when it first starts at power-on, the port is set as an input. This provides a safer option that running the risk of random information being sent out to whatever it is connected to.

PortB and TRISB registers

PortB is a 8-bit wide bi-directional port, each line being individually controlled using TRISB in the same manner as in PortA. Each of the PortB pins have a weak internal pull-up which can be switched on or off by the RBPU of the option register. The pull-ups are disabled when the port is being used as an output and also during power switch-on.

Any of the Pins RB4–RB7 that just happen to be configured as an input have an interrupt-on-change feature that can be useful. If any one or more of these pins have changed logic state since they were last read, it causes an RB port change interrupt. This interrupt can be used to wake the microcontroller from its ‘sleep’ mode.

Status register

This is very similar to the one we met when looking at the Z80 in Chapter 8.

The bits are:

Bit 0 is the C(carry/borrow) bit. 1 = a carry out from the MSB (most significant bit) of the result otherwise it is cleared to zero. For 'borrow' the values are reversed. Subtraction is carried out using the two's-complement method that we met in Chapter 4.

Bit 1 reflects the carry situation that last occurred from the 4th bit of the result. This is also called the half-carry bit.

Bit 2 is the zero flag. It is set to ONE when the result of the last arithmetic or logic operation is ZERO. Be careful not to misread this.

Bit 3 goes to 0 after running the SLEEP instruction.

Bit 4 goes to 0 when a watchdog time-out has occurred.

Bit 5 is used to select between the two memory banks. It is cleared to 0 to access Bank 0 and set to 1 if we need access to Bank 1.

Bit 6 and bit 7 – not used. It will be used in the future so by programming them for 0, future compatibility will be assured. This may save a lot of time if our program is used on an upgraded version.

Option register

As the name suggests, it offers a series of options. One example is the control of the prescaler.

The Prescaler

Two functions are affected by the prescaler, they are the timer, TMR0 (timer zero) and the watchdog timer. Each of these circuits provides an output pulse after the count overflows and restarts from zero. In the case of the watchdog, the time interval is about 18 ms. If a longer time interval is needed, we have three alternatives. We can simply switch the watchdog off but, of course, we lose the benefits of the watchdog if the microcontroller gets caught in a loop. A simple way is to place the software code CLRWDT (clear watchdog timer) in the program at anytime before the end of the countdown so it is reset for another 18 ms and we can repeat this as necessary. Lastly, we can use the prescaler to reduce the frequency of the incoming pulses and hence increase the time before the output signal is generated.

Bits 0, 1 and 2 PS2:PS0 (prescaler rate select bits) provide eight alternative pulse rates for use by the watchdog or TMR0. Setting the three inputs to 000 will provide a clock signal which is equal to an external signal or chip oscillator. Changing the setting to 001 will halve the frequency (or double the time). Increasing the setting to 010 will decrease the frequency to one quarter of the original. As the count increases, the frequency will halve for each count until we reach the maximum bit value of 111 which will cause the watchdog time to increase by a factor of 128. As it happens, the TMR0 has a divide by two circuit built in all the time so a setting of 000 will halve the frequency and the maximum count will reduce the frequency by a factor of 256.

Bit 3 PSA – (prescaler assignment bit) – Unlike most other PICs, the prescaler can be applied to the TMR0 or the watchdog, but NOT both, so the option register controls the choice. Bit 3 of the option register is set to 0 to prescale the TMR0 and a 1 selects the watchdog.

The other bits

Bit 4 T0SE (TMR0 source edge select). This controls the moment of timing the clock input. A '0' increments the count on the low-to-high transition and a '1' increments on the high-to-low transition.

Bit 5 T0CS (TMR0 clock source select). This decides where the clock pulses come from. The choices are a '0' to an internal clock as on the CLKOUT pin and a '1' counts the transitions on the RA4/T0CKI (timer zero clock input). This option allows pulses to be generated by any external source like cans of beans moving along the conveyor belt or the revolutions of an engine.

Bit 6 INTEDG (interrupt edge select). This is similar to bit 4 except we are controlling the moment at which an interrupt signal is recognized of the RB0/INT pin. A '0' uses the falling voltage edge and a '1' sets the rising edge.

Bit 7 RBPU (PortB pull-up enable). This controls the 'pull-ups' on the PORTB output. A '0' allows each line to have its own pull-up enabled or switched off as required. A '1' switches them all off. A pull-up circuit is shown in Figure 15.9. When the switch is closed by applying a '0' state to this pin it connects the output to the positive supply via a current limiting resistor. This ensures that in the absence

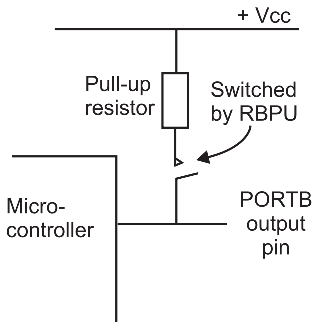


Figure 15.9
A pull-up circuit

of any input data the port will be pulled up to a positive state so that it doesn't wander about applying unpredictable inputs. The higher the value of the resistance used, the easier an incoming voltage finds it to bring the voltage down and this is referred to as a 'weak' pull-up like this one. Likewise, a reduction in the resistance value will make the port input more determined to stay high and this is referred to as a 'strong' pull-up.

PCL (program counter low)

PC is the program counter that keeps track of the instruction being executed at the time. It is a 13-bit register divided into PCL for low byte and PCH for high byte. Its behaviour is common with other microcontrollers and microprocessors as described in Chapter 8.

Stack

This register stores the return address when interrupts occur. It stores up to eight 13-bit addresses. Again, there are more details in Chapter 8.

Register 6–1: PIC16F84A configuration word

This is a special register which is just memory location 2007H which can only be accessed during programming. Unprogrammed pins are read as '1'.

It is a 13-bit word with the following options:

Bit 13–4. This protects the program code from being read after the programming is complete. 1 = no protection 0 = all program memory is code protected.

Bit 3 Power-up timer enabling. This allows a nominal 72 ms delay when first switching the microcontroller on to allow power supplies to settle. The delay can be activated by loading bit 3 with '0', no delay = '1'.

Bit 2 watchdog timer, disable with '0', enable with '1'.

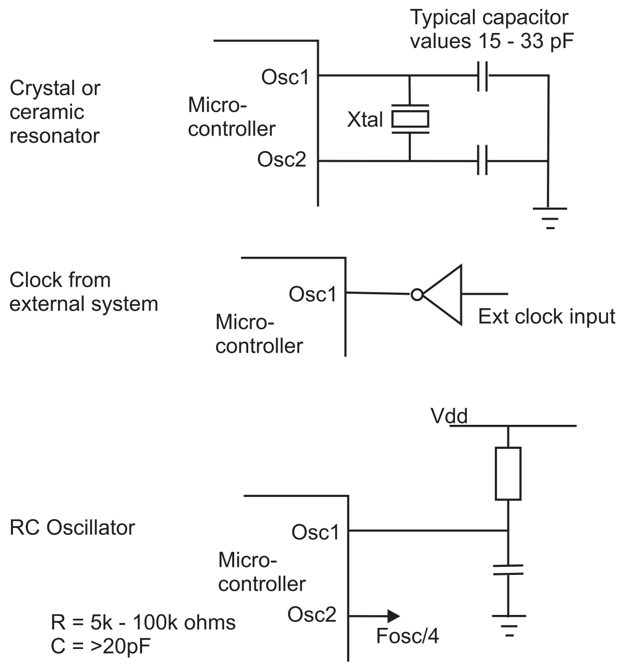
Bits 1–0 oscillator selection bits
00 = LP oscillator
01 = HS oscillator
10 = XT oscillator
11 = RC oscillator

Oscillators

The crystal or ceramic resonator options are shown in Figure 15.10 and the crystal oscillators are divided into the frequency ranges:

LP low power crystals 32 kHz–200 kHz.
XT crystal/ceramic resonator 100 kHz–4 MHz.
HS high speed crystal/ceramic resonator 4 MHz–20 MHz.
RC resistor/capacitor.

Figure 15.10
Oscillator options



A note on ceramic resonators

These offer an alternative to crystals for oscillators. They are generally lighter, more shock resistant and are usually somewhat smaller and cheaper to buy. The downside is that they do not have quite the frequency accuracy or stability and in many respects they are positioned somewhere between an RC network and the quartz crystal.

An interrupt summary

The PIC16F84A has four sources of interrupts:

(i) External interrupt on the RB0/INT pin.
(ii) TMR0 overflow.
(iii) PortB change detector pins RB7–RB4
(iv) EEPROM data write complete interrupt. This is used only when we are loading new programs into the EEPROM.

Quiz time 15

In each case, choose the best option.

1 An 8-bit microcontroller has:

(a) an 8-bit data bus.

(b) eight ports.
(c) an 8-bit address bus.
(d) eight internal registers.

2 Compared with the Pentium 4, a microcontroller:

(a) is faster.
(b) consumes less current.
(c) is more complex.
(d) is larger.

3 A watchdog circuit prevents:

(a) damage due to excessively high supply voltages.
(b) an overrun in timer/counter circuits.
(c) burglars.
(d) wastage of power while the microcontroller is not being used.

4 It is NOT essential for a microcontroller circuit to include:

(a) registers.
(b) a reset pin.
(c) some memory.
(d) a crystal.

5 The term 'ISR' refers to an:

(a) immediate service register.
(b) internal system reset.
(c) interrupt service routine.
(d) instruction set register.

16

Using a PIC microcontroller for a real project

PIC microcontrollers are a very convenient choice to get started in this field although, as we have seen, there are enormous areas of overlap between microprocessors and other microcontroller designs.

One convenience is that Microchip Technology has taken the RISC concept seriously. There are only 35 instructions and, of these, only a few are required to write quite usable programs.

All data movements are based around just a single register called 'W' for 'working'. This performs much the same function as the accumulator in the earlier processors like the 6502.

Getting started

To program the PIC we need a copy of software instructions and more details of the register layout.

There is no getting away from the fact that when we first meet a microcontroller, the information appears overwhelming – not only in the quantity but in the apparent complexity. This is despite the efforts of the designers to make it as simple as possible. It is very much like seeing a new foreign language, which it is really – and tackle it in much the same way, a bit at a time.

There are four golden rules which may help:

1 Don't get scared! From time to time, it will seem tempting just to throw it all away and find something easier to do.
2 Start really simple – one step at a time.
3 Be prepared for it to take a long time in the early stages.
4 When you are feeling low and despondent, remember that everyone else has felt the same. It will get better. Honest.

The hardware

Let's keep it really simple. We have to buy a microcontroller and connect it up and then program it.

The device that we are going to use is a PIC16F84A-04/P. The pinout is shown in Figure 16.1. The number PIC16F84A is the type of microcontroller. The 04 tells us that the maximum frequency is 4 MHz and the /P means that it is the standard plastic package. There is no minimum operating frequency and the slower it runs, the less power it consumes.

Figure 16.2 shows the most basic circuit. It includes only the chip supplies and a positive supply to prevent the MCLR (master clear) from

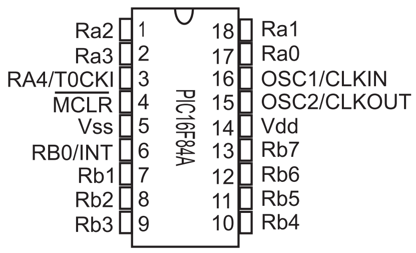


Figure 16.1

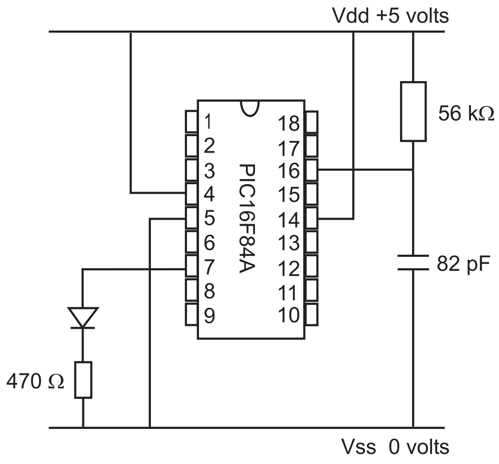


Figure 16.2

accidentally resetting the PIC during the program. Unless they are tied to a definite value all disconnected inputs will float up and down and can cause random switching.

We need a clock signal and the simplest and cheapest is just an RC combination. The resistor should be between 5 kΩ and 100 kΩ and the capacitor should be greater than 20 pF. The values chosen for this circuit were not selected carefully and are not critical. In practice, it is difficult to predict the operating frequency of an RC combination. If we need a particular frequency it is better to use a variable resistor and adjust it to give the desired frequency.

Later on, when we have written the program we must tell the assembler what clock signal we are going to use. This programs the PIC to expect an RC oscillator. This information can, as an alternative, be included into the program. It's our choice but it is slightly easier to do it at the assembly stage but we must remember to do it.

The software

There are three layers of information that we need to use the registers. Firstly, we need the names and addresses of the files as we met in the register file map in Figure 15.8. The second step is the function of each bit in each register, and this is shown in Appendix A, and finally, an explanation of these functions as we see in Appendix B.

All this data looks daunting but the thing to remember is that we only use one bit at a time and we can ignore the rest.

All PICs and indeed all microprocessors and microcontrollers use binary code and nothing else. To make life easier for us, we use assembly language or a high level language. To use these languages, we need a program which is able to convert them to binary code. For assembly language, the required program is called an assembler and for higher level languages a similar job is done by a compiler.

An assembler is part of the PICSTART PLUS development system but there are other assemblers available that will handle code for all the PICs. The code that we write as the input to the assembler is called 'source code' and the code that is supplied by the assembler is called 'object code'. The object code is really in binary but to make it easier for us, it is usually displayed on the screen in hex.

Important note

As we type in the program we must not leave any spaces in instructions or in the data.

The programming steps

The purpose of our first program

We want to do something really simple but not something that may have happened by accident. We are going to configure all of PortB as outputs and then make the output signal to be off, on, off, on, off, on, off, on. We can connect a series of LEDs to the output pins so alternate lights will be on.

1 We will start by selecting Bank 1 of the register file map. To do this we need to talk to the status register, so by referring to Figure 15.8, we can see that the status register is file 03. From our look at the Status register in Chapter 15, we can see that bit 5 is the RP0 which controls the selection of Bank0 or Bank1. The first line of the program must set bit 5 of register 3.

Program starts:

```
BSF     3,5    ; Sets bit 5 of register 3 to select Bank1
```

It is worth mentioning at this stage, that anything written after the semicolon is just a note for us and is ignored by the assembly program. This is called the comment field. It is always worth using this area to explain the program, and this often saves hours trying to remember what we were attempting to do when we first wrote the program. This can be an even larger problem if someone else writes a program and is then off sick and we are left to find out why the program doesn't work.

2 We now want to arrange for PortB to be an output. This involves loading a 0 into each of the controlling bits in the data direction register which we can see from Figure 15.8 is called TRISB and is register number 86. To do this, we are going to clear the W register using the CLRW instruction and then copy this zero into register 86. This involves two extra instructions so, at this stage, our program will read:

```
BSF      3,5    ; Sets bit 5 of register 3 to select Bank1
CLRW            ; puts a zero into register W
MOVWF  86       ; copies the zero into register 86 which is
                ; the PortB data direction register
```

3 Now we have sorted out the direction of the data flow and we can input the actual data. This time we are going to clear bit 5 of register 3 to allow us to access Bank0. We can now load our data into the W register. But what is the data? If we assume the LED is going to be connected between the port output and the zero volt connection, the voltages corresponding to the light sequence off, on, off, on, off, on, off, on will be 0V, +5 V, 0 V, +5 V, 0 V, +5 V, 0 V, +5 V and the data will be 0,1,0,1, 0,1,0,1. We could enter this as a binary number written as

B'01010101' or as the hex number 55 which is a little easier to read.

Once the 55H is in the W register, we can use the code MOVWF to copy it into register 06 which in Figure 15.8, we can see is the PortB data register.

The program is now:

```
      BSF     3,5    ; Sets bit 5 of register 3 to select Bank1
      CLRW           ; puts a zero into register W
      MOVWF   86     ; copies the zero into register 86 which is
                     ; the PortB data direction register
      BCF     3,5    ; clears bit 5 of register 3
      MOVLW   55     ; this is the output data to give the on, off
                     ; sequence
      MOVWF   06     ; this copies the data into PortB
```

4 As it stands, the micro will perform each of these steps once and then stop. We have a problem here because it will take only a few microseconds to complete these instructions – certainly too fast for us to see if the correct sequence of LEDs are illuminated. We need to give the micro something to do which will keep the LEDs operational and our choice here is to reload the output port continuously. Have a look at our new program:

```
      BSF     3,5    ; Sets bit 5 of register 3 to select Bank1
      CLRW           ; puts a zero into register W
      MOVWF   86     ; copies the zero into register 86 which
                     ; is the PortB data direction register
      BCF     3,5    ; clears bit 5 of register 3
      MOVLW   55     ; this is the output data to give the on,
                     ; off sequence
again MOVWF   06     ; this copies the data into PortB
      GOTO    again  ; this line forces the micro to return to
                     ; the previous line
```

The words 'again' are called labels and the assembler program notices that the two are identical and replaces them by the correct address. The fact that we have used 'again', a word that makes sense in the context is just to help us to understand the program, the assembler would accept 'asdf' or anything else just as happily. Some assemblers put restrictions on the names chosen. It may, for example, not allow it to start with a number, or use certain words or symbols.

5 At the end of the program, we have to put the instruction END to tell the assembler to stop. It is called an 'assembler directive' and is there to tell the assembler program that it has reached the end of our program. Directives are not instructions to the microcontroller and are not converted to machine code.

6 At the start of the program we can use the directive ORG which means 'origin' and gives the starting address for the assembled program. This has been added to our final program. If we had not done this, the assembler will assume the address to be zero so, in this case it would make no difference whether we added this directive or left it out. When the PIC is reset, it always goes to address 000 so if we wanted the program to start elsewhere we would have to leave a GOTO instruction at address 000 to tell the microcontroller where it is to start. Remember that the ORG is only an instruction to the assembler telling it where to start loading the program – the PIC doesn't know anything about this because directives are not converted to the program code.

So our final program is:

```
      ORG     000    ; Program starts at address 000
      BSF     3,5    ; Sets bit 5 of register 3 to select Bank1
      CLRW           ; puts a zero into register W
      MOVWF   86     ; copies the zero into register 86 which
                     ; is the PortB data direction register
      BCF     3,5    ; clears bit 5 of register 3 to select
                     ; Bank0
      MOVLW   55     ; this is the output data to give the on,
                     ; off sequence
again MOVWF   06     ; this copies the data into PortB
      GOTO    again  ; this line forces the micro to return to
                     ; the previous line
      END            ; the end of our code to be assembled
```

Notice how the program is written in columns or 'fields'. It is necessary to use the correct fields as this tells the assembler what the items are. Remember to use the semicolon to start notes that we wish the assembler to ignore.

Connecting the LEDs

For clarity, only one LED is shown but an LED and resistor should be joined to all the pins 7–13 to show the full output.

LEDs come in different colours and sizes and the cathode must be connected to a less positive voltage than the anode. The cathode is generally recognized either by a shorter connector wire or a flat moulded onto the body.

Component values

Looking at the data for a standard red LED, the typical voltage (V_f) across them when lit is 2 volts with a maximum current of (I_f) 20 mA. The small '$_f$' stands for 'forward'. The light lost by reducing the current below its maximum value is not very great and it would be quite reasonable to operate the LED on, say, 10 mA.

To limit the current flow, a resistor is connected in series. Now, if the supply voltage for the microcontroller is 5 volts and about 0.7 volts are 'lost' inside the PIC and the LED is using 2 volts, the series resistor must be 'using up' the other 2.3 volts. The value of the resistor is given by $R = V/I = 2.3/(10 \times 10^{-3}) = 230\,\Omega$. If in doubt start with 470 ohms and see how it goes – this is a generally safe value for all situations.

More labels

The use of labels not only makes the program more readable but it allows modifications to be accommodated. For example, if we put in the actual address instead of the label and then modified the program by adding an extra instruction, the actual addresses would all shuffle along a bit to make room for the new instruction, making our old address inappropriate. The program would not work and it might take us hours before we see what we have done whereas a label would be sorted out by running it through the assembler with the new instruction added.

There is another useful assembler directive, EQU, which is an abbreviation for equates or 'is equal to'. This can be used to make programs more readable by replacing some of the numbers with words. For example, register 86 is the PortB Data Direction register but the program would be easier to read if we replaced the number by the name. This would be done adding the line: PortBDDR EQU 86 before the program listing so as soon as the assembler spots the name PortBDDR it would replace it with 86. This has no affect on the final program but it makes life easier for us – which has got to be a 'good thing'.

If we add some other labels, the final program can now be written as:

```
;EQUATES

PortBDDR   EQU   86              ; PortB data direction reg.
                                 ; is register 86
PortB      EQU   06              ; PortB data register is
                                 ; register 06
Status     EQU   03              ; Status register is register
                                 ; 03
RP0        EQU   05              ; Bank1 is selected by bit 5
Data       EQU   55              ; Data used is 55H
           BSF   Status,RP0      ; Sets bit 5 of register 3 to
                                 ; select Bank1
```

```
               CLRW                    ; puts a zero into register
                                       ; W
               MOVWF   PortBDDR        ; copies the zero into
                                       ; register 86 which is the
                                       ; PortB data direction
                                       ; register
               BCF     Status,RP0      ; clears bit 5 of register 3
               MOVLW   Data            ; output data to give the
                                       ; on, off sequence
again          MOVWF   PortB           ; this copies the data into
                                       ; PortB
               GOTO    again           ; this line forces the micro
                                       ; to return to the previous
                                       ; line
               END                     ; the end of our code to be
                                       ; assembled
```

By making full use of labels, we have rewritten our program without any numbers at all. This is just a matter of choice – all labels, some labels or no labels, whatever we like.

Using a crystal

This gives a more accurate clock speed so that programs that involve real can be written. It may be that we want a display sequence to run at a particular rate.

To change to a crystal we need to set up the configuration bits in the PIC so that it knows that it is being controlled by a crystal instead of the RC method. This is most easily handled during the assembly process by clicking on 'configuration bits' and selecting the clock source from the options offered.

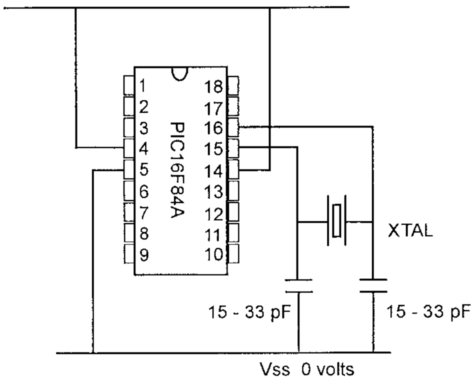


Figure 16.3
A Crystal Controlled Clock

The two capacitors shown in Figure 16.3 in the clock circuit always have equal values and the range shown is suitable for 200 kHz and all clocks of 2 MHz and over. Other recommended values are: 32 kHz – 68/100 pF and 100 kHz – 100/150 pF. The higher values in each range results in higher stability but slower startup times.

A ceramic resonator can be used as a plug-in replacement for the crystal.

A modification to the program

In the last program we controlled the voltages to each of the PortB outputs. With slight modifications we would be able to apply any combinations of voltages to control any external circuits. Even this first circuit has significant control capabilities but now we are going to extend the capability by applying a counting sequence to the output signals.

All programs are built on the backs of other programs that we have used before so we can save considerable time by keeping copies of our successful programs to be recycled whenever possible. This is well demonstrated in this example.

The program consists of three steps, two of which we have already designed and tested, so we know it works. If the new program refuses to work, we don't have to start from scratch, we know two–thirds of it is OK. This is a very powerful method of designing programs and whole libraries of programs are available so new developments can be reduced to slotting together ready-made program segments.

When we make changes to a previously program, it is important to save the new version under a new name so that, in the event of a disaster, we can retreat and start again.

Here is the section that we have 'borrowed' from our previous work:

```
ORG      000
BSF      3,5
CLRW
MOVWF    86       ; PortB data direction = output
BCF      3,5
MOVLW    55
MOVWF    06       ; PortB data set to a start value
```

At this stage we can, of course, set the start value for the output to any value between 00H to FFH which is binary B'00000000' to B'11111111'.

We have only one new instruction to worry about: INCF f,d. It increments or increases the value of a selected file 'f' by 1, and where the new value goes to is determined by the value of the 'd' term. If 'd' is 0 the new value is put into the W register but if it is 1, the new value is put back into the register in use.

PortB data register is register 06 so the code INCF 06,1 will take the current value of PortB data, increase it by 1 and put the answer back into PortB data so our starting value of 55 will change to 56 and the output voltages on the pins will change from 01010101 to 01010110.

This was just a single count, but for a continuous count we could use a label to make the program jump back and do the INCF trick again and again. When it reaches its maximum value, it will roll over to zero and start again so the count process can be continuous.
Our program would now be:

```
      ORG       000
      BSF       3,5
      CLRW
      MOVWF     86       ; PortB data direction = output
      BCF       3,5
      MOVLW     55
      MOVWF     06       ; PortB data set to a start value
again INCF      06,1
      goto      again    ; go back and INCF again
      end                ; end of source code
```

One more step

The speed at which the count continues is determined by the rate at which instructions are being followed.

Slowing things down

If we wish to slow things down, we can give the microcontroller something to do just to keep it busy. We have a NOP instruction which does absolutely nothing but takes one instruction cycle to do it. Since it doesn't do anything, it doesn't matter how many we include in a program, or where we use them. For a significant delay we made need hundreds, which is not an elegant way of solving a problem.

In the last modification to the problem, we made it count up on the PortB register. Now this takes time, so we could use this counting trick as a time waster. The PIC has 68 general purpose registers that can be made to count for us. Just choose any one of them and have it count for a set number of counts and then we can go back and count once on the PortB register, then go back to the time waste count. In Figure 16.4, we have loaded a register with a number, say

30H (48 in decimal). The next instruction decreases it by 1 to give 2FH (not 29!!!) and since the answer is not zero, we go around the loop and decrement it again to 2EH and so on until it gets to zero whereupon it leaves the loop to carry out 'instruction 2' shown in the figure.

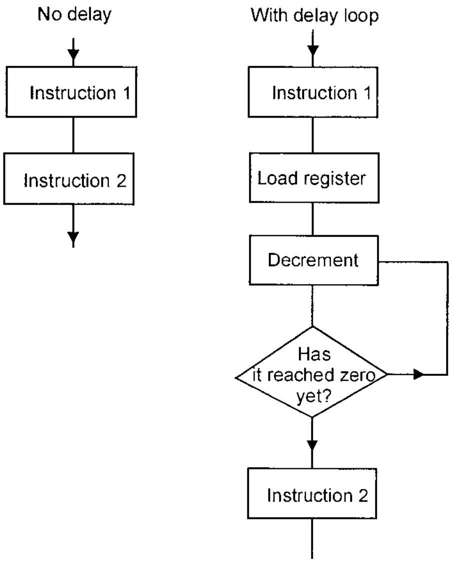


Figure 16.4
Using a timing loop

The instruction we are going to use this time is INCFSZ f,d. This is designed just for this type of counting job. It decrements the chosen register and if d = 0, the result goes into the W register but if it is 1, it will go back into the same register. For our purposes we would load the code as DECFSZ 20,1. This would decrement register 20 and put the answer back into register 20. When this register reaches zero, it will miss out the next instruction to stop it going around the loop again and will move on to the next instruction.

A slower count

Once again, this uses some of our previous programs.

```
      ORG     000
      BSF     3,5
      CLRW
      MOVWF   86     ; PortB data direction = output
      BCF     3,5
      MOVLW   55     ; PortB data set to a start value
      MOVWF   06
again INCF    06,1
```

```
      MOVLW    30      ; Loads W with 30H
      MOVWF    20      ; puts the number 30 into file 20
count DECFSZ   20,1    ; decrements register 20
      goto     count   ; keeps decrements until it gets to zero
      goto     again   ; returns to increment PortB
      end
```

For the slowest count on PortB, we would have to increase the count number in register 20 to its maximum number which, using this microcontroller is 7FH or 127 in decimal.

Calculating the delay

Starting from the moment that PortB is incremented:
MOVLW takes 1 count.
MOVWF takes 1 count.
DECFSZ takes 1 count normally but 2 when it leaves the loop.
As the register was loaded with the hex number 30, which is 48 in decimal, it will go around the 'count' loop 47 times at 1 instruction clock each and 2 clocks as it leaves the loop. This gives a total of 49 cycles.
goto will be used 48 times at 2 clocks each giving a total of 96 clocks.
goto will also be used once to return to the PortB, this is another 2 cycles.
Finally, INCF takes 1 count to increment the value on PortB.

The total is: 1 + 1 + 49 + 48 + 2 + 1 = 102 cycles

Assuming a crystal frequency of 32 kHz, we can divide it by 4 to give the instruction clock frequency and then by the delay of 102 cycles to give the rate at which the PortB is incremented of about 78 counts per second. PortB counts in binary from 0000 0000 to 1111 1111 and will finish its count after 256 counts so it will start recounting after 256/78 or roughly 3.3 seconds. We could reasonably double this time delay by a liberal sprinkling of NOPs or using a longer loop.

Longer delays

We have three alternatives.

1 For small changes, we could add some NOPs inside of the counting loop to boost the number of counts.
2 Our delay was built into the main program but we could have used it as a subroutine. A subroutine is any block of code that we may want to use more than once. In the main program we insert an instruction CALL followed by a label to identify the block of code so for our delay loop which we called 'count' we would insert the instruction 'CALL count' at any time we want to use our program to cause a delay. When

the delay loop 'count' has been completed, we insert the instruction RETURN at the end of this block of code and the microcontroller will return to the main program.

The benefit of using a subroutine is that we can run the 'count' delay twice just by inserting the instruction CALL count twice in the main program and we don't have to enter the delay loop again with the fear that we will mistype something and it will all collapse. We can make a subroutine as long as we want and use it as often as we want just by adding the CALL and RETURN instructions.

Here is our previous program but reorganized to use the delay loop 'count' as a subroutine.

```
count   DECFSZ    20,1   ; decrements register 20
        goto      count  ; keeps decrements until it gets to zero
        RETURN

        ORG       000
        BSF       3,5
        CLRW
        MOVWF     86     ; PortB data direction = output
        BCF       3,5
        MOVLW     55
        MOVWF     06     ; PortB data set to a start value
again   INCF      06,1
        MOVLW     30     ; Loads W with 30H
        MOVWF     20     ; puts the number 30 into file 20
        CALL      count
        goto      again  ; returns to increment PortB
        end
```

The subroutine is called count and has the instruction RETURN at the end.

The main program has the instruction CALL count which means 'go and get a subroutine and use the one called count'.

We can then put:

```
CALL count
CALL count
CALL count
```

In the main program which would be an easy way to treble the length of a delay. We could design a subroutine called '1second' and another for '0.1second'.

Then if we needed to insert a delay of 2.3 seconds, we could just add:

```
CALL 1second
CALL 1second
CALL 0.1second
CALL 0.1second
CALL 0.1second
```

All subroutines would end with the same code RETURN, so how do they know where they have to go back to?

The answer is a series of memory locations called a stack. Each return address is stored in the stack in order that each CALL occurs, the relevant address is sent to the stack and as each RETURN will occur in sequence, the addresses will be unloaded from the stack in the order required. This is a first-in last-out (FILO) organization. See Chapter 8 for more on the stack.

A subroutine can include a CALL to another subroutine. These are called nested subroutines – the PIC16F84A has room in its stack for eight return addresses – which is pretty small by microprocessor standards.

3 In the PIC, most instructions are completed in a single instruction cycle which is $\frac{1}{4}$ of the clock speed. To change the delay, we could always change the clock speed. There are two benefits, a slower clock speed reduces the power consumed, there is no low-speed limit for the PIC, unlike some devices. Generally subroutines are preferred as there are often other constraints on the clock speed.

Quiz time 16

In each case, choose the best option.

1 Return is:

(a) only used as part of a delay loop.
(b) a ticket to take you home again.
(c) an assembly directive.
(d) an instruction found at the end of a subroutine.

2 ORG is:

(a) never needed since the PIC always starts at address 0000.
(b) an assembler directive.
(c) short for orgasm.
(d) an instruction code.

3 An assembler converts:

(a) decimals into hexadecimals.
(b) main codes into subroutines.

(c) source code to object code.
(d) object code into binary code.

4 In choosing a clock circuit:

(a) a ceramic resonator is not as accurate nor so robust as a crystal.
(b) an RC runs at four times the frequency of a crystal.
(c) a crystal gives the most accurate and stable frequency.
(d) use an RC circuit and a crystal to get accuracy and robustness.

5 The normal execution time for when using 4 MHz crystal is:

(a) 0.25 microseconds.
(b) 1 millisecond.
(c) 4 milliseconds.
(d) 1 microsecond.

17

Interfacing

Interfacing is the process of connecting a microprocessor to the rest of the circuit or to external devices. Even in the simplest of computer systems, there is some input device like a keyboard. So how does the microprocessor know that we have pressed a key? When we send text to a printer, how does the printer tell us that it is ready for more input?

In a general purpose microprocessor-based system, if it is to do anything useful, there must be inputs and outputs. The external devices must therefore communicate with the microprocessor. In some cases, the microprocessor takes the matter into its own hands and sends data out as part of its program but even in this case it normally allows the external device to help.

If a microprocessor-based system were used to heat some water, it is easy enough to imagine the program switching on the power supply and sitting there doing nothing for 10 minutes. It would be a better use of the microprocessor to leave the heater running and wait for a thermostat to signal that the water has reached the required temperature. This thermostat signal would arrive at an interrupt pin on the microprocessor.

Interrupts

Interrupts were introduced in Chapter 8 when we looked at the operation of the interrupt flag in the status or flag register but we will now delve a little further into the system.

All microprocessors have interrupts that can be initiated either by the software being run at the time or by external hardware circuits. Microprocessors differ in the details of their response to hardware interrupts and in the number of different interrupt pins offered. Details are always itemized in the technical data supplied with the device.

The likely options are as follows

There are two basic types of hardware interrupt. The first is an interrupt request or IRQ (or INTR) pin. This tells the microprocessor that it would like to have some attention. Since it is a request rather than an order, the microprocessor is free to say 'yes' or 'no' or 'yes, but not at the moment – just wait till I'm ready'. This does not imply any intelligence on the part of the microprocessor – it must be told which response to give by the software that is being run at the time. In the absence of any instruction, it normally accepts the interruption. If a particular interrupt pin has been told not to respond to an interrupt, we say that the interrupt has been 'masked'.

The second type is called a non-maskable interrupt, that is, an unstoppable demand for attention. This will always assume top-priority. A typical use for this would be for an emergency shutdown in the event of a power failure. The program can also instigate an interrupt by means of a software instruction as part of a program.

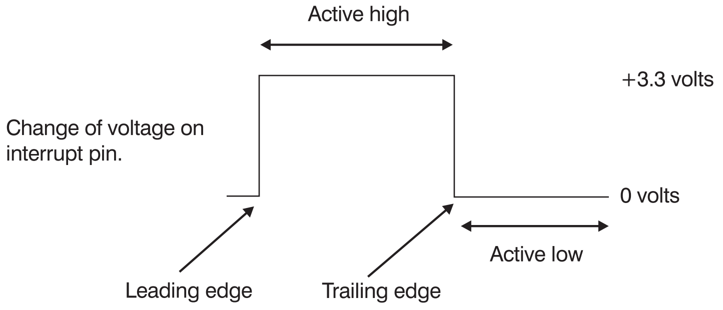
What is a hardware interrupt signal?

This is a change of voltage on an interrupt pin generated by the external device. The change required would be detailed in the technical data but there are four choices.

The first two choices are changes of voltage level. The pin can sit at +3.3 V or whatever the 'high' voltage happens to be, then responds when it falls to 0 V. We call this 'active low'. When the pin goes low, an interrupt is recognized. Alternatively, it could sit at 0 V and become activated by an increased voltage. This we call 'active high'.

Figure 17.1

Four ways of signalling an interrupt



The alternative approach is to use the sudden change of level. From low to high is called 'leading edge' or 'rising edge' and from high to low is the 'trailing edge' or 'falling edge' (See Figure 17.1 and have a glance at Figure 6.4 to see the alternative names.)

Once an interrupt is activated, the signal must be returned to its normal voltage before it can be triggered again. The input is masked as the interrupt program is running to prevent the interrupt pin from interrupting itself if the voltage remains at its active level.

Accommodating several external devices

The simplest and quickest way of connecting devices to the interrupts of a microprocessor is to have each device connected to its own interrupt pin. This is OK providing there are enough pins. Few microprocessors have more than two interrupt pins so we have to connect several devices to the same pin.

When an interrupt occurs, a program called the 'first level interrupt handler' or FLIH is activated. The function of the FLIH is to identify the device causing the interrupt and to pass the controls over to the 'interrupt handler' or 'interrupt service routine' program that has been written to deal with that device.

How does it know which device is crying for help? There are two options, polling and vectored interrupts.

Polling interrupts

This is a slow but sure way. Each possible device is interrogated in turn with an 'is it you?' signal until the source of the interrupt is found. The order of checking is prioritized so the most important device is checked first.

Vectored interrupts

Immediately after the interruption has occurred, the FLIH puts out a 'who's there?' signal and the interrupting circuit puts an identification signal onto the data bus. This signal is usually used as part of an address to identify the section of program to be executed.

What happens if an interrupt is received while the previous interrupt is being dealt with? Again we have a choice. We can disable the new interrupt until the first one is complete, then deal with the new one. Alternatively, we can check the priority of the new alarm and decide on the new priorities. A higher priority causes the present interrupt to be halted and its current state to be saved in the stack while the new one is worked on and then we return to unload the stack information and carry on with the original problem. If the new interrupt is less important it gets a 'wait a bit' message until the first one is finished.

In a microprocessor-based system it is up to the designer to decide on the priorities and uses of the interrupts. In a PC, the interrupts are prioritized in order of speed and importance. Top priority is given to the internal clock. This is the clock that tells us the time – not the square wave clock that synchronizes the circuitry. Thereafter, in order, we have the keyboard, two spare ones for any other operations, then comes the serial port, the hard drive, the floppy disk drive and finally the printer.

Parity

If you were to walk into a crowded room and say 'Burgers' and nothing further, many of those present would turn to their neighbours and say 'What was that?' and some would just stand and stare. (Some may even mis-hear and feel offended.) As a form of communication, this is not very efficient. Try instead, walking in and saying 'Lunch will consist of burgers'. Everyone would understand your message.

The first attempt was very efficient in terms of the number of words used but is probably likely to be inefficient communication since many people will not receive the message. In the second attempt, we have used five words to make sure that the one important one gets through. This is called adding 'redundancy'. The more redundancy we add, the more certain is the message but the slower and less efficient becomes the communication system. Data being returned from space probes use very high levels of redundancy, over 96%, which allows for correction of really scrambled signals due to the extremely low power levels involved.

We can use parity for alerting us to the possibility of an error in a stream of data or, in some cases, we can detect and correct the error. In its simplest form, we take a group of bits in a transmission, 4 or 8 bits are normally used though the idea is applicable to other values. In this example, we will look at a 4-bit group, say 1001. At the transmitting end, we add an extra bit on the end, either a 0 or a 1 to make the total number of '1's an even number. In this case, there are two '1's and so the number is already even, so we add a zero. The data now reads 10010. At the receiving end, if the data has been mutilated and now reads 11010, a quick count will show that there is an odd number of '1's and so an error has occurred.

This simple approach can be easily fooled. If there are two errors there will be an even number of '1's and passed as correct. And, another disappointment, if it shows an error, we cannot tell which bit is wrong and therefore cannot correct it. When this system is used, an error signal is sent back to the transmitter requesting a repetition but this assumes that the transmitter and the receiver are in communication with each other. We can modify the system to make limited automatic correction feasible.

Let's assume we have, say, 16 bits of data to send.

Step 1 Rewrite the data in the form of a square:

```
0 0 1 0
1 1 1 1
0 1 0 1
1 0 1 1
```

Step 2 Add parity bits. Across the top row, we have the numbers 0010 which includes a single '1'. In this system, which we will call 'even' parity, we add another '1' if necessary to ensure that there is an even number of '1's across the first row. As we have only a single '1', we add another '1' on the end. It now looks like this:

```
0 0 1 0 1
1 1 1 1
0 1 0 1
1 0 1 1
```

The top row now has an even number of '1's. The next row has four '1's which is an even number so we do not need to add another '1'. We therefore add a '0':

```
0 0 1 0 1
1 1 1 1 0
0 1 0 1
1 0 1 1
```

The third row will be completed with a '0' since it contains an even number of '1's and the last row, with three '1's will need an extra one to be added. The result is now:

```
0 0 1 0 1
1 1 1 1 0
0 1 0 1 0
1 0 1 1 1
```

Step 3 We now have five columns down the page and we can add extra '1's in the same way to make the total number of '1's an even number. The first two and the last columns each contain two '1's so zeros will be added. The third and fourth columns have three '1's so we need to add an extra '1' to each. The result is now:

```
0 0 1 0 1
1 1 1 1 0
0 1 0 1 0
1 0 1 1 1
0 0 1 1 0
```

Notice how we have now got a total of 25 bits to be transmitted. This represents 16 bits of data and 9 bits added to check the accuracy of the data. The final serial transmission is 0010111110010101011100110. This means that 9 out of 25 or 36% of the transmission is not actual data and represents redundancy.

Let's see how it works. We will assume an error has occurred and one of the bits is received incorrectly so here is the received transmission:

0010111110010101001100110

Step 1 Layout the data as a 5 × 5 square.

```
0 0 1 0 1
1 1 1 1 0
0 1 0 1 0
1 0 0 1 1
0 0 1 1 0
```

Step 2 Check the parity in each row across the square.

We decided to make each row and column to have an even parity.

The first row has two '1's, this is even – OK.
The second row has four '1's, this is even – OK.
The third row has two '1's, this is even – OK.
The fourth row has three '1's, this is odd – an error has occurred.
The last row has two '1's, this is even – OK.

We now know that one of the bits in the fourth row has been received incorrectly.

Step 3 Do the same for the columns.

The first column has two '1's, this is even – OK.
The second column has two '1's, this is even – OK.
The third column has three '1's, this is odd – an error exists in column three.
The fourth column has four '1's, this is even – OK.
The last column has two '1's, this is even – OK.

Step 4 Isolate the error and change the data.

The error occurs in the third column and the fourth row. Since this is now known to be an error and we only have a choice of 0 or 1, we can confidently change the 0 to a 1 and recover the correct data stream.

0	0	1	0	1
1	1	1	1	0
0	1	0	1	0
1	0	$\boxed{1}$	1	1
0	0	1	1	0

In this example, we chose to use even parity, that is, we made each row and column have an even number of '1's. It would work equally well if we used odd parity by making the number of '1's an odd number. It would also work just as well if we counted the zeros instead of the ones. If more than one error occurs, it will warn us of an error but it will be unable to make any corrections. If you try it, you will see that it indicates four possible positions for the two errors and nine for three errors.

Example

Correct this received data which includes one error. To provide automatic correction, odd parity on the '1's has been used.

The received signal: 1101001001111001010101101.

Step 1 Layout the data as a 5 × 5 square

1	1	0	1	0
0	1	0	0	1
1	1	1	0	0
1	0	1	0	1
0	1	1	0	1

Step 2 Check the columns and rows for an odd number of '1's

1	1	0	1	0	✓
0	1	0	0	1	✗
1	1	1	0	0	✓
1	0	1	0	1	✓
0	1	1	0	1	✓
✓	✗	✓	✓	✓	

Step 3 Isolate the error

1	1	0	1	0	✓
0	$\boxed{1}$	0	0	1	✗
1	1	1	0	0	✓
1	0	1	0	1	✓
0	1	1	0	1	✓
✓	✗	✓	✓	✓	

Step 4 The data where the column and row intersect is the error. So we simply change the '1' to a '0'.

Step 5 Strip out the parity bits to recover the original data

11010000111010100110

Data transmission

The most basic way of sending information from one place to another is simply to connect a wire to both ends of the system and apply a voltage to one end. By making the voltage vary, we can send different levels and even speech or music. These are called analogue signals and have many drawbacks.

The main one is the effect of noise. As the signal travels along a wire it gets weaker and it has noise induced into it by random electromagnetic signals and vibration of the molecules of the conductor. The overall effect is that the signal becomes degraded and weaker.

The 'weaker' bit is no problem, we can soon amplify it back to its original size but the noise is a different matter. The electrical noise has become embedded into the signal and has permanently distorted it. Amplifying it will amplify the noise and the signal equally.

We have no problem with digital signals since we know that they will all be rectangular in shape and so the amplifier can be used to regenerate the shape of the signal and hence strip out the effects of the noise. If we are faced with sending something inherently analogue, like speech or music, our first job is to convert it to a digital form.

Analog to digital conversion (A to D or ADC or A–D)

Inevitably these days, this is taken care of by an integrated circuit of which there are many different designs. It is quite possible, but totally uneconomic, to construct our own ADC so it is really a matter of flicking through the catalogues and choose the most appropriate one available.

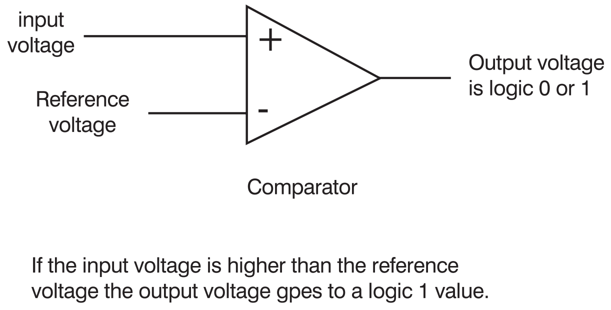
There are several different designs of ADCs, which are based on three basically different approaches.

Flash converter

The first is called a flash converter or parallel encoder. These use circuits called comparators. A comparator has two inputs, one is the analogue voltage being converted and the other is a known reference voltage.

All we ask of a comparator is to answer a simple question: 'Is the analogue input voltage higher or lower than our reference voltage?' It answers by changing its output voltage to a logic 1 to mean it is higher and a logic 0 to mean it is lower. They are so accurate that the chance of it accepting the two voltages as the same level are extremely slight and doesn't happen in practice (see Figure 17.2).

Figure 17.2
A comparator used in a flash ADC


input voltage
+
Reference voltage
-
Output voltage is logic 0 or 1
Comparator
If the input voltage is higher than the reference voltage the output voltage gpes to a logic 1 value.

So how do we use the comparator? If we had three of them with reference voltages set to 1 V, 2 V and 3 V and then applied the input voltage of 2.5 V to all of them, the first two would set their outputs to a level 1 and the last one would be unaffected at logic 0. The logic levels could be used to generate a binary number to represent 2.5 V.

An input voltage of 2.4 V or 2.9 V would also result in the same comparators being activated and hence the same output digital signal. This error occurs in all analog to digital converters. We can reduce the size of the error by increasing the number of comparators to detail more levels. Real ones have between 16 and 1024 different levels.

Ramp generators

These are a combination of a binary counter that simply counts up from zero to its maximum value, perhaps 1024 like the last type. As the binary count proceeds, a ramp voltage is made to steadily increase. A single comparator is used to compare the output of the ramp voltage with the analog voltage being converted. As soon as the ramp voltage exceeds the input voltage, the comparator signal stops the counter. The counter output is then the digital equivalent of the analog signal (see Figure 17.3).

Successive approximation

If we were to use a 3-bit digital signal to convert an analog voltage of between 0 V and 4 V we could have the 3 bits representing voltages of 4 V, 2 V and 1 V. This is how the circuit responds to an input of 3.5 V.

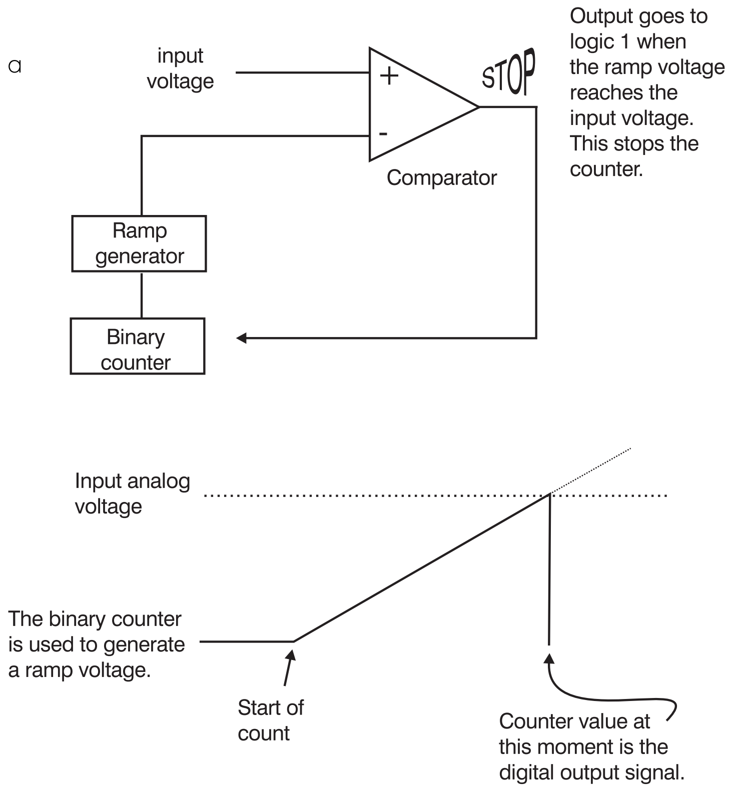

input voltage
+
-
STOP
Comparator
Output goes to logic 1 when the ramp voltage reaches the input voltage. This stops the counter.
Ramp generator
Binary counter
Input analog voltage
The binary counter is used to generate a ramp voltage.
Start of count
Counter value at this moment is the digital output signal.

Figure 17.3
An ADC that uses a ramp voltage

The digits are initially set to 000. The left-hand bit is switched on and its 4 V is compared with the input. The input is seen to be less than this so this digit is reset to zero. It then tries the next bit and its 2 V are compared and found to be less than the input so it remains set. The digital signal is now 010. The circuit now adds the 1 V from the last digit. The result is a total of 3 V, which is compared with the input analog signal. The input of 3.5 V still exceeds the current value of 3 V so the last bit is set. The final digital output is 011.

The circuit has tried all the available values until it finds the one that provides the result closest, but less than the input signal. As before, the more bits we are using, the more accurate is the result.

In checking the specifications of likely ADCs to use, we need to compare the following criteria.

Quantization error

In the above example using the flash converter, we can see that an analog input of 3.5 V would provide the same output as would any value between slightly over 3 V and slightly less than 4 V. This error means that small variation in the analog input voltage will be lost. The size of this error is equal to the space between the comparator reference voltages.

Changing from eight comparators to 1024, would mean that the voltage gaps would decrease from 1 V to 7.8 mV as would the quantization error. Regardless of the method used for A–D conversion, quantization error is always present.

Bits

The more bits, the merrier. Likely values will be between 8 and 16.

Speed

There are two factors here. How often can we get an updated value for the signal and how well can we follow any changes it is making? Even if we do not want the signal to be sampled at a very high rate, we still may want to take a quick sample so that the input value is unlikely to change very much as the sample is actually being measured.

For speed, you cannot beat the flash converter. It can sample for a period as short as 3 ns which compares very favourably with the typical values of 10 μs for the ramp generators and successive approximation types.

Digital to analog conversion (DAC)

Changing a group of digital bit values to an analog voltage is basically just the reverse process of the A–D conversion that we met in the previous section.

Most digital to analog converters operate by adding current together then converting the result into an analog voltage. The binary levels are used to switch currents on or off.

Let's assume a 4-bit digital signal in which the most significant bit is made to generate a current of 8 mA and the others produce, in turn, 4, 2 and finally 1 mA. If the digital signal to be converted happened to be 1011_2, then the first, third and fourth current sources would be activated giving a total of 8 + 2 + 1 = 11 mA (Figure 17.4).

In some DACs the final output is a changing current but in others it has been converted to a variable voltage. It just depends on which integrated circuit you choose to use. In the ones offering a voltage output, the total current is then passed through a resistor. If we choose a nice easy value like 1 kΩ, the voltage across the resistor would be 11 mA × 1 kΩ = 11 V.

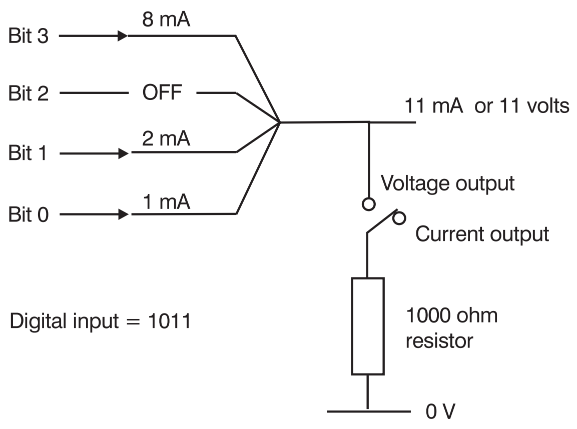


Figure 17.4
A DAC with a current or voltage output

In a similar way, we can see that all binary values between 0000 and 1111 would be converted into voltages between 0 V and 15 V. There are a couple of specifications that may need to be checked to decide on which one to use.

Resolution

This is the number of digital bits used to convert into an analog voltage. Typical values available are from 4 to 18 bits. As the digital input changes by a single bit, say from 1000 to 1001, the resultant voltage or current increases by a discrete step. The size of this step is determined by the number of bits used compared with the maximum value of the output current or voltage.

For example, if we used 4 bits then this would provide a total of 16 different steps and if the maximum happened to be 8 V, each step will represent a voltage change of 0.5 V. Thus, a steadily increasing digital signal will cause the analog voltage to increase in small discrete steps like a staircase. This is all very similar to the cause of quantization error.

Speed

The speed of operation is very dependent on the chip being considered. The conversion times available from an exceedingly fast 1 ns to a sluggish 5 μs.

Serial and parallel transmission

In sending information in digital form, we have a choice of using serial or parallel transmission. In the case of serial transmission, the binary values are represented by two different voltage levels and are sent one after the other along a cable. This is simple but slow. The alternative is to have several wires and thus be able to send several bits of data at the same time, one on each conductor.

In a microprocessor-based system, even if it is a one-off, it is usually better to conform to established standards for the cabling so that other instruments and circuits can be connected with a minimum of hassle.

Parallel connection

There are several different standards used for parallel connection of data but one of the most widely used, and most reliable, was produced by Centronics.

The Centronics system sends eight bits at a time and employs a 36 plug and socket system. To send data, there are four basic control signals as well as the eight data lines. It also stipulates a variety of other control wires that can be used if required.

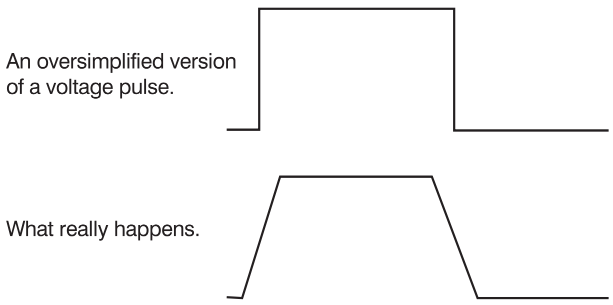
The important thing to remember about these standards is that you do not have to use all the connections listed but those that you do decide to use should conform to the stated specification and be on the correct pins. This ensures that if the plug is inserted into a new piece of equipment, it may not work but at least it will not be damaged.

Centronics data transmission

To see how the system works, we will use timing diagrams to show what happens and when. These diagrams, which all look very similar at first glance, are shown in all data manuals to show the sequence of events inside the microprocessor and in the surrounding circuit.

There are a couple of points that are worth mentioning. We have mentioned the problem of rise time in Chapter 7. You will remember that we cannot change a voltage level instantaneously. It may not seem an important delay when we think of switching a light on at home but when the microprocessor is handling data at millions of bits per

Figure 17.5
Rise and fall time may be important



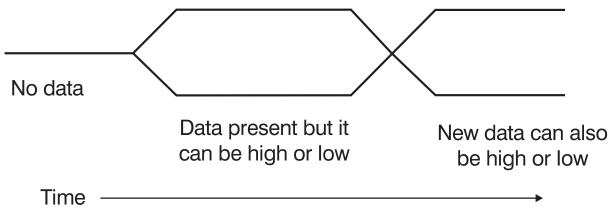

No data
Data present but it can be high or low
New data can also be high or low
Time

Figure 17.6
Showing alternative data levels

second, these delays can be important and is a common cause of failure in a circuit that 'should' work. Most waveform diagrams show the ends of a square wave as sloping lines rather than vertical ones.

In Figure 17.5 we have a positive-going pulse to represent a data value of 1 but, of course, data could equally well be at 0 V to represent a binary 0. In cases where we want to show that a level has changed, but it may go to either level, we redraw the diagram to show both possibilities at the same time, as in Figure 17.6.

Figure 17.7 shows the process of transferring eight bits of data from a microprocessor to an external printer or other device.

Step 1 The microprocessor puts the eight bits of data on the data wires.

Step 2 A short delay occurs while we wait for the data voltages to settle on all the eight wires. Then the strobe pulse occurs to tell the printer or other accessory that the data is ready. The line over the word strobe indicates that it is active low. No line would mean active high.

Step 3 The printer starts loading data and the busy line goes high to prevent more data being sent.

Step 4 When the data has been printed, the busy line goes down to tell the microprocessor to put the next piece of data onto the data wires.

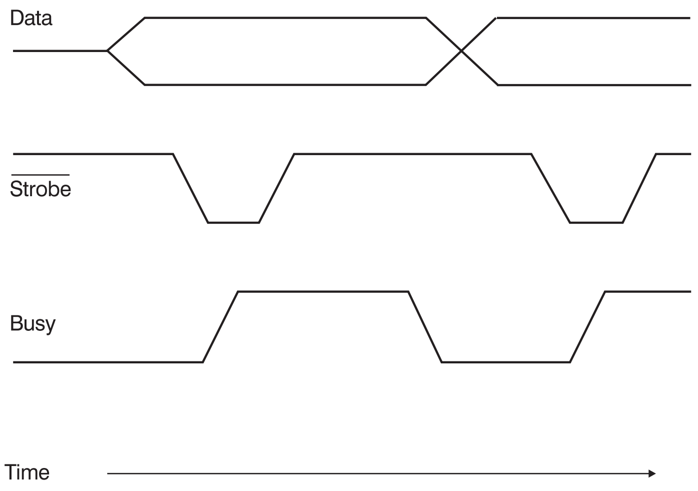


Figure 17.7
The timing of Centronics signals

The standard allows for many other control wires for other purposes. In this example, the eight data wires have been used as outputs from the microprocessor but it is quite possible to use them to carry input data although this is not allowed for in the design of PCs.

Serial transmission

To send information in serial form requires only a simple communication link but it is inevitably slower than parallel transmission since the data is only sent one bit at a time.

UARTs

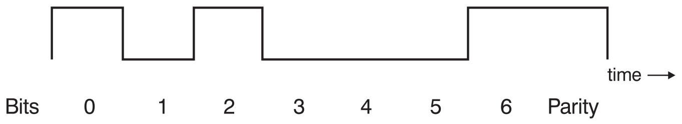
To convert the parallel data on the data bus of the microprocessor to a serial transmission we could use a shift register as in Chapter 6. The modern alternative is to use a chip called a UART (universal asynchronous receiver/transmitter) or USART (universal synchronous/asynchronous receiver/transmitter).

These are integrated circuits that convert data from parallel to serial transmission so instead of having eight wires, each carrying a single bit at the same time, the serial transmission passes the bits, one at a time along a single wire. It can also receive eight bits of serial data then pass it into the microprocessor in a single byte. Parallel transmission is obviously a lot faster since eight bits are moved at a time but it requires eight connections. To send a fax signal, for example, would require the use of eight telephone lines whereas the UART can convert it to a serial transmission and sent it over a single line at one-eighth of the speed.

UARTS do a lot more than a shift register. They include parity checking and buffers to enable it to handle about 16 bytes at a time without involving the microprocessor. Most transmissions involve the ASCII code to represent the characters to be transmitted. This is a seven-bit code to represent each alphanumeric character and a variety of control instructions. For example, the letter E is 45 in hex which, using only the lower seven bits, is 1000101. The ASCII code is used in both parallel and serial transmissions. Each letter and symbol has its own seven-digit code. A further bit is added on the end to provide a parity bit or it can be used to swap over to an alternative set of characters to allow mathematical symbols and Greek letters to be transmitted or, if unused, can be left at zero. Our letter E would then be represented as shown in Figure 17.8.

When using ASCII signals in a serial transmission, we need to be able to tell the receiving apparatus when a particular ASCII character has been sent. This is easily done in a synchronous system that ensures that the transmitter and the receiver are locked together running at the same speed. This is not the easiest way of operating the system owing

Figure 17.8
Coding in ASCII



The ASCII code is 45H or 1000101 + the parity bit. Notice that the bits are coded with the lsb first so we actually send 1010001 + parity bit, in this case using even parity.

to the difficulties of ensuring the two devices remain synchronous. Therefore, we tend to operate asynchronously. This means that we have to send a signal along with each ASCII code to tell the receiver when the code has started and when it has stopped. Otherwise the transmitter would send a continuous stream of data and if a bit were lost, the receiver would get out of step and would misread all subsequent data.

To get round this problem, a 0 V 'start' bit is sent at the beginning of the character and a positive 'stop' bit is sent at the end. This brings a seven-bit ASCII code up to a total of 10 bits. The start and stop bits ensure that there is at least one change of level for each character that can be used to keep the receiver clock nearly synchronized to the transmitter for the time taken to receive that character.

For distances over a few metres, we need to use a slightly more sophisticated transmission system to prevent random noise from interfering too much. There are several systems in use, the most popular being those created by the EIA (Electrical Industries Association).

As with most transmission media, there is a trade off between the speed and the maximum distance the system can be used for. If you intend pushing the transmission distance to its maximum value, you will have to accept a reduced speed. As a rule of thumb, halve the speed if you double the distance.

RS232C

This is one of the transmission standards created by the EIA committee. This standard allows for transmissions up to 50 feet (15 m) and at speeds of up to 20 kbaud (it can actually exceed this speed and distance but it's not guaranteed). The baud is the measure of the speed of transmission. It is the number of clock periods per second, which approximates to the number of bits per second.

The RS232C transmission is balanced at about 0 V. Here's the time to be careful, the binary one level is a negative voltage (between –5 and –15 V) and a binary zero level is a positive value between +5 and

+15 V. This seems upside down compared with all our previous uses of binary. Our letter E would be transmitted as in Figure 17.9. The transmitter levels are specified as ±5 V but the receiver limits are ∓3 V. This allows for a noise spike to be up to 6 V before there is any possibility of misreading a piece of data.

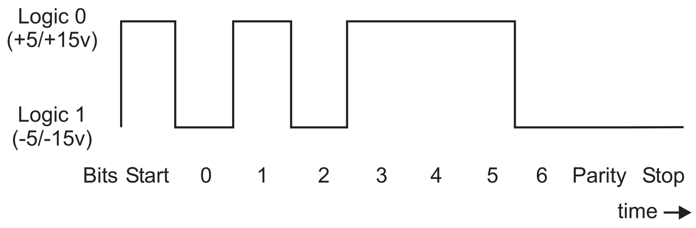


Figure 17.9 RS232C transmission

RS423A

This is an improved version having a maximum speed of 100 kbits/s and a maximum cable length of ¾ mile (1.2 km). The transmission voltages have to be between ±3.6 and 6 V and the receiver can go down to ±0.2 V.

Changing voltage levels

How do we change the binary or logic values into the RS232 voltage levels? If you are building a microprocessor-based system then the most obvious way is to use a pair of integrated circuits called the 1488 (transmitter) and the 1489 (receiver). These integrated circuits have been around for many years and are simple and reliable. They have a small snag in that they need 12 V supplies whereas nowadays 5 V supplies are much more common so you may find some new transceivers (made by Maxim) more interesting. These only require a single +5 V supply and generate their own ± voltages for the RS232C transmission. Each chip contains two transmitters and two receivers and operate up to 120 kbits/s. The devices are numbered MAX202, MAX208, MAX220 and MAX232 and others. PCs have a serial port that provides signals at RS232C levels.

Using RS232C in real life

Most RS232C links are via a 25-pin 'D' plug or a 9-pin 'D' plug and socket (Figure 17.10) but unlike the Centronics which is quite stable and usually work straight off, the RS232C can be a real nuisance. Before attempting to communicate, you must ensure that the transmitter and the receiver are using the same word length and parity values are set for the same speed of operation. Even then, it may take

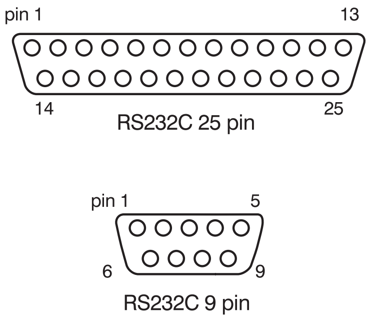


Figure 17.10
'D' connectors for RS232C

some experimenting before they spring into life. The problem is that there are many more options for the other connections. All have to be agreed between the receiver and the transmitter. The specifications are not detailed enough and can lead to different interpretations. It is not surprising that it is often insufficient to connect an RS232C cable between two pieces of equipment and switch on. You will need to get hold of the RS232C connection specification and settle down in a comfortable chair.

Modems

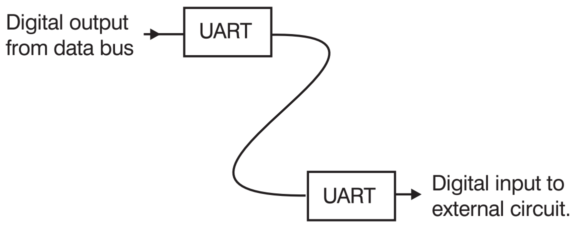
A modem (MOdulator DEModulator) converts a digital signal into two audio tones so that the transmission can occur along a telephone line. Telephones are generally designed to accept frequencies between 300 Hz and 3.1 kHz. This relatively narrow bandwidth was chosen to allow speech to be transferred with undue loss of quality while allowing the largest number of calls to be passed along the same cable. Once the digital signals are on a telephone line then the range is unlimited.

Choice of systems

A few metres

We can use the raw binary data transmitted over a simple cable (see Figure 17.11).

Figure 17.11
A very short range link up


Digital output from data bus
UART
UART
Digital input to external circuit.

Tens or hundreds of metres

We can convert the transmitted signal to RS232C or RS423A as necessary (see Figure 17.12).

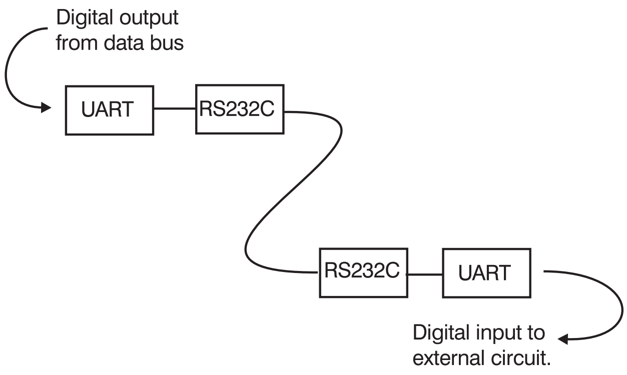


Figure 17.12
Around the building

Unlimited range

Add a modem and link by telephone or optic fibre (see Figure 17.13).

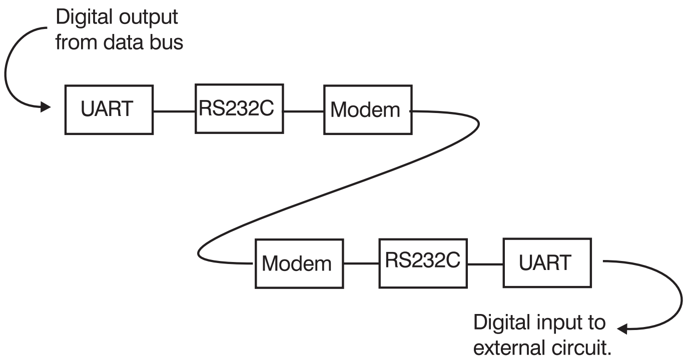


Figure 17.13
Around the world

An optic fibre link

A piece of optic fibre is a solid piece of glass or plastic. The plastic fibre is about 1 mm in diameter and is suitable only for short ranges of a few tens of metres but it has the advantage of being cheap and easy to use. Its useful range is limited by the clarity of current plastics. The special silica glass is incredibly clear and hence has much lower losses and able to be used over any distance, with suitable repeaters. It also has a much smaller diameter – only about 125 μm before the external protective layers are added.

If a light is shone into the end of an optic fibre, it will reflect off the inner surfaces along the cable. The light source used is a laser operating in the infrared region of the spectrum. To use it as a means of sending a digital signal we need to switch the light source on and off and then detect the flashes of light at the far end of the cable by a photoelectric cell. The losses can be made up by repeaters just as we do on copper-based systems, so range of operation is no problem. The optic fibre does not suffer from any electric noise pickup along the route and has an enormous bandwidth. In one sense, it is not really optional because nearly all long distance telephone cables are now optic fibres. (See Further reading for our companion volume *An Introduction to Fiber Optics*.)

If we are constructing our own fibre optic link, all we need to do is to buy the laser (or light emitting diodes (LEDs)) and the photocells and some plugs and sockets to connect it all up. It can be used to replace the copper cable in any of the systems described (see Figure 17.14). Careful! The infrared light from the lasers can cause immediate and irreversible eye damage. We must always remember that we are down to our last pair of eyes.

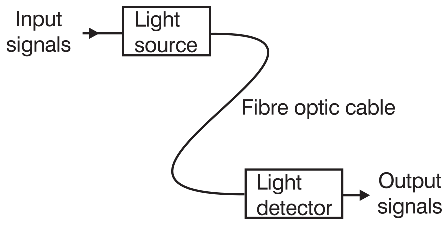


Figure 17.14
A fibre optic link – any distance

Data transfer rates

Using a single optic fibre for serial transmission, typical data transfer rates of 100 Mbytes/s are available up to 10 km. Very high speed data transfer of 1 Gbyte/s can be achieved up to 100 m using parallel transmission along a bunch of fibre optic cables.

Quiz time 17

In each case, choose the best option.

1 The fastest design of analogue to digital conversion is a:

(a) ramp converter.
(b) flash converter.
(c) comparator.
(d) successive approximation converter.

2 Quantization error can be reduced by:

(a) increasing the number of levels in the ADC.
(b) increasing the speed of the conversion.
(c) using vectored interrupts rather than polling.
(d) using a flash converter.

3 Using the RS232C standard, a binary 0 is most likely to be transmitted as:

(a) −4 V.
(b) +2 V.
(c) −5 V.
(d) +10 V.

4 A modem is:

(a) a type of USART.
(b) normally connected between the UART and the RS232C converter.
(c) only used in fibre optic systems.
(d) used to convert digital signals into audio tones for transmission over telephone cables.

5 This received transmission has sixteen bits of data and includes an error. It is using odd parity on the ones in a block of 25 bits:
0000111010111000100110011.
The corrected data is:

(a) 1000111010111000101110011.
(b) 0000111010111000101110010.
(c) 0000111010111000101110011.
(d) 0000111010111000100110011.

18

Test equipment and fault-finding

This chapter is intended to give some pointers towards finding faults in a microprocessor-based system. This chapter is firmly based on experience and could equally well have been entitled, 'Mistakes I have made'.

What's gone wrong now?

The whole process of fault-finding should be undertaken slowly and carefully. There is a popular misconception that you have to keep busy, taking measurements, making adjustments and changing components. But, in fact, most of the time is spent just sitting and thinking (don't forget the last two words!).

Collect the symptoms and write them down. Be wary of other people's idea of the symptoms. If they have misunderstood what is happening you could waste hours or days going off at a tangent. If you forget to write them down, then sooner or later you will be back repeating the same checks.

Don't make the problem worse

In most cases, a piece of equipment or a circuit fails due to a single fault. Two simultaneous but unconnected faults are very rare. There are two popular ways of converting a small problem into a large one. These are static electricity and plugs etc.

Static electricity

When two different materials rub against each other, some negative electrons tend to migrate from one material to the other. This results in a voltage difference between the two materials. The amount of voltages can be very high – several thousand volts. If we walk across a carpet or sit on a plastic covered chair, we can become lethal to an integrated circuit designed for 5 V. Many integrated circuits have anti-static precautions built in but they have limited success. There is a trade-off here in that the better we make the antistatic precautions, the slower the integrated circuit can switch.

We can overcome the problem by reducing the build up of static by allowing it to leak away. In carpets, clothes and furniture we can do this by adding a wax or polish that absorbs and holds a small quantity of moisture. A slight dampness is a very effective way of preventing static problems. For this reason, the weather and air humidity is important. The death rate of integrated circuits tends to vary seasonally! It is not helped by air-conditioned plant where the humidity is low. The effect of static electricity on integrated circuits is difficult to predict. It generally causes small localized failures which can have very peculiar effects.

Better than spraying ourselves with water, we can take a more high-tech approach but how far to go in this direction depends on what is at stake. If we are going to handle a couple of cheap AND gates once a week, then only the simplest precaution is worthwhile. However, sitting on a production line plugging in microprocessors will make any precautions economic.

The simplest method is to have a conducting band clipped around your wrist with a lead going off to a ground (earthed) point. These wristbands are made of rubber into which carbon has been amalgamated to allow it to conduct slightly. As well as the wristband we can place a sheet of this rubber on the bench top and ground the bench. Such antistatic workstations are very effective. A word of warning. Do not make your own wrist strap from a length of copper wire. This offers a very low resistance and provides no protection against electrocution in the event of accidentally touching a power line.

At home, just avoid working on a plastic table or chair or wearing clothing made from man-made fibres. Natural materials like cotton, wool and untreated wood naturally absorb some water and are fairly safe. A nice wooden bench coated with polyurethane varnish is effectively a plastic bench and should be avoided.

Problems with plugs

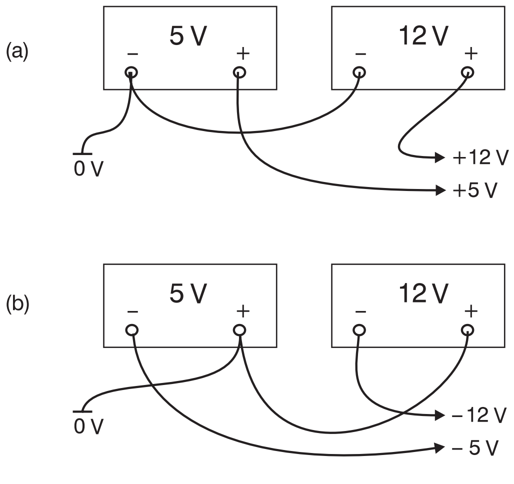
Many plugs used between pieces of equipment have a large number of pins. Pulling one of these out with the power connected is going to disconnect some voltages before others. This can prove fatal for integrated circuits. Either all the supplies must be on, or all should be off so never plug or unplug anything with the power on. For the same reason, never remove or replace an integrated circuit with the power on.

Tests we can make without test equipment

Are the power supplies turned on? Do you need two supplies? If you are using two supplies, are they connected together to keep their voltages in step with one another? If a ground connection is required, is it connected?

Most power supplies have floating outputs. That means that a 5 V supply, for example, will have a 5 V difference between its two terminals but neither is connected to the ground potential. This means that if we connect the negative terminal to earth, as in Figure 18.1(a), the other terminal goes to +5 V. If, on the other hand, we make the connection shown in Figure 18.1(b), the other terminal will become −5 V.

Figure 18.1
Connecting floating supplies



Have a look at the soldering if it is visible. It should be smooth and shiny. Any dull and craggy looking areas are suspect. If the integrated circuits are plugged into bases rather than being soldered, have a look to see if they have been inserted the right way round. Unfortunately, integrated circuit manufacturers take few precautions to prevent this type of error.

In most integrated circuits, the pins are numbered around the outside as shown in Figure 18.2. The position of pin 1 is always on the left-hand side of the end which has an indentation when viewed from the top as in Figure 18.2. When looking for the indentation don't be mislead by a small circular mark where the plastic has been molded. The printed circuit board usually has either a number '1' or a small square or other mark to indicate the position of the first pin.

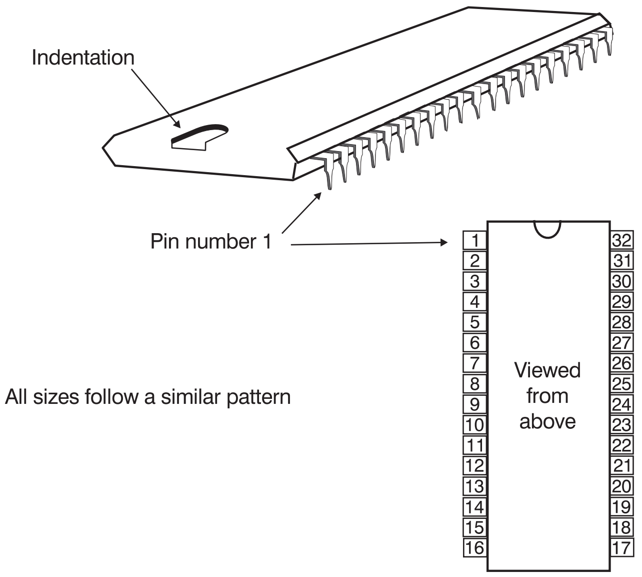


Figure 18.2
Pin numbering of 'dual in line' (DIL) chips

Figure 18.3 shows the pin grid array (PGA) layout. Notice that the letters skip from H to J because of the possible confusion between I and 1. The device determines the number of pins. The one shown happens to be the elderly Intel 80386. The Pentium has 21 pins along each side.

Simple test equipment

Apart from the standard voltmeter and an oscilloscope the only other simple piece of gear that may be helpful is the logic probe. It is better than the average oscilloscope at detecting very short voltage spikes and is faster to work with than a voltmeter.

Logic probe

A logic probe is a simple instrument that has two power connections and the other is a conducting tip that can be touched on points of interest. The general layout is shown in Figure 18.4. There are three LEDs on it. The first two show the logic states 0 or 1 and the third one indicates the presence of a high frequency square-wave or a single, very short duration, pulse, called a 'glitch'.

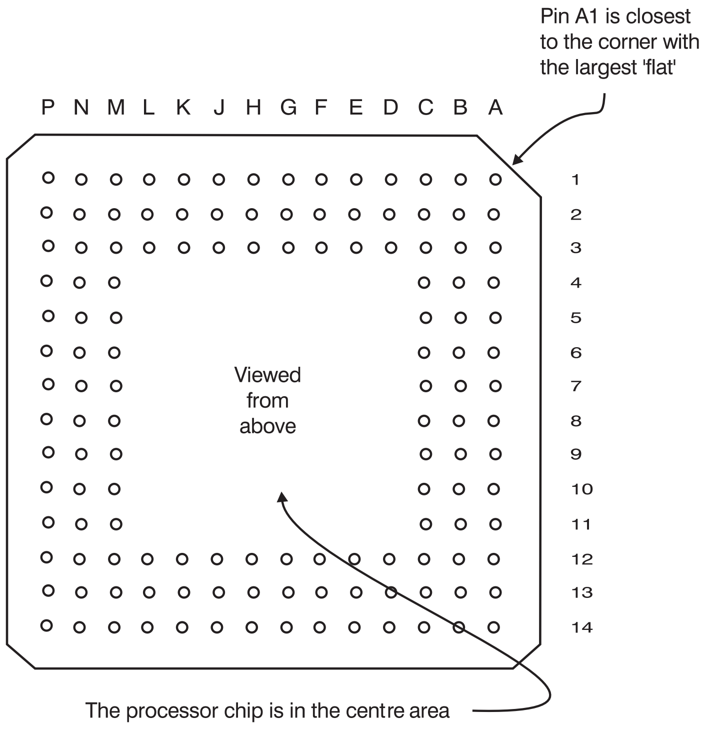

Pin A1 is closest to the corner with the largest 'flat'
P N M L K J H G F E D C B A
1
2
3
4
5
6
7
8
9
10
11
12
13
14
Viewed from above
The processor chip is in the centre area

Figure 18.3
Pin numbering of Pin Grid Array (PGA)

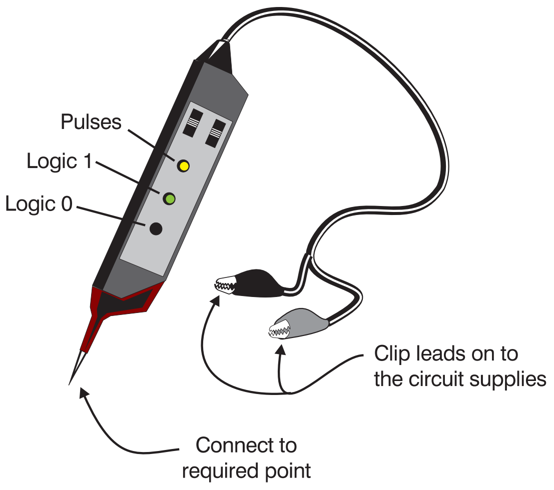

Pulses
Logic 1
Logic 0
Clip leads on to the circuit supplies
Connect to required point

Figure 18.4
A logic probe

Simple tests to make with these pieces of test gear

We can check some of the voltages on the microprocessor pins. If possible, it is a good idea to check on the actual pins rather than the base into which it is plugged. Doing this ensures that the base connections are also OK. It would also find the bent pin shown in Figure 18.5.

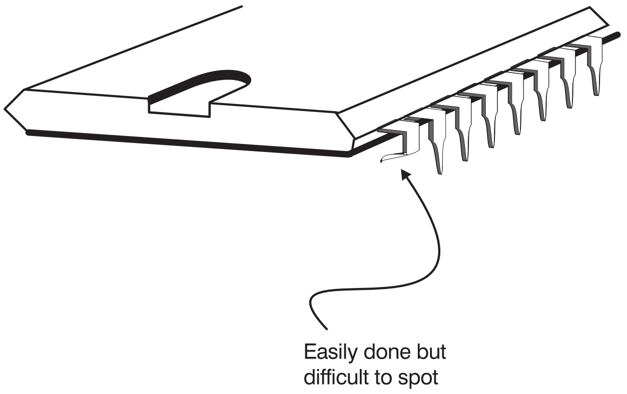


Figure 18.5
Pin bending not recommended

The likely pins that are worth checking are the ones carrying a dc voltage like the power supplies and the interrupts. It is worth keeping an eye on pins that should be at 0 V. When using a voltmeter, they can sometimes show 0 V when they are disconnected and floating. If you use your voltmeter to measure the voltage between the positive supply voltage and the suspect pin, it will still indicate 0 V showing that something is clearly amiss. A logic probe would not be fooled by a floating 'zero', it will not show a logic zero if it is floating.

The next job is to see if the microprocessor is running at all. We can do this by using the oscilloscope on a clock signal. Assuming that the clock signal is OK, we must next check that the microprocessor can follow an instruction and that the address and data bus are being read correctly.

A good check on the operation of the microprocessor can be arranged by getting it to do a simple repetitive program consisting of a permanent 'no operation' code. A no-operation code will instruct the microprocessor to do nothing except read the next instruction from the data bus by simply incrementing the value on the address bus. This new instruction will be another no-operation code and so the address bus will be continuously incremented. To provide a permanent no-op input we can solder or otherwise connect the required logic codes to the data bus. This is called hard-wiring the

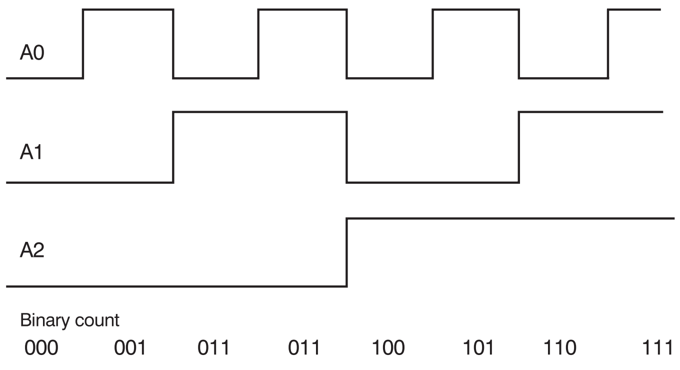


Figure 18.6 The address bus counts up in binary

This only shows the first three address lines.

Notice how the frequency changes.

data bus. As the address bus counts up in binary the lowest address line will be switching rapidly between zero and one giving a square wave output.

If we look at Figure 18.6, we can see that line A1 is running at half the frequency of line A0. Similarly address line A2 has half the frequency of A1 and so on all the way along the address pins. If we connect an oscilloscope to each line in turn, the frequency should reduce steadily. Check for the halving of frequency on each address line and errors in wiring like short circuits between address lines will become apparent.

If we get this far and still things seem wrong, we are into serious fault-finding.

A serious piece of test equipment

All the previous pieces of test gear have failed when we try to see what is happening on the address and data buses under real operating conditions. The oscilloscope cannot watch more than two different places at the same time but we may need to monitor a larger number, perhaps 50 or more places and then slowly check the information back or print it out. An instrument called a logic analyser can achieve all these functions and much more.

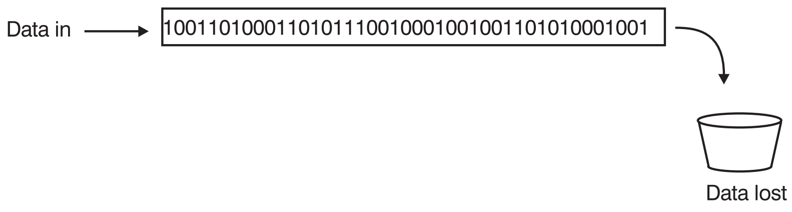
It can answer such questions as:

- ❍ What values actually appear on the address bus when we cause an interrupt to occur?
- ❍ Is the correct program actually being run?
- ❍ Are there any unwanted voltage spikes occurring?

The design of a logic analyser is basically a very simple combination of shift registers. You may remember we looked at shift registers in Chapter 6. The register was loaded with data and, on each clock pulse, the data is moved one place to the left or right as required.

Now imagine a shift-right register that can hold 36 bits of data. If we connect it to A0, the first line of the address bus, and run a program, the logic values of that address line will be copied onto the shift register, pass along to the end with each clock pulse and eventually start to fall out the far end (see Figure 18.7).

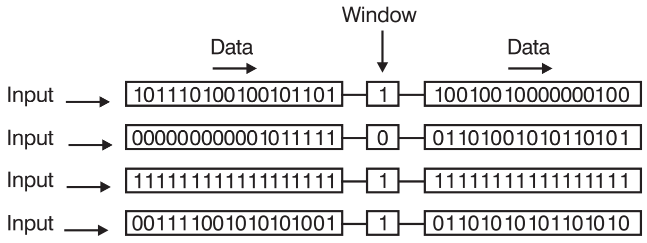
Figure 18.7



Now if we had four such registers, we could collect data from any four parts of a circuit at the same time. For example, we could monitor the lowest four address lines, which would be called A0, A1, A2 and A3.

In the centre of the register is a window. This means that we can access the centre of the shift register at this point to read off the data and to make comparisons. In Figure 18.8 only the four registers are shown for clarity. A simple arrangement like this would be referred to as a 4 × 16 (four by sixteen) logic analyser. In logic analyser specifications, the number of registers would be spoken of as the number of channels. In real life, we would never find such a simple arrangement of registers. Logic analysers could contain, say, 80 channels, each containing 4096-bit shift registers. This would be referred to as a 80 × 4 kbyte logic analyser. With one this size, we could monitor any 80 different points on a microprocessor-based system. Which points we choose are

Figure 18.8
Four shift registers can make a simple logic analyser



Window value = 1011

up to us, we could choose the whole of the address bus and the data bus and some control signals or any other points of interest. The choice is entirely ours.

If, in Figure 18.8, the four registers were being used to monitor four address lines, we may be suspicious of the line showing a constant value of logic 1. This may indicate that this line has become short-circuited to a positive power supply, or be disconnected and is floating high. Don't leap off your chair in excitement though – this is only one explanation. It could happen for these reasons or it could be running a part of the program where this would be the expected result.

So what about the window?

In the window, the logic analyser will 'see' a bit of data from each of the channels. We can load the combination that we are searching for. For convenience, we enter the values in hex numbers and as the clock pulses arrive from the microprocessor, the data moves across and is continuously compared with the number we have entered. When a match is found, the clock is switched off and the data is 'captured'. We can now move backwards and forwards along the registers and see the operation of the microprocessor 'frozen' in time.

The benefit of positioning the window in the centre of the shift register is that it allows us to observe the program action before, as well as after, the chosen moment.

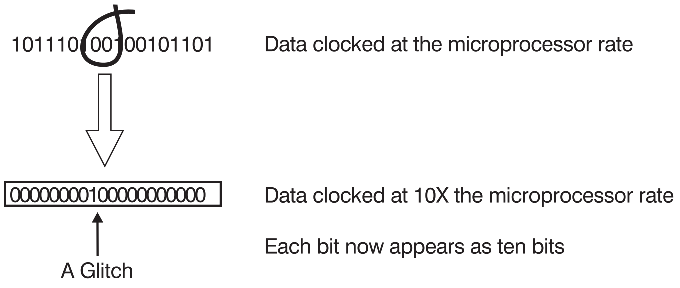
Extra facilities

A 'real' logic analyser has some extra facilities, like performing the capture not on the first time our input is seen but after perhaps the 200th occasion to take care of repetitive loops in the program. They also allow a 'don't care' condition on the inputs so in the window of a 20-channel logic analyser we could enter the hex code 7XXX2. This would perform a capture on any data that starts with 7H (0111_2) and ends with 2H (0010_2).

A glitch is a very short duration pulse that can occur in logic circuits, either from external interference or as a result of poor design. They can cause unwanted switching in the logic circuit and cause the microprocessor program to crash. They are exceedingly short, just a few nanoseconds and this makes spotting them very difficult. They are usually too fast for an oscilloscope but some logic probes have a 'glitch-catcher' built in, but they can only tell us that a glitch has occurred, not when it occurred. This is the information that will be needed if we are to track down a design problem.

A logic analyser may miss it because the incoming data is sampled once per pulse and if it misses the glitch it will not be recorded. To overcome this, the logic analyser can use its own internal clock that is

Figure 18.9
Glitch catching


101110100100101101
Data clocked at the microprocessor rate
000000001000000000000
Data clocked at 10X the microprocessor rate
Each bit now appears as ten bits
A Glitch

running much faster than the system clock so a single logic one may extend for 10 or 20 bits in the register and a glitch may well be recorded. Figure 18.9 shows the internal clock running at 10 times the microprocessor clock. Some logic analysers have a built-in glitch catcher and use it to capture the correct section of data. As we can see, the logic analyser is a very useful and sophisticated piece of kit. Using it, however, is a slow process. There are lots of connections to be made to the circuit and much sitting and thinking.

Quiz time 18

In each case, choose the best option.

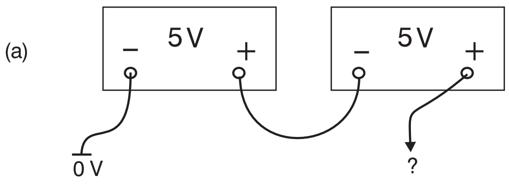
1 Damage due to static electricity:

(a) can only occur in the winter.
(b) is best prevented by wearing wet clothing.
(c) is only possible in carpeted areas.
(d) can be reduced by wearing a grounded wrist strap.

2 If the two power supplies were connected as in Figure 18.10, the result would be:

(a) smoke pouring from both power supplies.
(b) an output of +5 V but twice as much current.
(c) an output of +10 V.
(d) no output but no smoke either.

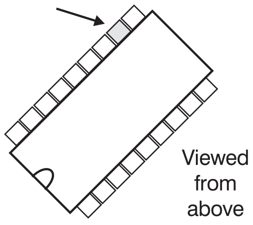
Figure 18.10



3 The arrow in Figure 18.11 indicates pin:

(a) 2.
(b) 8.
(c) 11.
(d) 16.

Figure 18.11



4 A logic probe:

(a) indicates whether your fault-finding technique is based on sound reasoning.
(b) can detect the difference between a disconnection and a grounded connection.
(c) can store a stream of data.
(d) detects the presence of static electricity.

5 A logic analyser quoted as 20 channels x 1024 bits:

(a) will show a four digit hex number in its window.
(b) can monitor any 1024 points at the same time.
(c) would store a total of 1044 bits of data.
(d) can monitor any 20 points at the same time.

Appendix A: Special function register file

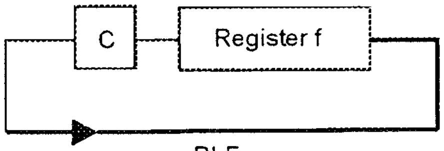
Addr	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Power-on Reset
BANK 0										
00H	Uses contents of FSR to address data memory (not a physical register)									
01H	TMRO	8-bit real-time clock/counter								xxxxxxxx
02H	PCL	Low order 8 bits of the program counter (PC)								00000000
03H	Status	IRP	RP1	RPO	TO	PD	Z	DC	C	00011xxx
04H	FSR	Indirect Data Mamory Address Pointer								xxxxxxxx
05H	PortA	–	–	–	RA4/T-CKI	RA3	RA2	RA1	RA0	---x xxxx
06H	PortB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RBO/ INT	xxxxxxxx
07	–	Unused								
BANK 1										
81H	Option	RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	11111111
85H	TRISA	–	–	–	PortA data Direction					---11111
86H	TRISB	PortB data Direction								11111111
87H	–	Unused								
88H	EECON 1	–	–	–	EEIF	WRERR	WREN	WR	RD	---0x000
89H	EECON 2	EEPROM control reg. (not a physical register)								--------

The full details (85 pages) of absolutely everything about the PIC16F84A (or any other PIC) can be downloaded from www.microchip.com.

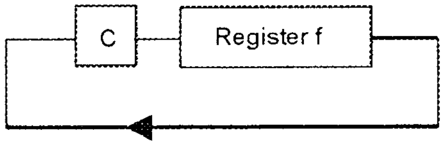
Appendix B: PIC 16CXXX instruction set

Syntax	*Description*	*Status affected*
ADDLW k	The contents of the W register are added to an 8-bit number and the result put in the W reg.	C.DC,Z
ADDWF f,d	Add the contents of the W and f registers. If d=0 the result goes to W. If d=1 the result goes to the f register	C,DC,Z
ANDLW k	The contents of the W register are ANDed with an 8-bit number and the result put in the W reg.	Z
ANDWF f,d	AND w with reg f. If d=0 the result goes to W. If d=1 the result goes to the f register	Z
BCF f,b	Bit b in reg f is cleared	
BSF f,b	Bit b in reg f is set	
BTFSS f,b	If bit b in reg f=0, the next instruction in executed. If it is 1 the next instr. is replaced with a NOP	1 or 2 cycles
BTFSC f,b	If bit b in reg f=1, the next instruction in executed. If it is 0 the next instr. is replaced with a NOP	1 or 2 cycles
CALL k	Call subroutine. Return address (PC+1) is pushed onto stack. The 11-bit immediate address is loaded into PC bits <1:0> The upper bits of the PC are loaded from PCLATH <4:3>.	2 cycle
CLRF f	Register f is cleared and Z flag is set.	Z
CLRW	Register W is cleared and Z flag is set.	Z
CLRWDT	Resets watchdog timer and watchdog prescalar	TO, PD
COMF f,d	Contents of 'f' are complemented (0∏1, 1∏0) If d=0 the result goes to W. If d=1 the result goes to f	Z
DECF f,d	Contents of 'f' reduced by 1. If d=0 the result goes to W. If d = 1 the result goes to f	Z
DECFSZ f,d	Contents of 'f' reduced by 1. if d=0 the result goes to W. If d=1 the result goes to f. If result = 1, the next instruction in executed. If it is 0 the next instr. is replaced with a NOP	1 or 2 cycles
GOTO k	The 11-bit immediate address is loaded into PC bits <10:0> The upper bits of the PC are loaded from PCLATH <4:3>.	
INCF f,d	If d=0 the result goes to W. If d=1, result goes to f.	
INCFSZ f,d	Contents of 'f' incremented. If d=0 the result goes to W. If d=1 the result goes to f. If rfesult = 1, the next instruction is executed. If it is 0 the next instr. is replaced with a NOP	1 or 2 cycles

Syntax	*Description*	*Status affected*
IORLW k	The contents of the W register are ANDed with an 8-bit number and the result put in the W reg.	Z
IORFWF f,d	The contents of the W register are Inclusive ORed with reg. F. If d=1 resuįlt goes back into f	
MOVF f,d	If d=0, contents of f goes to W reg. If d=1 it goes to f	Z
MOVLW k	The 8-bit number k goes into W.	
MOVWF f	Moves data from W register to f register	
NOP	Does nothing – įust a time waster (one cycle period)	
RETFIE	Return from interrupt. Top of stack∏PC, 1∏GIE	2 cycle
RETLW k	W reg loaded with number , return address∏PC	2 cycle
RETURN	Return from subroutine. Return address∏PC	2 cycle
RLF f,d	Contents of 'f' are rotated left one bit via the carry flag. If d=0 the result goes to W. Id=1, result goes back to f. See fig. below	C
SLEEP	Powerdown status bit PD is cleared, Timeout status bit TO is set WDT and prescaler are cleared, oscillator stops and controller goes to sleep.	TO, PD
SUBLW k	W register subtracted from the number k, result goes into W reg. (2's complement method)	C, DC, Z
SUBWF f,d	W register subtracted from the register f. If d=0 the result goes to W. If d=1 the result goes to f. (2's complement method)	C, DC, Z
SWAPF f,d	Upper and lower nibbles of f are exchanged. If d=0 the result goes to W. If d=1 the result goes to f.	
XORLW k	W register contents XOR'ed with the number k, result goes into W reg.	Z
XORWF f,d	W register contents XOR'ed with the register f, if d=0 the result goes to W. If d=1 the result goes to f	Z



RLF



RRF

Further reading

Bates, M. (2000) The Pic 16F84 Microcontroller. Arnold, London.

Bedford, M. (1996) Jubilee chips, *Computer Shopper*, December.

Bull, M. (1992) *Students' Guide to Programming Languages*. Butterworth-Heinemann, Oxford.

Carthy, J. (1996) *An Introduction to Assembly Language Programming and Computer Architecture*. International Thomson Computer Press.

Crisp, J. (1996) *Introduction to Fiber Optics*. Butterworth-Heinemann, Oxford.

Diefendorff, K., Oehler, R. and Hochsprung, R. (1994) Evolution of PowerPC architecture, *IEEE Micro*, April.

Digital Equipment Corporation (1996) Hardware reference manual of the Digital Semiconductor 21164 Alpha Microprocessor.

Horowitz, P. and Hill, W. (1989) *The Art of Electronics*. Cambridge University Press, Cambridge.

Intel (1988) *Microprocessor and Peripheral Handbook*, Vol. 1.

Krause, J.K. (1997) A Chip off the old block, *BYTE*, November.

Messmer, H.-P. (1995) *The Indispensable Pentium Book*. Addison-Wesley, New York.

Peleg, A. and Weiser, U. (1996) MMX technology extensions to the Intel architecture, *IEEE Micro*, August.

Predko, M. (1998) Programming and customizing the PIC microcontroller. McGraw-Hill, New York.

Wideman, G. (1986) *Computer Connection Mysteries Solved*. H.W. Sams, USA.

www.microchip.com

www.atmel.com

www.intel.com

Quiz time answers

Quiz time 1

1 (c)
2 (a)
3 (c)
4 (a)
5 (d)

Quiz time 2

1 (b)
2 (c)
3 (a)
4 (b)
5 (d)

Quiz time 3

1 (d)
2 (a)
3 (c)
4 (b)
5 (c)

Quiz time 4

1 (c)
2 (c)
3 (a)
4 (b)
5 (c)

Quiz time 5

1 (c)
2 (a)
3 (d)
4 (d)
5 (c)

Quiz time 6

1 (d)
2 (b)
3 (c)
4 (b)
5 (c)

Quiz time 7

1 (c)
2 (a)
3 (d)
4 (d)
5 (a)

Quiz time 8

1 (b)
2 (a)
3 (b)
4 (a)
5 (c)

Quiz time 9

1 (a)
2 (c)
3 (c)
4 (d)
5 (b)

Quiz time 10

1 (b)
2 (b)
3 (d)
4 (c)
5 (c)

Quiz time 11

1 (a)
2 (c)
3 (b)
4 (a)
5 (d)

Quiz time 12

1 (d)
2 (d)
3 (c)
4 (b)
5 (a)

Quiz time 13

1 (a)
2 (d)
3 (c)
4 (b)
5 (a)

Quiz time 14

1 (c)
2 (a)
3 (b)
4 (a)
5 (c)

Quiz time 15

1 (a)
2 (b)
3 (d)
4 (d)
5 (c)

Quiz time 16

1 (d)
2 (b)
3 (c)
4 (c)
5 (d)

Quiz time 17

1 (b)
2 (a)
3 (d)
4 (d)
5 (c)

Quiz time 18

1 (d)
2 (c)
3 (c)
4 (b)
5 (d)

Index