Microcontrollers
A Brief History of Microprocessors

The first microprocessor was developed by what was then a small company called Intel (short for Integrated Electronics) in the early 1970s. The client, a Japanese company called Busicon, declined to buy the chipset and Intel, faced with a development cost and no customer, decided to market the chipset as a "general purpose" microprocessing system for use in applications where digital logic chips would have been used. The chipset was a success and within a short while Intel developed a general purpose 4 bit microprocessor called the 4004.

In 1974 the more powerful second generation microprocessor (the 8008) was announced fabricated as a single chip. This was quickly followed by the Intel 8080. Both the 8008 and the 8080 operated from a single +5v power supply (using NMOS technology).

At about the same time Motorola released its first microprocessor, the 6800, which was also an 8 bit processor with about the same processing power as that of the intel 8080.

The architectures used in the Intel 8080 and the Motorola 6800 were very different.

The Intel 8080 used a register based architecture with registers AX, BX, CX, DX, and HL, all 16 bit but capable of being used as 8 bit register pairs so that the AX register could be used as two separate registers AH and AL. AH was really just the higher byte of register AX; and AL the lower byte. In this way, the AX, BX, CX, DX, and HL registers could be used as AH, AL, BH, BL, CH, CL, DH, DL, H, and L 8 bit registers.

Another thing about the 8080 was its separate I/O map. This meant that to perform byte-wide input/output to hardware, special instructions were used: IN to input from byte-wide input ports, OUT to output to byte-wide output ports. Access to memory involved access to a different memory map using typically the MOV instruction.

The Motorola 6800 microprocessor used what is called "Memory Mapped I/O" which means that both memory and byte-wide input/output share the same memory map.

The register set was much smaller, consisting of two 8 bit accumulators (A and B) and a 16 bit index register called X. These registers could however support a range of addressing modes which, in effect, made up for the fewer registers (and also made for simpler programming). Both microprocessors had other registers but we will ignore these since we have not yet formally looked at the microprocessor in detail.

To input data from memory or from I/O requires the use of the LDAA instruction, to write data to memory or I/O requires the use of the STAA instruction. Access to the X register was via its own set of instructions, ie. LDX and STX.
Intel and Motorola have maintained the fundamental differences in architecture during the development of later microprocessors.

In the case of the Intel range of microprocessors, the 8080 evolved into the 8085 (also 8 bit like the 8080), then the third generation 16 bit 8086 microprocessor which, in its 8088 pseudo 16 bit form, was used in the first IBM PCs. The 8088 was an 8086 but with only an 8 bit data bus. This made it easier to interface to the common 8 bit peripheral devices available at the time. In time this was followed by the 80186, the 80286, the 800386 (a 32 bit processor), and 80486, leading to the Pentium range of microprocessors (64 bit processors) available today. The 80x86 and Pentium processors have all been designed for use in personal computer type applications and have large memory maps.

The Motorola range of microprocessors followed a similar path with the 6800 replaced by the 6809 (8 bit), then the 68000 (16 bit), the 68010, 68020, and 68030 used in many workstations and of course the Apple MAC range of personal computers.

In due course the Intel 8080 core processor was used for a range of microcontrollers (8048 and 8051 to name but two). The 8051 microcontroller survives today but is now manufactured by Philips. Intel have gone on to develop a range of other microcontrollers that are more complex than the basic original devices (see [http://developer.intel.com/](http://developer.intel.com/) for more details).

Motorola followed in a similar vein with a range of microcontrollers based on the 6800 (6805, 6808, 6811 which survive to this day).

So, many of today's popular microcontrollers are based around two core architectures - that of the 8080 and that of the 6800 microprocessors.

Other manufacturers such as Rockwell produced microprocessors based on the 6800 architecture (6502), whilst Zilog developed the Z80 (based on the 8080 architecture).

The development of more recent microprocessor architectures such as the Harvard architecture and the use of Reduced Instruction Set Computers (RISC) have led to the development of microcontrollers such as the Microchip PIC.
1.1 Introduction to the microcontroller part of the course

The microcontroller is a very common component in modern electronic systems. Its use is so widespread that it is almost impossible to work in electronics without coming across it.

Microcontrollers are used in a wide number of electronic systems such as:

- Engine management systems in automobiles.
- Keyboard of a PC.
- Electronic measurement instruments (such as digital multimeters, frequency synthesisers, and oscilloscopes)
- Printers.
- Mobile phones.
- Televisions, radios, CD players, tape recording equipment.
- Hearing aids.
- Security alarm systems, fire alarm systems, and building services systems.
One could go on adding to this list, but the reader should now be getting the impression that there is a very wide scope for use of microcontrollers.

The following sections will attempt to answer the following questions:

1. What is a microprocessor?
2. What is a microcontroller?
3. What is the difference between a microprocessor and a microcontroller?

### 1.2 What is a microprocessor?
The microprocessor is the integration of a number of useful functions into a single IC package. These functions are:

- The ability to execute a stored set of instructions to carry out user defined tasks.
- The ability to be able to access external memory chips to both read and write data from and to the memory.

Microprocessors have been around for quite a long time now. For a brief history about the microprocessor read [here](#).

### 1.2.1 Types of memory
Memory can be obtained as either:

- Read Only Memory (ROM). This is memory that can only be read, the data being stored in the memory device during its manufacture.
- Erasable Programmable Read Only Memory (EPROM). This is similar to ROM type memory but the user can program it. The contents of the memory can be erased from the memory by exposing the memory chip to ultraviolet radiation for a short period of time. It can therefore be used many times over.
- Electrically Erasable Programmable Read Only Memory (EEPROM). Similar to EPROM but has part or all of the memory contents erased by the microprocessor.

Both ROM and EPROM memory are used to hold the program code of a microprocessor used in an embedded system, ie. a microprocessor used in an application where the program code is always the same and is designed to execute every time the system is switched on. Most development work is done using EPROM or EEPROM type memory, ROM memory being used in the final production version (when all the program code has been fully tested).
1.2.2 Random Access Memory (RAM)
All microprocessor systems need memory that can be both read from and written to. RAM memory is used to store dynamic data (that will change during the operation of the program).

So a typical microprocessor system will contain both ROM (could be EPROM, EEPROM, or ROM) to store the program code, and RAM to store dynamic data.

1.2.3 The Chip Select Line
The figure below illustrates the basic connections between the microprocessor and external memory.

![Basic connections between the microprocessor and its memory devices](image)

Figure 1. Basic connections between the microprocessor and its memory devices

In the next figure, we see the addition of address decoding logic to identify a particular range of address values to activate the chip select signal. This way, the memory device can only be selected within a particular range of addresses. The chip select line allows the memory to be either selected (/CS = 0, sometimes referred to as enabled) or not selected (/CS = 1, sometimes referred to as disabled).
The address decoding logic is simply a decoder that can be used to decode the particular combination of address inputs and activate one (of a number of) chip select outputs. See TTL data book for possible devices such as the 74138 and the 74139.

Example of Address Decoding Logic

Derive logic to select any addresses in the range A000 hex to AFFF hex. The bit pattern is illustrated here:

```
A15 A14 A13 A12 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0
  1  0  1  0  x  x  x  x  x  x  x  x  x  x  x
```
Figure 3. In this example, the chip select signal is active low when the input signal (A15, A14, A13, A12) is 1010. At all other times the chip select signal is logic 1 (disabled).

The address decoding logic can be used to generate a chip select signal for a number of memory devices. Each chip select output signal responds to a different address value, as in figure 4.

Figure 4. Three different Chip Select signals

The address decoding logic is usually used to select more than one memory chip since most microprocessor systems use a number of memory chips to make up a complete system. Only one chip select line will be enabled at any one time. This means that the
microprocessor sees only one memory device connected to the data bus at any one time, i.e. when the particular memory device is selected.

When no memory device is selected (as will be the case when the microprocessor is performing internal operations when memory access is not required), the data bus is placed into a mode called tri-state. This tri-state mode places an open circuit condition between the microprocessor and memory, thus disconnecting the data bus between memory and microprocessor. In fact, when the address decoding logic selects a particular memory device to be connected to the microprocessor, the other memory devices in the system are disconnected from the microprocessor. In this way, only one memory device can be connected to the microprocessor at any one time, the one selected by the address decoding logic.

1.3 What is I/O?
I/O is input or output (Input/Output). It can be:

- A number of digital bits formed into a number of digital inputs or outputs called a port. These are usually eight bits wide and thus referred to as a BYTE wide port. ie. byte wide input port, byte wide output port.
- A serial line from the microprocessor (Transmit or TX) and a serial line to the microprocessor (Receive or RX) allowing serial data in the form of a bit stream to be transmitted or received via a two wire interface.
- Other I/O devices such as Analogue to Digital Converters (ADC) and Digital to Analogue Converters (DAC), Timer modules, Interrupt controllers etc. (which will be discussed later in the context of microcontrollers).

These are relatively complex sub systems that can be obtained as separate ICs. They are connected to the microprocessor in a similar manner to that of the memory devices. Indeed, they often contain their own memory to support internal operations (ie. registers).

1.3.1 Digital I/O
A digital I/O port can be realised using a number of D type Flip-Flops.

1.3.2 Output Port
The figure below illustrates a four bit output port. The inputs are connected to the data bus whilst the outputs are connected to whatever output interface is to be controlled.
1.3.3 Input Port

Figure 5. A 4-bit output port

Figure 6. A 4 bit input port.
The input port allows outside world inputs (left hand side of the above figure) to be stored in the data latches so they can be read by the microprocessor via the data bus (left hand side of the figure).

The data bus connections must be via tri-state buffers so that the input port is only connected to the data bus when the input port is selected. This is achieved by connecting a chip select signal to the enable input signal line. Note that the tri-state enable is active low.

We have covered some of the basic concepts of the microprocessor. There is of course much more but we can pick up on these later. We now need to turn our attention to the question...

1.4 What is a microcontroller?
Basically, a microcontroller is a device which integrates a number of the components of a microprocessor system onto a single microchip.

So a microcontroller combines onto the same microchip:

- The CPU core
- Memory (both ROM and RAM)
- Some parallel digital I/O

Figure 7. Main components of a microcontroller
Most microcontrollers will also combine other devices such as:

- A Timer module to allow the microcontroller to perform tasks for certain time periods.
- A serial I/O port to allow data to flow between the microcontroller and other devices such as a PC or another microcontroller.
- An ADC to allow the microcontroller to accept analogue input data for processing.

![A Single Chip Microcontroller](image)

Figure 8. A single chip microcontroller

The above figure illustrates a typical microcontroller device and the different sub units integrated onto the microcontroller microchip.

The heart of the microcontroller is the CPU core. In the past this has traditionally been based on an 8-bit microprocessor unit. For example, Motorola uses a basic 6800 microprocessor core in their 6805/6808 microcontroller devices.

In recent years, microcontrollers have been developed around specifically designed CPU cores for example, the Microchip PIC range of microcontrollers.
1.4.1 Memory in a microcontroller

The amount of memory contained within a microcontroller varies between different microcontrollers. Some may not even have any integrated memory (eg. Hitachi 6503, now discontinued). However, most modern microcontrollers will have integrated memory. The memory will be divided up into ROM and RAM, with typically more ROM than RAM.

Typically, the amount of ROM type memory will vary between around 512 bytes and 4096 bytes, although some 16 bit microcontrollers such as the Hitachi H8/3048 can have as much as 128 Kbytes of ROM type memory.

ROM type memory, as has already been mentioned, is used to store the program code. ROM memory can be either ROM (as in One Time Programmable memory), EPROM, or EEPROM.
The amount of RAM memory is usually somewhat smaller, typically ranging between 25 bytes to 4 Kbytes.

RAM is used for data storage and stack management tasks. It is also used for register stacks (as in the microchip PIC range of microcontrollers).
1.4.2 The I/O Ports

The digital I/O ports are the means by which the microcontroller interfaces to the environment.

Digital I/O tends to be grouped into byte wide ports (8 digital bits) that can be configured as either input bits or output bits. There are some exceptions, such as the microchip PIC 16C54 with one 6-bit RA port and a byte wide RB port.

The number of I/O port bits varies, depending upon the size of the microcontroller. Some very simple 8 bit microcontroller have as few as 4 bits of I/O, whilst those at the high end range can have as many as 33 bits of I/O (some 16 bit microcontrollers could have around 78 bits of I/O).

A typical interface might look like the one illustrated below.
Microcontroller ports can be used to operate LEDs and relays, as well as input the state of switches and logic circuit inputs (not shown in this figure). Not all microcontroller port outputs are able to drive an LED directly; some need to be interfaced via a buffer such as a 7406 open collector inverting buffer. In the example above, the relay is driven by such a buffer.

Study Task 2
Look at the following web sites and discover the amount of memory used in both the microchip PIC, and the Motorola range of microcontrollers

- [http://www.microchip.com/](http://www.microchip.com/)
Self Assessment Questions

In answering these questions, just look for basic answers; don't try to go into too much depth at this stage.

1. What is the difference between a microprocessor and a microcontroller?

**Answer:** See section 1.2.1 to 1.3.3 for what a microprocessor is. Section 1.4 indicates what a microcontroller is. Essentially, a microcontroller is obtained by integrating the key components of microprocessor, RAM, ROM, and Digital I/O onto the same chip die. Modern microcontrollers also contain a wealth of other modules such as Serial I/O, Timers, and Analogue to Digital Converters. There are a large number of specialised devices with additional modules for specific needs. eg. CAN controllers, Serial Peripheral drivers.

2. A microprocessor is to be used in an application requiring 1Kbyte of data memory and a program of 8Kbytes. What type of memory is required in each case?
   a. For a prototype system
   b. For a final product.
   Also, how many bits would be required for the address bus?

**Answer:** 1KB memory for data requires the use of RAM memory since the data will change during the execution of the program.

8KB program memory will need ROM/PROM/EPROM/EEPROM memory since the program is static i.e., does not change during the life of the application.

The total amount of memory is 8KB + 1KB (there is no indication of any digital I/O) so the total memory space is 9KB.

To accommodate 9Kbytes of memory space will require an address bus with n bits where

\[ 2^n = (8192 + 1024) = 9216 = 2400 \text{ hex}. \]

2400 hex is 10 0100 0000 0000 in binary which is 14 bits so \( n = 14 \).

Put another way, \( n = \log(9216)/\log(2) = 13.17 \) which round up to 14 bits.

Another way of looking at it is to think of the binary weightings.
3. After looking at the Microchip web site you should gather that there are three ranges of PIC devices. What are the main differences between these three ranges of microcontrollers in terms of device physical size, memory, digital I/O, size of address bus and data bus, and the number of peripheral devices they contain (don't worry about what they are)?

**Answer:** The BASE LINE with ROM/EPROM memory from 512 to 2KB, between 25-72 bytes of RAM and a Timer module. These are designed for small applications and are contained in either an 18 pin or a 28 pin IC package, depending upon the amount of digital I/O offered. The smaller devices like the PIC16C54/PIC16F84 have only 12 bits of Digital I/O, whilst the PIC16C55 has 20 bits. The base line PICs use a 12 bit wide instruction set.

The Mid-range line are contained in 18 - 40 pin IC packages and contain between 1 and 4KB of program memory (ROM/EPROM) and 36 to 128 bytes of RAM. They all contain a Timer unit, but some also contain a serial port or an Analogue to Digital converter. The amount of digital I/O ranges from 13 bits to 33 bits. The Mid range PIC devices use a 14 bit wide instruction set.

The High end line of PIC devices are contained in 40 pin IC packages with a large amount of digital I/O (55 bits). They have at least 2Kbyte of ROM/EPROM and extended Timer module capability (more on this later). They have small amounts of RAM memory (around 128 byte) and can support serial I/O and a serial peripheral port. One of the most useful features of the high end line of PICs is that some can support external memory. This allows the addition of more RAM for data memory, or more ROM/EPROM for Program memory in applications requiring this. The High end PIC devices have a 16 bit wide instruction set.

4. Which of the PIC devices can be used to input analogue voltages.

**Answer:** The PIC16C7x range of microcontrollers all contain an Analogue to Digital converter that allows analogue signals to be converted into digital form for processing within the PIC.

5. After looking at the Motorola web site what is the range of memory size for program memory and data memory for the 6805 range of microcontrollers?

**Answer:** The Motorola 6805 microcontroller has a very wide range of different devices, each of which supports a number of internal function modules (you will cover module types in chapter 2). Program memory, which includes ROM/EPROM range from 256
bytes up to 32Kbytes, with some devices having 8/16 Kbyte of internal program memory.

6. What are the significant differences between the 6805 and 6808 microcontrollers?

**Answer:** The Motorola 6805 and 6808 both share the same internal CPU core architecture and hence the same basic instruction set. The range of devices within each range is large but similar. The 6805 does not contain Flash memory, whilst some of the 6808 family members can contain between 20K and 60K of Flash memory. This means that the 6808 can be used in applications where programs can be loaded into the microcontroller without removing it from its circuit board.

From your first browse of the above web sites you should now have a "feel" for the range of microcontroller products currently available. It is clear that these devices can be applied to many embedded system designs from the simple hardware control applications up to Signal Processing applications.

Note: Intel also produce a range of microcontroller devices. The 8051, 80151, and 8XC196 devices can be browsed at: [http://developer.intel.com/](http://developer.intel.com/) You don't need to look at this site in detail just yet but keep the website URL handy.
2.1 Introduction

Most microcontrollers contain a number of hardware modules. In the past many of these were designed as separate chips in a conventional microprocessor system. Integrating them into a single microcontroller chip allows for greater functionality in a single chip and saves space.

Typical devices are:

- Timer module
- Serial I/O module
- Analogue to Digital Converter module
The diagram above illustrates these modules within a typical microcontroller which also contains ROM and RAM memory and ports to access and control the outside world.

2.2 Timer modules

Timer Output Example

Can vary on time and off time dynamically by reprogramming the timer during program execution.
A common requirement of microcontrollers is the ability to be able to turn on an output (such as an LED or relay) for a period of time. This can be done by turning on the device, then causing the microcontroller to decrement a specified number to zero (a process that takes the microcontroller a specified amount of time to do), then turning the device off again.

This method of turning a device on for a specified period of time is wasteful in the sense that the microcontroller could be doing other things whilst it is waiting for the device on time to elapse. A hardware timer could be used instead and would perhaps allow the microcontroller to do other things during the on time period. Most microcontrollers have at least one timer module, usually with a number of inputs and outputs.

The inputs allow the timer to be used to measure the time of an input signal applied to the timer input, as illustrated below.

In this example the rotating shaft produces a single pulse every revolution. The microcontroller measures the time taken for a complete revolution and so determines the angular velocity of the shaft.
The above figure illustrates how a timer output could be used to produce an output pulse train, possibly to drive some external logic. The timer is set up so that an on time and off time can be generated by the timer unit.

**2.2.1 Internal Timer Functions:**

Timer interrupts CPU after pre-defined time period.
The timer unit can also be used for generating a delay. The timer is programmed with a specified count value which will be counted down (decremented) to zero. When the timer unit reaches zero a flag is set. This flag can be monitored by software so that the microcontroller will know when the timer unit has reached zero. Since the time taken by the timer unit to decrement from the specified count value to zero is simply the count value multiplied by the time between decrement operations, a known delay period is produced. The time between decrement operations is a function of the crystal clock oscillator frequency so it can be defined very accurately. The microcontroller can be doing other things between monitoring the flag bit so this is less wasteful in processor time than the simple software delay. (We will look at these aspects in more detail later on in the course).

In the above figure we see an LED prior to being turned on via the port bit PB0. Just prior to turning on the LED, the timer unit is loaded with a count value (the actual value being dictated by the delay time required). In due course the timer unit count value will reach zero, at which point a flag bit will be set. The CPU, on seeing the flag bit set, will turn the LED off. The time taken for the timer unit to decrement the count value to zero will be the time that the LED is on.

Another way of using the timer to create a delay is to cause the timer unit to interrupt the CPU when the timer reaches zero. This can be more efficient in processor time since the microcontroller can be doing other things uninterrupted until the timer reaches zero. The concept of interrupts will be expanded upon later in the course.
2.3 The Serial Port

Some microcontrollers have a serial port to allow data to be transmitted to another microcontroller, PC or remote system via a two wire pair. This can be a very convenient way of transmitting data between two devices. A disadvantage can be the slower speed of data transfer that a serial port has over a byte wide parallel data port. The transfer rate can be programmed from within a range of typically 300 baud (300 bits per second) to 38400 baud.

The serial port is usually referred to as a *Serial Communications Interface* (SCI). Most SCI units contained in microcontroller chips are a sub-set of the more traditional *Universal Asynchronous Receiver/Transmitter* (UART) unit that is available in the standard PC. The UART has additional signals that allow it to be connected to a modem. This should be borne in mind when considering microcomputer applications that require the use of a modem. Most SCI units can work in asynchronous mode, whilst some also support synchronous working.
In the above diagram we see a microcontroller with a SCI unit connected to another system (which could be a PC) via the transmit and receive pair. It should be pointed out that the logic levels from a SCI port are TTL. Therefore, if the microcontroller is to be connected to a PC, a TTL to RS232 interface chip is required to convert the signal levels from TTL to RS232 levels. A corresponding RS232 to TTL interface chip is included in the PC to perform the necessary signal level translation back to TTL levels for the UART chip used in the PC (a 16050 UART). This has not been shown in the above diagram.

In some applications, a special purpose interface might be designed with its own serial interface.
In the above diagram we see a remote display system controlled by a microcontroller via its SCI unit. This arrangement offers a simple interface between the two units and could be the basis for a number of remote “repeater” displays (by re-transmitting the data from one LCD sub-system to another).

2.4 Analogue to Digital Conversion

In the real world signals are often analogue in nature. For example, we might need to monitor (log) analogue signals from a strain gauge, or from a transducer converting temperature to an electrical analogue signal (temperature transducer). Since microcontrollers are sometimes required for these types of applications, some are fitted with an Analogue to Digital Converter (ADC).

In most cases, the ADC is of the Successive Approximation type and there are a number of analogue channels (between 4 and 8 is most common).

The analogue channel inputs are usually shared with one or more of the digital I/O ports. That is to say they are multiplexed with the digital ports and the decision to select whether the chip pins are to be used for analogue channel inputs or digital ports is made in software (during the initial configuration of the program on reset). More on this later.
It is uncommon for an ADC to derive more than 10 bits. For more demanding applications the ADC performance needs careful examination.

For example:

- On chip sample and hold to accommodate ac signal sampling.
- Control over the sample clock and synchronisation between channels.
- CPU loading at higher sampling frequencies.

*Control over the sample clock allows the system to sample at a rate required by some applications (which may be performing algorithms that require a specific sampling frequency for meaningful data calculations).*

*Higher clocking speeds means less time to do things.*

In general, most 8 bit microcontrollers have limited performance due to the limited ADC resolution and CPU performance. They are restricted to simple data logging and measurement applications not requiring high precision. Most 8 bit microcontrollers are unable to perform complex calculations owing to their limited instruction set and speed. For more intensive applications involving complex algorithms, a DSP is a better choice.

A simple typical application is illustrated below.

Here we see a microcontroller being used to measure an analogue signal level. The microcontroller includes an ADC module and one of its digital input ports has been configured (by software) to be an analogue input channel. The software running on the microcontroller will perform an Analogue to Digital conversion of the input signal, then determine whether the input signal is above or below some pre-defined limit. It will then turn on the appropriate LED.
We have now covered most of the units found in a microcontroller. Note that not all microcontrollers will contain all of these units.

Most manufacturers of microcontroller chips produce a range or family of microcontroller devices. The family will range from simpler (and hence cheaper) microcontrollers with perhaps only I/O ports, and a Timer unit, through to high end devices containing perhaps all of the units discussed so far.

### 2.5 Special Function Units

Some microcontrollers will contain other “special function” units such as

- Serial Peripheral Port (SPP).
- Controller Area Network (CAN) interface.

#### 2.5.1 Serial Peripheral Port

A *Serial Peripheral Port* (SPP) is a special interface that can be used to connect the microcontroller to a particular peripheral device such as an LCD display or a serial Analogue to Digital converter. This saves on valuable I/O pins and leads to a neat interface between the microcontroller and the peripheral device.

The same interface can be used to connect two microcontrollers together so that they can pass data between each other. This can allow a number of microcontrollers in a system to share data or allow one microcontroller (the master) to control a number of other microcontrollers (the slaves).

Some of the PIC microcontrollers have such a device called the *Synchronous Serial Port* (SSP). The Motorola 6811 microcontroller uses a device called the *Serial Peripheral Interface* (SPI) in a similar manner.

#### 2.5.2 CAN Interface

The CAN interface was originally developed for the automobile industry in order to reduce the amount of cabling in a vehicle. This has become particularly important since the wiring loom in an automobile can now account for a substantial proportion of the vehicle weight.

Serial data (following a pre-defined protocol) is transmitted around a number of subsystem units (which also contain CAN interfacing units). The CAN interface is used to control such things as the operation of the lights, monitor the engine parameters, even control the braking system, using a distributed CAN bus (twin wire pair). The heavy wiring can therefore be kept short and only a power distribution system is required to
get power to the individual unit being controlled. Each device to be controlled and each device sending data to another part of the automobile contains a CAN interface device. Each CAN interface is called a node.

Data messages transmitted from any node on a CAN bus do not contain addresses of either the transmitting node or of any intended receiving node. Instead, an identifier that is unique throughout the network labels the content of the message (e.g. revolutions per minute, headlights on, etc). All other nodes on the network receive the message and each performs an acceptance test on the identifier to determine if the message (and hence its content) is relevant to that particular node.

Some of the more prominent names manufacturing microcontroller chips have introduced CAN units into their range of microcontrollers.

For example,

- Microchip have introduced the MCP2510 stand alone CAN controller chip for use with their range of PIC microcontrollers, and say that they will have a PIC with an embedded CAN controller (the PIC18C641) available by the third quarter of 1999.
- Motorola have integrated CAN derivatives available in their 68H05, 68H08, and 68H12 families of microcontrollers.
- National Semiconductor have an integrated CAN unit in their COP888EB and COP884BC microcontroller devices.
- Philips also produces a stand-alone CAN controller, the SJA1000, and also produces an 8-bit microcontroller (P80C552 and P80C558) with integrated CAN controller.
- ST microelectronics manufactures the ST72E50, ST72511, and ST72531 8 bit micros with integrated CAN controller.

---

**Study Task 1**

Visit the Microchip web site and look at the range of microcontrollers that they provide. These microcontrollers are collectively called PIC microcontrollers, meaning that they are Peripheral Interface Controllers.

- [http://www.microchip.com/](http://www.microchip.com/)

In particular, look at the following devices:

- 16C54
- 16C84
- 16C71
- 16C73
- 16C42
Note the:

- amount of digital I/O offered by each device.
- amount of ROM and RAM memory offered by each device.
- type of memory (OTP, EPROM, Flash).
- number and type of peripheral modules available on each device (timers/serial I/O, ADC, etc).
- number of pins.

This will give you a feel for the support provided by the particular microcontroller manufacturer.

---

Study Task 2

Look at the following web sites:


Look for the following devices and determine the number and type of peripheral modules available.

- Intel 87C196
- Motorola M68HC12
- Motorola MPC500 family of microcontrollers

In particular, consider the type of application these devices are targeted towards and the size of their memory map.

---

Self Assessment Questions

1. What are the main reasons for having a timer module on a microcontroller?

**Answer:** The microcontroller can execute instructions in microseconds. External operations usually need to occur at very much reduced rates. For example, the turning on and off of a LED indicator needs to be performed much more slowly that the time to execute the turn on/off instructions. The main reason for using a Timer module to produce time delays is that it frees up the processor to do other things, or at least reduces the time the processor needs to spend performing these activities.
2. What is the typical resolution for an Analogue to Digital Converter in the microcontroller devices you have seen?

**Answer:** Most microcontrollers that you look at which have an Analogue to Digital Converter (ADC) are 8 bit types. There are a few with 10 bit ADCs, but these are still relatively rare.

3. Some microcontrollers have a serial port. Could this serial port be used to connect to an external modem?

**Answer:** The serial ports available on most microcontrollers do not allow a modem to be connected to the serial port. This is because the serial port is a simple device that does not have the additional signals, such as CTS/RTS and DTR/DSR, that are needed for interfacing to a modem. This means that an application requiring a modem will need to use an external serial port such as a UART to connect to the modem device.

4. Give some examples of microcontrollers that you have come across that contain a serial peripheral port. Are there any external devices that could be used with these ports?

**Answer:** In the PIC range there is the PIC16C74, PIC16C73, PIC16C65, PIC16C64, PIC17C42, to name a few. In the 6805 range there is the MC68HC05B176 and many more. Some support both a Serial I/O port and a serial peripheral port (but more on this later).

The serial ports could be connected to and UART TX/RX lines, or the serial lines of another microcontroller. To connect to the serial port of a PC would require TTL to RS232 interface device - see chapter 2 notes.

5. After looking at the Motorola web site, what are the differences between the 6805 and 6808 microcontroller devices in terms of the number and type of peripheral devices they contain?
**Answer:** Both the 6805 and the 6808 contain a very wide range of peripheral devices. Most contain the basic modules discussed in this chapter (Timer, Serial I/O, and ADC), whilst some do not have the ADC.

6. What is the maximum amount of external memory that can be connected to the 6811 microcontroller? Can this external memory be used for either program or data memory?

**Answer:** The 6811 microcontroller supports the use of external memory by using the Ports B and C as an external data/address bus. The address bus is 16 bits wide so it can handle up to $2^{16}$ (65536) memory locations. The total amount of external memory is therefore 65536 - the memory available on the microcontroller chip itself.

7. What are the main differences between the 6811 and 6812 microcontroller architectures?

**Answer:** Essentially, the 6811 is an eight bit microcontroller (having an 8 bit data bus width) whilst the 6812 is a sixteen bit microcontroller (having a 16 bit data bus). Both microcontrollers share the same basic architecture and instruction set. This does not mean that a program designed to run on a 6811 will run on a 6812.

8. What are the main features of the Intel 87C196 range of microcontrollers? What type of application have Intel targeted this microcontroller towards?

**Answer:** Look at the developer.intel.com/design/autoxpr.htm page for details. Essentially, the processor is targeted towards automotive applications, in particular, Anti-Lock braking. It also includes J1850 Communication Protocol (an in-vehicle communications standard for data sharing - CAN - Controller Area Network). The 87C196 is a 16 bit microcontroller with the J1850 protocol handler, watchdog timer, digital I/O ports, two timer modules, an ADC, a serial port, and 20KB of program memory. It also has an external memory interface unit.

From your browse of the above web sites you should now have a "feel" for the range of microcontroller products currently available. It is clear that these devices can be applied to many embedded system designs from the simple hardware control applications up to powerful signal processing applications.
### Chapter Information

<table>
<thead>
<tr>
<th>3.1 An introduction to software and the microcontroller instruction set</th>
<th>3.8 The instruction set and its use</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2 The programming requirements for a microcontroller</td>
<td>3.8.1 Bit operations and their use</td>
</tr>
<tr>
<td>3.3 Getting your program to work correctly</td>
<td>3.8.2 Using the W register</td>
</tr>
<tr>
<td>3.4 The PIC software development cycle</td>
<td>3.8.3 Port configuration</td>
</tr>
<tr>
<td>3.5 Source code writing</td>
<td>3.8.4 Some other PIC instructions</td>
</tr>
<tr>
<td>3.6 The reset vector</td>
<td>3.8.5 Bit setting and clearing by masking</td>
</tr>
<tr>
<td>3.7 The PIC instruction set</td>
<td>3.8.6 Instructions using the destination flag bit</td>
</tr>
</tbody>
</table>

#### 3.8 The instruction set and its use

- 3.8.1 Bit operations and their use
- 3.8.2 Using the W register
- 3.8.3 Port configuration
- 3.8.4 Some other PIC instructions
- 3.8.5 Bit setting and clearing by masking
- 3.8.6 Instructions using the destination flag bit
- 3.8.7 The auto-decrement and auto-increment instructions
- 3.8.8 The purpose of sub-programs
- 3.8.9 The stack and its use
- 3.8.10 Interrupts and why you need them

#### Self Assessment

- 3.9 Practical experimental work with MPLAB
  0. MPLAB Tutorial
  1. Introduction to MPLAB
  2. Simulation of Input / Output
  3. Programming a device
  4. Setting breakpoints and using the trace feature

- 3.10 Programming Exercise
3.1 Introduction to Software and the Microcontroller Instruction Set

In this chapter we will look at the assembly language programming requirements for a microcontroller. This will be followed by a detailed look at the instruction set of the PIC microcontroller with examples. Some fundamental programming requirements will be discussed along the way.

A microcontroller is designed to control hardware. To this end it has a number of digital I/O ports; it may also have ADC and DAC devices so that it can interact with analogue signals. And finally, it will contain a number of peripheral devices such as timer modules, serial I/O, an ADC unit and perhaps other specialised peripheral modules such as a CAN controller or serial peripheral controller.

The instruction set for a microcontroller will therefore be designed to allow efficient control of both its internal devices and the surrounding infrastructure (controlled by its ports).

The main requirements for the instruction set are:

1. Control of the register set of the microcontroller in an easy manner
2. Ability to access ports and other peripheral control and status registers
3. Ability to be able to access individual bits of a port or register
4. Good masking operation ability to test port and register bits individually
5. Good interrupt infrastructure
6. A rich set of addressing modes

The instruction sets for microcontrollers vary greatly. Some offer a very large number of instructions (eg. the Hitachi H83048 and Motorola MC6808), others have a very small set (eg. the PIC range of microcontrollers).

We will consider the PIC microcontroller instruction set in some detail and will then compare it with other microcontrollers such as the MC6805 and MC6808.

3.2 The Programming Requirements for a Microcontroller

These are much the same as for any computer. The microcontroller software development will be based around a programming environment which will normally include:

- A text editor to develop the source program code
- An Assembler to produce a machine executable program with linked library modules
- Some form of debugging to allow the program to be tested
- Access to a programmer to program the target device.
The job of the Assembler is to convert the source code produced by the programmer into an object code file. It works by replacing each line of assembly instruction code with the corresponding machine code instruction that will be executed on the target microcontroller.

The object code file cannot be executed on the target microcontroller. It needs to be passed through a program called a Linker.

The job of the Linker is to take the object file produced by the Assembler and link it with standard library modules (that have already been pre-assembled into object files) to produce a complete working executable file that can be loaded into the memory of the microcontroller.

Sometimes the Assembler and Linker are combined into one program so that assembling a source program automatically links it as well. This is the case with the PIC Assembler.

The above diagram shows the different stages in the production of a program. Note that the only stage that requires a lot of user input is the initial production of the user source code file. The rest of the program production involves using the Assembler and Linker tools.
A program called a Loader takes your executable .hex file from the hard disk and loads it onto the target microcontroller.

The original source file will contain instructions about where to place the .hex file in the target microcontroller memory. Typically, this instruction will be:

**ORG 0000H**  
This instruction ORGanises the target program to be loaded to the target microcontroller memory address 0000 hex.

ie. at the address 0000 0000 0000 0000 binary.

Other key addresses may also be needed, such as the interrupt vector addresses for any interrupt service routines (isr) that the program may use.

Generally, the Loader program is integrated into the development environment so that it is performed by the programmer tools.

For example, in the MPLAB PIC development environment, the PICSTART menu will deal with loading the target program into the memory of the EPROM Programmer system which will then program the target microcontroller which has been placed into the zero insertion socket of the programmer system.

---

### 3.3 Getting your Program to Work Correctly

If the Assembler and Linker report that they have successfully assembled and linked the program it might be assumed that the program will then run correctly. This is not necessarily the case.

All the Assembler will do is ensure that the instructions you have used in your program use the correct syntax (ie. they conform to the spelling and format required by the assembler). This does not mean that the program will run correctly since it is possible to create a program that is syntactically correct but is nonsense.

The next step is to execute the program and see if it does what we want it to do, and if not, modify it so that it does. This process is known as **debugging**.

Debugging can take up more time than the actual writing of the original program source code. There are a number of ways in which a program can be debugged:

- Use a **Simulator** program to run the executable (.hex) file. Typically the simulator runs on the development system (usually a PC).
- Use an **Emulator** system to run the executable (.hex) file. This is usually a hardware system that contains a version of the target microcontroller with additional hardware and software support to the program instructions to be...
monitored at the assembly level as they are executed. The Emulator system can also be used to allow the program to run on the target system while being monitored by the emulator system. The emulator system will also be connected to the development computer.

- Use an EPROM Emulator. This is a system that contains RAM. It is connected to the target system by removing any EPROM memory from the target system and placing the header attached to the cable from the EPROM Emulator to the EPROM socket of the target system. In this way the RAM memory of the EPROM Emulator is used in place of actual EPROM memory. The EPROM Emulator is connected to the PC and the executable program is placed into the RAM or the Emulator. It runs in this RAM, but the target system sees the program in its EPROM address space. With this arrangement, the PC can access the EPROM Emulator RAM and can thus control the program execution.
• Develop the program on the development system (PC) with Assembler and Linker, then download the program directly to a target system to run on that system. This method is not very productive since there is no control of the program execution on the target system.

The appropriate method for this course is to use a simulator on the PC to develop programs, then download the executable files to the PIC trainer board.

The PIC used is a PIC16c84, which contains EEPROM. Thus the executable file is downloaded from the PC into the EEPROM memory of the PIC16c84. The PIC trainer contains hardware to program the EEPROM memory of the PIC16c84, whilst the program used to download the executable file (LC16.exe) provides the necessary control to program the EEPROM memory.

3.4 The PIC Software Development Cycle

The development of a microcomputer-based application thus follows the paths shown in the following diagram.
Software design is usually based on a specification. The sequence of HW design, SW design, Assembly, Simulation and Debug may be repeated a number of times until the system meets the required specification.

The turquoise boxes represent areas which require the most user activity; the most intensive activities are HW design, SW design, Simulation, and Debugging.

3.5 Source Code Writing

This section will look at the way in which an assembly language program is structured and introduce some of the assembly language instructions of the PIC microcontroller.

The source code for a microcontroller can be written in assembly language or in a high level language such as C (providing a suitable compiler is available). For this part of the course we will use assembly language programming.

A typical assembly language program will consist of:

- some assembler directives
- sub-routines if they are needed
- the main program code
Assembler directives are a collection of commands that tell the assembler such things as the type of microcontroller being used, its clock speed, etc. They also allow names to be used for memory locations, ports and registers, so making the program more readable.

Some assembler directives:

The EQU directive associates a name with a physical value such as an address in memory.

\[ \text{Name EQU physical\_value} \]

RTCC EQU 1 ; the timer counter
PLC EQU 2 ; the program counter
STATUS EQU 3 ; the status register
CARRY EQU 4 ; the carry bit (in the status register)
DCARRY EQU 1 ; the digit carry bit (in the status register)
W EQU 0 ; result destination to W work register
F EQU 1 ; result destination to F (file) register
Z EQU 2 ; the zero bit (in the status register)

After the above equate directives the names can be used instead of register addresses within the PIC microcontroller, ie. meaningful names can be used instead of the numerical values.

eg. CLRF RTCC will clear all the bits in the timer counter register (which has the physical address of 1, or 0001 hex).

The instruction CLRF 1 would do the same, but the use of the label RTCC makes the instruction more readable.

Some of the equates refer to individual bits within a particular register.

eg. CARRY EQU 4 refers to the fourth bit in the status register.

\[ \begin{array}{c}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\text{Carry} \\
\text{bit} \\
\end{array} \]

The name CARRY only has meaning when referring to the status register.

eg. BCF STATUS,CARRY will clear the carry bit (bit 4) in the status register,
A program will typically also include equate directives for the I/O registers.

porta EQU 5 ; port A is at the address 5, or 00 0000 0000 0101
PA0 EQU 0 ; these labels simply provide names for each bit
PA1 EQU 1 ; of port A
PA2 EQU 2
PA3 EQU 3
PA4 EQU 4
PA5 EQU 5
PA6 EQU 6
PA7 EQU 7
portb EQU 6 ; port B is at the address 6, or 00 0000 0000 0110
PB0 EQU 0 ; these labels simply provide names for each bit
PB1 EQU 1 ; of port B
PB2 EQU 2
PB3 EQU 3
PB4 EQU 4
PB5 EQU 5
PB6 EQU 6
PB7 EQU 7
More of these register labels can be declared for other PIC registers.

We can also declare memory equates such as

    ORG 00h ; This defines the start address of a program
    GOTO init ; This directs to the program start
    init
    ; Your program goes in here
    END ; Defines the end of the program so the assembler knows where to stop

So a complete program structure would look like this:
RTCC EQU 1 ; the timer counter
PLC  EQU 2   ; the program counter
STATUS EQU 3 ; the status register
CARRY EQU 4 ; the carry bit (in the status register)
DCARRY EQU 1 ; the digit carry bit (in the status register)
W   EQU 0 ; result destination to W work register
F   EQU 1 ; result destination to F (file) register
Z   EQU 2 ; the zero bit (in the status register).

porta EQU 5 ; port A is at the address 5, or
PA0  EQU 0 00 00000000 0101
PA1  EQU 1 ; these labels simply provide names for each bit
PA2  EQU 2 ; of port A
PA3  EQU 3
PA4  EQU 4
PA5  EQU 5
PA6  EQU 6
PA7  EQU 7

portb EQU 6 ; port B is at the address 6, or
PB0  EQU 0 00 00000000 0110
PB1  EQU 1 ; these labels simply provide names for each bit
PB2  EQU 2 ; of port B
PB3  EQU 3
PB4  EQU 4
PB5  EQU 5
PB6  EQU 6
PB7  EQU 7

ORG 00h ; This defines the start address of a program
GOTO init ; This directs to the program start

init ; ***** The subroutines start here *****

; **** the main program starts here *****

; Your program goes in here

END ; Defines the end of the program so the assembler knows where to stop

To avoid having to enter all these equate directives each time a new program is started, it is convenient to place them in a separate file and include them into the new program using the assembler command include

eg. include "picreg.h"
In this example, the register equate directives have been placed in a file called picreg.h so that they can be included into the program thus:

```
include "picreg.h"

ORG 00h ; This defines the start address of a program
GOTO init ; This directs to the program start

; **** The subroutines start here ****
; **** the main program starts here *****

init

; Your program goes in here

END ; Defines the end of the program so the assembler knows where to stop
```

---

3.6 The Reset Vectors

All microprocessors (and microcontrollers) need a reset vector. This is a special memory location (in ROM memory space) that is used to contain the first memory location of a program. Without it the microcontroller would not know where to start program execution.

The PIC devices use the following memory locations for the reset vectors:
PIC Reset Vectors

<table>
<thead>
<tr>
<th>PIC</th>
<th>Vector</th>
<th>These memory locations are</th>
</tr>
</thead>
<tbody>
<tr>
<td>16C54</td>
<td>1FFh</td>
<td>all held in ROM or EPROM</td>
</tr>
<tr>
<td>16C55</td>
<td>1FFh</td>
<td>or EEPROM memory</td>
</tr>
<tr>
<td>16C56</td>
<td>3FFh</td>
<td></td>
</tr>
<tr>
<td>16C57</td>
<td>7FFh</td>
<td></td>
</tr>
<tr>
<td>16C58</td>
<td>7FFh</td>
<td></td>
</tr>
<tr>
<td>16Cxx</td>
<td>000h</td>
<td>replace with appropriate</td>
</tr>
<tr>
<td>17Cxx</td>
<td>0000h</td>
<td>reset vector.</td>
</tr>
</tbody>
</table>

To use:

```
ORG 1FFh ; the reset vector
GOTO init ; GOTO to direct program to
            start of program
```

You can see now how the assembler directive command ORG is able to direct
the PIC microcontroller to the start of the user program. The start address of the
program is placed into the reset vector locations. Thus for a PIC16C54:

```
1FFh init where init is the physical start address of the user
    program.
```

Each PIC device has its own reset vector. The PIC16c5x series place the reset
vector at the top of the available ROM, which can be 1FFh, 3FFh, or 7FFh,
depending upon the amount of ROM memory available for the particular PIC
type. The PIC17cxx and PIC17Cxx series place the reset vector at the bottom of
the memory map (in a ROM area). (This change in position of the reset vector
might be to standardise the position of the reset vector in the memory map of
later devices produced by Microchip).

3.7 The PIC Instruction Set
We will now look at some of the instructions used by the PIC microcontroller.
The PIC instruction set falls into three groups:

- Bit operations
- Byte operations
- Literal/Control operations
Most of these instructions execute in a single clock cycle. The duration of a single clock cycle depends on the frequency of the clock oscillator which is usually between 4MHz and 20MHz, depending upon the PIC device.

To keep EM levels low most general purpose applications will use a 4MHz clock and the execution time is therefore 1µS per instruction.

Each memory cycle is made up of four states so the duration of a clock cycle (and hence the instruction execution time) is four times the oscillator period.

If the oscillator frequency is 4MHz, the execution time will be 1µS per instruction
If the oscillator frequency is 20MHz, the execution time will be 200nS per instruction.

A complete breakdown of all the PIC16c84 instructions is to be found in the PIC16c84 Data Sheet.

Figure 3.2, on page 10 of the data sheet, illustrates the relationship between the oscillator clock and the fetch and execute cycles of the PIC.

3.8 The Instruction Set and its Use
The following section looks at the PIC instruction set, giving examples of how the instructions are used.

3.8.1 Bit Operations
One of the principal requirements of a microcontroller is to control the external environment via the digital I/O ports. Indeed, the first microcontrollers (Intel 8048/8031, Motorola 6801) were basically generic microprocessors with integrated digital I/O and a timer module.

Controlling digital I/O often involves turning individual bits on and off, taking care not to affect any other I/O bits. Traditionally this has been achieved by using ANDing and ORing mask operations, but most modern microcontrollers now provide special instructions to selectively set or reset individual bits of a selected port.

The PIC has two instructions, one to turn on a selected bit of a port, the other to turn it off.

BSF  f,b used to turn on (set) bit b of file register f.  
eg. BSF portb,5  will set bit 5 of portB

BCF  f,b used to turn off (clear) bit b of file register f.  
eg. BCF portb,5  will reset bit 5 of portB.
Note that there is no instruction for turning on or off a group of bits. To do this it is necessary to set or reset each bit in turn or use a mask.

For example, assuming that all bits of portB start at zero, the effect of three such operators in sequence is shown below:

```
7654  3210  port bits
BSF  portb,0  0000 0001
BSF  portb,1  0000 0011
BCF  portb,0  0000 0010
```

---

Branching Instructions

All microcontrollers need an instruction to jump out of the program sequence. The instruction to do this on the PIC is the GOTO instruction.

```
GOTO K
```
causes the program to jump to the label/address k.

eg. GOTO 0050 causes the program to go to program memory address 0050 hex

GOTO instructions cause the program to branch to another part of the program. They are often used to cause the program to loop back to repeat an earlier section of code.

For example, the following program code will turn on and off bit 5 of port A in an endless loop:

```
agn BSF porta,5 ;turn on bit 5 of portA
BCF porta,5 ;turn off bit 5 of portA
GOTO agn ;go back and do it again
```

A typical PIC application may involve testing an input bit and performing some action when the bit changes state. For example the input could be the state of a door switch and the action could be to turn on a relay which then switches on a motor.

The PIC provides two instructions, Bit Test and Skip if Set (BTFSS) and Bit Test and Skip if Clear (BTFSC).

```
BTFSS f,b  Test bit b in file register f and skip the next instruction if it is set (ie.1)
BTFSC f,b  Test bit b in file register f and skip the next instruction if it is clear (ie.0)
```
Now consider the following example:

```
agn    BSF    portb,0  ;set bit 0 of portB (the LED will turn off)
   BCF    portb,0  ;clear bit 0 of portB (the LED will turn on)
   BTFSC   porta,2  ;test bit 2 of portA and skip the next instruction if S1 closed
   GOTO    agn    ;branch back to label agn
   BCF    portb,1  ;clear bit 1 of portB and so energise the relay
```

This program will cause bit 0 of portB to pulse on and off while input bit 2 of portA is a 1, ie. while the switch S1 is open. As soon as bit 2 of portA becomes a 0 (when the switch is closed), the program will clear bit 1 of portB and so energise the relay.

---

**Delay Routine**

In the program above the LED will be turned on and off so rapidly that the flashing action will not be noticed - the LED will appear to be lit at half power.

The time to execute an instruction is 1 clock cycle except for the BTFSS/BTFSC instructions which have an additional clock cycle if a skip is involved. Therefore, the number clock cycles for a traverse around the loop of the program is:

\[ 1 + 1 + 1 + 1 = 4 \text{ clock cycles} \]
Assuming a 4MHz crystal for the PIC clock, the clock cycle is 1µs (oscillator frequency /4). The time to complete the program loop once is 4µs. The LED will be off for 1µs and on for 3µs and the flash rate will be 250kHz. Clearly, we will not be able to see the LED flashing.

A delay between turning the LED on and off, and another between turning it off and on again will allow the LED flashing to be observed. We will look at how to produce a delay later, but for now assume that a delay routine has been produced as a sub-program. This can be invoked using the instruction CALL delay, where delay is the name of the sub-program.

Our program now becomes:

```asm
agn BSF portb,0 ;set bit 0 of portB (the LED will turn off)
   CALL delay ;call the delay sub-program to introduce a delay
   BCF portb,0 ;clear bit 0 of portB (the LED will turn on)
   CALL delay ;call the delay sub-program to introduce another delay
   BTFSC porta,2 ;test bit 2 of portA and skip the next instruction if S1 closed
   GOTO agn ;branch back to label agn
   BCF portb,1 ;clear bit 1 of portB and so energise the relay
```

The delay sub-program will simply take up processor time to ensure the LED remains on or off for a reasonable amount of time.

---

**Exercise 1**

Produce a fragment of assembly program to turn on the relay when switch S1 is operated twice.
3.8.2 Using the W Register

The W register (working register) is particularly important as it is used to obtain literal data values and pass them to other registers in the file register.

NOTE: A literal is simply a data value (not an address).

Two instructions used to do this are:

```
MOVLW    n    move (copy) the value n into the W register.

eg. MOVLW  36h  will copy the literal 36 hex (0011 0110) into the W register.

MOVWF    f    move (copy) the contents of the W register into register f

eg. MOVWF portb will copy the contents of the W register into portB (assuming that portB has been set up to be an output port).
```

So now we have a way to place literal data into a selected register.

```
eg. MOVLW  12h
MOVWF  OPTION
```

moves the literal 12 hex into the OPTION register via the W register.

3.8.3 Port Configuration

The ports of the PIC (and indeed of most microcontrollers) can be configured as input or output ports. In fact the individual port bits can be configured as input or output bits independently.
The above diagram illustrates how the PIC portA can be configured for input or output operation.

The TRISA register is located at address 85h in the file register map and is used to configure portA for input or output by placing a 1 or a 0 into the bits of the TRIS register.

Placing the bit pattern 0110 1110 into the TRISA register would configure portA as illustrated below.

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Bits 7, 4, and 0 are configured as outputs, whilst bits 6, 5, 3, 2, and 1 are inputs.

The PIC instructions to do this would be

```
Config BSF STATUS,5 ; set bit 5 of STATUS
MOVELW 6Eh  ; copy literal 6E hex to W register
MOVWF TRISA ; copy contents of W register into TRISA
BCF STATUS,5 ; clear bit 5 of STATUS register
```

This code needs some explanation.

The default register file is called page 0. The TRISA and TRISB registers are located in the second bank of register files known as page 1. The figure below illustrates the arrangement of the two banks of memory register files. Note that in register bank 0 address 05h accesses porta, while the configuration register TRISA is in register file bank 1 at address 85h.
To gain access to register bank 1 we need to set bit 5 of the STATUS register to 1. Bit 5 is the *register file page select* bit. This is most easily done by using the bit set instruction:

BSF STATUS,5

To gain access to the default register file bank 0, we simply clear bit 5 in the status register with:

BCF STATUS,5

You will notice that some registers (such as the STATUS register) are duplicated in both register banks.

The most commonly used bits in the status register are illustrated below.

<table>
<thead>
<tr>
<th>Bank 0</th>
<th>Bank 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>Indirect Reg</td>
</tr>
<tr>
<td>01h</td>
<td>TMR0</td>
</tr>
<tr>
<td>02h</td>
<td>PCL</td>
</tr>
<tr>
<td>03h</td>
<td>STATUS</td>
</tr>
<tr>
<td>04h</td>
<td>FSR</td>
</tr>
<tr>
<td>05h</td>
<td>PORTA</td>
</tr>
<tr>
<td>06h</td>
<td>PORTB</td>
</tr>
<tr>
<td>08h</td>
<td>EEADATA</td>
</tr>
<tr>
<td>09h</td>
<td>EEADR</td>
</tr>
<tr>
<td>0Ah</td>
<td>PCLATH</td>
</tr>
<tr>
<td>0Bh</td>
<td>INTCON</td>
</tr>
<tr>
<td>0Ch</td>
<td>General purpose RAM</td>
</tr>
<tr>
<td>7Fh</td>
<td>Mapped Access in Bank 0</td>
</tr>
</tbody>
</table>

To gain access to register bank 1 we need to set bit 5 of the STATUS register to 1. Bit 5 is the *register file page select* bit. This is most easily done by using the bit set instruction:

BSF STATUS,5

To gain access to the default register file bank 0, we simply clear bit 5 in the status register with:

BCF STATUS,5

You will notice that some registers (such as the STATUS register) are duplicated in both register banks.

The most commonly used bits in the status register are illustrated below.

<table>
<thead>
<tr>
<th>7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRP</td>
</tr>
</tbody>
</table>

Where:

IRP is the *indirect address select* bit
RP0 is the *register file page select* bit (as discussed above).
TO is the *Timer Out* bit
Z is the *Zero flag*, set to 1 by any operation resulting in a zero
C is the *Carry flag*, set to 1 by any carry bit generation in an addition operation.
Exercise 2
Try producing a segment of PIC assembly program to configure portA as an input port and portB as an output port.

The TRISB register is located at 86h in the file register map.

3.8.4 Some other PIC Instructions
The following PIC instructions are commonly used:

**CLR**
CleaR the **W** register. ie. set all its bits to 0.

**CLRF** f
CleaR the **File register f** to zero.
This last instruction is useful for setting all the bits in a particular register to zero.

eg. **CLRF** OPTION would set all bits of the OPTION register to zero.

Another use:

**CLRF** portb would clear all bits of portB (assuming that it has been configured as an output).
Note that a CLRW or a CLRF instruction will set the Z bit in the STATUS register.

---

### 3.8.5 Bit Setting and Clearing by Masking

The logical AND and OR operators can be used to clear or set bits according to the bit pattern specified as a literal. The literal is known as a *mask*.

Logical anding is used to clear selected bits in a byte to zero.

#### ANDLW

The value `n` is logically ANDed to the contents of the W register and the result placed into the W register.

Example:

```
ANDLW 7Bh would logically AND the contents of W with literal 7Bh
```

<table>
<thead>
<tr>
<th>W</th>
<th>1001 1101</th>
<th>the contents of W (assumed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>0111 1011</td>
<td>the literal value 7Bh (the mask)</td>
</tr>
<tr>
<td>Result</td>
<td>0001 1001</td>
<td>bits 7 and 2 have been cleared by the AND operation</td>
</tr>
</tbody>
</table>

---

**Exercise 3**

Which bits are cleared in the W register with the following instructions?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVLW</td>
<td>DBh</td>
</tr>
<tr>
<td>ANDLW</td>
<td>ACh</td>
</tr>
</tbody>
</table>

Logical ORing is used to set selected bits in a byte to 1.

```
IORLW n Logically inclusive OR n with the contents of W.
```
Example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IORLW</td>
<td>7Bh</td>
<td>would logically OR the contents of W with literal 7Bh</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>1001 1001</td>
<td>the contents of W (assumed)</td>
</tr>
<tr>
<td>N</td>
<td>0111 1011</td>
<td>the literal value 7Bh (the mask)</td>
</tr>
<tr>
<td>Result</td>
<td>1111 1011</td>
<td>bits 1, 5 and 6 have been set by the OR operation.</td>
</tr>
</tbody>
</table>

Exercise 4

Which bits are set in the W register with the following instructions?

```assembly
MOVLW 85h
IORW 22h
```

This method of clearing and setting bits is sometimes known as masking. The literal used in both the ANDLW and the IORLW instructions is the mask byte.

We could use this idea to set or clear multiple bits of an output port.

Example:

```assembly
MOVLW 85h
MOVWF portb ; output 1000 0101 to portB
IORLW 22h    ; now do it
MOVWF portb ; set bits 1 and 5 to logic 1
ANDLW EEh    ; in portB
MOVWF portb ; now clear bit 0
```

; in portB.
3.8.6 Instructions Using the Destination Flag Bit

Many of the PIC instructions can place their result into either the W register (as in previous examples) or into one of the selected file registers. The destination is determined by the value of d (destination select).

\[ d=0 \quad \text{destination W register} \]

\[ d=1 \quad \text{destination file register f} \]

(this is the default)

eg.

\[
\text{MOVF} \quad f, d \quad \text{move/copy the contents of the file register f into the destination d}
\]

Example

MOV\text{LW} \quad \text{FCh}
MOV\text{WF} \quad 35h \quad ; \text{move literal FC hex into location 35h}
IN\text{CF} \quad 35h, 1 \quad ; \text{increment contents of memory location 35h and store it there}
MOV\text{F} \quad 35h, 0 \quad ; \text{move contents of 35 hex into W register}

The second instruction moves the contents of W register (the FCh literal) into the memory location 35 hex. The operand 35h is a free memory location in register page 0 (see the register file memory map we looked at earlier; it's also on page 12 of the PIC16C84 data sheet). The third instruction increments (adds one to) the contents of memory location 35h and stores it in 35h again. The final instruction moves (copies) the contents of memory location 35 into the W register (overwriting the value FC placed there by the first instruction). Note that in the last instruction, bit \( d=0 \) indicating the destination as the W register.

It is good programming practice to use an equate to give the \( d \) bit a more meaningful name.

\[
\text{W} \quad \text{EQU} \quad 0
\]

\[
\text{F} \quad \text{EQU} \quad 1
\]

So that the use of this type of instructions used in the above example becomes more readable:

\[
\text{MOVLW} \quad \text{FCh}
\text{MOVWF} \quad 35h \quad ; \text{move literal FC hex into location 35h}
\text{INCF} \quad 35h, 1 \quad ; \text{increment contents of memory location 35h and store it there}
\text{MOVF} \quad 35h, 0 \quad ; \text{move contents of 35 hex into W register}
\]
The following instructions all use the d bit to indicate their destination (d=0 W register, d=1 file register):

\[
\begin{align*}
\text{ADDWF} & \quad f,d \\
\text{IORWF} & \quad f,d \\
\text{ANDWF} & \quad f,d \\
\text{XORWF} & \quad f,d \\
\text{ADDWF} & \quad f,d \\
\text{SUBWF} & \quad f,d \\
\text{COMF} & \quad f,d \\
\text{INCF} & \quad f,d \\
\text{DECF} & \quad f,d \\
\text{RRF} & \quad f,d \\
\text{RLF} & \quad f,d \\
\text{SWAPF} & \quad f,d
\end{align*}
\]

Look up these instructions in the PIC16C84 data sheet.

---

3.8.7 The Auto-decrement and Auto-increment Instructions

There are two instructions that can be used to create loops that execute for a particular number of times.

\[
\begin{align*}
\text{DECFSZ} & \quad f,d \quad ; \text{decrement register f and skip the next instruction if zero} \\
\text{INCFSZ} & \quad f,d \quad ; \text{increment register f and skip the next instruction if zero}
\end{align*}
\]

Example

```
DECFSZ count
next GOTO next
BSF portb,1
```

The first instruction decrements the contents of memory location count. If it is not equal to zero, the GOTO instruction will be executed. This of course causes the program to loop back to the DECFSZ instruction to decrement count once more.

This will continue until the DECFSZ instruction decrements count to zero, at which point the DECFSZ instruction will skip the GOTO instruction and execute the next instruction (in this case setting bit 1 of portB).

The two instructions DECFSZ and GOTO take up processor cycles and thus constitute a "delay".

The DECFSZ instruction can be used to keep a count of some event taking place, for example we may wish to keep a count of the number of times an input changes from 1 to 0.
The INCFSZ instruction behaves in much the same way, but adds one each time it is executed.

### 3.8.8 The Purpose of Sub-programs
A sub-program (or subroutine as it is sometimes called) is a piece of program that can be called from anywhere from within a program by using the instruction

**CALL name_of_sub_program**

We have already seen this idea before with the delay sub-program.

**CALL delay**

The program instructions comprising the delay sub-program are:

```
delay MOVWLW FFh
      MOVWF count,F
next   DECFSZ count
        GOTO next
      RETURN
```

The sub-program needs to start with a label (delay) and end with the instruction RETURN.

When the CALL instruction calls sub-program, the program branches off to the group of instructions following the label delay.

On completing the sub-program (ie. on reaching the instruction RETURN) the program branches back to the next instruction in the main program following the CALL instruction.

In order to know where to return from a sub-program the PIC microcontroller must know where to branch back to in the main program. This is achieved by storing the next instruction address after the CALL instruction in a special area of memory called the STACK.

The RETURN instruction will look at the stack and obtain the return address which it will load into the PIC's program counter (which keeps track of where the next instruction is to be executed in program memory).

The next diagram illustrates the STACK mechanism used in the PIC 16Cxx series.
3.8.9 The Stack and its use

The CALL instruction (ignore word Interrupt in the diagram for now) places the return address onto the stack and the RETURN instruction gets it back off the STACK and places it in the Program Counter register (PC). These actions all take place without any further action being taken by the programmer.

The complete program containing both the main program sequence and the sub-program looks like this:

agn  BSF   portb,0  ;set bit 0 of portB (the LED will turn off)
CALL  delay   ;call the delay sub-program to introduce a delay
BCF   portb,0  ;clear bit 0 of portB (the LED will turn on)
CALL  delay   ;call the delay sub-program to introduce another delay
BTFSC porta,2 ;test bit 2 of portA and skip the next instruction if S1 closed
GOTO  agn     ;branch back to label agn
BCF   portb,1  ;clear bit 1 of portB and so energise the relay

delay  MOVLW FFh
       MOVWF count,F
next   DECFSZ count
       GOTO next
       RETURN

Note: The delay produced by this simple sub-program is not very long and something a little more complex would be used in practice.

3.8.10 Interrupts

When looking at the diagram for the stack we saw that the stack was also used for interrupts. What is an interrupt?

There are situations where we wish to take some kind of action whenever a particular event occurs, but in order not to waste time, we do not want to keep looking for the event to happen. A very good example is when you use the keyboard of your computer.
The PC only looks at the keyboard to discover which key has been pressed after it happens - pressing a key generates an interrupt.

When an interrupt occurs, the processor is automatically directed to a particular part of memory called an interrupt vector where it finds the address of the part of program to be executed in response to the interrupt service routine (ISR).

The arrangement is illustrated below.

In the case of the keyboard interrupt we have:

1. A key is pressed, generating an interrupt.
2. This causes the processor to store the next instruction address on the stack.
3. The processor then goes to the interrupt vector (each interrupt input has its own interrupt vector).
4. The interrupt vector contains the address of the interrupt service routine (ISR) which is loaded into the program counter so that the next instruction to be executed will be the first instruction in the ISR.
5. The ISR will contain instructions to read the keyboard data and place it into data memory to be used by the main program, or perhaps act on the key operation there and then. The last instruction in the ISR will be a RETURN instruction that will obtain the return address (the address of the next instruction to be executed in the main program) and place it into the program counter.
6. The next instruction in the main program will then be executed.
The advantage of using an interrupt is that the interrupt source (e.g. the pressing of a key) can occur at any time. The processor just completes the current instruction that it is executing, then performs the isr, and returns to execute the next instruction as if nothing had happened. This means that when writing the main program we do not have to worry about when the interrupt will occur since it is taken care of by the interrupt mechanism.

An important aspect of interrupts is that they are serviced straight away rather than when the processor happens to notice (as would be the case if the processor kept polling the source waiting for the event to occur).

Most microcontrollers are required to work in an embedded real time environment. Therefore they are provided with interrupt capability; usually an input pin that can be connected to whatever is to cause the interrupt. All the programmer has to do is:

1. Enable the interrupt (by setting an interrupt enable bit in a control register).
2. Place the address of the isr into the interrupt vector.
3. Write the required isr program code to be placed at the isr address (using the ORG assembly language directive)

Look at the PIC16C84 data sheets and discover:

1. What is the interrupt vector address?
2. Where is the interrupt pin?
3. Where is the interrupt enable bit?
4. What is the instruction for returning from an interrupt?

Click on the button below to go to the Interactive Self Assessment

Self Assessment

3.9 Practical Work With MPLAB Software

These four practical sessions involve using the Microchip PIC MPLAB programming environment to enter, assemble and run a number of assembly language programs. They will also help you to become familiar with such things as single stepping through the program, setting breakpoints, making use of the simulator and using a stimulus file.
Happy programming!

**Practical Session 1**: Introduction to the MPLAB Development Environment

**Practical Session 2**: Simulation of Input/Output

**Practical Session 3**: Programming a Device

**Practical Session 4**: Setting Breakpoints and using the Trace Feature

When you have completed the practical work, try the programming exercise below.

---

### 3.10 Programming Exercise

Use the MPLAB development software to develop a program to output a binary count sequence on portB after pressing a switch on portA bit RA0.
Microcontrollers
Downloading Inprise/Borland Turbo C Compiler

The Inprise/Borland website is constantly changing so you may need to snoop around a bit to find the C compiler. You should be looking for the History zone which is usually accessed from the Inprise/Borland home page. At the time of writing, this method worked:

2. Click on C++ Builder
3. Click on History and fill in the form for access. After a few hours at most you should receive an email giving a password to access the rest of the site. Meanwhile, you can download a different compiler: See item 5. below.
4. Access the History Zone again, this time with your password, and locate the Turbo C (or Turbo C++) IDE for download. Download the file and follow the installation instructions.
5. Alternative compiler avoiding the wait. Click on Compiler (to download). This compiler is suitable for practicing programming in C. (A tutorial for Borland C++ Builder version 3.0 is available from the tutor or via the AMI Office if required).

If you have Windows XP home edition you might not be able to run the DOS version – in which case you will have to use the windows version and use the help information to find out how to use it.

If you use Windows 98, or Windows 2000 you should be able to access the DOS prompt (i.e., run 16 bit applications) and hence run DOS versions of Turbo C/ or Turbo C++. The advantage of running the DOS versions is that you can access the PC’s Input/Output.
Microcontrollers
Testing the Matrix Multimedia PIC Development Board
- a walkthrough of the tools

Introduction

The Matrix Multimedia PIC development kits are first required towards the end of Chapter 3 of the module, in Practical Experiment 3.

The kits are ready to be posted from Bolton at the start of each module. However, they are too large to fit through a letter box and to avoid causing unnecessary inconvenience or possible loss, we ask that you to tell us an appropriate address for delivery: home, work or perhaps a neighbour or relative. If you are enrolled on the module and have not already done so, please could you email this information to Debbie Higham on d.higham@bolton.ac.uk.

A Matrix MultiMedia (MMM) PIC development kit is issued on loan to each student studying the module. The kit comprises four items:

- the MMM PIC development board
- cable to connect the development board to the printer port of a PC
- a mains power supply
- a CDROM "C for PICmicro Microcontrollers".

Please check everything on arrival and report any problems to the AMI Office. Students based outside the UK should check that they have received the correct adapter and international version of the power supply.

The CD-Rom contains a C programming environment and the software to program the PIC. Before installing the software, read the README file. Then click on the "install C4CPICs" executable to install the software, choosing the FULL installation rather than the MINIMAL option. The executables also include MPLAB assembler, but not the MPLAB programming environment or the simulator needed for this course.

Once the software has been installed, click on "Start", then on "Programs", then "C for PIC Micros", then select "PIC Micro Development Board Documentation" and "Open".

This is a pdf file which can be read using the Adobe Acrobat reader (downloadable free from www.adobe.com). It is advisable to print the contents of the file as it provides useful information about the MMM board. Section 6 describes how to test the MMM board.
The board will already have a test program in memory and will automatically run this application on power up. You should see all the Port B LEDs flash on and off at an approximately one second intervals. If your MMM board does this, you can assume it is working OK.

To up load a program to the MMM board use the following procedure.

First ensure that the PIC is going to be configured correctly.
a) select the PIC16C84 as the PIC to be programmed.
b) select oscillator XT
c) select watchdog timer OFF
d) select power up timer OFF
e) select code protect OFF

To do this click on "Start" then "Programs" then within the menu select "Matrix Multimedia", then "C for PICMicros", and finally select "Config the PICMicro".

The following window should appear.

Make sure that your window looks like the one shown here. When you are happy with this, click on OK to select your choice.

Now click on "Start", then "Programs", then within the menu select "Matrix Multimedia", then "C for PICMicros", and finally select "C2C Compiler". The following window should appear.
To test the programming aspect of the MMM board I have created a very simple test program called io.asm. The program is reproduced below and you will need to type it into the C2C IDE environment. Click on the "FILE" dropdown menu item, then select "New". Your screen should look something like this:

Now type in the program carefully. You can ignore lines beginning with a semicolon - they are just comments and are ignored by the assembler.
; Test program io.asm
; Test Program to check MMM PIC Micro board.
; A very simple test to prove that the board can be programmed
; and a hex program run.

list p=16f84

; Program Equates:
pc equ 0x02
status equ 0x03
porta equ 0x05
portb equ 0x06
trisa equ 0x05
trisb equ 0x06
rp0 equ 5
w equ 0
f equ 1
RA3 equ 3
RA0 equ 0

; Main Program
org 0x00 ;reset vector

goto init

init bsf status,rp0 ;select page 1
movlw 0x00 ;set portb all output
movwf trisb
movlw 0xff ; set porta all input.
movwf trisa
bcf status,rp0 ;switch back to page 0
movlw 0xff
movwf portb ;reset the portb

; now for the main task!

zero btfs porta,3

goto zero ;read switch LA3 of porta till 1
bsf portb,7 ;turn LB7 on when porta switch LA3 pressed.

one btfs porta,3

goto one ;now read switch LA3 of porta till 0
bcf portb,7 ;turn LB7 off when porta switch LA3 released.
goto zero ;repeat the whole thing forever

END

After typing the program into the C2C environment, you need to assemble it. Click on
the "A" speed button to Assemble the program (see window below).
Now select the "Compile" drop-down menu item, then "Assemble". (Alternatively, click on the "A" [Assemble] button, but note that you need to click on the program file before the assembler will work).

The C2C environment will assemble the program and generate a list file (.lst) containing both your source and binary executable code. The Output window should display "No errors".

Now close the Output window and the List file window.

Make sure that the MMM board is connected to the power supply and switched on, and that the cable supplied with the MMM board is plugged into the printer port of the PC and the MMM board.
Now click on the "Program" button (the one with the microchip symbol, next to the "Assemble" button. The .hex executable file will be programmed into the PIC and the program should now be ready to run. At this point all the LEDs connected to the PortB should be turned on.

Pressing SA3 switch on and off should cause the Port B LED LB7 to flash on and off.

Pressing the Reset button will reset the program (but you will not see anything in this case).

You have now tested the MMM board for both programming and running an application program.

Note: An alternative way of programming the MMM PIC micro board is to use the PPP programmer. This can be found in the folder Program Folder/ Matrix Multimedia/Common/PPP. (It's labelled PPP).

When you double-click on the PPP icon the following screen will appear.

Click on the "File" drop-down menu item and select "Open". Now navigate your way to the hex file, io.hex. This is what it looks like:
Note that there are a number of test file in the folder Test Files (see below) which you can experiment with in order to try out other parts of the MMM PIC development board.

This concludes the walkthrough for using the MMM PIC development board.