2.1. Discrete Logic

By selecting appropriate ICs, it is possible to construct a circuit to achieve any function or functions required. This is the original way of working with digital circuits.

Benefits

There are a number of potential benefits in using discrete logic to implement a design:

- Any function can be achieved. By selecting the appropriate combination of logic ICs any required function is achievable.
- Parallel processing is possible. Systems can be designed which achieve one task at a time, or achieve many tasks in parallel.
- Design methods exist to enable a structured approach to the design process. Because discrete design originated in the days before computer-aided design, many of the design processes developed are designed to be implemented manually. This includes the use of Karnaugh maps which we looked at in P21389, and the structured approach to the design of what are called *finite state machines* which we will look at in the next few weeks.

Issues

The biggest issue with a discrete logic implementation is that many of the ICs contain reasonably simple, basic, functions. As a result, a large number of ICs are likely to be required in order to implement more complex functions. ICs need to be a certain size if they are to be hand-soldered. This is because we, as humans, can only work with components which are a certain size. If they are too small, we cannot see them clearly, or hold them. It is also a strange function of the manufacturing process that unless a highly complex IC is being produced the cost of producing a complex IC is not much more than producing a simple IC. Any costs associated with producing the case and transportation will be almost the same. As a result, a solution using many simple ICs will be much more expensive to implement than a solution using a single IC which can do all that is required.

Using many ICs in the solution makes the design of the *printed circuit board* much more complex because a large number of connections will be needed between the various ICs. Achieving complex interconnections inside an IC is generally easier than on a PCB.

A final issue, is the size of the final solution. Most of you will not be old enough to remember the earliest mobile phones, when the phone was the size of a suit-case, and comparing that with a modern phone which can do a huge number of other things in addition to making calls. This is as a result of miniaturisation, using highly complex ICs which have very small package sizes. Discrete logic solutions will necessarily be much larger than a highly integrated solution.

2.2. Microcontrollers

As we started to see in P21389 last year, and will see further this year, a large number of different microcontrollers exist, which can be used and selected according to the requirements of the system into which it is being installed.

The circuit inside a microcontroller is fixed by the chip designers, but is designed to be highly flexible. Much of this flexibility is achieved by the inclusion of a large number of *special function registers* which are designed to allow the operation of various subsystems to modified by loading ones or zeros into specific SFR locations. This allows subsystems to be switched on or off, or for the operation of those systems to be modified.

In addition, the central processing unit is designed to be highly flexible allowing a wide range of different functions to be achieved.

Microcontroller processing units only allow one thing to be done at a time. Microcontrollers mimic achieving many processes in parallel by the use of *time division multiplexing*. This is where several processes are handled simultaneously by processing a command on process 1, storing that in memory whilst a command on process 2 is processed. The processor will work through as many processes are a required in order to make it seem like all are being processed in parallel. The key to this is that the processor must run fast enough that the time delay to get round all the processes is small enough that it is not really noticeable.

2.3. Programmable Hardware

There is a third way which sits between discrete logic and software-based processors, that is programmable hardware.

Here, the IC itself is reconfigurable or freely designable by the system designers. Three different levels of configurability exist, depending on the size and complexity of the implementation required.

Programmable Logic Devices.

Programmable logic devices were the first of the various programmable hardware devices. A number of different variations have been developed over time, some as proprietary names for a similar solution. These include:

- Programmable Logic Device (PLD)
- Programmable Array Logic (PAL)
- Generic Array Logic (GAL), and more recently,
- Complex Programmable Logic Device (cPLD)

You will remember from P21389 that the Karnaugh map is a simple and reliable way of determining the appropriate logic implementation. In general, a Karnaugh map implementation consists of three logic layers. The first is a possible layer of inverters, the second is a layer of AND gates, and the final layer is a layer of OR gates. Such an implementation is known as a sum-of-products implementation. PLDs are designed around this basic structure, as shown in figure 1.

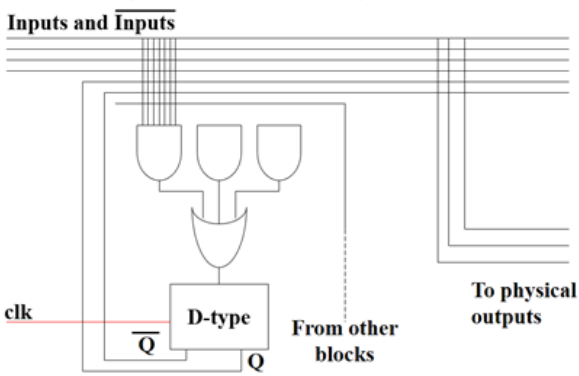

Figure 1: A Basic PLD Implementation

The PLD consists of a number of functional blocks. Each block consists of a number of AND gates routed into an OR gate. The one development from the standard Karnaugh map implementation is that a D-type latch is included after the OR gate to allow for the implementation of sequential systems.

Central to the reconfigurability of the PLD is the *crosspoint matrix*. All inputs and their complements are made available, together with all outputs from the functional blocks. These are not connected directly to the inputs to the functional blocks during manufacture, but are connected via the matrix. The required connections are then made by the user by programming the device. This will then fuse some connections and blow others. This will allow only the required connections to be routed to the inputs to the functional blocks.

A second part of the matrix increases flexibility by routing the output connections via the matrix too. This means that a functional block does not necessarily have to be connected to an output.

Early PLDs were one-time-programmable. When the device was programmed, connections in the matrix were physically fused or blown, permanently changing the internal connections in the device. Modern devices are reprogrammable, with the matrix being in the form of an EPROM storing connection information.

PLDs are normally defined by the number of inputs and outputs available, and the number of functional blocks in the IC, for example a GAL22V10 contains 22 pins, and 10 functional blocks.

Field Programmable Gate Arrays

The second form of programmable hardware device, allowing a greater level of complexity and flexibility in implementation is the field-programmable gate array, or FPGA.

The basic functional block of an FPGA is smaller, as shown in figure 2, than the PLD, and the flexibility in interconnection is greater. In addition, a much larger number of blocks may be included in the IC, up to 2 million in some cases.

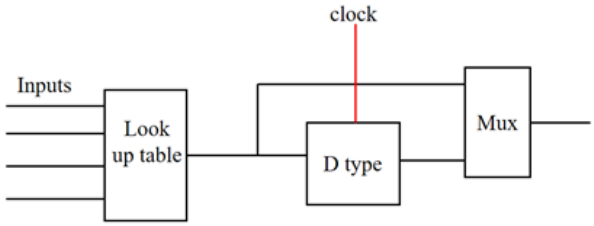

Figure 2: FPGA Functional Block.

The look-up table is a way of implementing generalised logic functionality between the inputs and the latch. Effectively, the inputs form the address of a 1-bit memory array. The memory stores ones or zeros depending on what combinations of inputs should give a one at the output.

The multiplexer (mux) at the output allows the latch to be bypassed if it is not required.

Unlike PLDs, by default the programming on an FPGA is lost when power is removed.

2.4. Application Specific Intergrated Circuits

The application-specific integrated circuit (ASIC) is the most general-purpose logic device. However, unlike PLDs and FPGAs, once an ASIC is fabricated, its design and function is fixed.

The starting point for an ASIC is a completely blank canvas. This allows the designer complete freedom to specify exactly the design you require for the system.

Complete design from scratch is known as *full custom design* and a very highly specialised task. This allows design even down to the level of specifying the sizes of individual transistors. In many cases this level of specialisation is not required, and so *semi-custom* design is used in a lot of situations. Here, basic building block are available at a number of different levels, from basic gates to more complex subsystems. The transistor sizes will have been standardised, but the actual connections and design of the circuitry is still completely open to the designer.

Benefits

Complete freedom in the design is the main benefit of an ASIC-based solution. In addition, a complex system which only requires a few inputs or outputs can be packaged in a suitably small outline saving space in the system.

Issues

ASICs are very expensive, and so their use is generally restricted to large volume situations. Once the design is produced, it is sent to a fabrication house. The design is converted into a number of masked which are *photoetched* on to the silicon wafer. Many ICs can be produced on a single wafer.

Generally speaking, there will be a one-off tooling cost for setting up the masks, and then a unit cost for each IC produced.

2.5. Designing with Programmable Hardware

The implementation of systems on programmable devices is based either around bespoke or generalised computer-based *integrated design environments* or IDEs. These systems allow for the development of the design, the conversion of the design into a file which can be used to programme the device, and, in some cases, the control of the programming phase. Certainly, in the case of ASIC design, but also for FPGA and PLD design, simulation facilities are also available. Simulation is vital in the case of ASIC design, because the entire system must be built and tested in the IDE before committing to fabrication. Any change once the design is submitted would incur a further set-up fee.

Initially, design of programmable devices was undertaken exactly as discrete design was. A schematic diagram, or a set of schematics were developed which diagrammatically showed the circuit required. The IDE would then convert that into a *JEDEC* file, which is the file format used to programme the device. For complex designs, however, it was realised that this approach was not the most efficient. Decisions made by the designer in producing the schematic may lead to a less than optimal implementation. Over about the last 30 years, the design has moved to a language-based design approach, similar to programming a microcontroller.

In theory, a computer language such as C may seem suitable for use in designing hardware too but a microcontroller has a fixed architecture and a single processing path. Computer programming languages are designed for this purpose. Programmable hardware has freedom in implementation, and the possibility of multiple parallel processing paths. As such, a hardware programming language requires the ability to handle parallel processing, and also both the functional and structural aspects of the design.

Hardware Programming

Two main programming languages have come to the fore in recent years, the first is *Verilog* and the second, VHDL. VHDL stands for VHSIC Hardware Description Language. VHSIC in turn stands for Very High-Speed Integrated circuit. VHDL is an IEEE standard 1076, originally developed in 1987 and updated at various times, most recently in 2008.

It contains constructs for specifying both the function (behaviour) and hardware structure (architecture) of the system. It also has constructs for handling a variety of physical digital inputs and outputs.

VHDL is a digital-only language, but an analogue and mixed-signal extension has been developed to provide a similar approach for analogue and mixed signal ICs.

You will learn about VHDL and FPGA design in more detail in the third year.

3. Finite State Machines

In P21389 last year, we used Karnaugh Maps to determine the logic which provided a ‘one’ at the output for a given combination (or combinations) of ones at the inputs. The value of the output or outputs of the system changed depending on the combination at the inputs.

We also saw, in the case of counter circuits, how a sequential logic circuit could step through a series of output combinations (or states) in order. In this case, the output after the next clock pulse depended on the current output value. This is not unusual. When we count, we work through the numbers in order and the next number will depend on what number we have currently got up to.

A *finite state machine* combines both of these ideas. It is a logic system where the output after the next clock pulse depends *both* on the current state *and* the current inputs to the machine. We met a simple example of such a system last year in the lab when we used the up/down counter circuit. Technically, we could class this as a state machine because when the current output is four, for example, and the direction input is ‘up’ then the next state and so the next output is five. If the direction is ‘down’ then the next output is three. So the operation and so the next state of the system is governed both by the current state, and by the current inputs.

In the same way that the Karnaugh Map is a formalised way of determining the relationship between the inputs and outputs of a combinational logic system, so there is a formal process which has been developed in the days when most systems were still designed manually, for developing and implementing a finite state machine.

In our project this year, we will be implementing the system using the PIC microcontroller, so the implementation phase will be slightly different, but we will still look at how you might implement such a system using logic, either based around D-type flip-flops or JK flip-flops.

3.1. Overview and Theory

A generalised block diagram of a state machine is shown in figure 1. It consists of two functional blocks, one of which consists of a set of memory elements to store the number of the current state of the machine, and one which consists of combinational logic, to relate the current state and current inputs to the required current output and the inputs to the memory element block (which will in turn give the next state of the machine).

In the case of a circuit like a counter, the state and the output value are the same but we cannot assume that this will always be the case, hence why the state output from the memory elements is not connected directly to the output.

Also, if we use D-type flip-flops then we know that the values at the input to the memory element block will be the next state but again, this is not always the case. If we use JK flip-flops, different input combinations could give the same next state, and so in the generalised form this needs to be considered.

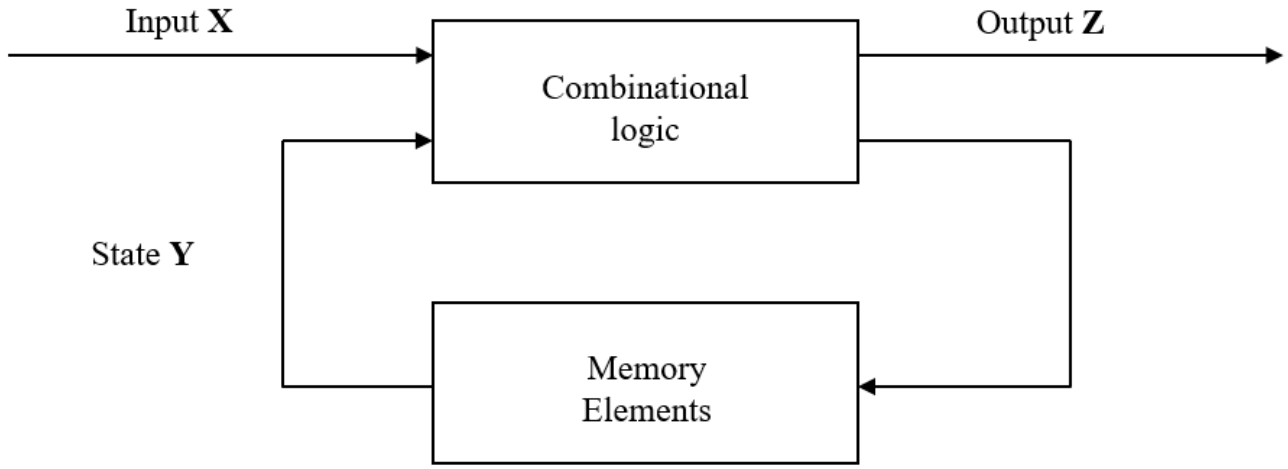

Figure 1: Generalised State Machine Block Diagram

Finally, there is the possibility, not only that the selection of the next state is affected by the current inputs, but also, that the current output values are affected both by the current inputs and the current state.

Formally we can consider all the signals shown to be logic vectors where:

$\mathbf{Y_K} = [Y_0^K, Y_1^K, \ldots, Y_n^K]$ is the current state vector.

This is a mathematical way of expressing the outputs of all the memory elements after K clock pulses, with each 'y' representing one bit of the output. The number of bits 'n' required in this vector will depend on the number of states required.

$\mathbf{X} = [x_0, x_1, \ldots, x_i]$ is the input vector.

Notice that the number of bits in the input doesn't have to be the same as the number of bits in the state hence the suffix 'i' here.

$\mathbf{Z} = [z_0, z_1, \ldots, z_m]$ is the output vector, and:

$\mathbf{Y_{K+1}} = [Y_0^{K+1}, Y_1^{K+1}, \ldots, Y_n^{K+1}]$ is the next state vector.

That is, it is the input to the memory element block to give the required next state.

We can now formally, and generally relate these vectors together:

$\mathbf{Z} = f(\mathbf{Y_K}, \mathbf{X})$ (eqn 1) and

$\mathbf{Y_{K+1}} = g(\mathbf{Y_K}, \mathbf{X})$

Given that f and g are Boolean functions.

In other words, the current output is some logic function of the current state and current inputs, and the next state is some other logic function of the current state and current inputs.

In many cases, the output might only be affected by the current state. For example, our up/down counter did not change its current output based on whether it was counting up or down, it only changed which state it went to next. In such a case the output vector can be expressed:

$\mathbf{Z} = f(\mathbf{Y_K})$ (eqn 2)

The output is a Boolean function of the current state.

Mealy and Moore Machines

This leads to the definition of two different types of state machines, the *Mealy Machine* and the *Moore Machine*. A Mealy machine is one where the output vector is defined according to equation 1, and a Moore machine is one where the output vector is defined by equation 2.

Both are valid state machine designs, and are used as appropriate depending on the requirements of the particular system.

3.2. Design Considerations

We will look in detail at the state machine design process, the various stages involved and how to achieve an overall solution in the next section. Here we are considering in general terms what we wish to achieve.

At the start of the design, we need to know how many inputs there are and what they represent. We also need to know how many outputs there are and when they need to be high and low.

From this starting point, the design process needs to answer two questions:

1. How many memory elements are required? This will depend directly on the number of states required. Since it is a binary system, it will be the next binary multiple large than the number of states, so for six states, three memory elements would be required.
2. What is the logic required in the combinational logic block to relate inputs and current state to outputs and next state?

Once these questions have been answered in general, there are certain considerations such as the type of memory element to use, which will in turn affect some of the logic, because JK flip-flops will required different logic inputs form D-types. They will also require twice as many, so that would change the size of the next state vector.

We will look at the design process for implementing a finite state machine in the next session.

3.3. Design Process Overview

So far, we have looked in general terms at what a state machine is and how it is expressed algebraically. We now need to get somewhat more practical and look at a suitable process for determining what we need the state machine to do, what interfacing logic is required, and how many memory elements are needed. The generalised block diagram of a state machine is included again in figure 1 below for quick reference.

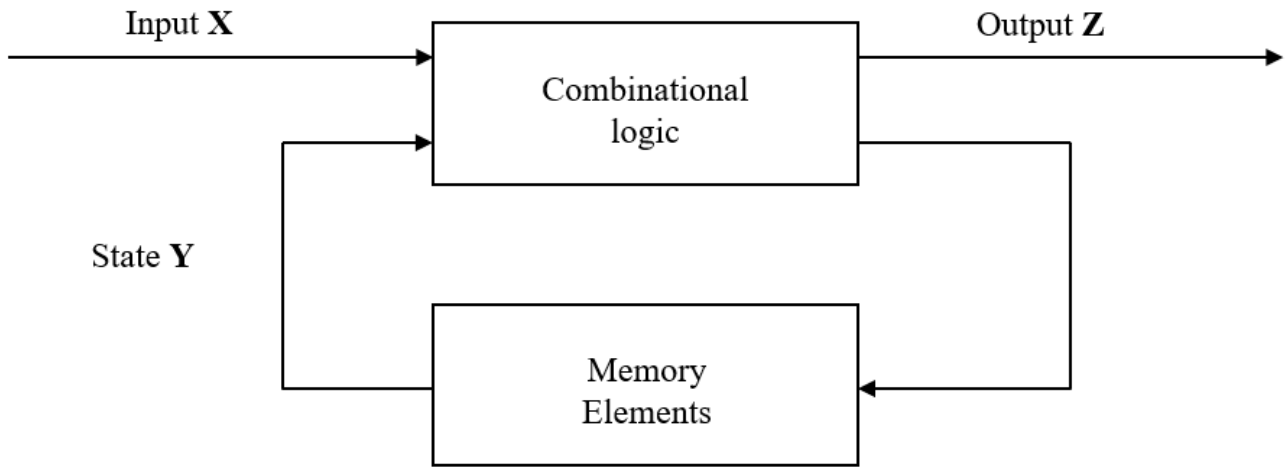

Figure 1: Generalised State Machine Block Diagram

There are a number of stages in the design and implementation of a state machine:

1. Draw a *state transition diagram*. This is a pictorial way of determining the operation of the system. At this stage, some elements such as inputs and output may be expressed in binary, but much of the system operation is just expressed in plain language.
2. Translate this into a *state transition table*.
3. Optimise the state transition table. We now need to look to see if there are any states which we've repeated (if so they can be removed). Because we know that the number of states needs to be a power of two, we need to consider what we do with any unused states. For example, if our machine needs six states, we have two free (the next power of two is eight). What outputs and what next state do we specify if the machine gets in to these states? You may think that this is unlikely, but this could happen on power up for example.
4. Assign binary values to the states. The obvious thing to do here is to allocate in order but it is worth just taking a moment to see if a different allocation might lead to a simpler logic implementation.
5. Construct a *coded state transition table*. Basically, here we repeat step three, but translating all the values to binary.
6. Select your memory element. Normally either a JK or D-type.
7. Rewrite the *coded state transition table* using the memory excitation table, so you have a table listing the values to be fed into the memory elements in each state. If using D-types, this will be the same as the coded state transition table, if using JKs the values of both J and K are each 'next state' will need to be determined. This is known as the *state machine excitation table*.
8. Optimise the *state machine excitation table* to determine the memory element input logic functions.
9. Determine the output logic functions
10. Draw the circuit diagram
11. Implement

We will now work through these stages to look at how a state machine can be designed and implemented in logic.

3.4. The State Transition Diagram

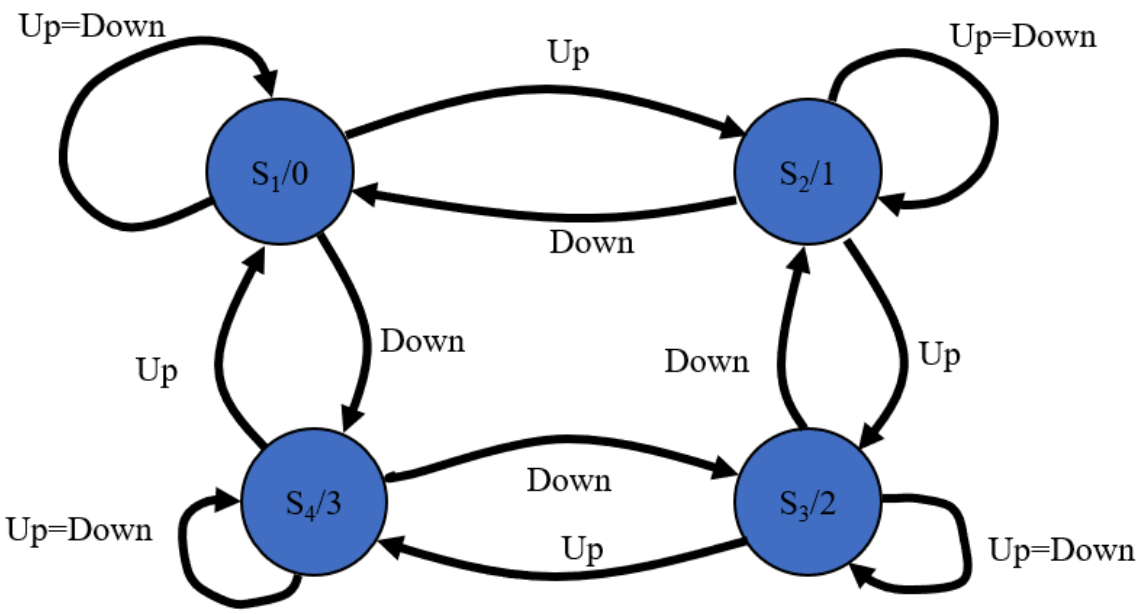

Figure 2: A Simple State Transition Diagram

Figure 2, above, shows a simple state transition diagram. You will notice that it is a *circle and arrow* type diagram with each of the required states represented by a circle, and each of the transitions represented by an arrow.

States

If we look in detail at the states, you will notice that the each circle contains a certain amount of information.

Firstly, you will notice that the information is in plain English, not binary. That is perfectly fine at this stage. If we look at the state 'S_1', inside the circle is written '$S_1/0$'. The S_1 is the name of the state. Depending on the type of system you are designing, you can use a system like this to name the states or something more descriptive. If you were designing the state machine for some traffic lights, you might choose to use names like 'All Red', 'Side Road Amber' and so on. The second part of the information displayed here is the required output in this state.

From this you can also deduce that this is a Moore machine. Why? Well, because there is only one output shown for each state, so the output is only dependant on the state and not on the inputs too. If you were designing a Mealy machine, you would need to include some form of truth table in the state circle to show the outputs in that state for each possible input combination.

Transitions

Each of the arrows on the diagram represent a transition (or state transition- hence the name). You will notice that there is a label above each of the arrows. This label describes the input condition (or conditions) which cause that transition to occur.

There are a number of things to note about the transitions:

1. The arrows show the direction of the transition. So, for example, in figure 2, if you are in state S_1 and the input is 'up' you move to state S_2.
2. Transitions can go from one state to another, or can be looped back round to the same state if there is an input condition which requires the state to be held for more than one clock pulse. You can see this in figure 2 in state S_1 where there is a looped back arrow for when the input Up=Down occurs.
3. **All input combinations must be accounted for**. Again, because we are working in binary, this means that two inputs would give four possible combinations, three would give eight and so on.
4. You do not need to include a separate arrow for each transition if they go to the same place. Again looking at figure 2, there are two inputs, so four combinations exist, yet from the state S_1 there are only three arrows departing. This is because 'Up=Down'

actually expresses two input combinations, 00 and 11. It is perfectly acceptable to do this provided you don't confuse yourself. If for example, in a state transition diagram, the input combination '010' moves you from S1 to S2, otherwise you stay at S1, you could express this just using two arrows. The arrow from S1 to S2 with the label '010' and a loopback arrow to S1 with the label 'All others' or something like that. Similarly, if the transition from S1 to S2 always occurs, you could draw one arrow and give it the label 'All'.

When drawing the state transition diagram, try to make is as neat as possible and make it nice and large. You may find you'll need to redraw it a couple of times if it gets too crowded but it's important that you can accurately read it so that mistakes don't creep in.

State transition diagrams are the sort of thing which are very quick and easy to draw by hand, but are rather more difficult and involved on computer. If you have a tablet with a stylus, you might find that drawing on to the computer is okay, otherwise, you might find it easier to hand-draw the diagram and then scan or photograph it to include in your log book and design submission.

In this example, transitions are only around the outside of the diagram but it doesn't have to be like that. You can have transitions from any state to any other as required.

When developing a state machine, you need to remember that state machines are driven by some form of clock pulse. On each clock pulse, the state machine will look to move from its current state to the next state as specified in the diagram. Because digital systems don't like uncertainty, you need to ensure that all possible combinations of inputs have a clear transition to a next state associated with them even if that transition is back round on itself so that the state machine stays in the current state.

3.5. The State Transition Table

The second stage in the design process is to convert the diagram into a table. By doing this it helps to start to formalise the transitions and outputs.

The state transition table consists of three sections as shown in table 1.

Section 1 The present state of the machine	Section 2 (this will probably contain a number of columns) For each input combination list the next state	Section 3. If it is a Moore machine, this will contain one column. If this is a Mealy machine, this will contain a column for each input combination This section lists the output in the current state.

Table 1: Sections of the State Transition Table

It the same way that the state transition table can be simplified by grouping inputs together, this can also be done for the state transition table, provided you don't end up tripping yourself up. Otherwise, separate columns can be included in section 2 (and section 3 if needed) for each input combination.

Again, at this stage, the entries in the table are just in plain English. Excel, or a similar spreadsheet program can be ideal for completing these tables.

	Next State				
Current	Input	Input	Input	Input	Current
State	Up /Down	Down /Up	/Up /Down	Up Down	Output
S1	S2	S4	S1	S1	0
S2	S3	S1	S2	S2	1
S3	S4	S2	S3	S3	2
S4	S1	S3	S4	S4	3

Table 2: Example State Transition Table

The state transition table for the state transition diagram in figure 2 is shown in table 2.

You can see in the leftmost column, that the current states are all listed. These are still just using the state names in English from the diagram.

There are four input combinations and these are all listed as sub-columns in the central section of the table. In each of those columns, the required next state for the combination of current state and current inputs is given. The current inputs are the column, and the current state is the row.

The third section is the current output. Here there is only one output for each state so the third section only has one column. Again, this column lists the output for this state. Again, in the case of a Mealy machine, the output section would look similar to the next state section.

3.6. Optimisation

At this stage it is important to check that there aren't any unused states or repeated states.

Repeated states will be states with different names but where:

- The output from the states are the same.
- *All* the transitions from both states are the same.
- *All* the transitions in to the states are the same.

If all of these conditions are met, then the states are repeated so you only need one of them. The repeat can be deleted and removed from the table. Similarly, if you need to include additional states to make it up to a power of two, these need including at this stage.

The remaining stages of the design process are implementation specific so have been included in a separate document. You can select the implementation phase based on your particular target.

4. Implementing a Finite State Machine in Discrete Hardware

So far, we have looked at the general design stages of the implementation of a finite state machine. These are stages which would be the same regardless of the way that the state machine is finally implemented. In the list below which is repeated from the previous section, these are steps one, two and three. In this session, we will look at steps four to eleven, which are more specific to a discrete hardware or schematic-capture-based implementation.

1. Draw a *state transition diagram*. This is a pictorial way of determining the operation of the system. At this stage, some elements such as inputs and output may be expressed in binary, but much of the system operation is just expressed in plain language.
2. Translate this into a *state transition table*.
3. Optimise the state transition table. We now need to look to see if there are any states which we've repeated (if so they can be removed). Because we know that the number of states needs to be a power of two, we need to consider what we do with any unused states. For example, if our machine needs six states, we have two free (the next power of two is eight). What outputs and what next state do we specify if the machine gets into these states? You may think that this is unlikely, but this could happen on power up for example.
4. Assign binary values to the states. The obvious thing to do here is to allocate in order but it is worth just taking a moment to see if a different allocation might lead to a simpler logic implementation.
5. Construct a *coded state transition table*. Basically, here we repeat step three, but translating all the values to binary.
6. Select your memory element. Normally either a JK or D-type.
7. Rewrite the *coded state transition table* using the memory excitation table, so you have a table listing the values to be fed into the memory elements in each state. If using D-types, this will be the same as the coded state transition table, if using JKs the values of both J and K are each 'next state' will need to be determined. This is known as the *state machine excitation table*.
8. Optimise the *state machine excitation table* to determine the memory element input logic functions.
9. Determine the output logic functions
10. Draw the circuit diagram
11. Implement

4.1. Assign Binary Values to States and Outputs

The output of our system will be binary, and because we are working with a digital system, the states will be numbered in binary. Strictly speaking, this will be the case if we implement in discrete hardware, programmable hardware or software, but if we implement in programmable hardware or software, we can continue to use the state names, identifying them as variables and allowing the system to assign the values automatically.

The default would be to assign values to the states in numerical order, so for our example which we started in the last section, we would assign as follows:

State Name	Binary Value
S_1	00
S_2	01
S_3	10
S_4	11

Table 1: Default State Value Assignment

It is worth just taking a moment to reflect, however. There may be a different assignment which is either preferred, or which would lead to a more efficient implementation.

For example, in our example here, the output values are 0, 1, 2, and 3, but suppose they had been 1, 2, 3 and 4. In this instance, a more efficient implementation would be as shown in table 2.

State Name	Binary Value
S_1	01
S_2	10
S_3	11
S_4	00

Table 2: Alternative Value Assignment

In this case, this would be the preferred assignment, because three of the outputs could be generated directly from the state values, with the fourth (S_4) being achieved through the use of a single NOR gate. If the default assignment from table 1 is use here, then decoding logic would be required for each of the output states.

Practice Exercise

In this example, the output vector would be $\mathbf{Z} = [z_2, z_1, z_0]$ where z_2 is the most significant bit, and the present state vector $\mathbf{Y^K} = [y_1^K, y_0^K]$ is the current state vector assignment as set out in table 1 or table 2 above.

Show that if the default assignment, shown in table 1 is used, then the output logic is:

$$z_0 = \overline{(y_0^K)}$$

$$z_1 = y_0^K \oplus y_1^K$$

$$z_2 = y_0^K \times y_1^K$$

Whereas if the alternative assignment of table 2 is used then it is:

$$z_0 = y_0^K$$
$$z_1 = y_1^K$$
$$z_2 = \overline{(y_0^k + y_1^K)}$$

4.2. Completion of the Coded State Transition Table

We now need to replace the 'English' values of the states in the state transition table with the assigned values we have chosen.

The state transition table is repeated as table 3 below, together with the coded version in table 4.

	Next State				
Current State	Input Up	Input /Down	Input Down/Up	Input /Up/Down	Current Output
S1	S2	S4	S1	S1	0
S2	S3	S1	S2	S2	1
S3	S4	S2	S3	S3	2
S4	S1	S3	S4	S4	3

Table 3: Original State Transition Table

	Next State				
Current State	Input Up	Input /Down	Input Down/Up	Input /Up/Down	Current Output
00	01	11	00	00	00
01	10	00	01	01	01
10	11	01	10	10	10
11	00	10	11	11	11

Table 4: The Coded State Transition Table

Practice Exercise

Have a go at constructing the coded state transition table for the alternative state assignment of table 2.

4.3. Selection of Memory Element

It is now necessary to determine how we are going to implement the state memories in the machine. There are two main alternatives here, we could use D-type flip-flops, or we could use JK flip-flops.

At an initial glance, D-types might seem easier because whatever is at the D input is what is transferred to the Q output on the next clock pulse, but in the case of the JK flip-flop, there are two ways to achieve each transition, the output can be directly loaded either '1' or '0', or it can 'change state' or 'remain at state'. This tends to mean that for each change one of J or K is a 'don't care' whilst the other must be set to a specific value. In many cases, the resulting logic can be simplified because the 'don't cares' can be arbitrarily set to simplify the logic.

Here we will develop the solution using JK flip-flops and leave you to complete the solution using D-types.

4.4. State Machine Excitation Table

In order to achieve the required 'next states' as listed in table 4 above, we need to supply the appropriate inputs to the memory elements such that they give the required outputs after the next clock pulse.

Hopefully, you will recognise that, in the case of the D-type, this is a trivial task because the input of the D-type transferred to its output on the next clock pulse. This means that the *excitation table* is identical to the *coded state transition table* in table 4 above.

In the case of the JK flip-flop, which we will look at in detail here, it is slightly more involved. Effectively, we need to combine the JK flip-flop excitation table with the coded state transition table to show what inputs need to be fed to the JK flip-flops in each state and for each combination of inputs to give the appropriate Q output from those flip-flops.

The excitation table for the JK flip-flop is given in table 5.

J	***K***	Q_{n+1}
0	0	Q_n
1	0	1
0	1	0
1	1	$\overline{Q_n}$

Table 5: JK Flip-Flop Excitation Table

In general, we will want to do one of four things as we move from one state to the next:

1. Output remains at 1
2. Output remains at 0
3. Output switches from 0 to 1
4. Output switches from 1 to 0

In each of these four cases, we have two options as far as the JK flip-flop is concerned:

1. Force the output to the required value
2. Use the *hold* or *toggle* states to achieve the required output

Table 6 shows the required inputs for all these cases.

Current Output	***Next Output***	***Set Output***	***Hold/Toggle***		
0	0	0	Hold	0	Don't care
0	1	1	Toggle	1	Don't care
1	0	0	Toggle	Don't Care	1
1	1	1	Hold	Don't Care	0

Table 6: J and K Input Requirements to achieve all Possible Outputs

You will notice that in each case, only one of J or K needs to be set to a specific value. If you cannot immediately see how table 6 was generated, take a few moments to work through the requirements in columns 3 and 4 using table 5 for reference to prove that the J and K requirements are correct.

The don't care states are normally shown as Φ because it looks like a zero and one superimposed on each other.

We now need to combine table 6 with table 4. The number of columns in the central section will need to double because each memory has two inputs.

	Next State															
	Input				Input				Input				Input			
Current	Up /Down				Down/Up				/Up/Down				Up Down			Current
State	J1	K1	J0	K0	J1	K1	J0	K0	J1	K1	J0	K0	J1	K1	J0	K0	Output
00	0	Φ	1	Φ	1	Φ	1	Φ	0	Φ	0	Φ	0	Φ	0	Φ	00
01	1	Φ	Φ	1	0	Φ	Φ	1	0	Φ	Φ	0	0	Φ	Φ	0	01
10	Φ	0	1	Φ	Φ	1	1	Φ	Φ	0	0	Φ	Φ	0	0	Φ	10
11	Φ	1	Φ	1	Φ	0	Φ	1	Φ	0	Φ	0	Φ	0	Φ	0	11

Table 7: State Machine Excitation Table.

4.5. Determining the Logic

We are now in a position where we can determine both the logic required to achieve the correct state transitions, and the log required to achieve the correct outputs in each state.

This can be done using Karnaugh maps, although in many cases you may have too many inputs to achieve this in one go.

Next State Logic

In this example, we have four outputs J1, K1, J0 and K0 and we need a separate Karnaugh map for each. Fortunately, we have two inputs and two 'current state' outputs which also act as inputs to our circuits here. As a result, we have just four signals to consider and so can use a Karnaugh map directly to generate the logic. If we had a third input, we might need to generate two Karnaugh maps for each output and combine the logic (in this instance you would have one Karnaugh map for when the third input is one, and the other for when it is zero).

The first thing we need to do is to swap the lower two rows of our excitation table around, so that they are in the correct order for the Karnaugh map. We can then select the four 'J1' columns for the J1 Karnaugh map and so on. These are shown in table 8a-d.

Up Down

J1	00	01	11	10
00	0	1	0	0
01	0	0	0	1
11	Φ	Φ	Φ	Φ
10	Φ	Φ	Φ	Φ

Q1 Q0

(a)

K1	00	01	11	10
00	Φ	Φ	Φ	Φ
01	Φ	Φ	Φ	Φ
11	0	0	0	1
10	0	1	0	0

(b)

J0	00	01	11	10
00	0	1	0	1
01	Φ	Φ	Φ	Φ
11	Φ	Φ	Φ	Φ
10	0	1	0	1

(c)

K0	00	01	11	10
00	Φ	Φ	Φ	Φ
01	0	1	0	1
11	0	1	0	1
10	Φ	Φ	Φ	Φ

(d)

Table 8: Karnaugh Maps for the Memory Inputs

We can now use standard techniques to determine the logic required. Remember when performing the grouping to make use of the Φ cells to minimise the amount of logic needed. For example, in the case of the J1 logic, there are only two cells which actually contain a '1'. This would suggest that the logic should be:

$$J1 = \overline{Q1}.\overline{Q0}.\overline{Up}.Down + \overline{Q1}.Q0.Up.\overline{Down}$$

But each of these has an adjacent 'don't care' state which we could set to a '1' giving the equation:

$$J1 = \overline{Q1}.\overline{Up}.Down + Q0.Up.\overline{Down}$$

Hopefully, looking at table 8b, you can see that:

$$K1 = J1$$

From table 8c and d, and setting the appropriate Φ terms to 1:

$$J0 = K0 = Up \oplus Down$$

Again, if you cannot immediately see these, take a few moments to work through them for yourself to confirm that they are correct.

Output Logic

In this case, determining the output logic is trivial, because it matches the corresponding output of the memory element, so z_0 is driven directly from y_0 and z_1 from y_1.

4.6. Draw The Circuit Diagram

We are now at a stage where we have completed the actual design of our state machine. We can represent the result as a circuit diagram which we can fabricate.

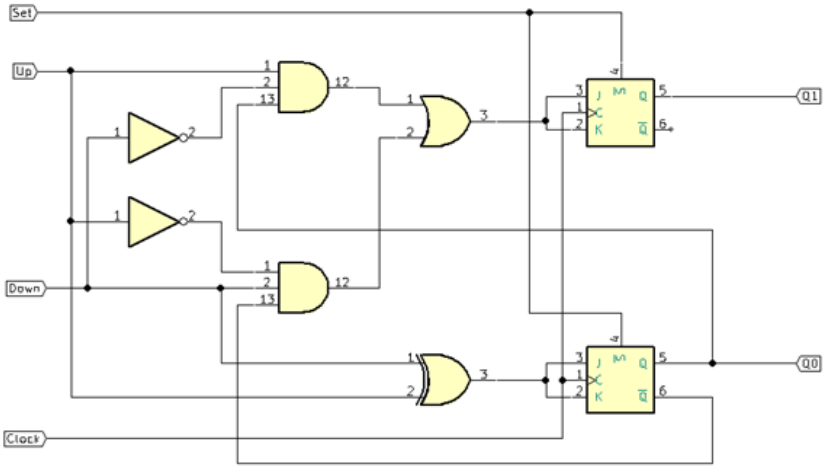

Figure 3: Memory Element and Control Logic Circuit Diagram

Figure 3 shows the circuit diagram for the implementation of the state machine using JK flip-flops. Note the inclusion of the clock signal which is required, and the inclusion of the set input, which is needed because these JK flip-flops include a set input but which is not part of the design.

Practice Exercise

Try to complete this design using D-type flip-flops, including determining all the required logic and drawing the circuit diagram. Remember, neatness is important when drawing circuit diagrams, consider how you will lay things out in order to minimise the number of tracks which need to cross each other.

Practice Exercise 2

At the start of this session we considered what we could do if the outputs counted 1-4 rather than 0-3. Was saw that numbering the states differently would make the output logic simpler (and so have already done part of the final stage of this design. Complete the design for both JK flip-flops and D-type flip-flops.

4.7. Implementing a Finite State Machine in Programmable Hardware or Software

So far, we have completed the generalised stages of the state machine design process which would be required whatever the final method of implementation was. In a separate session, we have considered how such a design would be implemented in discrete hardware- or using schematic capture on a programmable hardware system. Today, many solutions use programmable hardware with an HDL-based design entry system, or are software solutions and we need to consider how a state machine might be implemented here.

Implementing in Code

In some ways, implementing a state machine in code, be that VHDL, C or some other language, is almost easier than developing the solution in hardware.

Central to the code implementation is the use of the 'case' statement.

Hopefully, you have covered case statements elsewhere in the course, but briefly, a case statement give a list of code segments, one of which will be executed based the value of some variable. It is not too difficult here, to see how this structure fits nicely with the state machine concept.

We could define our variable *states*. If we wanted, we don't even need to assign numerical values to our different states, we could select the cases based on the names. If we consider the example we have been developing as we considered state machines, we would need four sections in our case statement:

Select Case 'States':

    Case 'S1':

        If Up=1 then States=S2

        Elseif Down=1 then States=S4

        Output=1

    Case 'S2'…

End Select

The listing gives a general idea (in pseudocode) of how the case statements might be constructed. Both VHDL and C contain this code structure, although the syntax is different in each case.

We have the various cases listed, and again, notice that we can just stick with using the names we have given them rather than converting to numerical values. Then inside each of the 'cases', we have a set of if statements to determine whether the various input combinations are occurring. If they are then we update the state according. Effectively, each of the 'if' lines forms a state transition arrow on the diagram. Finally, we update the output for the state.

We would need to put the whole thing in some overall loop which is controlled in some way by a clock but essentially that is all that is required.

In VHDL clock signals can be defined. In C, we could use a delay. On a microcontroller, we could use a timer for this purpose.

5. Digital To Analogue Conversion

In this session we will look at digital to analogue conversion. Practically, digital to analogue conversion is generally easier to achieve than its counterpart, as evidenced by the way that many analogue to digital converters have a digital to analogue converter at their heart.

Whereas an analogue converter tries to determine the appropriate digital number to represent an analogue input, the digital to analogue converter, will represent the value placed on its digital inputs as a voltage or current whose value is in proportion to that digital input value.

In many cases, it is easiest to consider the digital input value as a fraction of the maximum, and the output analogue value should be the same proportion of the maximum output voltage or current. So if an 8-bit converter is used and the value 128 is loaded, then the 'digital fraction' being loaded into the DAC is $\frac{128}{255} \approx 0.5$. If our output voltage range is 0 – 10V, then we would want the actual output voltage to be the same proportion of the maximum, so:

$$0.5 \times (10 - 0) + 0 = 5V$$

This may seem like rather a cumbersome way to express the voltage, but this is done to take account of any offset in the range, that is, in case the minimum voltage is not zero. If this is the case, then in general terms we can say:

$$V_{out} = \frac{(dig_{value})}{(dig_{max})}(V_{max} - V_{min}) + V_{min}$$

Suppose for example our DAC gave an AC output of ±15V. This would give:

$$V_{out} = \frac{128}{255} \times (15 - -15) - 15$$

$$= 0.5 \times 30 - 15 = 0V$$

This is what you would expect, if we have a bipolar output, then the 50% value should be zero.

In general, there are two ways that digital to analogue conversion can be achieved and we will look at both here.

5.1. The Weighted Resistor Digital to Analogue Converter

If we start by considering a digital output such as a port pin on a microcontroller such as a PIC. This pin will adopt one of two values, if the pin is outputting a '0' then 0V will be output, and if it is outputting a '1' then 5V will be output. If a resistor is connected to this pin then the current through this resistance will also switch between two distinct values, as determined by Ohm's law to be:

$$I_{a(b.0=1)} = \frac{V_{(1)}}{R_a} = \frac{5}{1k} = 5mA$$

$$I_{a(b.0=0)} = \frac{V_{(0)}}{R_a} = \frac{0}{1k} = 0mA$$

This arrangement is shown in figure 1 below.

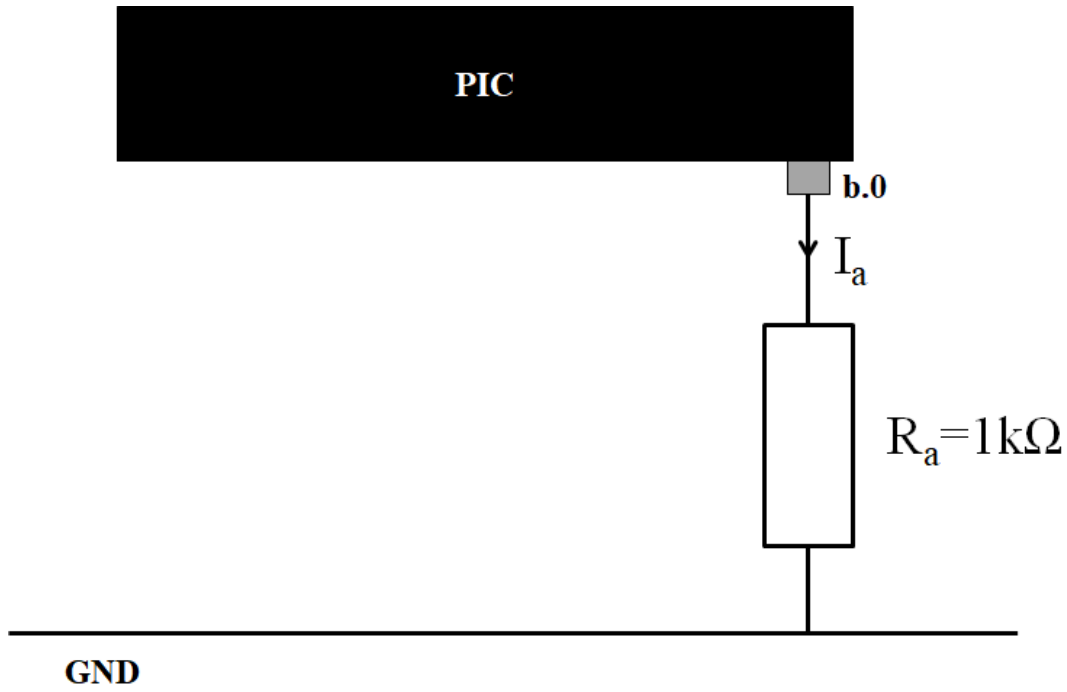

Figure 1: Connecting a Resistor to a Port Pin

Initially, this may not seem to be particularly helpful, since all we have achieved is to convert a pair of voltage levels to a pair of currents- and normally voltages are easier to handle.

Suppose, we now connect a second resistor with a value $2R_a$ to port pin b.1 as shown in figure 2.

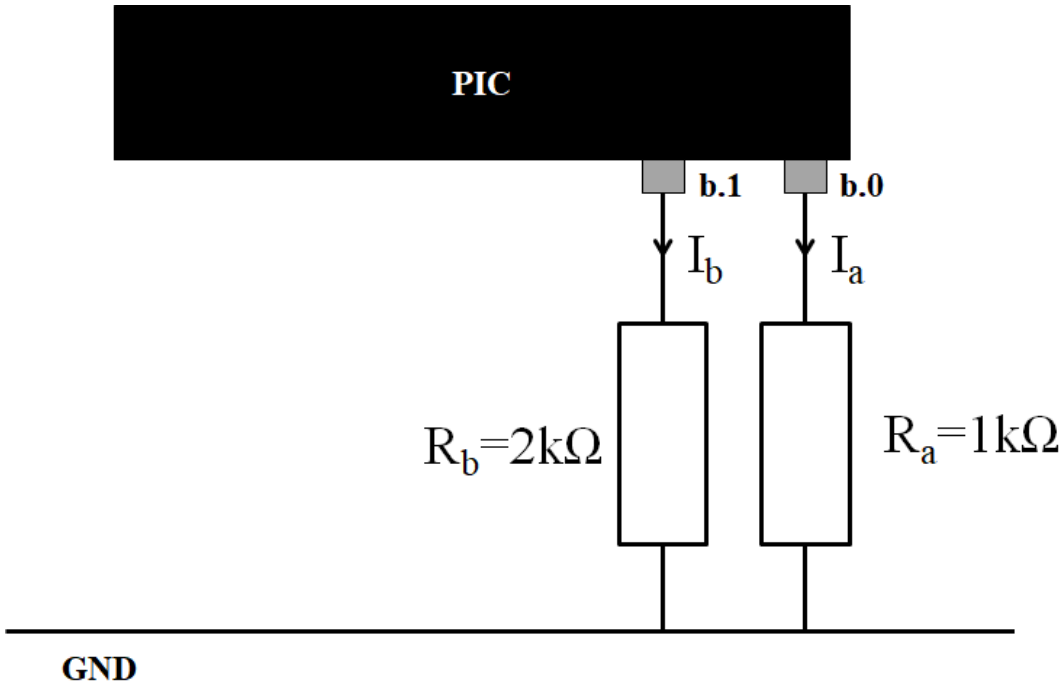

Figure 2: A Second Resistor Added to the Port

If we now use Ohm's law to determine the current though R_b we get:

$$I_{a(b.1=0)} = \frac{V_{(0)}}{R_b} = \frac{0}{2k} = 0mA$$

$$I_{a(b.1=1)} = \frac{V_{(1)}}{R_b} = \frac{5}{2k} = 2.5mA$$

We now have a situation where the current I_b is half of the current I_a when the port pin is high.

If we extend this to each of the eight port pins we get the circuit in figure 3.

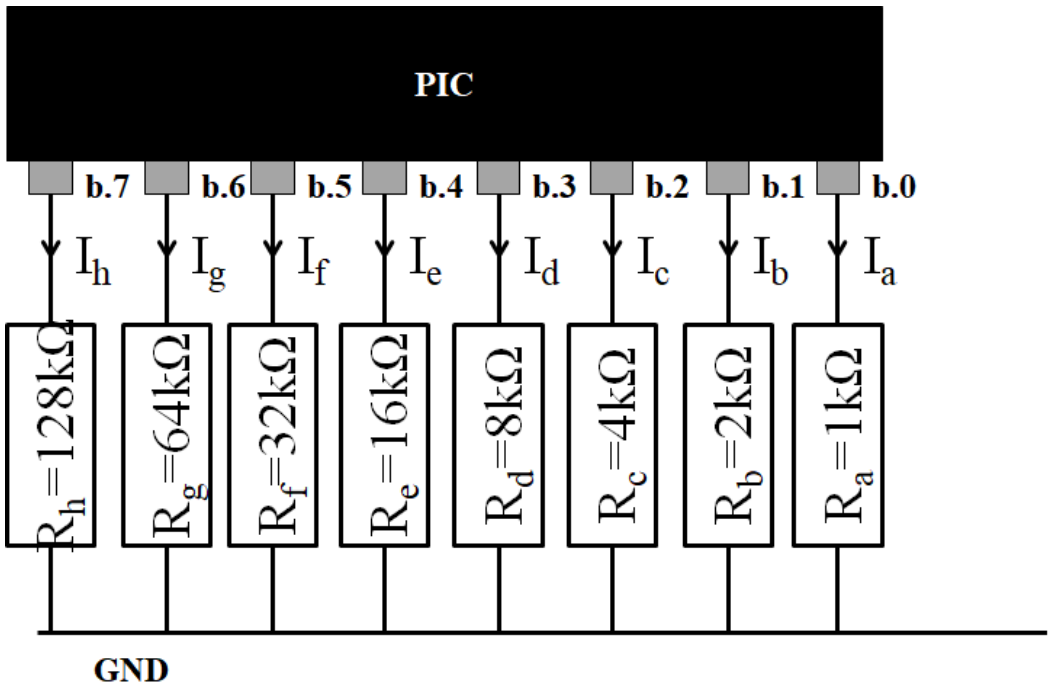

Figure 3: Weighted Resistor Digital to Analogue Converter

Across the whole port:

$$I_a = 2I_b = 4I_c = 8I_d \cdots$$

So we have now converted our binary output on the port pins to a binary weighted current though the attached resistors. There are two issues with the basic circuit as presented, however, the first is that the greatest contribution is given by port pin 0, whereas we normally represent binary numbers with the most significant bit on the highest numbered pin. This can easily be solved by reversing the order of the resistors as shown in 4.

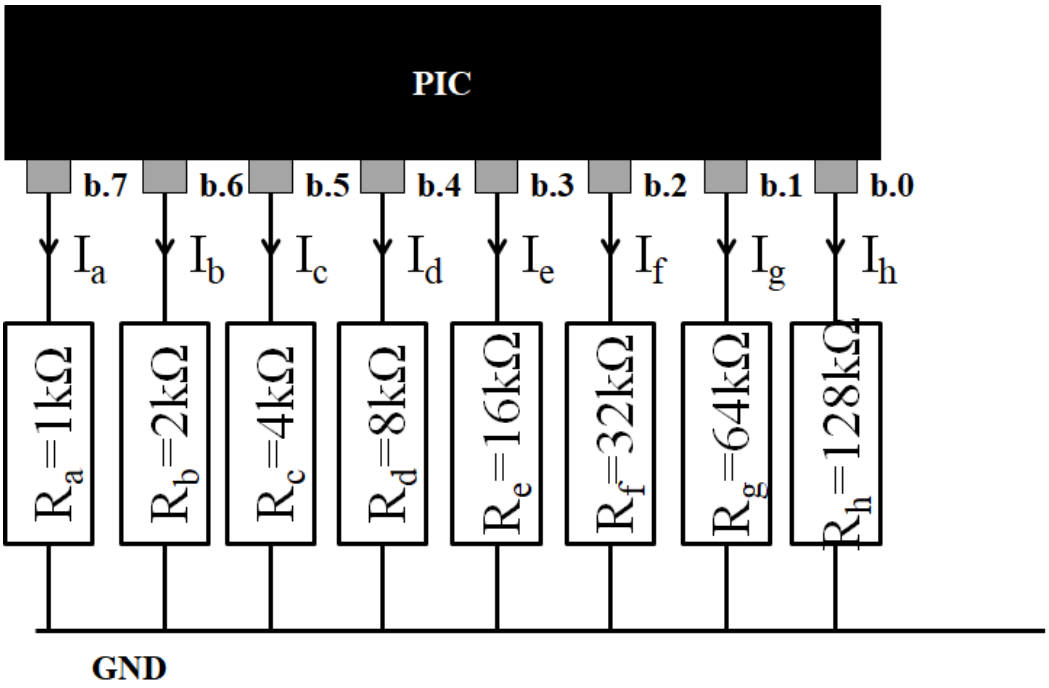

Figure 4: Correctly Weighted Digital to Analogue Converter

The second issue is that, ideally we need a single output which represents the total value of the binary number on the port pins. Currently, we only have a binary weighted set of currents through different resistors.

Here, we can make use of an analogue circuit. We know from Kirchhoff's current law that the sum of the currents arriving at a node is the same as the current leaving that node. Since one end of each resistor is connected to ground then this current summation occurs in the ground node. In order to successfully sum the currents we need to keep the summation voltage at zero because changing the voltage will change the potential across the resistors and the current through them as a result, but still be able to read off the total current in that node- and then, ideally, convert to a voltage.

Fortunately, the op-amp-based, inverting summing amplifier is a perfect solution to this problem.

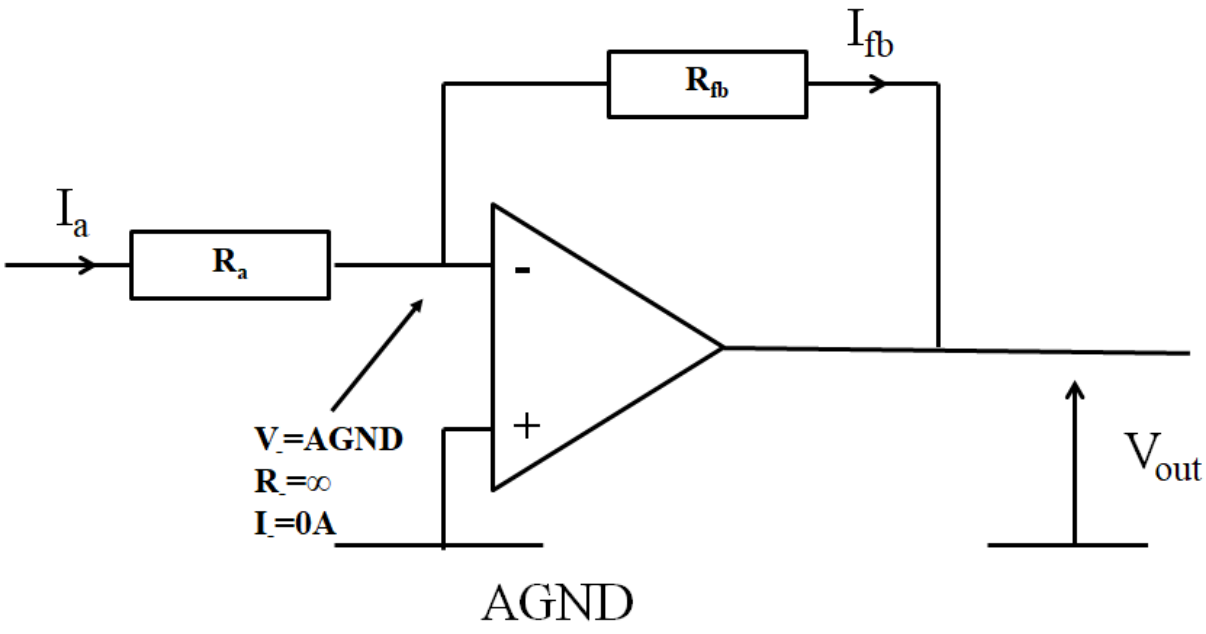

Figure 5: Inverting Amplifier Circuit

By way of a reminder, a standard inverting amplifier circuit is shown in figure 5. The principle of operation of the inverting op-amp is that the inverting input to the IC itself presents a high impedance to the signal connected to it so no current flows into this terminal. The internal gain of the op-amp means that there is no potential difference between the inverting and non-inverting inputs for any value of output voltage in the linear region. As a result, if the non-inverting input is connected to ground, then the inverting input is held at 0V too (the principle of virtual earth), and all current arriving at that node through the input resistor flows round the feedback loop ($I_{in} = I_{fb}$).

$$V_a - 0 = I_a R_a$$

$$0 - V_{out} = I_{fb} R_{fb}$$

$$\therefore -V_{out} = I_{fb} R_{fb}$$

The output voltage is negative but, apart from that, we now have a voltage output proportional to the current through the resistor. The value of R_{fb} can be selected to control the voltage gain of the output.

Since

$$I_{inv_{in}} = I_a + I_b + I_c \cdots$$

$$\therefore -V_{out} = (I_a + I_b + I_c \cdots) R_{fb}$$

In general terms, however, we don't want to talk in terms of the voltages or currents at the output pins, but in terms of the 1s and 0s, since this is normally how we express digital values.

We know that the output voltage will be 5V if the output is a '1' and 0V if the output is a '0'. As a result we can express the voltage as:

$$V = 5(b.n)$$

Where b.n is the value of bit n of port b.

We have already said that the values of the resistors halve between each increasing output bit. We know that the resistance of bit 7 is 1kΩ, so we can say that the value of the resistance attached to any of the bit is:

$$R_n = 2^{7-n} R_a$$

Using Ohm's Law, we can now express the current from any bit in general terms as:

$$I_n = \frac{V(b.n)}{2^{7-n} R_a} \qquad \text{Equation 1}$$

And we can say that the total current is:

$$I_{tot} = \sum_{n=0}^{7} I_n$$

Whilst this is helpful, what we really want is to be able to relate the output voltage directly to the full byte loaded on to port b. In order to achieve this we need to look at how the bits of the byte are related together. We can represent the numerical value of a whole digital value, B, of length, N, bits as:

$$B[N] = [\frac{b.0}{2^0} + \frac{b.1}{2^1} + \frac{b.2}{2^2} + \cdots + \frac{b.(N-1)}{2^{(N-1)}}]$$

The fact that we have ascending powers of two in the denominator here and descending powers of two in equation 1 means something rather interesting happens when we combine these equations:

$$I[N] = V[\frac{b.0}{2^0 \times 2^{(N-1)} \times R_a} + \frac{b.1}{2^1 \times 2^{(N-2)} \times R_a} + \cdots + \frac{b.(N-1)}{2^{N-1} \times 2^0 \times R_a}]$$

What is interesting here, is that when you multiply powers together, you add the indices together. As a result, each of the power of two multipliers in the denominator is the same:

$$I[N] = V[\frac{b.0}{2^{(N-1)} \times R_a} + \frac{b.1}{2^{(N-1)} \times R_a} + \cdots + \frac{b.(N-1)}{2^{(N-1)} \times R_a}]$$

$$= \frac{V}{2^{(N-1)} R_a}[b.0 + b.1 + \cdots + b.(N-1)]$$

$$= \frac{V}{2^{(N-1)} R_a} B[N]$$

And:

$$V_{out} = -\frac{VR_{fb}}{2^{(N-1)}R_a}B[N]$$

We now have a way of determining the output voltage of the DAC based on the binary value loaded, the digital supply voltage used, and the input and resistor values.

Example

In order to prove that this works, it is helpful to check both this equation and using the individual bits, to check that it gives the same result.

$B[N] = [01100000]_2 = 96_{10}$

$N = 8$

$R_a = 1k\Omega$

$R_b = 2k\Omega$

$R_c = 4k\Omega$
$R_{fb} = 1k\Omega$

$V = 5V$

As a byte:

$$V_{out} = -\frac{VR_{fb}}{2^{(N-1)}R_a}B[N]$$

$$\therefore V_{out} = -\frac{5\times1000}{2^{(8-1)}\times1000} \times 96$$

$$= -\frac{5{,}000}{128{,}000} \times 96$$

$$= -3.75V$$

By bits:

$$I_{tot} = \frac{5}{2000} + \frac{5}{4000}$$

In this example I have kept things simple by only setting two bits high (b.6 and b.5)

$$I_{tot} = 3.75mA$$

So showing that the equation is indeed valid.

$$V_{out} = -I_{tot}R_{fb}$$

$$= -0.00375 \times 1000 = -3.75V$$

Weighted Resistor Issues

We have seen that each additional bit requires the value of the least significant bit to double. This is not too much of a problem with 8 bits, since we have a minimum value of 1kΩ and a maximum of 128kΩ. If we wished to use this method for a 16 bit converter, however, the range would be:

This is a problem for two reasons:

1. As resistance values increase, the inherent noise voltage produced increases so, at these values, the noise may be come noticeable.
2. Probably the more important issue is that although we assume that the input impedance of the op-amp is infinite, in actual fact it is somewhere in the region of a 10-100MΩ. The assumption of no current flowing into the input of the op-amp only really applies if the external resistance is less than about 10% of the input impedance, so less than about 1MΩ. Here, this is no longer

the case, so we can no longer assume that all the current from that input flows round the feedback loop. As a result, we will end up with an error in our output voltage. Consequently, we need a solution whereby any length converter can be produced whilst the resistor values are kept within sensible limits.

An initial thought might be to reduce the value of the resistance connected to the most significant bit but this is only possible to a certain extent because each pin has a current limit, and the IC as a whole has a current limit. As a result 500Ω or 250Ω might be the practical lower limit on resistor values which would still leave the maximum resistance at 8MΩ.

5.2. The R-2R Ladder

The solution to the issue of needing a large range of resistor values is to use the R-2R ladder. As the name suggests, this type of DAC design only requires two resistor values, one of which is twice the other. This is regardless of the resolution of the converter.

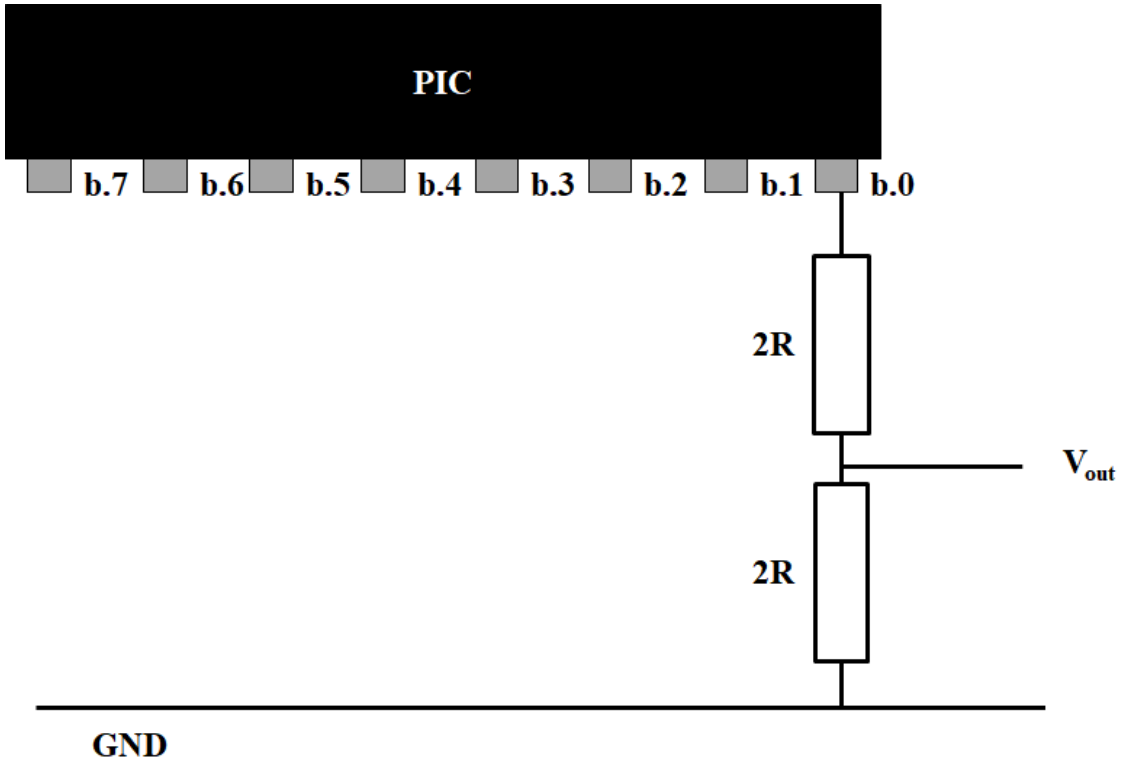

Figure 6: A One-Bit R-2R Ladder DAC

Figure 6 shows a single bit of an R-2R DAC. You will notice that it forms a potential divider and, hopefully, you can intuitively see that when b.0 is high, the output is half of the input, and is zero when b.0 is zero. More formally we can say:

$$V_{out1} = V_{b.0}\frac{2R}{2R+2R}$$

The other important thing to notice here is the source impedance of the divider. We can determine this by applying Thevenin to the divider. You will remember that a perfect voltage source is considered to have 0Ω output impedance. The port pin of the PIC will not quite be 'perfect' in that sense but provided the output impedance is only a very small fraction of the external resistor value, the effect will be negligible. This means that as far as the output is concerned, it will 'see' the two resistors in parallel, so the source (or Thevenin) impedance is:

$$2R \parallel 2R$$

$$\frac{1}{R_{source}} = \frac{1}{2R} + \frac{1}{2R} = \frac{2}{2R}$$

$$\therefore R_{source} = R_{Th} = R$$

Similarly, the Thevenin voltage is the open circuit output voltage V_{out1}. Because the value of $V_{b.0}$ can change, we need to express this voltage as a function of $V_{b.0}$.

$$V_{th1} = \frac{V_{b.0}}{2}$$

Whilst this may seem somewhat irrelevant, it is useful when we come to adding additional stages to our converter. If we now expand our converter to two-bits, as shown in figure 7, we can see why the source impedance is useful.

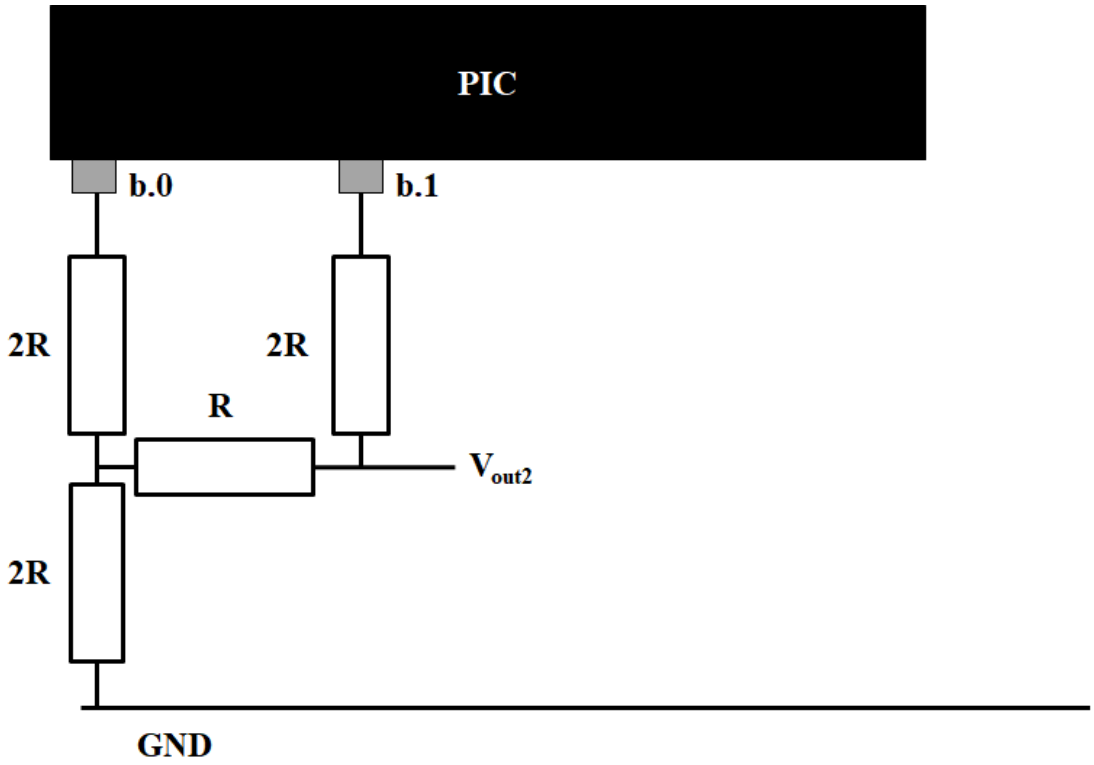

Figure 7: A Two-Bit R-2R DAC

The first thing to note is that all of the resistors shown are either R or 2R. This means that when we come to analyse the operation of the circuit, we need to include all the resistors in our analysis because they will load each other and so will all interact together.

When we come to analyse this circuit, we now have four possibilities because we have two binary outputs, and each of these can have one of two values.

Output '10'

If we analyse the situation where $b.0 = 0$ and $b.1 = 1$, initially we can draw up an equivalent circuit as shown in figure 8.

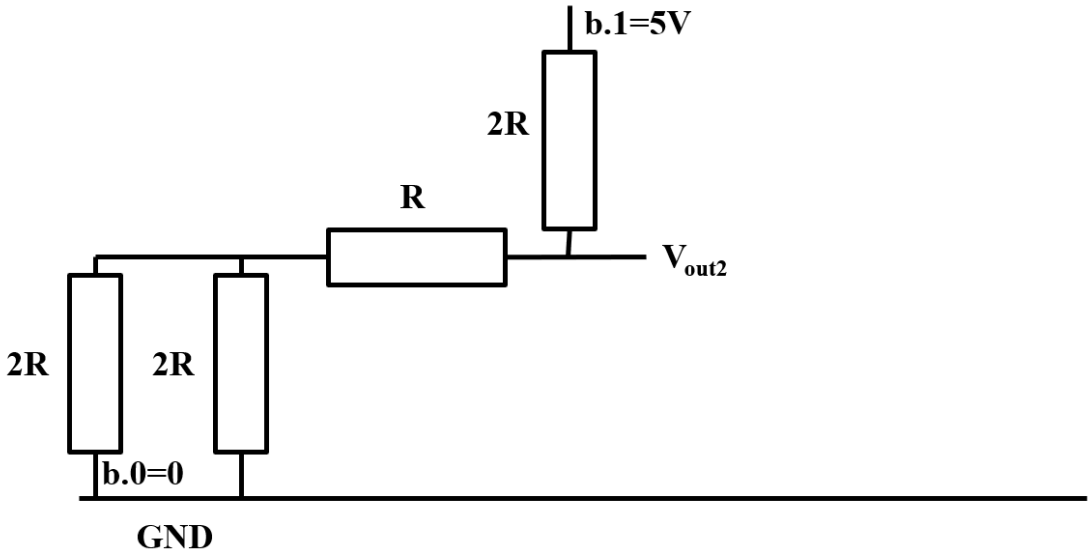

Figure 8: R-2R DAC Equivalent Circuit for 10 Output

Again, hopefully you can see how various resistors combine in parallel and series to give a final equivalent circuit as shown in figure 9.

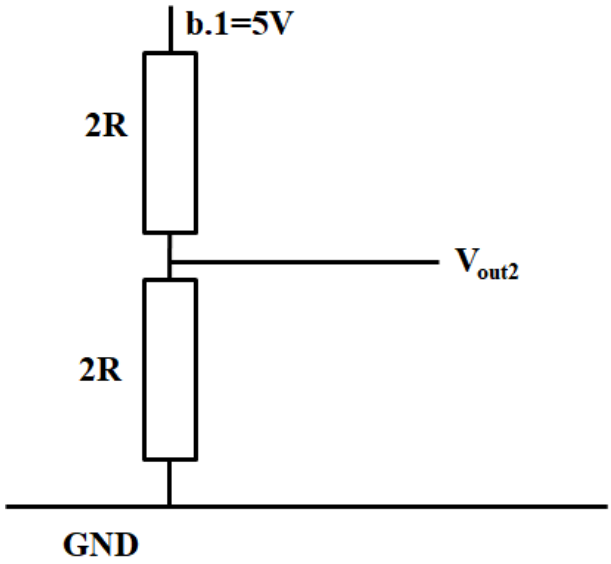

Figure 9: Simplified Equivalent Circuit for '10' Output Combination

Bit one is now the most significant bit of our DAC, and you will notice that, because of the way in which the resistances combine, the equivalent circuit, as shown in figure 9, is the same as that in figure 6 when bit zero was the only and therefore most significant bit.

This implies something significant and fundamental to the design of the R-2R DAC. That is that, regardless of how many bits there are in an R-2R DAC, the source impedance, and hence the Thevenin impedance, when looking back into the output of the DAC is always R.

Because the circuit is the same as previously, we can also see that the contribution to the output from b.1 alone is 0.5, which again is the same as when b.0 was the most significant bit.

Determining the Thevenin voltage for this circuit is a little more complex because we have two sources involved. The important thing to remember is that the Thevenin equivalent circuit for b.0 is valid in all situations, so we can replace the part of the circuit associated with b.0 by its equivalent, as shown in figure 10.

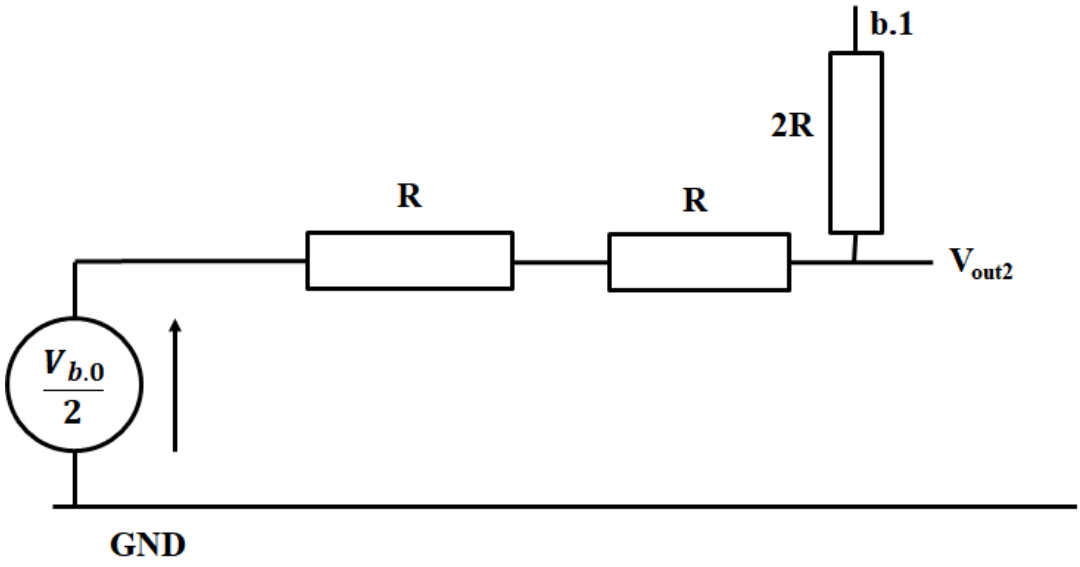

Figure 10: The DAC with the b.0 Contribution Replaced by its Thevenin Equivalent Circuit

Hopefully, you can see without the need for a full proof that the overall Thevenin resistance is still R. The Thevenin voltage is still determined from the potential divider as previously but now we have both ends connected to voltage sources so we need to use a more complete form of the potential divider rule:

$$V_{out} = \frac{R_1}{R_1+R_2}(V_{pos} - V_{neg}) + V_{neg}$$

In general we have:

$$V_{out} = Division \times Range + Offset$$

In many situations $V_{neg} = 0V$ and so can be left out of the equation but here, depending on the value of $V_{b.0}$ it has the possibility of shifting the output voltage up and so must be included. Substituting our values in to the general equation we get:

$$V_{Th2} = \frac{2R}{4R}\left(V_{b.1} - \frac{V_{b.0}}{2}\right) + \frac{V_{b.0}}{2}$$

$$= \frac{1}{2}\left(V_{b.1} - \frac{V_{b.0}}{2}\right) + \frac{V_{b.0}}{2}$$

We can now multiply out the brackets and collect terms to give:

$$V_{Th2} = \frac{V_{b.1}}{2} + \frac{V_{b.0}}{4}$$

You can see here that the addition of the extra bit has halved the weighting of the contribution from $V_{b.0}$. We can again take out the voltage and express the bits and 1s and 0s:

$$V_{Th2} = 5\left(\frac{b.1}{2} + \frac{b.0}{4}\right)$$

In general, we can develop this equation to express the total output from an N-bit R-2R DAC as:

$$V_{out} = \sum_{n=0}^{N-1} \frac{V_n}{2^{(N)-n)}}$$

Output '00'

To check our analysis above and for completeness, it is helpful to consider the remaining three output combinations.

When b.0 and b.1 are both 0, the equivalent circuit will be as in figure 11.

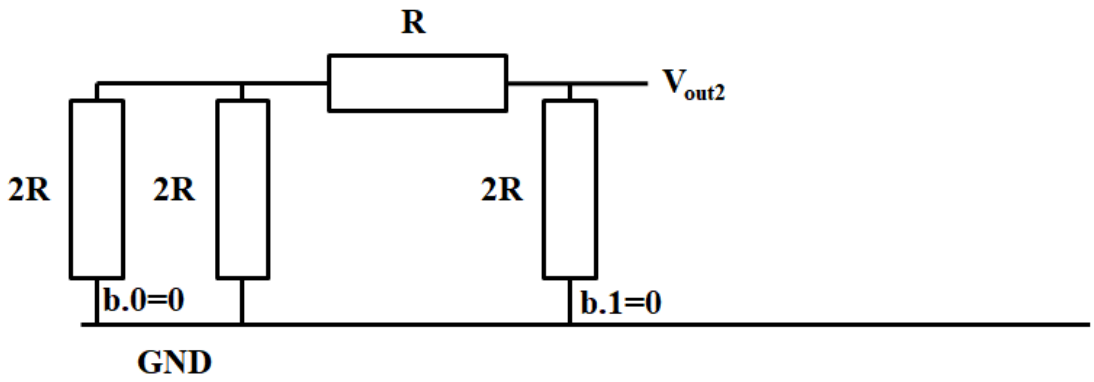

Figure 11: The R-2R Equivalent Circuit for a 00 Output

Since all resistors are connected to ground, it is easy to see that the overall output is 0V.

Output '01'

For this combination, we again need to undertake circuit analysis to determine the output voltage, because we cannot assume that the output from the b.0 stage is still 2.5V since its resistor network will be loaded by the b.1 network. We need to work backwards combining groups of resistors either in parallel or series.

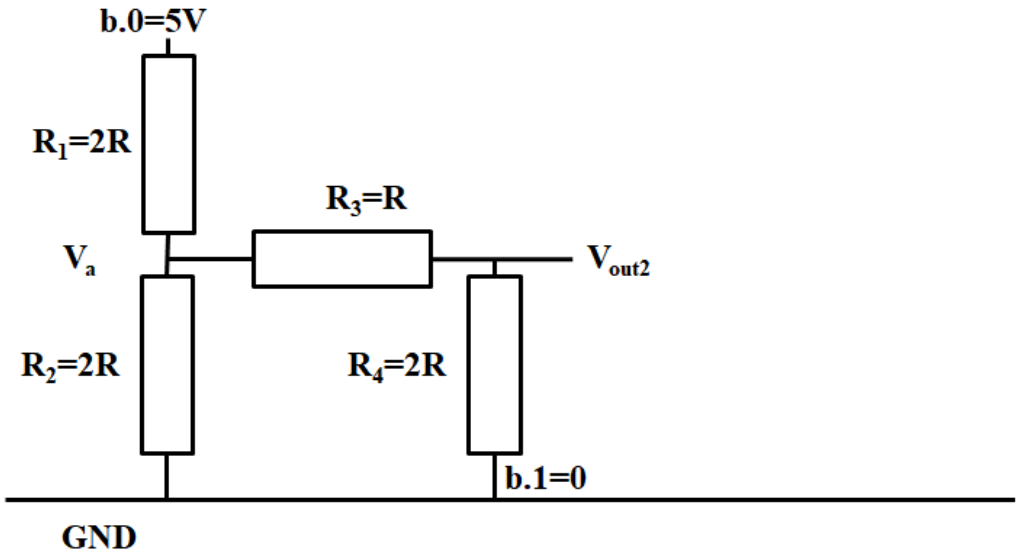

Figure 12: The R-2R DAC Circuit for a '01' Output

In figure 12, the various resistors have been labelled for clarity, and a node voltage V_a added.

Analysis can proceed as follows:

1. Combine R_3 and R_4 in series to give R_5
2. Combine R_5 in parallel with R_2 to give R_6.
3. Use the potential divider rule with R_1 and R_6 to determine V_a
4. Use the potential divider rule again on R_3 and R_4 to give V_{out2}.

$$R_5 = R_3 + R_4 = R + 2R = 3R$$

$$\frac{1}{R_6} = \frac{1}{R_2} + \frac{1}{R_5} = \frac{1}{2R} + \frac{1}{3R} = \frac{5}{6R}$$

$$\therefore R_6 = 1.2R$$

$$V_a = 5 \times \frac{1.2R}{3.2R}$$

$$V_{out2} = \frac{2R}{3R} \times \left(5 \times \frac{1.2R}{3.2R}\right)$$

$$= \frac{2}{3} \times 5 \times \frac{1.2}{3.2}$$

$$= 1.25V$$

Output '11'

Much of the analysis for the '11' output combination is the same as for the '01' combination above. The only difference is that in step 4 above, the potential divider is now between V_a and 5V not V_a and 0V, so we need to account for that in the calculation:

$$V_{out2} = (V_{b.1} - V_a) \times \frac{R}{3R} + V_a$$

$$= (5 - 3.125)\,\frac{1}{3} + 3.125$$

$$= 3.75V$$

Note that, in this case, the voltage of interest is across R_3 so the fraction is 1/3 not 2/3.

5.3. Issues

Resistor Accuracy

We have already seen that there are issues with the weighted resistor DAC, and how the R-2R DAC can overcome these, but there is an issue common to all DACs and that is to do with the resistors.

In both of the examples here, the output voltage is determined by applying the port pin voltages to a network of resistors. As a result, any error in the values of these resistors will lead to an error in the analogue output, and as we have said previously, any such errors will be detected as noise in the output of the system.

Variations could occur due to tolerances due to the manufacturing process, or as a result of changes due to a change in temperature. Because each of the bits carries a different binary weighting, the resistors connected to that part of the circuit will have that amount of effect on the output if they are incorrect. This means that resistors connected to the more significant bits will have a greater effect on the output than the less significant bits and any errors will be magnified too.

The only solution to this issue is to use high precision (0.1% or better) low temperature coefficient resistors for the DAC network.

A second related issue is that resistors are not easy to fabricate on silicon. Higher values in particular are large, and so expensive, and generate heat which can lead to other problems. When they are fabricated the tolerances are, in general, much worse than for discrete resistors. As a result, when fabricating the resistors for DACs special techniques are used such as *laser trimming* to adjust the values of the resistors in the network to make them as accurate as possible.

Output Smoothness

Looking closely at the output of the DAC will show an analogue waveform which is actually made up of a number of tiny steps. In many cases this is acceptable, but in others it is necessary to put filtering on the output to smooth the output and remove any resulting distortion.

5.4. The ∑Δ DAC

It is possible to remove the need for large numbers of resistors through the use of the ∑Δ technique discussed in relation to analogue to digital conversion in the last session. As with the ADC, there are a family of related DAC structures which can be used, but all are essentially the 1-bit DAC and integrator elements of the ∑Δ ADC. If the DAC input is a *pulse code modulated (PCM)* data input, that is, it is a binary value representing the size of the signal, then the inverse DSP algorithm will need to be employed to convert it to a suitable ∑Δ bit-stream

5.5. Summary

In this session we have looked at digital to analogue conversion, discussing the use of binary weighted resistors and the R-2R ladder to achieve an analogue output. We highlighted the issue of large resistor values in regard to the weighted resistor DAC, and resistor tolerances in both cases.

Finally, we highlighted the possibility of using the ∑Δ technique discussed in relation to analogue to digital conversion to achieve digital to analogue conversion without the need for large numbers of precision resistors.

6. Theory of Analogue to Digital Conversion

In this section we will look at the theory behind analogue to digital conversion. The companion handout looks more at the specifics of analogue to digital conversion on the PIC.

Analogue and digital describe the way in which a signal is represented. Analogue refers to a signal which is continuous in both time and amplitude. Digital refers to a signal which can adopt one of a predefined number of distinct values and, in the case of a sequential system, one where those values are constrained to exist at defined intervals in time. In the majority of situations, the analogue signal level will be expressed as a voltage and that voltage will exist between specified limits. The limits will normally be set by the supply voltage, or some other limitation on the circuitry in the analogue signal path.

Analogue to digital conversion is the process by which an analogue input signal is converted to an accurate digital representation of that signal. In many situations input signal conditioning may be employed either to convert the source signal format to a voltage if, for example, the source is an opto-sensor which outputs a current, and/or modify the range of the analogue signal to make is suitable for conversion. This second aspect may involve either increasing or reducing the voltage range and commonly includes converting an AC, bipolar, signal to a varying DC, unipolar signal. This is required because digital systems tend to work on voltages between 0V and some positive voltage. Originally, this was standardised at 5V but increasingly is 3.3V or 2.5V.

6.1. The Principles of Analogue to Digital Conversion

If we start, by way of an example, by thinking about how long the queue is for the coffee shop in the Anglesea foyer area, we might thing about doing this by counting the number of people in the queue on the hour every hour. Doing such an experiment, however, would give very misleading results because it only details how many were waiting exactly on the hour, and at no time in between. To get a more detailed idea of the way in which the queue changes throughout the hour much more regular measurements would need to be taken, maybe every minute, or every second. Taking more measurements would give a much more detailed picture of the changes, but at the expense of recording much more information. Because the queue is not likely to be that long, determining how accurately we record the number if probably a trivial matter, because we would probably just record exactly how many people were in the queue.

If we were looking to record something which changed more slowly such as atmospheric pressure or temperature, then taking a reading once an hour might be perfectly acceptable to show the broad way in which change was occurring throughout the day. The question in this instance might be how accurately we should take each reading. Should we read to the nearest degree centigrade or millibar, or to the nearest half, or the nearest tenth?

How Much, How Often?

The questions of how much or more precisely how accurately, we take each reading or *sample* and how often we take readings are central to the process of analogue to digital conversion.

How often we sample is known as the *sample rate* or *sampling frequency* and is normally given the symbol f_s. It defines the number of samples taken per second, and is normally expressed as a frequency in Hertz.

The accuracy of the conversion is governed by the number of *quantization levels* used. The quantisation levels define the number of discrete output levels used to represent the analogue input signal. The number of quantisation levels available is directly related to the *wordlength* of the converter, that is the number of bits used in the conversion:

$$quantisation_{levels} = 2^{bits}$$

This is also referred to as the *resolution* of the converter.

Wordlength Issues

As wordlength increases, the amount of data generated also increases, which can cause an issue. Also, the design of the converter becomes more complex and the tolerances on the conversion become that much tighter requiring higher design and manufacturing standards. Noise can also be an issue in that a small amount of noise on the analogue input can translate into an incorrect reading at the output by causing the reading to increase or decrease by one or two.

As the resolution decreases, the conversion process becomes simpler, but the conversion is coarser because fewer values are used to represent the whole analogue input range. As a result some of the finer details in the input signal might be missed.

A second problem also occurs as the resolution decreases, however, in that the *signal to noise ratio* also gets worse.

Without going into too much of the technical detail at this stage, any conversion is something of an approximation. Effectively, we are rounding to the nearest whole number on each conversion whereas the input could take any value.

When rounding a number in maths, we either round up or down depending on whether the decimal is <0.5 or not. The same happens when analogue to digital conversion occurs. The analogue value will sit somewhere between two of the quantisation levels and the output will either round up or down depending on which is closer. If we looked at the probabilities involved, we would discover that numbers are rounded up as often as they are rounded down. As a result, the value of the least significant bit is effectively random, and so can be considered to be noise since noise is defined as a random signal.

The signal to noise ratio is defined as:

$$SNR(dB) = 20log\left(\frac{Signal}{Noise}\right)$$

Normally we measure these values as voltages, but here we can measure them in terms of their binary values. If we have an 8-bit system, the most significant bit would represent the sign of the input (positive or negative) leaving 7-bits for the value. The maximum number achievable with 7-bits is:

$$Signal = 2^7 - 1 = 127$$

The noise level here is 1 as this is a single bit change. The resulting signal to noise ratio is:

$$SNR(dB) = 20log\left(\frac{127}{1}\right)$$

$$\therefore SNR(dB) = 42dB$$

If a 16-bit system is used then the maximum signal level is:

$$Signal = 2^{15} - 1 = 32,767$$

Since the internal system noise remains the same, the signal to noise ratio becomes:

$$SNR(dB) = 20log\left(\frac{32,767}{1}\right) = 90dB$$

This gives a measure of the inherent limit on the signal to noise ratio for a given resolution converter not taking into account any additional noise due to poor design or implementation. To give some idea of what is good or not, anything above about 60-70dB would not really be noticeable in an audio single when we listen to it.

Sampling Issues

Again, as the sampling rate increases, the amount of data generated increases so sampling excessively fast is not ideal, but if we sample too slowly we might 'miss something'. When we sample a signal we only have information about the signal level at the particular points at which we sampled. For example, if we took a temperature reading at 10am each day we could plot the results on a graph and then draw a curve of 'best fit' but in doing so, we would have to assume that we can link adjacent points by a straight line. The temperature could have dropped to -10^{o}C overnight, but we wouldn't know that without having taken that reading. The same applies when converting analogue to digital. We have to assume that we can draw a straight line linking adjacent samples so any other changes between these points is missed. This requires us to ensure that we sample fast enough not to miss anything in the input signal.

6.2. Nyquist, Shannon and Aliasing

Nyquist and Shannon, amongst others, variously described the minimum sampling frequency required to accurately convert a certain frequency analogue signal in digital. In different places it is variously described as the *Nyquist Frequency*, the *Nyquist-Shannon Frequency*, and the *Shannon Limit* amongst others. All basically state that:

$$f_{sampling(min)} \geq 2 \times f_{sig(max)}$$

Effectively, this is saying that the sampling frequency must be at least twice the maximum signal frequency. So if your maximum signal frequency is 20kHz, then your minimum sampling frequency is 40kHz, or 40,000 samples per second.

Figure 1 shows graphically the effect of not sampling a sine wave fast enough. The blue line shows several cycles of a sine wave, and the orange line is a line of best fit against three samples taken. You will notice immediately that the implied frequency of the orange line is different to, and lower than the blue line.

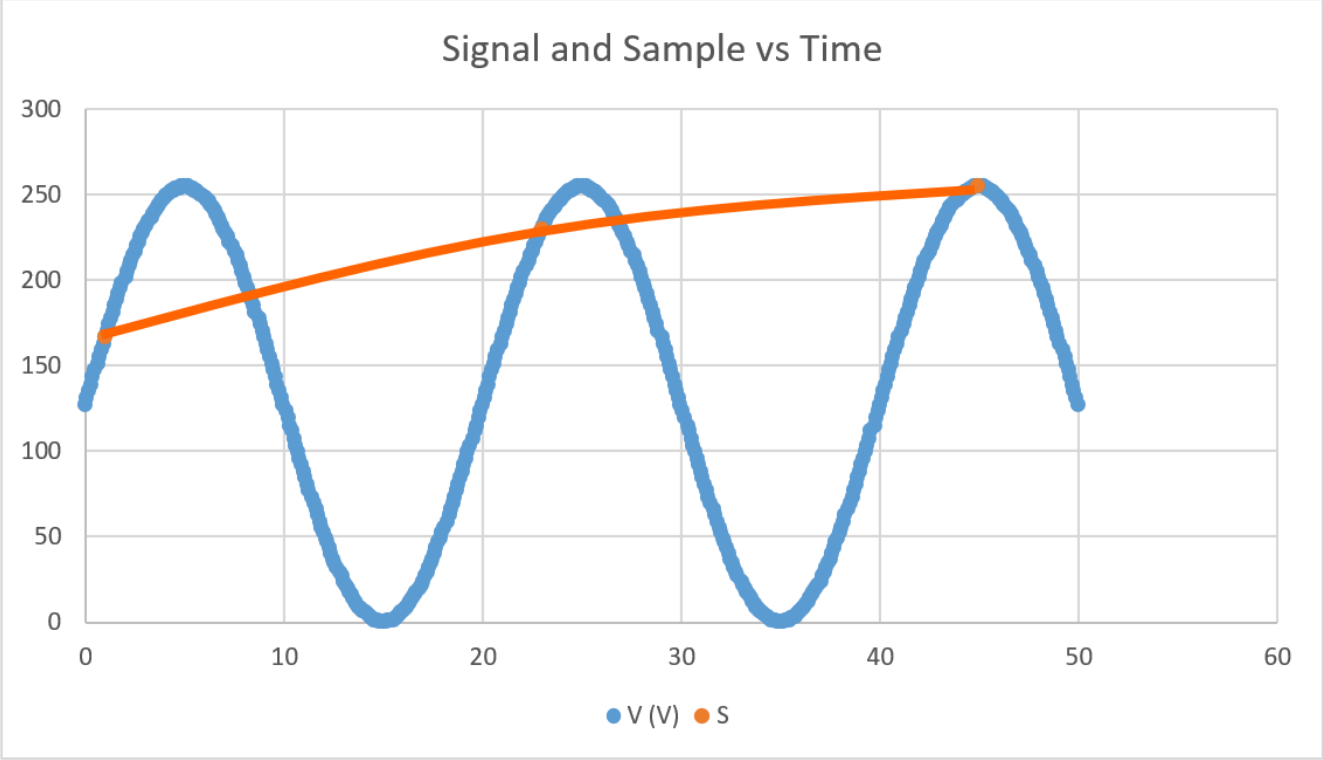

Figure 1: The Effect of Sampling at lower than the Nyquist Frequency

The issue of sampling at less than the Nyquist frequency can be seen to be more profound than might initially have been expected. It is not the case that frequencies greater than half of the Nyquist frequency do not appear in the output. The issue is that they *do* appear in the output but at a frequency which is different from, and lower than that of the original. You can possibly imagine the effect in the case of sound where random frequencies, unrelated to anything in the music, start appearing, it would not sound pleasant.

This effect is known as *aliasing* and is a fundamental consequence of sampling. The mathematical proof as to why this happens is too involved to cover in detail here but I'm sure you will do in the DSP module next year if not before. Briefly, if you take a sampled signal and plot a graph of amplitude against frequency (rather than time) an interesting effect becomes apparent in that the input frequency response appears as a repeating pattern above and below integer multiples of the sampling frequency. This is a similar effect to what occurs in radio with modulation.

If the sampling frequency is less than twice the highest signal frequency then the repeat which sits below the sampling frequency on the graph overlaps the original, as shown in figure 2. In the graph it is clear that the bands overlap as highlighted by the blue areas. Note that including negative frequency is formally correct but normally we ignore it.

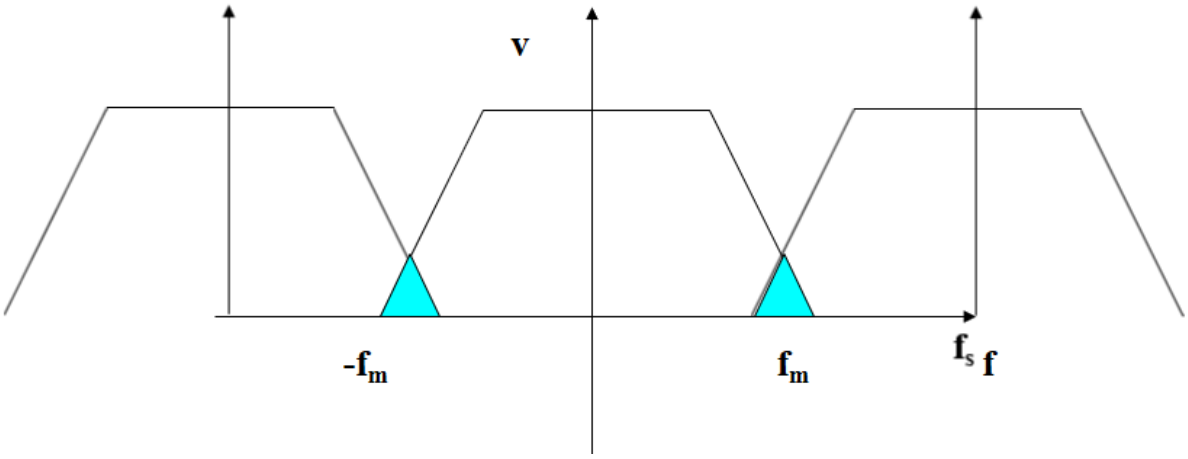

Figure 2: A Graphical Representation of Aliasing in the Frequency Domain

Further considerations with aliasing

If we consider an audio signal, the generally accepted frequency range for audio is 20Hz to 20kHz based on the frequencies that an average human with good hearing in their early to mid twenties can hear. As a result we could say that the maximum signal frequency is 20kHz and so the Nyquist frequency is 40kHz. However, just because *we* can't hear it does not mean that there are no 'sounds' in the air at frequencies greater than 20kHz. You may be aware, for example, that bats use ultrasonic echo-location to find their way around at frequencies up to around 40kHz. Just because we cannot hear frequencies above 20kHz does not mean that they might not be picked up by a microphone. If they are picked up by the microphone, they could arrive at the input to the ADC and so could cause aliasing. As a result, it is not enough to say that a certain frequency is the maximum frequency we are interested in, we have to take steps actually to ensure that frequencies higher than our desired maximum are prevented from reaching the ADC and being converted. This is normally done through the use of a low pass filter known as an *anti-aliasing filter*. Again, an issue exists here, however, because as the sampling frequency gets closer to the Nyquist limit the 'space' between the original band and the lower band of the first repeat gets smaller. As a result, if aliasing is to be prevented, the filter has to become of a higher order to adequately remove unwanted frequencies.

The sampling rate for CD is 44.1kHz. This is primarily because it was about a fast as could reasonably be achieved at 16-bit resolution without the converters becoming excessively expensive when CDs were introduced. This is a little above the Nyquist frequency but not by much. As a result, a very sharp *anti-aliasing* filter was required to allow through 20kHz unattenuated, but to attenuate above 22.05kHz by over 80dB. More recently, other techniques have been employed to resolve this issue.

6.3. A Very Basic Analogue to Digital Converter

Having looked at the requirements on the input side in terms of sampling frequency and resolution, we will now turn our attention to the process of converting the analogue input to digital, starting with the most basic of ADCs as shown in figure 3.

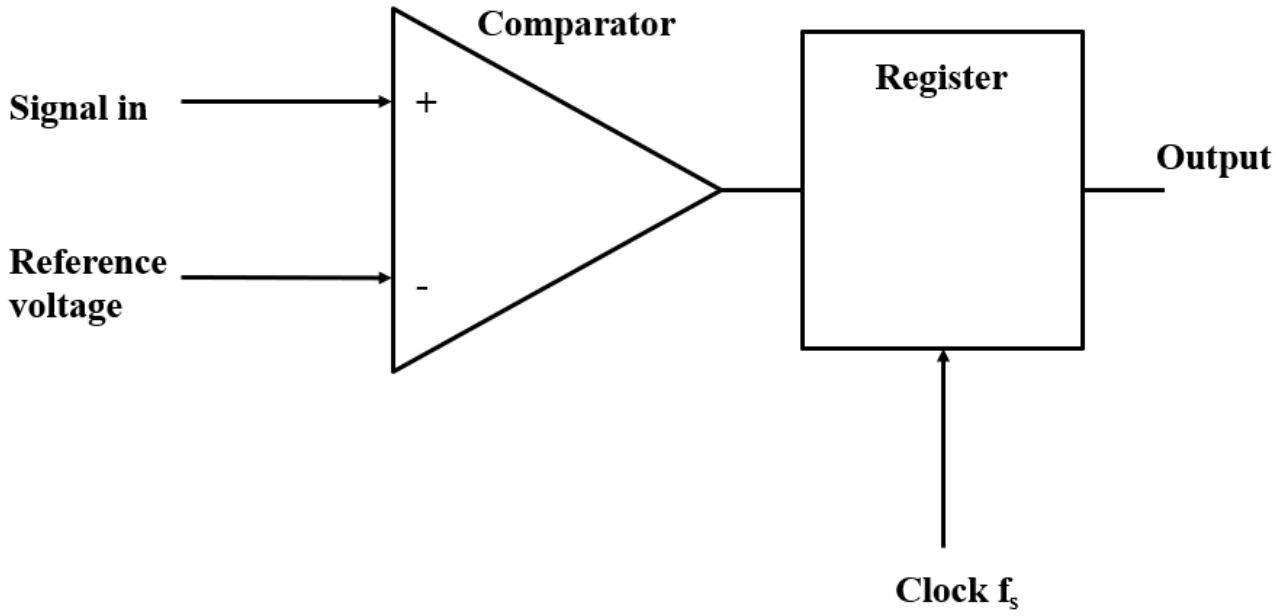

Figure 3: The Comparator as an Analogue to Digital Converter

The signal we wish to convert to digital is fed in to the 'signal in' which is the non-inverting input of the comparator. A steady DC voltage (normally half of the signal range) is fed in to the reference voltage input. Table 1 below gives a summary of the output of the comparator.

Input	Output
$v_{sig} > V_{ref}$	1
$v_{sig} < V_{ref}$	0

Table 1: Output of a Comparator ADC

Note that $v_{sig} = V_{ref}$ is not given as an option because it is assumed that the two inputs can never be exactly the same and the smallest difference between the two will be enough to cause the appropriate inequality.

The clock driving the output register is the sampling clock. On each clock pulse the 1 or 0 will be read out of the 'ADC'.

This converter, in its current form is not really practical, although it can be modified to produce a practical ADC. It does show, however, some of the basic concepts which are employed across many ADCs. The most important of these is the concept of comparing the analogue input with some known reference voltage and determining whether the input is greater than or less that this value. It is very difficult to achieve analogue to digital conversion without employing this technique at some point.

6.4. Types of Analogue to Digital Converter

We will now move on from our very basic and not very practical ADC to look at a range of converters, all of which could be used for practical analogue to digital conversion.

6.5. The Flash ADC

The first practical ADC we will look at is the flash ADC. Conceptually, it is the easiest to understand because it is a direct development of the initial idea presented above. The main issue with the initial ADC was that it only had two possible output values and so didn't have very good resolution. The flash ADC solves this problem by using a set of comparators, each with their own reference voltage equally spread across the input voltage range. As the input voltage rises, it passes more of the reference voltages and so more of the comparators are switched. The output of the comparators could be considered to be a 'bar graph' reading of the analogue input because all comparators below the current level will give a one. The final stage is a digital encoder which takes this bar graph reading and converts it to the corresponding binary value. A simple example of a flash ADC is shown in figure 4.

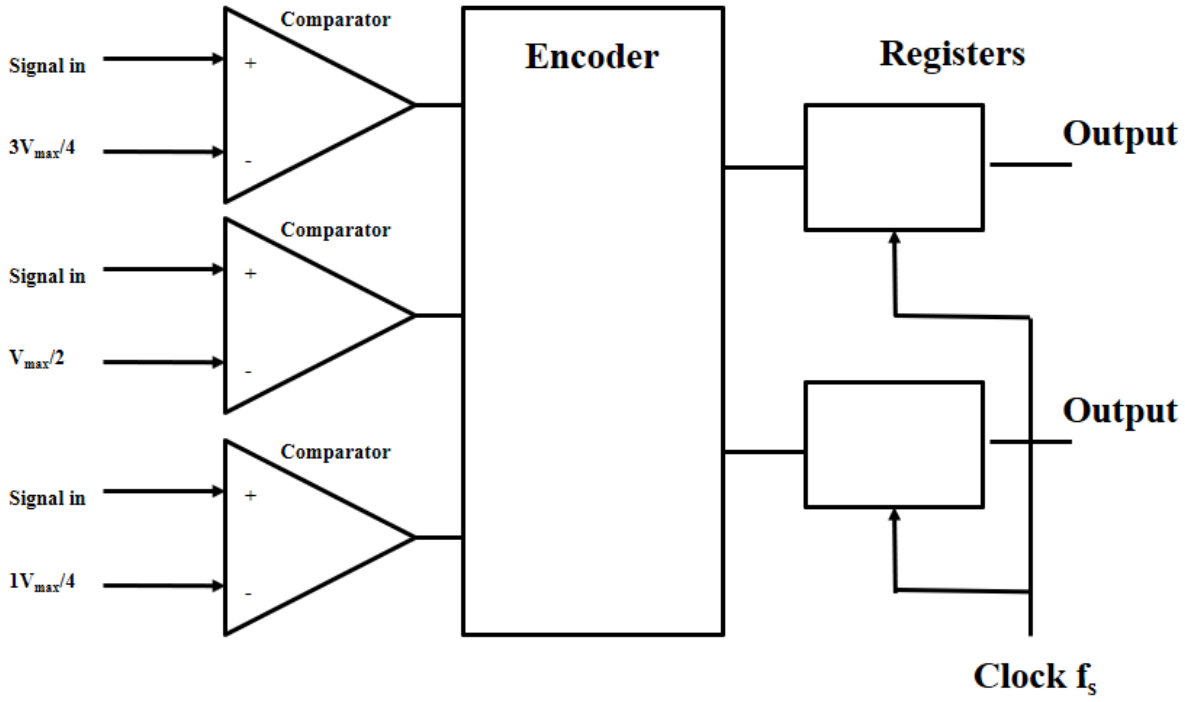

Figure 4: A Two-bit Flash ADC

The main benefit of the flash ADC is speed. For any given input value, the output is directly available. As a result conversion is very fast and so flash ADCs tend to be used where speed is of the greatest importance, such as in video ADCs.

The main issue with Flash ADCs is the requirement for numerous analogue reference voltages. All of these voltages need to be very stable and accurate to achieve stable and accurate conversion. Because they are the reference for the comparators, any error or drift in the reference voltage will cause an error in the switching voltage of that comparator which in turn will translate to an error in the value at the output. Flash ADCs do not require one reference per bit, but one reference per number, so to implement an N-bit flash ADC requires 2^N reference voltages. This means that an eight-bit flash ADC would require 255 reference voltages. As a result six or seven bits is normally about the limit for practical flash ADCs.

6.6. A Counter-Based ADC

The flash ADC worked by comparing the current input to a fixed set of reference voltages in order to determine its current value. The next two converter designs compare the current input to a reference voltage which is derived in some way from the current output. In order to achieve this, the heart of the analogue to digital converter is actually a digital to analogue converter.

An up/down counter will generate a binary output which increments or decrements on each clock pulse based on the value on an 'up/down' control input. In a counter-based ADC, this direction input is fed from the output of a comparator which compares the current input with the current output of the counter to determine whether the output should increase or decrease. If the current input is greater than the current output then the counter will be set to increase. If the current input is less than the current output then the counter will be set to decrease. A diagram of a counter-based ADC is shown in figure 5.

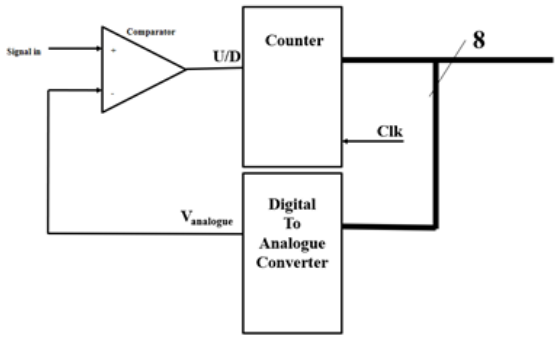

Figure 5: A Counter-Based ADC

Note that the thicker lines with the '8' above them on the output side are indicative of an eight-bit bus but shown as a single connection of clarity.

This type of ADC is reasonably simple to construct and will work satisfactorily, but has one major issue and this relates to the speed at which the output can change. We stated above that to successfully convert a signal at a certain frequency to digital we needed to sample at at-least twice that frequency. However, in the case of a counter-based ADC this doesn't quite work.

Figure 6 shows a possible analogue input to a counter-based ADC. The scale on the x-axis is 'samples' so the frequency of the input is not that high since one complete cycle contains 20 sample points. To put this in context, that would be the equivalent of a sine of about 2kHz on a CD-quality analogue to digital converter. The amplitude is given as a numerical value rather than as a voltage, and the peak amplitude of the signal is 255, so to encode the whole signal a full scale signal would be achieved on a 9-bit ADC or a -6dB(FS) reading would be achieved on a 10-bit ADC.

Figure 7 shows this signal converted to digital using a counter-based ADC, where we assume that the ADC starts at zero (it is important to state this since it works by incrementing or decrementing the previous value). Conversion of the same analogue signal is completed over 20 samples, using the function that if the input is greater than the current output, the output is incremented, if less it is decremented.

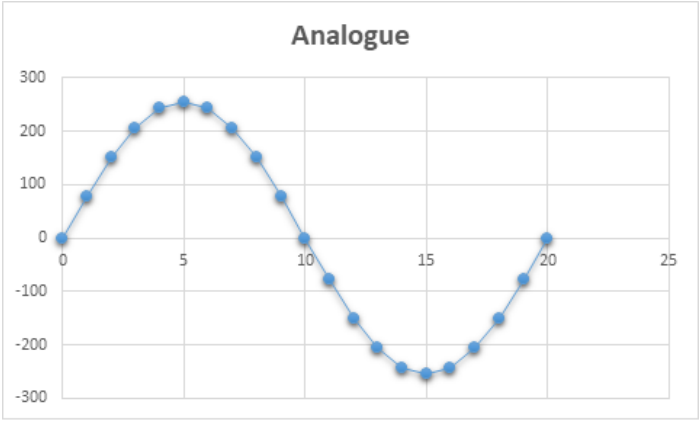

Figure 6: A possible Analogue Input to a Counter-Based ADC

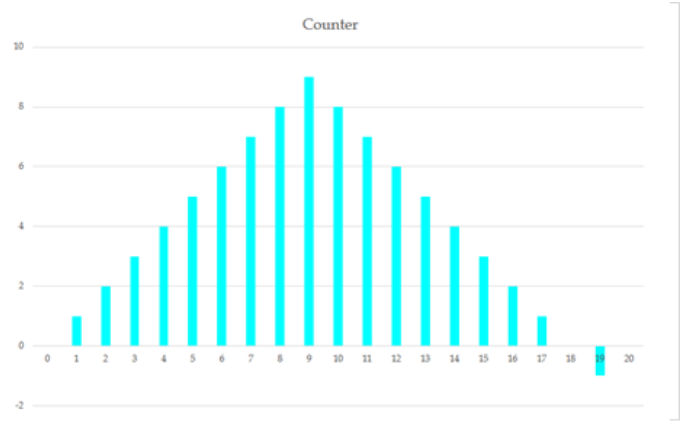

Figure 7: The Binary Output of the Counter-Based ADC

There are two things to notice about the output of the counter-based ADC in this example, firstly, the scale on the y-axis. Whereas the input signal in figure 6 has a range of ±255, the output signal only achieves a maximum positive value of 20, but also the output is now a triangle wave and only just manages to go negative before the input returns positive. This is just a numerical example so negative numbers at the output would be perfectly possible. It is possible to achieve a satisfactory output signal in this example as shown in figure 8 but notice the new scale on the x-axis showing how many samples are now being taken in one period of the input signal. If we put this in context again, using the 2kHz audio signal referred to above, we are now taking 2,000 samples in one cycle of a 2kHz signal, which means that the effective sampling frequency to achieve this would need to be $2,000 \times 2,000 = 4MHz$. This is also for an 8-bit converter and before we consider that the full audio frequency range extends to 20kHz.

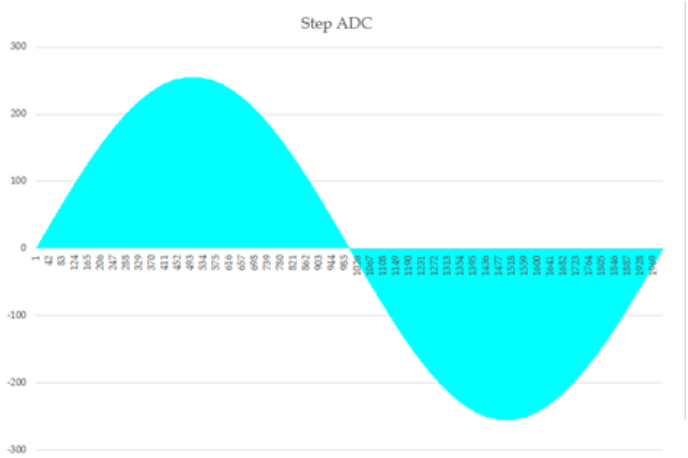

Figure 8: Counter-Based ADC Output with Higher Sampling Frequency

In effect you could think of the changing of the digital output of a counter-based ADC in the same way that you might think of the slew rate of an operational amplifier. The slew rate defines the maximum rate at which the analogue output of an op-amp can change, normally in V/µs, with a good op-amp having a slew rate in the hundreds of volts per microsecond range.

If for example we have an op-amp a slew rate of 100V/µs and compare that with an 8-bit ADC running of a 5V supply. The time taken for the output of the op-amp to change by 5V will be:

$$\frac{5}{100} = 50ns$$

If the counter-based ADC is to achieve the same rate of change then it will need to count through all 255 possible states in the same time. We can then determine the time for each state as:

$$\frac{50n}{255} \approx 196ps$$

Taking the reciprocal of this will give the clock frequency we require for the counter to keep up:

$$\frac{1}{196p} = 5.1GHz$$

By comparison, computer clock speeds top out at less than 4GHz, so this is not really feasible.

The big problem is the requirement to step through each state in turn. This requires a lot of states in a short amount of time to keep up with a sudden change in input. What is needed is a simpler converter where the output can be directly determined each time.

6.7. The Successive Approximation Analogue to Digital Converter

Arguably the most common commercially available ADC is the successive approximation converter. Certainly this is true for medium quality, medium speed applications. Again, as with the counter-based converter, it is based around comparing the current input with a digital value fed through a digital to analogue converter.

The basic principle behind the successive approximation converter is based around a form of a guessing game. In this game one person thinks of a number between 1 and 1000 say, and the other person has to guess the number as quickly as possible but the only responses can be 'higher' or 'lower'.

The most efficient way to solve this problem is to divide the range in half each time. So the first suggestion is 500 (half of the original 1,000). If the response is 'higher' then divide the range from 500 to 1,000 in half and ask if the number if higher than 750. If the response is 'lower' then do the same between 0 and 500 and suggest 250. You keep on going, halving the remainder each time until you alight on the number thought of.

This is exactly how a successive approximation analogue to digital converter works. The 'guesses' are made by setting each bit high in turn starting with the most significant bit. The 'higher' or 'lower' answer is then given from the output of a comparator. If 'higher' is returned the current bit remains set and the next most significant bit is set and tested. If 'lower' is returned then the bit is cleared before the next most significant bit is set and tested. Generally speaking, it will take one clock cycle of the ADC to set one bit. There may also be an additional overhead of one or two clock cycles in preparing the input and releasing the final output once the conversion completed. A block diagram of a successive approximation ADC is shown in figure 9.

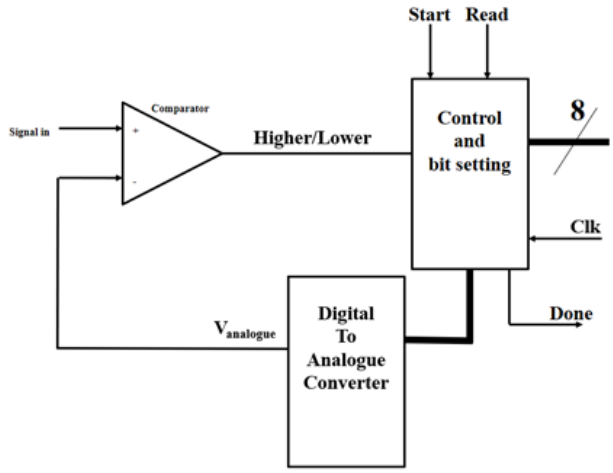

Figure 9: Successive Approximation ADC Block Diagram

If we consider a 16-bit ADC being clocked at 750kHz, we have already said that one bit is determined on each clock pulse and that there may be up to two additional clock pulses required for starting the conversion and for releasing the output. As a result, the 16-bit conversion will take 18 clock cycles to complete. This will take:

$$\frac{18}{750k} = 24\mu s$$

Again, if we take the reciprocal of this, it will give us the maximum sampling frequency which will in turn give us the maximum signal frequency which could be handled.

$$\frac{1}{24\mu} = 41.\dot{6}kHz$$

So our converter could just about manage audio frequencies up to 20kHz.

24µs may not sound like a very long time but it is possible that the analogue input level could change during that time, and this could mess up the conversion. As a result it is necessary to prevent the input to the ADC from changing whilst the conversion is taking place. To achieve this a *sample and hold* circuit is normally attached to the input of a successive approximation ADC. A sample and hold circuit is effectively an analogue latch. The analogue input value is loaded into the sample and hold circuit (normally based around a small capacitor) the input is then disconnected and the voltage maintained whilst the conversion is completed.

Successive approximation ADCs are reasonably easy to implement and have a good balance between speed of operation and output

resolution. A basic successive approximation ADC can be made up to about 16-bits in resolution and can run up to audio frequencies.

6.8. ΔΣ or ΣΔ Analogue to Digital Converters

In different places, you will find the Δ and ∑ in a different order. You may also see this referred to as a ∑Δ Modulator. All refer to the same basic circuit, or more correctly, the same basic family of circuits. In the same way that filters could be classified as a family of circuits, the same is true here. Again, similarly to filters, it is possible to get various *orders* of ∑Δ modulators.

The name comes from the mathematical symbols ∑ and Δ, where ∑ is the mathematical symbol for summation and Δ is the mathematical symbol for difference (or $\frac{d}{dt}$).

The basic idea behind a ∑Δ modulator is that a simple analogue to digital converter section is used but then the output is reconstructed in the digital domain using digital signal processing. As such, high resolutions and high speed can be achieved at moderate cost. As a result, such converters are extensively used in high end audio. There are some slight variations on the implementation of this circuit, but one form of a first order modulator is shown in figure 10 below.

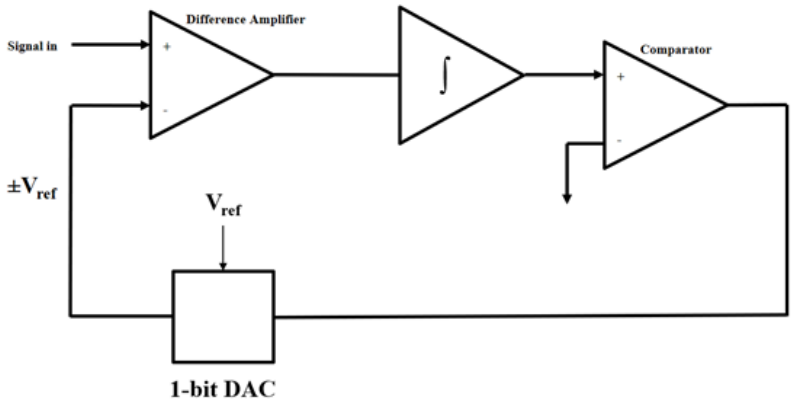

Figure 10: First Order ∑Δ Modulator Block Diagram

If we work through the modulator a section at a time, the input arrives at one terminal of a difference amplifier. Note that this is different from the comparator we have used previously. With the difference amplifier, the voltage output is a measure of the difference between the voltages at the two inputs. The second input is a reference voltage which switches between plus and minus V_{ref}, which is the output of a 1-bit digital to analogue converter. The output of the difference amplifier is then fed to an integrator circuit. The greater the voltage fed into this circuit, the faster the output of the integrator changes. If the output of the difference amplifier is positive, the output of the integrator raises, if the output of the difference amplifier is negative, the integrator reduces.

The integrator output feeds into a comparator which will convert the smoothly rising and falling output of the integrator into a square pulse. The faster the rise and fall as a result of the bigger difference at the output of the difference amplifier will cause a faster switching of the output of the comparator.

The output of the comparator is the output of the modulator which is also fed back to a 1-bit digital to analogue converter which in turn feeds back in to the difference amplifier.

Effectively, the output of the modulator is a form of voltage to frequency circuit. At a first glance, the output may not seem to bear any relation to the analogue input waveform but it is possible to reconstruct a standard, what is known as a *pulse code modulated*, digital output. This is achieved using digital signal processing on the output of the modulator.

6.9. Oversampling

One technique which is commonly used in analogue to digital conversion is *oversampling*. This is sampling the input signal at a rate which is (normally) an integer multiple of the desired sampling rate.

If we consider a signal which is sampled at 8-bit resolution, once every second (in order to make the maths easier), then the total amount of information we have regarding that signal could be expressed as 8 bits/second or 256 levels/second. If we take this reading and get a result of 64, then that tells us that the input signal level is $\frac{64 \pm 0.5}{256}$ for that second.

Suppose we were to change the sampling rate so that we took two readings per second, and that the first was 64 but the second was 65. We could now refine our initial reading by averaging the two values:

$$\frac{64+65}{2} = 64.5$$

Alternatively, we could avoid the division by just adding the two values together to give an output of 129 across the whole sample period.

However, as you know from maths, whatever we do to one side of an equation we must do to the other in order to keep everything balanced. If the original level was 64 out of 256 levels in that second then the new level cannot be 129 out of 256, but must be 129 out of 512 levels in that second.

So, by sampling twice as fast, we have increased the output resolution of the converter by one bit.

This process can be continued. Oversampling by a factor of 16 will equate to a resolution increase of 4-bits ($2^4 = 16$) and so on.

Because we are sampling the same signal, the various values read are said to be *coherent*. This means that when we come to determine the average value we can just add the various values together as we have done above.

6.10. Noise and Oversampling

We have said above that the noise in a sampled signal is effectively equivalent to the least significant bit. Something rather interesting happens to the noise in a signal when you oversample however. Because noise is by definition random, you cannot average the noise signal just by adding all the individual signals together. To find the total noise in the final signal you must use Pythagoras to find the average:

$$Noise_{total} = \sqrt{n_1^2 + n_2^2 + \cdots}$$

If the noise level remains constant ($n_1 = n_2$) then oversampling by a factor of 2 means that:

$$Noise_{total} = \sqrt{2} \times n_1$$

Whereas the signal is:

$$Signal_{total} = 2s_1$$

So oversampling by a factor of two has also improved the signal to noise ratio by a factor of $\sqrt{2}$. Hopefully, by extension, you can see that oversampling to a greater level will continue to improve the signal to noise ratio.

6.11. Nyquist and Oversampling

The Nyquist limit still applies to an oversampled signal but an important benefit results from the oversampling process. We have already stated that the maximum signal frequency which should be allowed to enter the ADC to avoid aliasing is less than half the sampling frequency, that is:

$f_{sig(max)} \leq \frac{f_{conv}}{2}$

If we oversample, we make the sampling frequency higher by a certain factor. This, in turn, means that the Nyquist frequency is made higher by the same frequency. This has the benefit that any anti-aliasing filter used can be of a much lower order whilst still removing stray input frequencies to a suitable degree.

For example, if we think of an audio signal which needs to be sampled at 44.1kHz. We have already seen that sampling at this frequency requires a very steep anti-aliasing filter to remove unwanted frequencies whilst keeping wanted audio signal frequencies (up to 20kHz). If we now oversample this at 16 times oversampling, the new sample rate is:

$16 \times 44,100 = 705,600 = 705.6ksps$ (kilosamples per second)

The Nyquist limit still applies but based on this new, elevated, sampling frequency:

$f_{sig(max)} \leq \frac{705.6k}{2} \leq 375.3kHz$

This is over four octaves above 20kHz. Very little audio will be likely to be picked up and fed to the input to the ADC at those sorts of frequencies by any microphone or input electronics because it is so far output of the frequency range of interest, but in addition, a simple 2nd or 3rd order filter would reduce signal which would get through by between 48dB and 60dB which should be enough to avoid problems.

A Filter Aside

The way we, as humans, hear frequency is as a *geometric series*. This means that multiples of a frequency sound like equal divisions. As a result, two multiples are important. The first is a doubling (halving) of frequency which is known as the *octave* and will be familiar to anyone who plays a music instrument. The second is the factor of 10, which is known as the *decade*.

The way in which a filter works is by defining the amount of change in signal level applied for each octave or decade change in signal frequency. Capacitors and inductors are the important components in making filters and, broadly speaking, each additional capacitor or inductor added to the filter circuit increases the amount by which the signal level changes in each octave. Note that this is a very general and approximate description of how filters work. You will cover the theory in much greater detail in the analogue electronics module in teaching block 2. The number of capacitors of inductors used is known as the *order* of the filter and the amount of reduction which results is given in table 2 below.

Filter Order	Reduction per octave	Reduction per decade
1	6dB	20dB
2	12dB	40dB
3	18dB	60dB
4	24dB	80dB
5	30dB	100dB

Table 2: Table of Filter Attenuations

6.12. Summary

In the session we have looked at the theory behind analogue to digital conversion. We have looked at some of the ways in which analogue to digital conversion can be achieved and some of the more modern developments, particularly oversampling.

More modern systems may also employ further digital signal processing on the output of the ADC to further enhance the quality and perceived resolution of the input signal.

7. Pulse Width Modulation

As we have said previously, when discussing analogue to digital conversion, analogue signals are continuous in time and can adopt any value in a given range. An example of a section of an analogue signal is shown in figure 1.

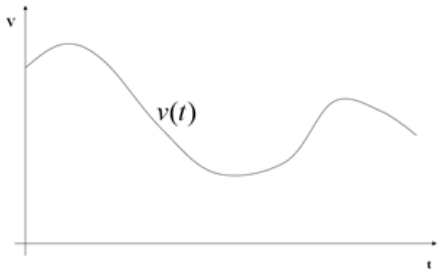

Figure 1: A Generalised Analogue Signal

Although the signal is varying in level over time, if we zoom right in, the signal can be approximated to a straight line as shown in figure 2.

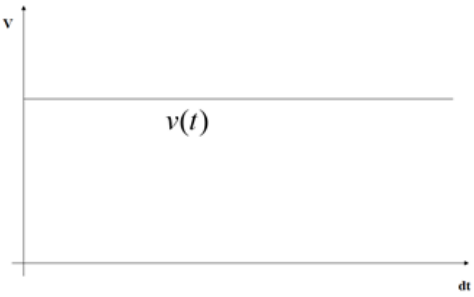

Figure 2: A Close-up of the Analogue Signal

You will notice that the scale on the x-axis has now changed to *dt*, so we are very much zoomed in.

7.1. Analysis of the Average Signal Voltage

We now need to do a little bit of maths to understand this signal more closely. If we consider our generalised signal over the time, t, the average voltage can be expressed as:

$$\overline{V} = \frac{1}{t}\int_0^t v(t)dt$$

It is important to remember here that the bar over the V indicates the average value- not logic inversion.

Because we have very much zoomed in, we can also consider our function over this very much zoomed-in timeframe:

$$\overline{V} = \frac{1}{dt}\int_0^{dt} V dt$$

Here, the period of analysis has become dt, as has the period of the integral. Also, because we have zoomed in to such an extent that V is now a constant, the function of voltage has changed to a constant voltage. Because this is now a constant, we can treat is as such and take it outside of the integration:

$$\overline{V} = \frac{V}{dt}\int_0^{dt} dt$$

If we now solve this integral we get:

$$\overline{V} = \frac{V}{dt}[t]_0^{dt}$$

We can now substitute the limits to give:

$$\overline{V} = \frac{V}{dt}[dt - 0]$$

$$\therefore \overline{V} = \frac{V \times dt}{dt} = V$$

This may seem like an unnecessarily involved and mathematically stress-inducing way of showing what we all know, that the average voltage of a constant voltage is that voltage, but it is helpful to develop this as the base for our further discussion.

If, rather than just expressing the voltage as V, we express it in terms of the maximum possible voltage, then if the voltage is half of the maximum we get:

$$\overline{V} = \frac{V_{max}}{2}$$

7.2. Analysis of a Changing Signal Voltage

Now let us suppose that we have a signal voltage which changes half way through our time period, such that is has the voltage V_1 in the first half and V_2 in the second, as shown in figure 3.

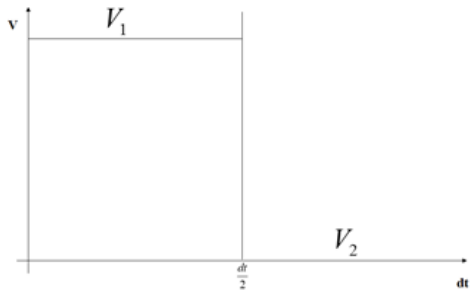

Figure 3: A Signal which Changes at $\frac{dt}{2}$

In each of the two periods the voltage is constant, so the analysis can proceed almost as before. The only change is that we need to analyse the signal in two parts, the first for V_1 and the second for V_2.

$$\bar{V} = \frac{1}{dt}\int_0^{\left(\frac{dt}{2}\right)} V_1 dt + \int_{\left(\frac{dt}{2}\right)}^{dt} V_2 dt$$

If we solve these integrals, this gives us:

$$\bar{V} = \frac{1}{dt}\left([V_1[t]]_0^{\frac{dt}{2}} + V_2[t]_{\frac{dt}{2}}^{dt}\right)$$

Although this initially seems more complex because of the number of elements, we can treat each individual element as we did previously.

$$\bar{V} = \frac{1}{dt}\left(V_1[\frac{dt}{2} - 0] + V_2[dt - \frac{dt}{2}]\right)$$

$$= \frac{1}{dt}\left(\frac{V_1 dt}{2} + \frac{V_2 dt}{2}\right)$$

If we now take out the common factors we get:

$$\bar{V} = \frac{dt}{2dt}\left(V_1 + V_2\right)$$

$$= \frac{V_1 + V_2}{2}$$

If we now set:

$$V_2 = 0V$$

We get

$$\bar{V} = \frac{V_1}{2}$$

If V_1 is our maximum voltage, this then gives:

$$\bar{V} = \frac{V_{max}}{2}$$

This is the same equation we had earlier when we halved the voltage for the whole time. As a result, we appear to have two alternatives: Firstly, we can vary the voltage between a minimum and maximum but keep it constant over the whole time and this will give a varying average voltage; secondly, we can switch the voltage between the maximum and minimum for a certain proportion of

the time and this will give the same average.

Does it work for any proportion?

If we set the proportion of the time for which the signal is at V_1 to:

$$t_{V_1} = \frac{a}{n}$$

Then our integration becomes:

$$\bar{V} = \frac{1}{dt}\int_0^{\left(\frac{adt}{n}\right)} V_1 dt + \int_{\left(\frac{adt}{n}\right)}^{dt} V_2 dt$$

Here we have V_1 from the start until the time $\frac{adt}{n}$ and the voltage V_2 from time $\frac{adt}{n}$ to dt. Again, the maths continues as before:

$$\bar{V} = \frac{1}{dt}\left([V_1[t]]_0^{\frac{adt}{n}} + V_2[t]_{\frac{adt}{n}}^{dt}\right)$$

$$\bar{V} = \frac{1}{dt}\left(V_1[\frac{adt}{n} - 0] + V_2[dt - \frac{adt}{n}]\right)$$

$$= \frac{1}{dt}\left(\frac{aV_1 dt}{n} + \frac{(1-a)V_2 dt}{n}\right)$$

$$= \frac{1}{n}\left(aV_1 + (1-a)V_2\right)$$

This looks rather complicated as is but if we now apply the simplifications we used before that:

$$V_2 = 0; V_1 = V_{max}$$

Then

$$\bar{V} = \frac{aV_{max}}{n}$$

Notice here that the average voltage is of the maximum and that this was also the proportion of the time window for which the voltage was set to its maximum. We have a system here where we can vary the time in any given period for which the voltage is at its maximum and this will average to that proportion of the maximum voltage. As a result, we can trade time and voltage both giving the same varying average over a given time period.

This is a useful approximation because digital systems are good a supplying one of two set voltages on a pin and switching between these very quickly and can do this much better than supplying a constantly varying output voltage.

7.3. Definitions

By constantly varying the proportion of the time for which the signal is high, the output voltage can be constantly varied between some minimum and maximum values. Because this requires the width of the voltage pulse in any given widow to be modulated, this is known as *pulse width modulation*.

As with analogue to digital conversion, there are limits on the timing window which can be used. This is because the constant average voltage assumption can only be made when the analogue voltage is constant over the timing window. In general, the frequency of modulation needs to be 'much higher' than the analogue signal frequency. Whilst there is not a formal limit in the same way as with Nyquist, in many places at least a 10:1 ratio of modulation frequency to signal frequency is recommended for accurate reproduction.

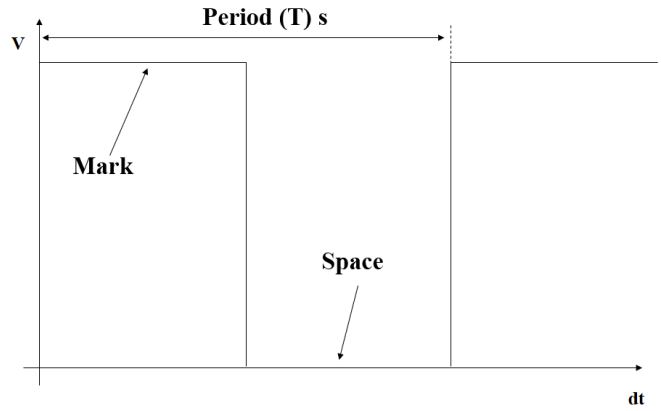

Figure 4: Pulse Width Modulated Signal Definitions

Figure 4 shows a one and a half repeats of a pulse width modulated signal with definitions marked.

As with all repeating signals, the time for one complete cycle is known as the *period* of the signal and is given the symbol T.

The high output is known as the *mark* and the low output is known as the *space*. This gives rise to a number of definitions:

Mark Space Ratio

$\frac{t_{mark}}{t_{space}}$

This expresses the high time to low time as a fraction.

Duty Cycle

$D=\frac{t_{mark}}{T}$

This is an alternative expression for the proportion of the time for which the signal is high, defining the high time compared to the whole period. You will notice that this is actually the same expression we used in the initial development of the theory of the pulse width modulator in the previous section:

$D=\frac{t_{mark}}{T} \equiv \frac{a}{n}$

If

$V_1=V_{mark}$

$V_2=V_{space}$

Then

$\overline{V}=DV_{mark} : V_{space}=0V$

Or

$\overline{V}=DV_{mark} +\left(1-D \right)V_{space}: V_{space} \neq 0V$

7.4. The Benefits of Pulse Width Modulation

There are two main benefits of pulse width modulation. The first has already been noted, digital systems work based around two values on a wire or pin. Pulse width modulation works by approximating a continually varying voltage only using two fixed voltage values. As such, PWM is ideally suited to use in digital systems.

The second benefit may not be immediately obvious. Suppose we set up a simple circuit as shown in figure 5 below.

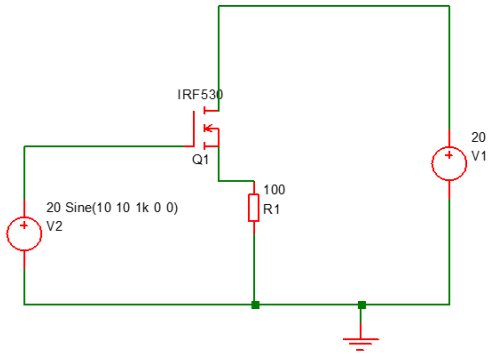

Figure 5: A Transistor-Based Source Follower

The circuit is built around a MOS transistor, Q_1, with R_1 acting as the load. Hopefully, you are familiar with the operation of such circuits but briefly, V_1 is the DC power supply to the circuit and V_2 is the input signal, which is set as a sine wave of 20V peak. The output voltage across the resistor is the transistor gate voltage (V_2) minus the threshold voltage of Q_1, which is the voltage required on the gate to turn the transistor on.

As V_2 increases the voltage across R_1 increases, current flows through R_1 and so the power in R_1 increases according to the equations:

$$I_{R_{1}}=\frac{V_{R_{1}}}{R_{1}}$$

$$P_{R_{1}}=\frac{V^2_{R_{1}}}{R_{1}}$$

The current through the transistor Q_1 will be the same as the current through R_1 according to Kirchhoff's current law, because they are in series. The voltage across Q_1 is:

$$V_{Q_{1}}=20-V_{R_{1}}$$

Because there is a voltage drop across Q_1 and current flowing in Q_1, power is dissipated in Q_1:

$$P_{Q_{1}}=V_{Q_{1}} \times I_{R_{1}}$$

As $I_{R_{1}} \Rightarrow 0A$, $P_{Q_{1}} \Rightarrow 0W$.

As $V_{Q_{1}} \Rightarrow 0V$, $P_{Q_{1}} \Rightarrow 0W$. This occurs when $V_{R_{1}} \Rightarrow V_2$.

As a result, the power dissipation in Q_1 can be plotted against the signal voltage, V_2 as shown in figure 6.

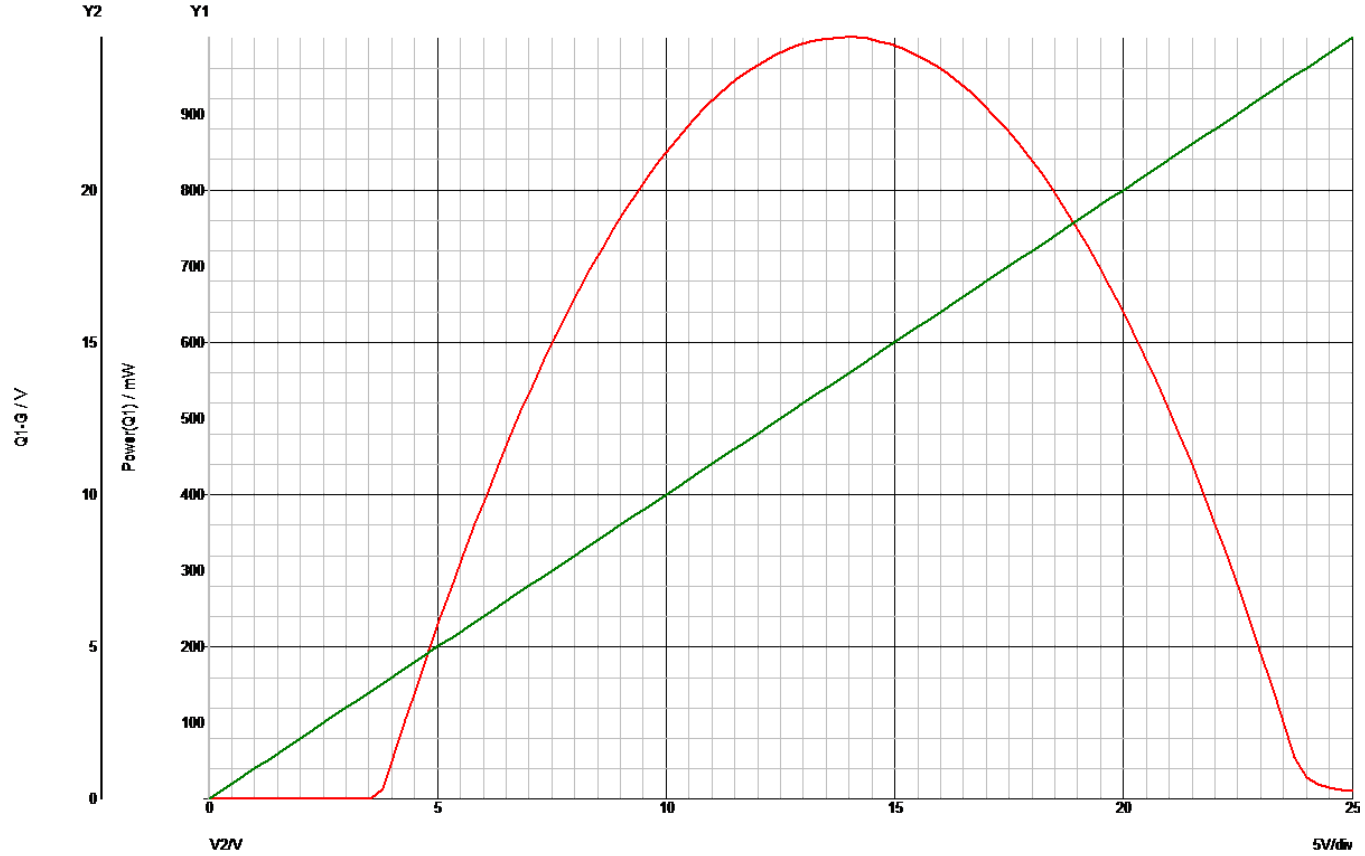

Figure 6: Power in the Transistor Plotted Against Gate Voltage

You can see that initially, there is now power dissipated until the threshold voltage of the transistor is passed, then power increases until it reaches its maximum when half the supply voltage is dropped across the transistor. It then reduces until almost no power is dissipated when all the supply voltage is across the resistor. This may not be an issue in low power systems but if driving larger loads such as loudspeakers from digital sources, there are efficiency gains to be made by driving the speaker using a pulse width modulated output.

7.5. Pulse Width Modulation Issues

Whilst the previous section may make pulse width modulation sound wonderful and the obvious solution in all situations, there are also issues.

Continuing the previous discussion, if we use pulse width modulation to drive a loudspeaker, the modulation period would need to be greater than 200kHz to cover the whole audio frequency range. Loudspeakers are large inductive loads and switching inductors on and off in this way can lead to issues with high voltages which can damage the power transistors if not correctly handled.

The second issue is that of distortion. We are synthesising an analogue signal using a fast changing square wave. This switching can lead to additional harmonics being introduced to the output which were not in the original signal. If we use the circuit shown in figure 7 by way of an example.

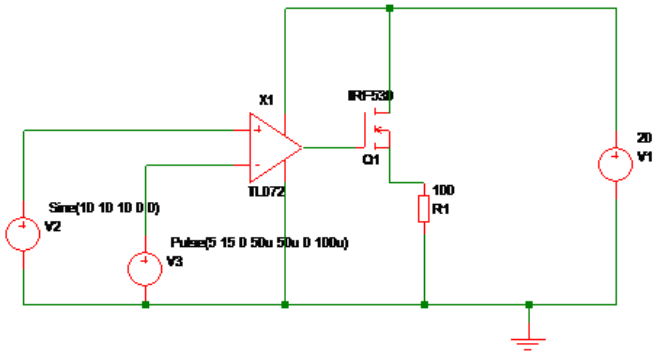

Figure 7: A Simple Analogue Pulse Width Modulator

The op-amp, X_1, is configured as a comparator, with a sine wave fed into the non-inverting input and a triangle wave fed into the inverting input. The circuit is configured as a single supply circuit because the transistor operates best in this format. The sine and triangle voltage sources are set up so that the waveforms are offset by 10V. The principle of operation is reasonably simple. If the sine wave is more positive than the triangle wave the op-amp output is 20V, the transistor is fully on and so the full voltage is across the resistor. If the triangle wave is greater than sine, the op-amp output is fully negative, the transistor is off and so there is no voltage across the resistor.

In this instance, we are not so interested in the actual output of the circuit, but in a frequency analysis of both output and input, as shown in figure 8.

A sine should contain just one frequency, and the frequency of the sine wave itself. Here, we can see a small amount of other frequency components showing in the input waveform (the red trace). If you look at the scale on the y-axis (and notice that this is a log scale), you will see that there is a big spike at the fundamental on the y-axis, but that the rest of the frequencies are very low (of the order of 10pV). Looking at the output, however, you will notice that the frequencies are nearer 10mV with noticeable spikes at multiples of 10kHz. Using the pulse width modulator has noticeably increased the levels of other frequencies in the output signal- and as a result the level of distortion in the signal.

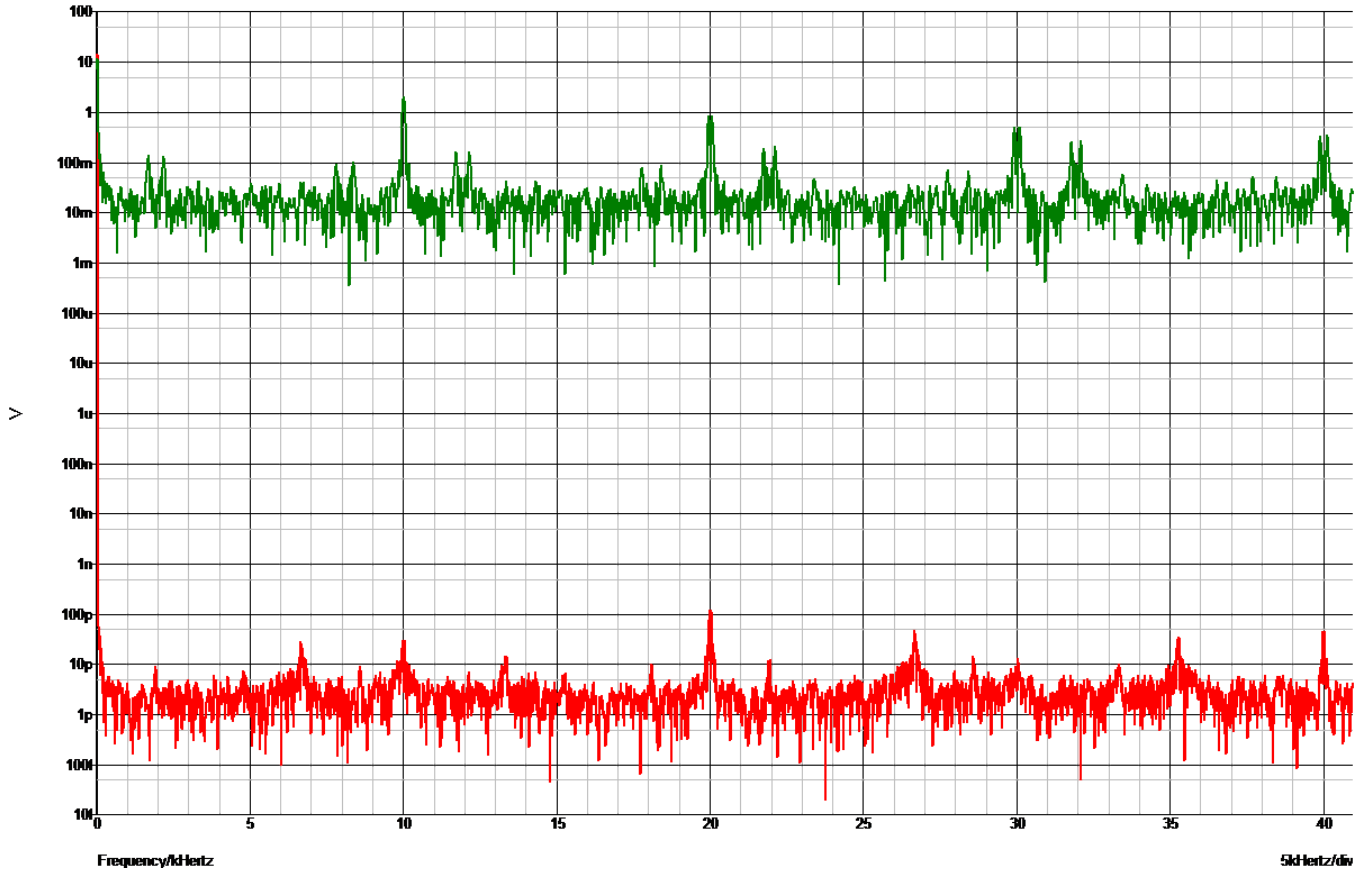

Figure 8: A Frequency Plot of the Input and Output of the PWM.

7.6. Generating a PWM Digitally

Counters are the basic building block of a digital PWM. There are several ways they can be set up and the best or preferred method might depend on the system being used. In most cases, the basic counter (or timer on a microcontroller) would be set to roll over at the end of each period. At the start of the period, the PWM output pin would be set high. The PWM 'value' would then be loaded into a suitable memory. As the count proceeds, the output of the counter is compared with the value in memory, when the counter value passes the stored value, the pin is reset and remains low until the end of the period. At the end of the period, the stored value can be updated as required. Care normally needs to be taken to ensure that the stored value is not updated mid-count in case this causes the switching point to be missed. In many cases, two registers are used. The first is loaded into by the user and can be loaded at any time. This is then transferred to the working register only at the appropriate time in the cycle and this transfer is controlled by the PWM control system.

7.7. Summary

In this session we have looked at the theory behind pulse width modulation, how it works and why it is used, why it is a beneficial way of converting outputs from digital systems such as microcontrollers to analogue, and the benefits and issues.

8. Universal Synchronous/Asynchronous Transmitter Receiver

In many cases, sending data in serial, as opposed to parallel, form is more efficient. This connection may either be synchronous, in which case the data is sent on one wire and the associated clock on a second wire, or it may be asynchronous, in which case the data is transmitted, but the clock signal is generated locally at the receiver.

Both types of transmission are used and have their benefits in different situations. On the PIC the transmission and reception of both types of data is handled by the *Universal Synchronous/Asynchronous Transmitter Receiver* module or USART, although a second *Master Synchronous Serial Peripheral Module* (MSSP) is also available and is used for some protocols.

In this session we will look at the theory behind both synchronous and asynchronous serial transmission, focussing on I^2C and RS232. We will also look at the USART and MSSP modules on the PIC and see how PIC implements serial transmission and reception.

8.1. Serial Data-An Overview

In order to send data over a serial link, the data byte, or bytes, must first be converted from parallel to serial form. They must then be sent out over the link and, at the receiver end, converted back from serial to parallel.

In converting from parallel to serial, a decision must be made as to the order in which the bits are sent. On the data bus, it is clear that bit D_7 is the most significant bit and bit D_0 the least significant bit. On a serial link, however, two possibilities exist. The most significant bit could be sent first, or the least significant bit could be sent first. To a certain extent it does not matter which it is as long as both the transmitting and receiving ends are the same.

The next requirement is that both ends of the link must expect the same length data blocks. If the transmitter is sending out 8-bit blocks but the receiver is expecting 7-bit blocks a problem will result because the received data will not be formatted correctly.

8.2. Clocking

As with any sequential system, the clock is central to all that happens on the serial link. The data will be clocked out of the transmitter at a certain number of bits every second, normally called the *baud rate* and the receiver must be set up to receive that number of bits per second. Again, if this is not the case, problems will result.

This leads us to the consideration of how the clock signal reaches the receiver. Here, there are three basic options.

Synchronous Serial Link

The first option is to run the link in synchronous mode. This means that two connections are used (plus ground). The first will carry the data being transmitted and the second, the clock signal. In this situation it doesn't matter so much what clock speed is used, provided the link and the receiver can run that fast because the clocking is all controlled from the transmitter.

Asynchronous Serial Link

The second option is to run the transmitter and receiver in asynchronous mode. This means that the clock is not transmitted alongside the data. As a result, care must be taken to ensure that the clocks at both the transmitter and receiver ends are running at the same speed so that the two halves stay in step. We will look shortly at how that is achieved. This has the benefit that it saves a wire because only the data and ground are required. The down side is that more care needs to be taken to package the data up so that the receiver can recognise that data is being sent, and decode it appropriately.

Embedded Clock Serial Link

There is a third way of encoding a serial link. This is, in some way to encode the clock within the data stream. This is commonly used where long sets of data are sent and where those data sets may contain large numbers of consecutive ones or zeros. Sending large numbers of the same value one after another on an asynchronous link could cause an issue. If the clocks at the transmit and receive ends drift slightly, it might not be clear whether 14 or 15 zeros have been sent. If the clock is in some way embedded within the data stream this helps to keep the receiver in time, but without the need for a separate clock wire.

This sort of encoding is commonly referred to a *Manchester Coding* although there are many variations and only one is strictly Manchester coding.

By way of an example, to explain how this works, we will look at a timing signal used in film and TV, known as SMPTE Timecode, which standard for Society of Motion Picture and Television Engineers. If different devices are being used on a film set to record the pictures and the sound, it is important that these stay in time with each other. Also, if one is fast-forwarded, or rewound, it is necessary for the other to track it. For this reason SMPTE Timecode was developed. It is a digital way of encoding time, down to the nearest frame number and is an embedded clock serial signal.

We don't need to worry too much here about how the data is structured, what we are more interested in is the way that the clock signal is embedded.

With SMPTE, the actual value transmitted is not really important. What is important are the changes of state. Every time a '1' is transmitted, the value sent is the opposite of what was just sent:

$$d_{n+1}=\overline{d_n}$$

If a zero is to be sent, then the output is switched half way through the bit-period as well. Effectively, we has a simple form of frequency modulation as a result:

$$f_c \rightarrow 1$$

$$2f_c \rightarrow 0$$

Because the changes in output for each bit transmitted mean that the clock frequency is readily available at the receiver, is doesn't matter what frequency the clock actually runs at. As a result it can be sped up or slowed down and the receiving device will still be able to track.

Figure 1 shows an example of a length of SMPTE Timecode with bit values included.

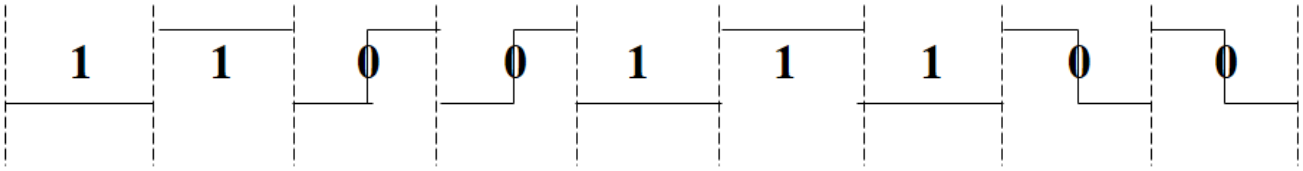

Figure 1: A Length of SMPTE Timecode with Values.

8.3. Simplicity of Use and Simplicity of Implementation

The various serial standards around are designed, for the most part, for connecting different pieces of equipment together. As such, anything could be being connected using the serial link. The desire has always been to make the connection simple to use and robust. This can be seen in the case of more modern standards such as USB. USB is designed to be *hot plugged*, that is, you can safely plug and unplug devices without turning the power off. This is achieved by designing the connector to ensure that the ground connection is the first to be made and the last to release. Although many people might plug in other, older, standards without powering down first, technically this could lead to damage due to electrostatic discharge as the connection is made. You will also be familiar with the way in which the computer will automatically set about determining what has been plugged in to the USB port. It will then respond in the appropriate manner, assigning a drive letter, or downloading a printer driver for example. Unseen by the user, quite a lot of data has to pass back and forth between the device and the PC to enable this to happen. This requires a standard form for presenting this information and a standard set of information to be stored and supplied. As a user, this is helpful, because you purchase your new memory stick, plug it in, it is identified and given the appropriate name in the file explorer. The system will also determine at what speed to run the USB link according to the capabilities of the device.

As a user, this makes life simple but from an implementation perspective, this makes designing and using a USB link more complex. This is generally the case, the more set-up is automated to make life simpler for the user, the more complex the link becomes from the engineer's perspective when trying to implement it. It also tends to be the case that newer protocols will try to automate more of these processes and so will be simpler to use but more complex to design whereas older protocols will be more basic to implement but at the expense of requiring more intervention from the user to set them up correctly.

8.4. Synchronous Serial Data

A number of protocols have been developed to allow for efficient communication between devices on a PCB. In many cases using a parallel bus arrangement is inefficient in terms of board space, so serial links have been developed. Two examples are SPI, or *Serial Peripheral Interface*, and I^2C or *Inter-integrated Circuit*. It is the latter we will look at briefly here.

I^2C is actually a proprietary standard developed by Philips and now 'owned' by NXP. The specifications are freely available, and it can be implemented without the need to pay a licence fee or any sort of royalty. However, I^2C is a serial bus arrangement and each device on the bus must have an address in the same way that all devices on a normal computer data bus have an address. These addresses are reserved and to ensure interoperability between all devices on a bus, manufacturers and implementers need to apply to NXP for an address in order to implement I^2C on their system. I^2C can run in two address modes, 7-bit and 10-bit.

I^2C is a master/slave arrangement with the ability to connect multiple devices to a single bus. The bus consists of two wires, the serial data *SDA* and the serial clock *SCK*. The clock is generated by the master and data is transferred along the data line in sync with the clock.

I^2C operates on an 8-bit data format, but communications can be several bytes in length and can be unidirectional or bi-directional as controlled by the master. The serial clock can run at varying frequencies and, in theory, could be slowed down for different transmissions because data is clock in sync. Unusually, the clock is not always transmitting but is only active during data transmissions.

Both the clock and data lines are designed to operate as a 'wired-OR' where each device has the capability to pull the line low or to go high-impedance. An external resistor must be used to pull the line high. This means that simultaneous activity on the line will not damage the ICs and also that any device can be responsible for controlling the value on either the clock or data lines. Because devices can only pull a line low, all definite communication must be in terms of a '0' on a line, for example initiating a communication, or providing a response.

Although we might think in terms of a single maser device on a system, I^2C is designed to operate with multiple master devices on a single bus if required. As a result two devices could try to initiate communication at the same time. Should this occur, a *collision* will result. Devices need to be able to detect such an occurrence and respond accordingly. For a master, this will require waiting and attempting to restart communication. As a slave, collision detection is still important because it could mean that data received might be corrupted

Communication Overview

Initially the clock is idle and is held high by the pull-up resistor. The data line is then taken low to indicate a start of communication (known as a START condition).

All devices will then be monitoring the line for the next transmission which will be the address of the device with which the master wants to communicate (or a *global request* to all attached devices). The next byte will contain 7-bits of address information with the final bit indicating whether a read (from the slave) or write (to the slave) is being initiated. If 10-bit addressing is being used, a further three address bits will then need to be transmitted.

The device at that address will acknowledge by pulling the data line low on the ninth clock pulse. This tells the master device it has received and recognised the address and is ready. In the case of a write, the master will then start to write data to the line in bytes. After each byte it will wait for an acknowledge back from the slave device to indicate that data has been successfully received and removed from the buffer. In the case of a read, the slave will start placing data on the line, waiting for an acknowledgement from the master after each byte.

The protocol is designed to be as flexible as possible, so it is designed to allow for devices which take time to process incoming data. As a result, it is possible for the receiving device to cause a 'pause' between bytes. It does this by holding the clock line low between the eighth and ninth clocks. The master will then wait for the clock to be released before sending the ninth clock and looking for the acknowledge.

Once the read or write is complete, a STOP is sent, and the clock line will then be released high.

Bidirectional Communications

Because the first byte transmitted contains a *direction bit*. Communications in theory could only be unidirectional. This would require the master to release the channel using a STOP procedure and then initiate a new communication. In multi-master situations this may not be ideal because another master may jump in and take the line. As a result, a process known as *repeated START* is used. Here, the master can initiate a new communication without stopping the previous one. This would be used for example if information needs to be transferred both to and from the slave. The first communication might write data, a repeated start is then issued with the direction bit set to read and the required data read back.

9. RS232

There are many different asynchronous serial standards, one of the earliest being RS232. Officially, it is now known as TIA/EIA232 but everyone still calls it RS232. RS stands for *Recommended Standard*. It was originally developed in the 1960s and as such is technically a reasonably simple standard to implement.

In order to understand some of the terminology used, it is helpful to consider for a moment the original purpose for which RS232 was developed and the environment into which it was introduced. In the 1960s digital technology was very much in its infancy but was starting to be used for devices such as 'teleprinters' which were effectively typewriters but with the keyboard half and the printing half at opposite ends of a telephone line. In the days before broadband, connection to the telephone line was made via a *modem* which stands for modulator demodulator, which took the digital signal and converted it into two frequencies to send down the telephone line. RS232 was developed as a standard connection between the keyboard and the modem. In RS232 terminology, the keyboard is known as *Data Terminal Equipment* or *DTE* and the modem as *Data Communications Equipment* or *DCE*.

The standard is a *point-to-point* standard, which means that an RS232 connection is designed to connect one DTE device to one DCE device. It allows for bidirectional communication either in *full-duplex* or *half-duplex* modes. Full duplex means that data can be sent and received at the same time and will require the link to have separate send and receive data connections. Half duplex means that communication can happen in both directions but not at the same time. This would be required over the telephone system since only a single voice circuit is available, so communication must be organised to ensure that they signal gets through cleanly.

With a modem link, the connection along the telephone line would need to be established before data could be sent. This was achieved through the use of a number of control signals which ensured that the channel was correctly set and ready for the transmission to occur and would also manage direction control in a half-duplex situation.

9.1. RS232 Connections

The RS232 specification officially defines a 25-pin D connector as the required connector for an RS232 link. What is somewhat surprising is that 22 of these pins are officially used. The signal names and pin numbers on the 25-pin connector are listed in table 1 below. Some of the signals are discussed in the table, the more common ones are discussed in more detail below.

Pin	Name	DTE I/O	Notes
1	Frame Ground*		Effectively the 'earth' connection. This is different from the signal ground on pin 7 below.
2	Transmit Data*	Output	The serial data signal from the DTE
3	Receive Data*	Input	Data arriving at the DTE
4	Request to Send*	Output	Handshaking
5	Clear to Send*	Input	Handshaking
6	Data Set Ready*	Input	Handshaking
7	Signal Ground*		
8	Data Carrier Detect*	Input	Handshaking
9,10	Reserved		
11	Unassigned		
12	2nd Carrier Detect	Input	Second Circuit Handshaking
13	2nd Clear to Send	Input	Second Circuit Handshaking
14	2nd Transmit Data	Output	Second Circuit Data Output
15	Transmit Clock	Output	The transmit clock output wire. Note that although almost always used in asynchronous mode today, RS232 can run either asynchronous or synchronous
16	2nd Receive Data	Input	Second Circuit Data Input
17	Receive Clock	Input	The clock for synchronous data input
18	Local Loopback	Output	Used for testing the link to check for breakages
19	2nd Request to Send	Output	Second Circuit Handshaking
20	Data Terminal Ready*	Output	Handshaking
21	Remote Loopback	Output	Loopback connection requested from the remote end
22	Ring Indicator*	Input	Detects the telephone line 'ringing' so that the system can respond to an incoming message.

23	Data Rate Select		Designed to be used to select 'high' or 'low' speed connection.
24	Transmit Clock	Output	
25	Test Mode	Input	Used to enter line test mode

Table 1: RS232 D25 Pin Assignments

* Required Signals

You will notice a number of things looking at the full signal listing. The standard does appear to be somewhat over-engineered with a large number of control and data signals. You will notice that there is provision for two data channels, both with transmit and receive capability. Although RS232 is generally thought of as an asynchronous protocol, you will notice that it was originally designed to run either in asynchronous or synchronous modes, with transmit and receive clock lines. There was originally provision for dual speed operation, using the data rate select pin and a number of system test pins are included to allow for line tests.

Today many of these signals are never used, and others which have to be used for compatibility purposes are connected in such a way as to 'con' the transmitter into thinking that appropriate handshaking responses have been received.

Because not all of the signals are required, this leaves many of the 25 pins on the original connector unconnected. As a result, in most cases, an RS232 connection is now made using a 9-pin D connector, with the pin assignments as shown in table 2 below.

Pin	**Name**	**DTE I/O**	**Notes**
Shell	Frame Ground		Effectively the 'earth' connection. This is different from the signal ground on pin 7 below.
3	Transmit Data	Output	The serial data signal from the DTE
2	Receive Data	Input	Data arriving at the DTE
7	Request to Send	Output	Handshaking
8	Clear to Send	Input	Handshaking
6	Data Set Ready	Input	Handshaking
5	Signal Ground		
1	Data Carrier Detect	Input	Handshaking
4	Data Terminal Ready	Output	Handshaking
9	Ring Indicator	Input	Detects the telephone line 'ringing' so that the system can respond to an incoming message.

Table 2: RS232 9 Pin D Pin Allocation

You will notice in this format, the second data link is missing, as is any provision for line testing and the ability to run in synchronous mode. Understanding the timing and purpose of all the handshaking signals is easiest with reference to a timing diagram as shown in figure 2.

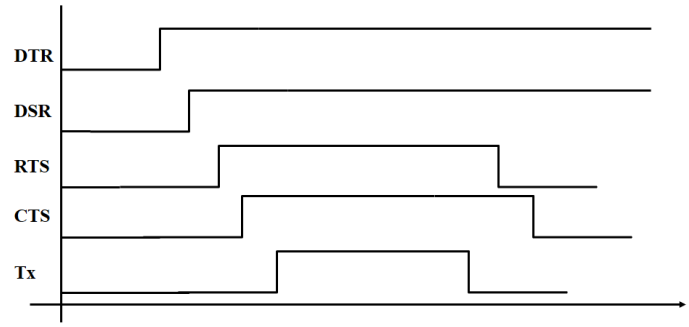

Figure 2: Timing of the RS232 Handshaking Signals

If the data terminal wishes to transmit data, the *Data Terminal Ready* is taken high. The receiving data communications equipment with then respond by setting its *Data Set Ready* pin high. This has now readied the channel but before the transmission actually occurs, there is some further handshaking. The data terminal will assert *Ready To Send*, which is as the name suggests. The data communications equipment will respond with *Clear To Send*. Only when the data terminal receives this final clearance will it start to transmit data.

You will notice that as soon as the data transmission is complete the *RTS* signal is de-asserted in advance of *CTS*. This is to allow a response to occur in a half-duplex system as part of the same communication.

This handshaking process assumes that there is a date terminal equipment (transmitter) at one end of the link and a data communications equipment (receiver) at the other, and that both have fully functional RS232 links. In such cases the wiring will be as shown in figure 3 below.

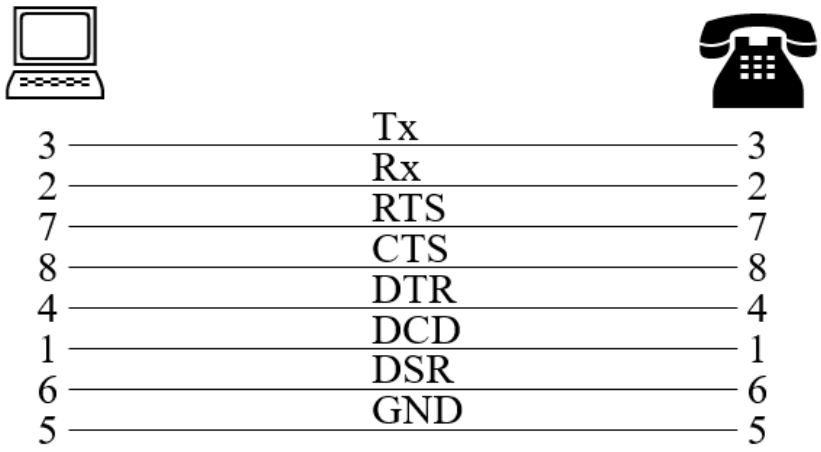

Figure 3: Fully Wired DTE-DCE RS232 Link

You will notice that corresponding pins are wired the same on both ends, so the function at the DCE end is the opposite of that at the DTE end. That is, the transmit is actually and input and the receive and output for the DCE.

Over time, strict adherence to the DTE-DCE connections has somewhat slipped, so in many cases both ends may consider themselves to be data terminals. In order to connect two such devices together, what is known as a *Null Modem* connection must be used. Effectively, this is a connection which doesn't include the modem part of the standard link (as the name suggests). On such a link the handshaking connections are swapped, as are the data lines, so that outputs at one end connect to the corresponding input at the other, as shown in figure 4.

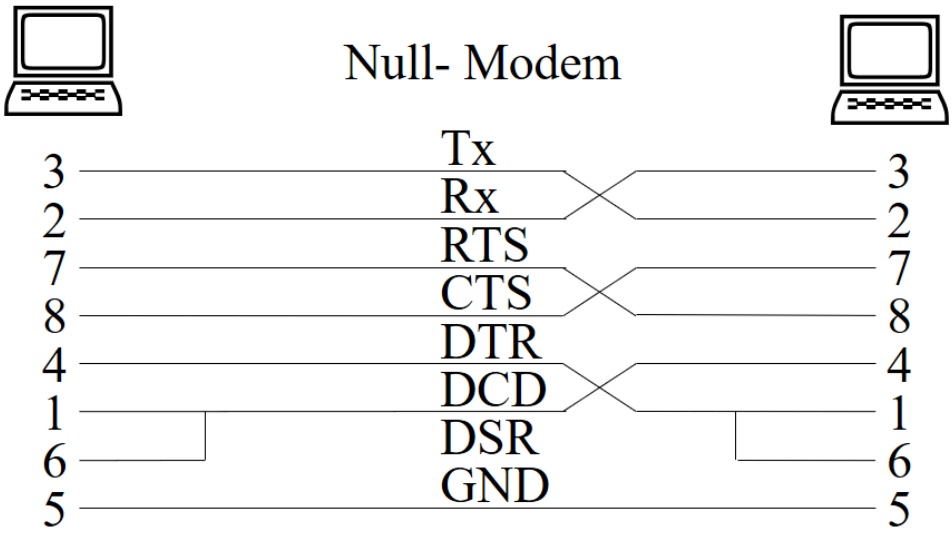

Figure 4: RS232 Null Modem Connection

As stated above, many systems today do not fully implement the handshaking signals, reducing the connection to the minimum set of signals required to enable bidirectional communication to take place, that is the transmit, receive and ground.

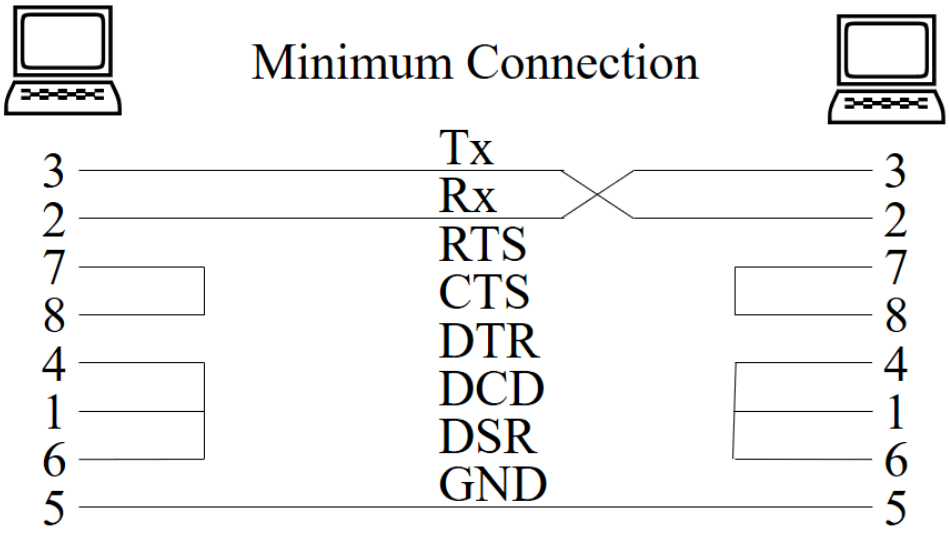

Figure 5: RS232 Minimum Connection Wiring

Figure 5 above shows the minimum connections required to achieve data transfer. You will notice that it is wired as a null-modem connection, although it could also be wired as a straight connection depending on the situation. You cannot assume that the 'other' end of the link is also minimally wired so the handshaking signals are locally looped back on themselves. The data terminal ready being asserted causes the carrier detect and data set ready inputs to be asserted, allowing the system to respond by asserting ready to transmit. This loops round to assert clear to send, allowing data transmission to occur.

9.2. RS232 Voltages

Looking at an RS232 connection on a digital board such as the PIC, you will notice that microcontroller is not directly connected to the RS232 link, but is connected to an RS232 driver IC, commonly the MAX232, before being connected to the connector. This is not to protect the PIC as such, but because the RS232 link does not run on the standard 5V used in many modern logic systems. This is primarily because the standard predates the adoption of 0V and 5V as the standard voltages for digital circuits.

Signal	'Minimum'	'Maximum'
Data '1'	-3V	-15V
Data '0'	+3V	+15V
Line Idle	-3V	-15V
Handshaking '1'	+3V	+15V
Handshaking '0'	-3V	-15V

Table 3: RS232 Signal Voltage Ranges

Table 3 shows the specified signal level voltages on an RS232 link and there are a number of things to notice. Firstly, 0V is not used at all. This is a form of link failure checking. If any pin detects 0V, then this indicates that an issue has occurred on that wire. In modern systems, such checking is not normally implemented. Secondly, a range of voltages is acceptable but the same values positive and negative should be used. Anywhere between ±3V and ±15V is allowable. The third thing to notice is that the data lines are active low (1 is negative and 0 is positive) but the handshaking lines are active high (1 positive). This is not an issue if using a commercially designed interfacing IC because the levels and polarities will be taken care of but it needs remembering if designing the driver from first principles.

9.3. RS232 Data

Up to now we haven't actually looked at how the data itself is transmitted. Initially, the line will be idle, so will be at a voltage of between -3V and -15V. The first thing that will happen is that a *start bit* will be sent. This is a zero, so the line voltage will change from negative to positive. This tells the receiving device that a transmission is about to start (even if handshaking is not used). A number of data bits will then be sent. This may be followed by a parity bit, which is a simple form of error checking and by stop bits. The stop bits are the line returning to its idle state. It will need to remain at this voltage for a certain period of time in order for the receiver to register the end of the transmission. This is easier to understand diagrammatically as shown in figure 6.

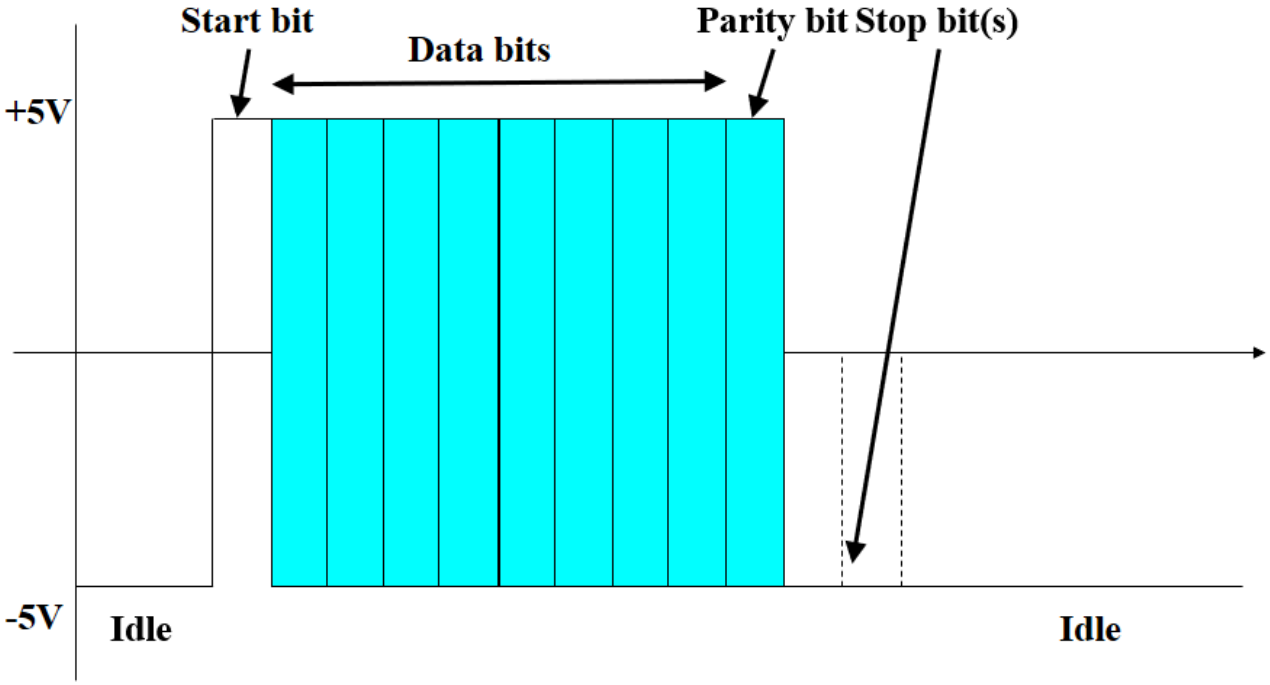

Figure 6: An RS232 Data Packet

Central to the ability of the receiver to be able to accurately decode the transmissions it received is the fact that all data transmissions are formatted the same. They will be transmitted at the same baud rate, they will all have the same number of data bits and they will all have the same number of stop bits.

According to the RS232 specification the following is allowed:

Data bits	5, 6, 7 or 8
Parity	Odd, Even or not transmitted
Stop bits	1, 1½ or 2. 1½ is only allowed with 5 bit data

Table 4: RS232 Valid Data Structures

One thing which is not actually specified in the standard is the order in which the bits are transmitted. Normally, data is transmitted least significant bit first.

You will notice that the number of data bits transmitted can be anything between 5 and 8 bits. As far as the specification is concerned any of these is acceptable but to successfully transmit, both the transmitting and receiving ends must be set to work with the same values. At the end of each transmission, the stop bits effectively return the channel to its idle state. Specifying the number of stop bits sets a minimum time between the end of one message and the start of the next.

Parity

Parity is a very simple form of error-checking, which can be transmitted as part of the RS232 data format. Before a message is transmitted, the value of all the bits in the message is added up. This isn't the binary weightings of each bit, literally, the ones and zeros. Once this addition is completed, the result will either be odd or even. The parity bit is then either set or reset to ensure that all

messages contain either an odd or even value.

For example:

Suppose we wish to transmit the value 0010 1001 with *even parity*.

Firstly, add the bits together: \(0+0+1+0+1+0+0+1=3 \)

Three is an odd number, so we set the parity bit to ensure that the total number transmitted is even.

Had the parity been odd then the parity bit would not have been set to ensure that the value of the bits transmitted was odd.

Using parity is a basic way of spotting errors, if one bit gets corrupted and so is the wrong value, then the sum at the receiving end will give an odd when it should be even. A parity error can then be flagged. What parity cannot do is detect even numbers of errors, if two bits both get corrupted then the parity check at the receiver will still yield a valid transmission although two bits have changed values.

Clocking and Synchronisation

Central to the ability of the receiver to successfully detect and decode the received messages is its ability to synchronise to the incoming data. Most modern systems will have a clock which runs much faster than the speed of the incoming data stream, so there will be many system clock pulses during on bit period. Figure 7 below shows a common start-bit validation and synchronisation process.

The rising edge (which is actually a 1 to 0 transition) is detected. It is common for the serial interface clock to be set up by specifying the number of system clock cycles in one bit-period of the serial interface. For example, there may be 16 system clocks per bit. The receiving system will know this, so it will know that 8 system clocks after the leading edge was detected is the middle of the pulse and so is the best place to synchronise to. From that position it will then take a reading every 16 clock pulses until all the bits in the message have been read in.

In figure 7 three other red lines are also shown. In order to check that a valid start bit has been received rather than a spurious edge, it is common for several readings to be taken during the start bit period to ensure all show a valid start bit has been received.

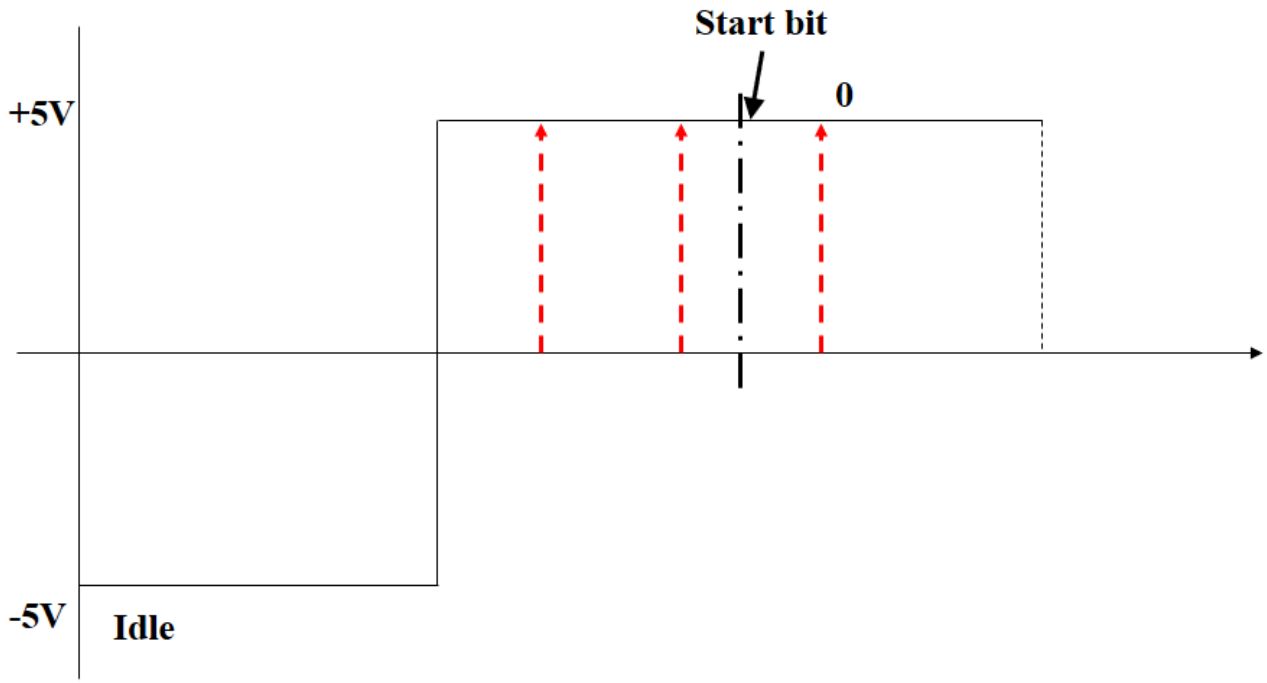

Figure 7: RS232 Clock Synchronisation

These readings may be taken after 4, 8 and 12 clocks for example.

Why clock from the centre of the bit?

It is normal to clock from the centre of the bit because this gives the maximum room for drift in the clock timing without running in to difficulty. The transmit and receive clocks will never run at exactly the same frequency so over time one will gain or lose compared to the other. The maximum message length in an RS232 transmission is 12 bits (1 start, 8 data, 1 parity and 2 stop bits). As we have

said, the stop bits are a return to idle and the start bit determines the synchronisation point, so we only need to remain synchronised for a maximum of 11 bits.

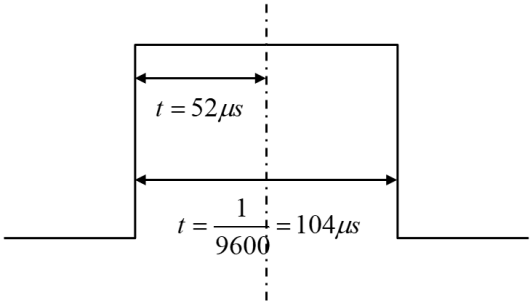

Figure 8: RS232 Bit Timing

Figure 8 shows the timing for an RS232 bit if running at 9600kbd, or 9,600 bits per second which is one of the standard frequencies used.

The width of each bit is 104µs, as shown. This means that if the synchronisation point is the middle of the start bit, then the maximum drift can be 52µs over 11 bits.

$\frac{52\mu}{11}=4.7\mu s/bit$

For this we can work out the maximum and minimum receiver clock frequencies to still receive the data accurately:

$f_{max}=\frac{1}{104-4.7}=10.07kbd$

$f_{min}=\frac{1}{104+4.7}=9.1996kbd$

Normally, we would express this as a percentage:
$drift\leq\frac{9199.6}{9600}\rightarrow 4.17\%$

10. The PIC UART

The PIC has the ability to simultaneously transmit and receive serial data. The port can work synchronously, so could be used for I^2C communications or asynchronously for RS232 or other transmissions. There is variation across the range, with some PICs containing a single USART, whilst others have two. Full information on setting up and using the USART on the PIC can be found in chapter 16 of the PIC18FXX2 data sheet.

10.1. RS232 on the PIC

In regard to the transmission and reception of RS232 data, as we have said above, the voltage levels on an RS232 line mean that an interfacing IC would be required to match the voltage levels. Once this is done, the PIC can directly generate and receive RS232 data but the interface is not fully comprehensive so there are limitations.

1. The PIC can only handle 8-bit data. This is not too much of an issue because most modern RS232 links will work using this data format.

2. The PIC does not include a parity system by default.

3. The PIC runs with one stop bit.

What the PIC interface does provide, however, is the ability to transmit a freely assignable 9th bit. This bit can be selected or not as required by the user, but could then be used as a second stop bit, or be handled in software as a parity bit if needed. There is also an inbuilt facility to use the ninth bit to transmit address information.

Internally, the PIC contains a number of special function registers to control both transmission and reception, *TXSTA* and *RCSTA*. In addition, there are receive and transmit data registers *RCREG* and *TXREG*, and a register to control the timing of the bit clock, known as the baud rate generator, *SPBRG*. Note that there is only one baud rate generator for the interface, so both the transmitter and receiver need to run at the same baud rate. If the PIC contains two USARTs, the register names are appended by the USART number.

Serial Peripheral Baud Rate Generator SPBRG

The baud rate generator SFR is very similar in its operation to the timer registers and also similar to the timers, the baud rate generator effectively has a prescaler in that it can run in *high speed* or *normal speed* modes as set by the value of a bit in the transmitter status register as we will see shortly.

In normal speed mode, the baud rate is set by:

$$f_b=\frac{f_{osc}}{64\times SPBRG}-1$$

In high speed mode, the baud rate is:

$$f_b=\frac{f_{osc}}{16\times SPBRG}-1$$

So if, for example, we wish to transmit at 19.2kbd using our PIC with a 24MHz crystal:

$$19.2k=\frac{24M}{64\times SPBRG}-1$$

$$19,201=\frac{24M}{64\times SPBRG}$$

$$\therefore SPBRG=\frac{24M}{64\times 19,201}=19.5$$

19.5 isn't ideal because we can only load a whole number so would have to load either 19 or 20, both of which will give something of an error

In terms of the error this would be:

$$f_b=\frac{24M}{64\times 20}-1=18,749Hz$$

$$err=\frac{18749}{19200}=97.7\% \Rightarrow 2.3\%$$

$$f_b=\frac{24M}{64\times 19}-1=19,736Hz$$

$$err=\frac{19736}{19200}=102.8\% \Rightarrow 2.8\%$$

So the error would be ±3%

Alternatively, if we run the baud rate generator at high speed we get:

$$SPBRG=\frac{24M}{16\times 19,201}=78.121 \simeq 78$$

We would only tend to round down here, giving an error of:

$$f_b=\frac{24M}{16\times 78}-1=19,230Hz$$

$$err=19230/19200=100.16\% \Rightarrow 0.16\%$$

In this case, we could run with the baud rate generator in high speed or normal speed modes, but the error is smaller in high speed mode so this would be preferable.

The Status Registers RCSTA, TXSTA

There are separate status registers controlling the receiver and transmitter halves of the USART, *RCSTA* and *TXSTA* although care needs to be taken because some bits in the 'other' SFR may still need to be set. For example the serial port enable is in the receive status register but needs to be set for both receiving and transmission and similarly the high speed mode bit is in the transmit status register.

Bit	Mnemonic	Function
7	SPEN	Serial Port Enable Turns on the serial port. Note that there is a single enable for both the receive and transmit sides, so this bit needs to be set in order to transmit even though it is located in the receive status register
6	RX9	9^{th} Bit Enable Setting a 1 here enables the transmission of bit 9. The value transmitted is loaded into bit 0 of RCSTA.
5	SREN	Single Receive Enable This bit only operates in synchronous mode and allows reception of a single byte. This bit is reset on reception. This enables the receiving of a single byte of data as opposed to…
4	CREN	Continuous Receive Enable This enables continuous reception of data and is the only bit used in asynchronous mode. CREN takes precedence over SREN
3	ADEN	Address Detect Enable Setting this bit allows the ninth bit to transmit an address 1-bit per message.
2	FERR	Framing Error Flag- see below This bit can be cleared by reading the receive buffer and receiving a next valid byte.
1	OVERR	Over-run Flag- see below This can be reset by disabling and re-enabling CREN
0	RX9D	This is the data received in the 9^{th} bit

Table 5: The Receive Status Register

Bit	Mnemonic	Function
7	CSRC	Clock Source Select When running in asynchronous mode this bit has no effect. In synchronous mode it selects between 0- Internal (master) 1- External (slave)
6	TX9	9th Bit Enable Setting a 1 here enables the transmission of bit 9. The value to be transmitted is loaded into bit 0 of TXSTA.
5	TXEN	Transmit Enable This enables the transmit side of the USART
4	SYNC	This selects synchronous or asynchronous operation. 0- Asynchronous (default) 1- Synchronous
3		*Unused*
2	BRGH	Baud Rate Generator High 1- High Speed 0- Normal Speed
1	TRMT	Transmit Buffer Status Bit This bit flags when completion of a transmission 0- Transmit register is full 1- Transmit register is empty
0	TX9D	This is the data to be transmitted in the 9th bit

Table 6: Transmit Status Register

Receiver Operation

In order to receive serial data, the USART must be set up and enabled.

1. The appropriate value needs to be loaded into the baud rate generator SFR.
2. The correct speed must then be selected but setting or resetting BDGH in the TXSTA SFR.
3. The appropriate options must be selected in the RCSTA register including whether and how the 9th bit is used.
4. Port C pin 7 is used as the serial port input and so must be configured as an input in *TRISC*.
5. *Either* SREN or CREN must be enabled depending on whether single or multi-byte reception is required in synchronous mode or CREN must be set in asynchronous mode.
6. Finally, bit 7 of RCSTA must be set to enable the USART as a whole.

The USART is now ready and waiting for data to arrive. When it does so the following will happen:

1. The received data is loaded into the RCREGx buffer (where x is the number if the PIC has two serial ports)

2. The USART will check that the correct stop bit is received (a '1'). If not, the framing error flag is set. If this is set, it may indicate a problem with the settings on the link. You need to clear this in code.

3. The RCxIF interrupt flag is set.

4. In an interrupt service routine, the framing and overrun flags need to be cleared if necessary and the data moved out of the RCREGx buffer. When this happens RCxIF is automatically reset.

5. If a second byte of data is received before the first byte is cleared from RCREGx, the overrun flag will be set. This is to show that data in the buffer has been overwritten without being read.

Transmitter Operation

The setting up of the transmit side of the USART is very similar to the receive side. Internal or external clocking must be selected and the baud rate set if internal as above. Whether the transmission is synchronous must be selected as must the use of the 9^{th} bit. The transmit side must then be enabled before the USART as a whole is enabled by setting bit 7 of the receive status register. The serial port output pin is pin 6 of port C, which must be configured as an output in *TRISC*.

10.2. I2C on the PIC

It can be seen from tables 5 and 6 in the previous section that the USART can be configured for either synchronous or asynchronous data transmission and reception. This means that it could be used to implement an I^2C or similar synchronous protocol either as a master or slave. However, it is not the best option in this situation. One of the issues being that the USART transmit and receive pins are not open drain as required by the standard.

For I^2C connections either as the master or slave using the Master Synchronous Serial Peripheral Module (MSSP) is a more efficient way of working. This module is specifically designed to implement I^2C so many of the requirements of the protocol are handled automatically within the module without the need to implement them manually as part of the program. This includes both master and slave acknowledge timing and handling, correct start and stop assertion whilst the clock is high, and bus arbitration should a collision occur. Full details of the implantation of the MSSP module can be found in chapter 15 of the data sheet, with details its implementation for I^2C in section 15.4. The module includes two control registers, *SSPCON1* and SSPCON2, a status register, *SSPSTAT,* a data buffer, *SSPBUF*, which is common to both the transmit and receive sides and the address register, *SSPADD*. There is also a shift register which is used to actually shift the data in and out this connects to *SSPBUF* but is not directly addressable by the user.

Bit	Mnemonic	Function
7	SMP	Slew Rate control In both master and slave modes, this controls the maximum rise and fall times as required by the standard.
6	CKE	SMBus Select bit (1 to select).
5		This is the Data/Address bit. It is only valid for use by the programmer in slave mode and will indicate whether the last byte received as a data (1) or address (0) byte
4	P	A 1 in this position indicates that a STOP was the last signal received. This is cleared when the module is reset or disabled.
3	S	A 1 in this position indicates that a START has been received. This is cleared when the module is reset or disabled.
2		In slave mode, this indicates if a read or write request was received from the master. In master mode a 1 indicates that a transmission is currently in progress.
1	UA	When running in 10-bit address slave mode, a one here indicates that the address buffer needs updating
0	BF	Data buffer is full (1) or empty (0)

Table 7: The SSPSTAT Register

Bit	Mnemonic	Function
7	WCOL	Write collision detect. A 1 indicates a collision occurred. In slave mode this may indicate that the data received is corrupted. In master mode it will indicate a need to restart the transmission. This must be cleared in software
6	SSPOV	Data buffer overflow. The serves the same function as the overflow in the USART.
5	SSPEN	Enable This enables the SSP module.
4	CKP	Clock hold/release. If additional processing time is required before the next communication, the clock line is held. This is used to initiate holding the clock in slave mode. The bit is reset to release the clock.
3-0	SSPM3:SSPM0	I^2C Mode bits. Only six mode combinations are valid for I^2C as listed in the data sheet. Others are reserved or used in other SPI modes. These include Master and Slave selection, and 7 and 10 bit address selection.

Table 8: The SSPCON1 Register

Bit	Mnemonic	Function
7	GCEN	General Call Enable A 1 here allows an interrupt to be generated in slave mode when a general call (to all devices) is received.
6	ASKSTAT	Used in master mode When transmitting, a 1 here indicates an acknowledge has been received from the slave and so the next action can occur.
5	AKDT	Used in master mode When receiving from the slave, a zero here will be used to place an acknowledge on the line to tell the slave to proceed.
4	AKEN	Acknowledge Enable Initiates the acknowledge sequence on the data line when in master receive mode

3	RCEN	Receive Enable A 1 here places the data line in receive mode when acting as the master.
2	PEN	STOP Enable Initiates the generation of a STOP on the line when in master mode.
1	RSEN	Repeated START Initiates the generation of a repeated start condition on the line in master mode.
0	SEN	START Enable Initiates generation of a START condition on the line when in master mode.

Table 9: The SSPCON2 register (Most of these are specific to running in master mode)

When running as the master device much of the signal timing is handled automatically but the user must initiate communication by writing the required enable bits to *SSPCON2*. The user must also ensure acknowledges are received before continuing.

10.3. Summary

In this session we have looked at serial interfacing, looking at RS232 and how it is implemented on the PIC. We have also briefly looked as I^2C and how it can be implemented both in master and slave mode on the PIC.

11. The PIC Overview

To maximise the usefulness of microprocessor-based systems it is necessary to interface the processor with the outside world. This is achieved through a variety of peripherals such as screens, keyboards, a printer, the mouse, and so on. In general, the microprocessor cannot interface directly with the peripheral and so a certain amount of *interfacing logic* would be required to translate between the processor itself and the peripheral. As peripherals became more complex, the complexity and amount of interfacing logic also increased. At the same time, there was a drive towards further miniaturisation, meaning that large amounts of interfacing logic were not an ideal solution. Not only does the use of a large amount of interfacing logic increase the circuit size, it also increases the cost of the circuit and can reduce reliability because there are more components which could fail, or interconnections which could fail.

To solve this problem, the *peripheral interface controller* (PIC) was developed. Strictly speaking, the PIC is a whole family of programmable devices designed principally for replacing the interfacing logic between the processor and the peripheral. Because they are programmable devices, they have found a wider range of uses as one of a range of different *microcontrollers*. Microcontrollers contain a programmable core or *central processing unit* (CPU) very similar to that found in a microprocessor, but also contain a range of interfacing circuitry as part of the same *integrated circuit* (IC).

Over time, the ability of these devices to perform a range of different tasks means that they are now regularly used in more general control and processing situations, and the development of newer devices has sought to build on this.

11.1. Why use interfacing circuitry at all?

Before we progress, it is probably helpful to address the question of why any form of interfacing logic or controller is required. Why could the microprocessor not be used to provide all the control direct to the peripheral?

There are a number of reasons why this might not be a possible, or an ideal solution:

1. Interconnections: Microprocessors have bus connections for data transfer. These may not be suitable for connecting to peripherals. PICs use *ports* for input and output connections. These can be controlled more directly to allow more efficient interfacing with the peripheral.
2. Processor Load: Running a function on the processor to convert the outputs to interface to the peripheral takes up processor time which could be better used in other ways. Offloading the control interfacing to the PIC is generally a more efficient solution.
3. Standardisation: If the central processor contains the interfacing program, it would need reprogramming each time a different peripheral was attached, for example, a different screen is connected. By using a dedicated controller, the microprocessor can output information in a standard format and the interface controller can be specifically designed for that particular device.

11.2. A Range of Devices

PICs consist of a large number of different devices with different capabilities and features and with differing numbers of pins.

In some cases, the control function might be complex but only one or two pins might be required on both the input and output sides. In this case, a controller is needed with the processing capability to handle these complex functions, but with a low pin count, and the resulting small footprint, to only provide those connections required without filling unnecessary board space. If all devices contained a lot of inputs and outputs then many of these would be unused in this type of situation.

Of course, in other situations, a large range of I/O may be needed, so alternatives to cater for this are also required.

Processor Size

PICs are available with a variety of different processor sizes: 8-bit, 16-bit and 32-bit. This defines the physical size of the processor inside the PIC and hence the default size data it can process. Again, the selection of which to use would depend on the data which will be processed. If the system the PIC is being used in only needs to process 8-bit data, then an 8-bit PIC will suffice. If the system will need to process mainly 16-bit data, this could be achieved using an 8-bit device, but each calculation would have to be divided into parts and so would take longer to complete. For example, if an 8-bit device is used to add to 8-bit values together, this could be achieved in one processor cycle. If an 8-bit device is used to add two 16-bit values together, firstly the 8 less significant bits would be added, then the 8 more significant bits would be added, taking twice as long. Obviously a 16-bit device could achieve this in one processor cycle.

8-Bit PICs

The 8-bit devices are divided into families, the two most common of which are the PIC16 and PIC18. Within these families, there are a number of ranges. Devices within the same family are designed to have a large amount of their structure in common, so switching between one device and another in the same family can be achieved with a minimum of reprogramming.

For example, within the PIC16 family is the PIC16F18xxx range. This consists of a total of 17 devices ranging from the PIC16F18313 which is an 8-pin device, up to the PIC16F877A which is a 40-pin device. Different devices within the range also contain different amounts of program and data memory, and a different combination of input and output facilities, for example differing numbers of analogue to digital converter channels.

Also available in this family are the PIC16F15xxx range, the PIC16F17xxx range and the PIC16F19xxx range.

Full information about the range of different devices available can be found on the microchip website: www.microchip.com.

Looking at the full specification chart for any of the ranges available shows a large variety of possible different types of connection and interface for which a device might be tailored. For example, some are designed with Ethernet connectivity included, others for USB or more specialised standards such as CAN bus; some are designed for high voltage situations and others optimised with inputs to detect certain types of signals on an input, for example zero crossing (when a voltage crosses over a reference line either negative to positive or positive to negative) or a certain slope.

12. The Structure of a PIC

More information about the structure of the PIC can be found here. This site gives quite a good brief overview of the PIC.

A generalised block diagram of a PIC processor is shown in figure 1.

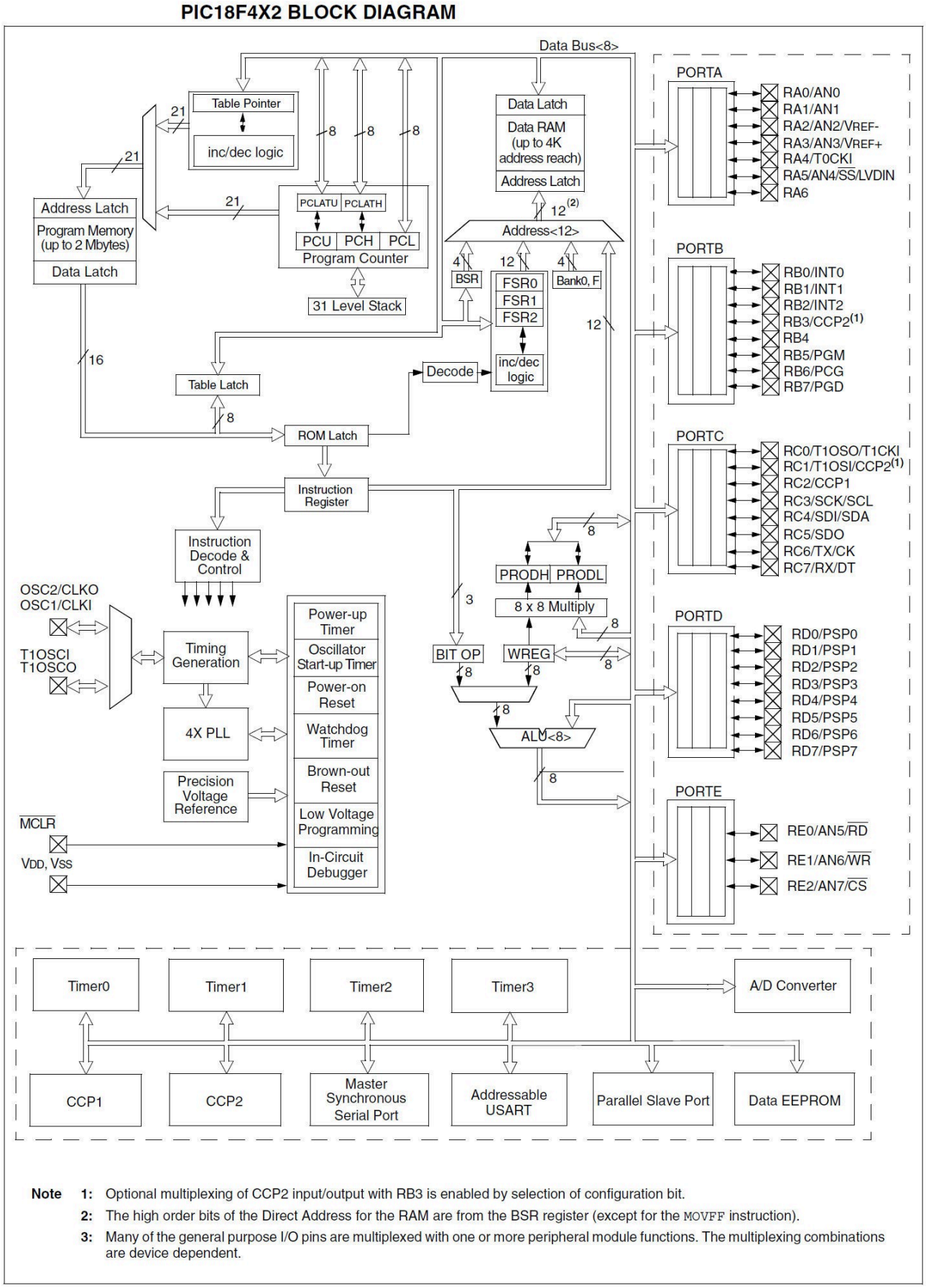

Figure 1: PIC Block Diagram (Note image is PIC18F4x2- No Port D or E on 252)

12.1. The Central Processing Unit

Central to the operation of the PIC is the CPU which is a fairly standard processor based around what is called the *Harvard Architecture*. Many microprocessors and microcontrollers are based around this general structure. The CPU consists of an *arithmetic logic unit* or ALU which is capable of performing a wide range of arithmetic, logic and data manipulation functions basic on settings given by the current instruction. Instructions will be decoded to adjust various settings and options within the ALU to make it perform the required task. This may be to add or subtract values, or to perform logical operations such as AND or OR on a value. Situated alongside the ALU as part of the CPU will be a number of *registers* which will hold the data values to be processed and to store other information about the operation of the processor such as the current instruction, the number of the next instruction to read and so on.

12.2. Memory

Data Memory

The second type of memory on the PIC is data memory, and this is mostly *Random Access Memory* (RAM) which is also known as *volatile memory*. The two important differences between RAM and ROM are:

1. It can be both read and written to by the CPU when the program is running.
2. Its contents are lost when the power is switched off.

The data memory is normally referred to as *registers* and is normally smaller that the program memory.

General Purpose Registers (GPR)

The general-purpose registers can be used to hold any data required for the operation of the program. Depending on the device used this may range from a few tens of bytes to a few thousand bytes (kbytes). To make the design of the PIC easier, the memory is normally arranged into *pages* with only one page accessible at any one time. To change pages the number of the new page needs to be selected.

Special Function Registers (SFR)

Selecting the new page of general-purpose registers is achieved by writing the new page number to a different type of register known as a *special function register* or SFR. SFRs are memory elements in that they will remember the last value written to them, but rather than for general use, they are used to control the operation of various aspects of the PIC. The PIC designers have tried to make the devices as flexible as possible, so, for example, rather than fixing a certain number of pins as inputs and a certain number as outputs, any of those available can be inputs or outputs. In order to make this possible, however, the programmer needs to set up the pins either inputs or outputs and this is done by writing 1s (1 for in) or 0s (0 for out) to a special function register known as the port control register (of which more later). The 1s and 0s written to the various bits in the SFRs effectively act as switches turning on or off various options of the many functions available in the PIC. Some of these may literally act as switches, turning on or off different parts of the PIC. This can be useful if a battery-powered system is being designed. Turning off a part of the circuit which is not required will reduce the amount of power required by the PIC and so will lengthen battery life.

Internally, SFRs are just memory locations with numerical addresses but to make programming easier, SFRs are normally given names within the programming environment so that they are easier to remember and so that the programs are easier to read.

SFRs and GPRs occupy the same *address space*. Thinking in terms of the postal system, this is like saying that they are on the same road. Following this analogy through, however, it is better to think in terms of an American street. Why? Travelling down an American street it is noticeable that all houses will have numbers, but not every number is used. For example, one house might be number 1384, and next door might be 1392. The numbers in between do not exist. The same is true with the PIC. It might use an 8-bit value to access the address space. 8 bits give 255 possible addresses, but not all 255 might actually be used. Writing to a non-existent address will cause the data to get lost. The reason for the missing addresses is to maintain compatibility between devices. Some devices have more memory than others and those with less have gaps. Also, some devices have functions others do not. It is better from a compatibility perspective to leave those locations vacant if that function is not implemented rather than using them for something else. Otherwise a programmer could write to the location on one device to achieve one result, but find something strange happening on a different device because the location is used for something different.

Data EEPROM

Having said that data memory is RAM, some PICs do also include some data EEPROM. This memory area cannot be directly written to but is accessed via settings in some of the special function registers. This memory could be used for holding important constants needed by the program so that they don't have to be reloaded each time the system powers up.

12.3. Peripheral Functions

What makes a microcontroller different from a microprocessor is that many of the peripheral functions are included within the same IC. These can be many and varied, and much of what differentiates the different devices in the PIC range is the inclusion of different (or differing amounts of) various peripheral functions. As indicated above, many of these are designed to be as flexible as possible, with their exact operation controlled by loading the required settings into SFRs.

Ports

The most fundamental peripheral on the PIC (and other microcontrollers) is the *port*. Ports are general purpose digital input and output from the PIC. The number of ports available will vary depending on the device, and the width (number of bits) in the ports may also vary. Again, in order to maximise flexibility whilst minimising the number of pins on the IC, ports pins may double up as other inputs and output too. Care needs to be taken in system design to make sure that each pin is only being used for one purpose.

The ports are given letters, for example, a PIC may have three ports, A, B and C.

Each port has two SFRs, one to control the direction of the port pins and one to hold the data input or output from that port. The direction SFRs have the name 'TRIS' and the data SFRs 'PORT'

So to set port B to all inputs, use the command:

TRISB = 0b11111111; (The semicolon is required for correct C syntax)

To set port A to all outputs:

TRISA = 0b00000000;

To actually set the pins on Port A to 1s write:

PORTA = 0b11111111;

To read in the value on port pins:

MyVariable=PORTB;

Which will transfer the value of the port B data register to the variable called 'MyVariable'.

Note that each pin on a port can be set up independently, the whole port does not need to be input or output. So it is possible to write:

TRISB = 0b01010101;

This would set the port up as follows:

Port pin	Direction
7	Output
6	Input
5	Output
4	Input
3	Output
2	Input
1	Output
0	Input

Note also that the numerical values here are written in binary. This can be helpful when setting SFRs because it is easy to see which bits are set (1) and which are clear (0). The binary value will always read left to right from the most significant bit (bit 7) to the least significant bit (bit 0). It is also possible to specify these values in decimal or hexadecimal if preferred.

Remember that computers like things to be very clear and explicit. Lots of numbering systems contain 1 and 0 so it is necessary to identify which numbering system is being used:

TRISA=0b11111111;	Binary
TRISA=O377;	Octal (Note this is a capital O not zero) Not really used now.
TRISA=255;	Decimal (actually the default if no identifier is included)
TRISA=0xFF;	Hexadecimal

Table 1: Numbering Systems

Each of these will do the same, setting all bits of port A to inputs.

The Timer

In many situations a single of a specific frequency, or at a regular time interval needs to be produced. A simple example of this would be to flash the indicator on a car.

A simple way to do this would be to turn the indicator on, get the processor to wait (waste time for a certain time) and turn the indicator off. This is perfectly possible to do, but during the 'wait' periods, the central processor is tied up doing nothing, but could not respond to anything else it might need to do.

The timer is a specific module designed to generate signals at fixed intervals, but without the CPU needing to be involved.

Again, SFRs are used to set up the timer (or timers as some PICs have more than one) and to control its operation. Control options include:

- Setting the speed of operation. Essentially a timer is a counter which starts at a certain number (loaded from the program) and counts up until the maximum value is reached. When this occurs the timer will tell the CPU time is up. It does this by generating an *interrupt*. Interrupts will be looked at in more detail later, but they are a means of telling the CPU that something has occurred which needs responding to *now*.
- Internal or External Trigger. In different situations, the timing period might need to be started internally by the program, or by some external signal.
- Continual or once only. Again, different situations may have different requirements. In some cases a frequency might be required. Here, the timer would need to run continually. At the end of each count, the timer will interrupt the CPU, but it will also automatically reload its starting value and start counting up again. In a *single shot* situation, the timer will stop once it has finished.
- Generate time or measure time. This may be used with the external input, and possibly with other modules on the PIC. If an external input is set as the trigger and the timer is in measuring (count) mode, the timer receives a trigger signal to start, and starts counting. It will continue to count until the next trigger signal is received. At that point it will stop counting an interrupt the CPU. By doing a little bit of maths, the time between the two external trigger events can be calculated. If these triggers are happening too quickly for a meaning value to be generated, it is also possible to adjust settings in counting mode to allow for it to count the time taken for a number of triggers to occur.

Analogue to Digital Conversion (ADC)

The PIC is digital but many of the signals fed into it may be analogue, from sensors etc, for example, from a temperature sensor to monitor how hot the inside of a PC case is getting. The temperature sensor may give an analogue voltage out. This is then input to the PIC. It needs converting to digital for the PIC to be able to process it, then when its value exceeds a set level an output is set to turn a fan on to cool the PC down. The analogue input would continue to receive the output of the temperature sensor as the temperature drops. When it is low enough it will turn the fan off.

Again, as with other modules, there are a range of options for controlling the ADCs. These include:

- Resolution. The resolution is the number of bits used to represent the analogue voltage. In some PICs this can be selected between 8 and 10 bits for example. If the additional accuracy of a 10-bit output is not needed, it is better not to use it on the 8-bit PICs because of the additional processing overhead which results.
- Number of channels. Many of the PICs have more than one ADC input. It is possible to control how many are used. Not using those which are not required reduces power consumption.
- Reference Voltages. Normally, the ADC converts over the full 5V power supply range, with 0V=0 output and 5V=1023 (10 bit) or 255 (8 bit). If an input voltage range was only between 2V and 4V, then converting over the full 5V range is a waste. A more accurate result will be achieved by setting the ADC up so that 2V=0 and 4V=1023 (or 255). The ADCs can be set up to use some of their pins to act as minimum and maximum reference voltages.

The ADC functions independently of but controlled by the CPU. The CPU sends a signal to the ADC to start a conversion. As normal, this is achieved by writing a 1 to the appropriate bit of an SFR. Once the conversion is complete the ADC tells the CPU by generating an interrupt.

Capture/Compare/Pulse Width Modulator Module (CCP)

Depending on the device used, several of these may be included. This module is a powerful resource in detecting and generating all sorts of specific external signals.

It is commonly used alongside the timers to measure the frequency of a change of level at an input. It may also be used to detect the rising or falling edge of a signal on an input (A rising edge is when a normally 0V signal changes to 5V, a falling edge being the opposite). This may be important in a number of protection situations, for example, if there is a power failure, an input which is held at 5V would drop to 0V because the power is off. This could then trigger the PIC to fire up a backup generator or sound an alarm.

Pulse Width Modulation (PWM) is commonly used as a form of digital to analogue conversion. A signal is output at a set frequency, but whereas a frequency signal is normally thought of as being high for 50% of the time and low for 50% of the time, a with a PWM signal the ratio of high to low is varied depending on the value to be output.

For example, an analogue voltage input of ½ of the supply (2.5V) would give an 8-bit binary value of 127. If 127 is then fed into the PWM, this would give 50% high and 50% low. Because the output is on for half of the time that equates to an average of half the maximum voltage.

If this was repeated with the analogue voltage at ¾ of the supply (3.75V) this would give a binary value of 191 (¾ of 255). Feeding 191 into the PWM module would give the same frequency output but high for ¾ of the time and low for ¼ of the time. Because the output is now high more than it is low, the average also rises to ¾ of the supply voltage.

Note it is also possible to get outputs where the ratio of high to low stays at 50:50, but the frequency changes as the control value changes but this is a frequency modulator not a pulse width modulator.

12.4. The Clock and System Control

All microcontrollers are *sequential circuits*. That is, they carry out a sequence of operations in order, one after another. A clock is needed to control this process and, in the case of the PIC, there is an internal clock generator circuit. The frequency of the clock can be controlled by settings loaded into the PIC. This cannot be controlled by the program which is running, however, and must be set up before the program starts. The programming environment contains special facilities to allow this and a number of other features which must be set up before the program starts to be configured.

The system control block also allows the PIC to *go to sleep* if nothing is happening. In a lot of situations, the PIC will not need to do anything most of the time. It will just be waiting for an interrupt to occur to which it needs to respond. It the system is battery powered, having the PIC fully operational when nothing is happening is a waste, so the PIC can go to sleep. Many PICs have different levels of ‘asleep’ depending on how much of the PIC is shut down, and how many different signals can wake it up again.

13. Programming the PIC

The PIC can be programmed from a Windows PC (or a Mac) using an *integrated Design Environment* (IDE) known as MPLAB. This program is specifically designed for programming the PICs and not only allows the programmer to write the code for the PIC (technically this could be done on any text processor) but also to check the code for errors, simulate the code to see that it works as intended and download the code to the PIC itself.

As with any 'computer' the PIC could be programmed using any programming language provided the IDE was capable of converting the commands in that language into *machine code* which is the list of instructions that can actually run on the PIC itself. For programming microcontrollers, two different languages are most commonly used, and MPLAB is able to handle both.

13.1. Assembly Language

Assembly language is what is known as a *low-level* language. The commands in assembler directly relate to the machine code instructions which run on the PIC. The only change is that rather than having to input a set of numbers representing the instructions, mnemonics are used. These are a form of shorthand which is easier for the programmer to understand.

Assembly language assembles rather than compiling, and can be the most efficient way of coding. The down side is that very little is done automatically.

13.2. C

You should already have some familiarity with C. It is a *high-level* language and as a result, much more is done automatically, within the C function commands. C cannot directly convert to machine code but must be compiled first. This is where a possible issue can occur. There may be many ways of converting a line of C into machine code instructions, different compiler writers may have different solutions. All of these may achieve the same final result, but some may be more efficient either in terms of program memory usage or execution speed.

Here, C will be used to program the PIC.

13.3. Programming the PIC in C

C for the PIC is just the same as C for any other system. The syntax and the overall program structure is the same, and the same commands are available.

One slight difference from 'standard' C is that there are certain aspects of programming any microcontroller which do not exist as normal C functions. The most obvious example of this is the handling of interrupts. Interrupts exist on PCs but are handled at a by the operating system as a much lower level than the programmer would ever work at. On a microcontroller interrupts form an integral part of the system and programmers need to be able to write the *interrupt service routines* which are the lines of code the microcontroller will run when the interrupt occurs.

Standard C functions are handled the same in all compilers because they are standard. However, because the extensions are non-standard, different compilers may handle them in different ways. Some handle them by allowing the inclusion of assembly language commands to access the interrupt routine; others by having additional commands which will tell the compiler to compile the following function in a certain way so that it is identified in machine code as an interrupt service routine.

You will be aware that everything in C is considered as a *function*. Even the declaration of variables is approached in the same way. 'char' is a function which sets aside memory for the storage of that data. You will also be aware that some functions are automatically available in the compiler, others are defined within the program, but others are made available through *header* files. These are effectively standard additional functions and declarations which are made available for inclusion in your program via the '#include' command. These are used in programming for the PIC in just the same way as programming for the PC. The 'standard' files are available if required, but also specialised headers are made available which define things like the names of the SFRs and their memory locations, and other specific instructions which might be particular to the PIC.

You will be aware that C requires everything to be declared before use, so in order to use these names and functions the relevant header file must be included at the start of the code.

A 'C' file for the PIC normally consists of three parts:

Configuration

The first part of the C file will be to configure the compiler and the PIC. This is done through the use of *directives* which are commands with start with a '#'.

The #include command is already familiar. There is a second directive which is commonly used, *#pragma config*. This command is used to configure the PIC.

Setup

The second part of the program file will be to set up the PIC. This will be where values are loaded in to various SFRs to set certain port pins to inputs or output for example, or to activate and set up modules such as the timers, ADC or CCP.

Why configure and setup?

The set up phase runs as part of the program. So when a program is run on the PIC, the first things it will do is to load the values into the various SFRs. You will remember, however, that some settings on the PIC need to be made before the program executes. Two of the most common of these are to set the clock speed, and to switch encryption of the program memory on or off. These commands could not be included as part of the program itself, but need to be sent to the PIC as part of the programming process. The '#pragma config' command tells the compiler to send these settings to the PIC before the program is uploaded.

The Actual Program

The final aspect of the code will be the actual program the PIC is to run. Normally, microcontroller programs are intended to run continually, so the main body of the code will sit within a never-ending loop. This is most commonly achieved using a 'while(1)' function although it can be achieved using a never-ending 'for' loop.

As stated above, the main function may not be required to do anything- just wait for an interrupt and then respond. As a result, the 'main' function might literally be:

```
Void main(void)

{
```

```
While(1);

}
```

All this function does is to wait indefinitely. Obviously, if a program with this main function is to do anything useful, additional functions would be required which respond to interrupts as they occur.

13.4. Structuring your Program

You will already have some ideas about structuring a C program from your programming course, and this is just as relevant here. In particular, make sure the program is well commented- that is that each function has a comment describing its purpose at the top, and that each line or set of lines has a comment describing its purpose. The comments should be clear enough that in a year's time, they should tell you exactly how the program works.

Heading Information

At the start of each file, include a title, date and revision number. You may also wish to include your name. Remember each of these lines need to be included as comments.

If you are working on a bigger project, rather than keep updating the same file, it may be helpful to archive a file once you have a certain element working and then move increment the revision number before working on the next bit. In this way, if you make a change which causes a major problem you will still have the last working version available to go back to.

Your heading may look something like this:

```
/*
 * filename:
 * Author:
 * Date:
 * Revision Number:
 * Target Device:
 * Compiler:
 * Summary
*/
```

This can seem like a lot of information to include before you actually start writing any code, but it is good to develop good practice from the start, and if you were to write code for an actual project, information like this can be vital if you need to revisit the code at a later date.

Header Files to Include

You will need to include some files in your program. This may be standard C functions such as *stdlib.h* or *stduo.h* and you will definitely need the compiler-specific header *xc.h*.

Your header section may look like this:

```
#include <xc.h>     // XC8 compiler specifics
#include <stdio.h> // standard C I/O library
#include <stdlib.h> // standard C library
```

You will notice I've also included a fourth file here which contains a number of compiler-specific functions.

Configuration Settings

This is where you make use of the #pragma config statement. These are fairly standard for our designs, so this section will look like this:

```
#pragma config OSC = HS // Oscillator = Set High Speed
#pragma config PWRT = OFF //Power Up Timer disabled
#pragma config WDT = OFF // Watchdog timer disabled
```

#pragma config LVP = OFF // Low Voltage ICSP disabled

Because the different PICs have different configuration bits, it is normally best to use the configuration bit setting tool within MPLAB to set these. The same bit function doesn't always have the same name.

Definitions

The SFRs for the ports have understandable names, but these are generic and do not relate to what the ports will actually be doing. As a result it can be confusion, particularly when you come back to a program at a later date.

To overcome this, it is normally considered good practice to give inputs and outputs names which mean something in the context of this project. This is achieved using the *#define* directive.

For example, if we were to design a stopwatch, we could have four seven-segment displays connected to the PIC. You could then define the strobe signals driving these as:

#define CATHODE_4 PORTAbits.RA1

#define CATHODE_3 PORTAbits.RA2

#define CATHODE_2 PORTAbits.RA3

#define CATHODE_1 PORTAbits.RA4

Or you may prefer:

#define HUNDREDTHS PORTAbits.RA1

#define TENTHS PORTAbits.RA2

#define SECONDS PORTAbits.RA3

#define TENS PORTAbits.RA4

Of course, in other situations or for other projects, you can define these as required. It is useful to define external connects in this way so that if you change PIC and the port connections change, you only have to make the changes in one place at this point in the file. Otherwise, you are hunting through the whole file, or set of files, to try to find all instances of the names.

Global Variable Declarations

If you need a value to be readily available throughout the program, it is necessary to define it as a global variable. Amongst programmers, opinions vary regarding the use of global variables. Some stating that with good code design, global variables should not be needed, the ideal being that if a value from one function is needed in another function which is called, that value should be passed to the function and then returned.

We won't worry too much about that here. Make sure that you only declare variables globally which are truly global and not just through laziness but if you need a value in more than one function declare it globally.

Function Declarations

It is normal practice for any other functions used to be placed *after* the 'main' function whilst interrupt functions are placed *before* the main function. Any function placed after the main function needs to be declared first so that the main function knows that they exist, and these should be included here. Note that you do not need to declare the *main* function or the interrupt functions.

If you need to do any amount of setting up of the PIC, for example, setting the direction of the ports, or loading initialisation information into other SFRs, it is probably sensible to put all this information in a 'setup' or 'init' function to be called at the start of the main function. This would need to be declared before being called of written.

Code Template

A basic code template is given below, this is also available as a downloadable file:

/*

* filename:

```
 * Author:

 * Date:

 * Revision Number:

 * Target Device:

 * Compiler:

 * Summary

*/


//Included Files


#include <xc.h>     // XC8 compiler specifics

#include <stdio.h> // standard C I/O library

#include <stdlib.h> // standard C library


//Configuration statements


#pragma config OSC = HS //Oscillator = Set High Speed

#pragma config PWRT = OFF //Power Up Timer disabled

#pragma config WDT = OFF //Watchdog timer disabled

#pragma config LVP = OFF //Low Voltage ICSP disabled


//Definitions




//Global Variables




//Functions




//Interrupt Functions (if used)
```

```
//Main Function

Void main(void){

//Local Variables



//Call initialisation function if used



//Local default set-up parameters



//The actual code

/*At the start of the code include a description
 *This will probably be in the form of a ‘while(1)’ loop
*/



}
```

14. PIC Ports

Ports are general purpose input/output pins on the PIC (or other microcontroller) and can be used for interfacing to a range of different digital sources or destinations.

Some microcontrollers (such as the 8051 series) have true bidirectional ports. That means that they will act as both an input and output at the same time. If a signal is input they work as an input and if a value is output they act as an output.

The PIC does not work quite like that. The ports pins can act as either inputs or outputs, but not both at the same time. So as part of the set-up routine, it is necessary to state whether the port pin is an input or output.

The PIC18F452, which we are using has about 60 possible input and output signals available but only about 40 actual pins. This means that each pin, in addition to being a port pin, also has at least one other function. This is important to realise because in some cases it affects the way the pin is connected and in other cases it may mean that SFRs associated with different modules in the PIC affect the operation of the ports.

14.1. Port Overview

The PIC18F452 has five ports: Port A, Port B, Port C, Port D and Port E. Ports B to D are all 8-bits wide (which would probably be thought of as *normal* as the 452 is an 8-bit controller). Port A, however, is only 7-bits side and Port E, 3-bits wide.

We also use some PIC18F252s which only have three ports, A, B and C.

Each port has three SFRs associated directly with it.

TRIS

TRIS is short for tristate, and is the SFR used to determine whether a port pin is an input or an output. When writing in binary a 1 represents setting that pin to input and 0 sets it to output.

You can set each pin independently, so you could write:

TRISB=0b01010101;

Notice the addition of 'B' to the TRIS command to specify the port referred to. This command sets bits 7, 5, 3, and 1 to outputs (0) and 0, 2, 4 and 6 to inputs (1). Use TRISA for Port A and TRISC for port C.

Each of the SFRs can be bit-addressed using TRISAbits, TRISBbits, TRISCbits, TRISDbits or TRISEbits respectively. Table 1 below gives both alternate names for the bits for each port. Note that Port E bit 3 and Port A bit 7 are blank because Port E is only 3-bits wide (0:2) and Port A only 7-bits wide. Note also that Port E bits 4-7 are used as flags for the second function of Port E. See the Port E specific Information section for more details on these.

By default, all ports configure as inputs on power up or reset but it is good practice to include configuration of TRIS registers for all ports used.

Bit number		TRISAbits.	TRISBbits.	TRISCbits.	TRISDbits.	TRISEbits.
0	Name	TRISA0	TRISB0	TRISC0	TRISD0	TRISE0
	Alternate Name	RA0	RB0	RC0	RD0	RE0
1	Name	TRISA1	TRISB1	TRISC1	TRISD1	TRISE1
	Alternate Name	RA1	RB1	RC1	RD1	RE1
2	Name	TRISA2	TRISB2	TRISC2	TRISD2	TRISE2
	Alternate Name	RA2	RB2	RC2	RD2	RE2
3	Name	TRISA3	TRISB3	TRISC3	TRISD3	
	Alternate Name	RA3	RB3	RC3	RD3	
4	Name	TRISA4	TRISB4	TRISC4	TRISD4	IBF
	Alternate Name	RA4	RB4	RC4	RD4	
5	Name	TRISA5	TRISB5	TRISC5	TRISD5	OBF
	Alternate Name	RA5	RB5	RC5	RD5	

6	Name Alternate Name	TRISA6 RA6	TRISB6 RB6	TRISC6 RC6	TRISD6 RD6	OBOV
7	Name Alternate Name		TRISB7 RB7	TRISC7 RC7	TRISD7 RD7	PSPMODE

Table 1: TRIS bit names

Port

The second SFR associated with the ports is helpfully called *port* and is the data register for the port. This is the register you read from or write to in order to input or output via the port.

Again, the main *port* name has A, B, C, D or E appended to access the required port and again the port bits can be accessed individually using *PORTxbits*.

It can sometimes be more helpful to access port pins individually, particularly if some pins are inputs and others outputs or if some port pins are used for alternate functions.

To access the port bits individually, you use the alternate name given to that bit in the TRIS bit table above, that is Port A bit 0 is:

PORTAbits.RA0

To access Port C bit 3:

PORTCbits.RC3

In general:

PORT**x**bits.R**xn**

will access the bit of the port where **x** is the port letter, A, B, C, D or E, and **n** is the bit number 0-7 (6 for Port A, 2 for Port E).

Because these pins have alternate uses, as stated above, several alternate names are given to the port bits based on each of these uses. For example the three Port B pins which can act as external interrupt inputs can also be identified as such:

PORTBbits.INT0

PORTBbits.INT1

PORTBbits.INT2

Shortly we will look at each of the ports in turn and there we will discuss the various alternate functions and names available.

LAT

There is a third SFR associated with each port, this is the LAT or latch register. This is the actual physical output latch for the port pin. It remains in position and active regardless of whether the pin is an input or an output. *However*, most of the time we don't need to worry about it.

In output mode, the *port* command will write to the LAT latch automatically and read from the LAT latch output. In input mode, the *port* command will read the voltage at the input pin itself. For completeness, these latches can be directly addressed as LATA, LATB, LATC, LATD or LATE.

They can also be accessed as individual bits either by:

LATAbits.LATA0

LATAbits.LA0

Both of these will access bit 0 of Port A's latch. In general:

LAT**x**bits.LAT**xn**

LAT**x**bits.L**xn**

will access the bit of the port where **x** is the port letter, A, B, C, D or E, and **n** is the bit number 0-7 (6 for Port A, 2 for Port E).

14.2. Port Specific Information- Port A

Each of the ports have slight variations in their implementation, primarily to meet the requirements of the alternate functions.

The main alternate function of Port A is to act as the analogue to digital converter module analogue inputs. Analogue and digital signals have distinctly different requirements in terms of their input circuitry so it is necessary to formally switch between them. The section on the analogue to digital converter module will look at how to set the port up to operate as an ADC but it is important to note that there are ADC port configuration bits located as part of the ADC control SFRs which need setting up to ensure that the pins function correctly in the digital domain.

Table 2 shows the alternate functions for Port A with alternate function bit names where appropriate.

Port A bit	Alternate Function 1	Alternate Function 2	Alternate Function 3
0	Analogue Input AN0		
1	Analogue Input AN1		
2	Analogue Input AN2	ADC V_{ref-}	
3	Analogue Input AN3	ADC V_{ref+}	
4	Timer 0 Clock in T0CKI		
5	Analogue Input AN4	/SS	LVDIN
6	OSC_2	Internal Clock Out CLKO	

Table 2: Port A Alternate Functions.

Basic Port Configuration

Below, a basic digital port set-up configuration is given, showing which SFRs need to be set and the values which need loading. In each case **x** is either a 1 or 0. In some cases this may be because either will work, in other cases, the setting may be situation-specific, that is in the TRISA register set 1 for input and 0 for output; and **p** is used to set the ADCON1bits according to the values in table 3.

TRISA=0b**xxxxxxxx**;

ADCON1bits.PCFG=0b**pppp**;

pppp	AN7/ RE2	AN6/ RE1	AN5/ RE0	AN4/ RA5	AN3/ RA3	AN2/ RA2	AN1/ RA1	AN0/ RA0
0000	A	A	A	A	A	A	A	A
0001	A	A	A	A	V_{ref+}	A	A	A
0010	D	D	D	A	A	A	A	A
0011	D	D	D	D	V_{ref+}	A	A	A
0100	D	D	D	D	A	D	A	A
0101	D	D	D	D	V_{ref+}	D	A	A
011x	D	D	D	D	D	D	D	D
1000	A	A	A	A	V_{ref+}	V_{ref-}	A	A
1001	D	D	A	A	A	A	A	A

1010	D	D	A	A	V_{ref+}	A	A	A
1011	D	D	A	A	V_{ref+}	V_{ref-}	A	A
1100	D	D	D	A	V_{ref+}	V_{ref-}	A	A
1101	D	D	D	D	V_{ref+}	V_{ref-}	A	A
1110	D	D	D	D	D	D	D	A
1111	D	D	D	D	V_{ref+}	V_{ref-}	D	A

Table 3: Port Configuration Settings (A- Analogue in, D- digital I/O).*

Note that not all port configuration options exist on the PIC18F252. Those which are not implemented should not be used.

At power-on, Pins RA0 to RA3 and RA5 are configured as analogue inputs, with the remainder digital inputs. TRISA needs to be correctly configured as inputs when using port A in analogue mode.

It is also useful to note that pin RA5 is an open-drain. This means that it requires an external resistor to allow the output to go high. If this is omitted the output will go low or float. This is required because pin RA5 can also be used to detect a low voltage on the power supply line.

* From PIC18FXX2 datasheet p182 Microchip Corp

14.3. Port-Specific Information- Port B

Different parts of port B have different alternate functions. Bits 0, 1 and 2 double as the external interrupt pins. Bits 5, 6 and 7 are used for programming the PIC, this includes when using the PICkit.

Port B can be configured to operate in open-drain mode, where the output can pull down or float, or can be configured with an internal pull-up. This is described as a *weak pull-up*. This means that when the output is not set low, it will be able to pull the output high, but it will not be capable of delivering much in the way of current. So if you connect any form of load between the output and ground, the PIC will struggle to pull the output high. It is better therefore to connect the load between 5V and the PIC pin.

In addition to being able to configure port B bits 0, 1 and 2 as external interrupts, it is also possible for the interrupt system to be triggered by a change on any of the remaining port B pins. There is more information about setting this up in the session on PIC Interrupts.

Port B bit 5 can also act as the CCP2 I/O pin, this is in preference to the port C pin.

On power up, the port configures as an input.

Basic Port Configuration

To configure Port B, it is necessary to configure the port B tristate register, and switch on the pull-ups if required. The pull-up switch is located rather surprisingly, the one of the interrupt control registers. A basic configuration would be:

TRISB=0b**xxxxxxxx**;

INTCON2bits.RBPU=0;

Note that the RB pull-up is active low, that is, a 0 needs to written to this bit to turn on the pull-up, and a 1 switches the pull-up off.

In addition, if a port pin change needs to trigger an interrupt, or if the port pins are to be configured as interrupts then these need to be configured using the interrupt control registers as detailed in the Interrupts section.

Table 9.4 in the datasheet gives full information about all the SFRs associated with port B operation.

14.4. Port-Specific Information- Port C

Port C has alternate functions as various forms of serial input and output, and as the external connections to the *CCP*, that is *Capture Compare and Pulse Width Modulation* modules. These are shown in table 4.

Port Pin	Alternate 1	Alternate 2
RC0	Timer 1 oscillator output T1OSO	Timer 1 external clock input T1CKI
RC1	Timer 1 oscillator input T1OSI	CCP2 module input/output (direction depends on CCP configuration)
RC2	CCP1 module I/O	
RC3	Serial Peripheral Interface Clock SCK	I^2C Clock SCL
RC4	Serial Peripheral Interface Data In SDI	I^2C Data I/O SDA
RC5	Serial Peripheral Interface Data Out SDO	
RC6	UART Asynchronous Transmit output TX	UART Synchronous Clock CK
RC7	UART Asynchronous Receive Input RX	UART Synchronous Data DT

Table 4: Port C Alternate Functions

Port C does not have any specific registers in addition to TRISC which need setting to control the port. On power up or reset, the port configures as an input.

14.5. Port Specific Information- Port D

Port D has alternate functions as a 'Parallel Slave Port'. This port is designed for interfacing to a microprocessor if used in its primary function of an interface controller. These are shown in table 5.

Port Pin	Alternate
RD0	PSP0
RD1	PSP1
RD2	PSP2
RD3	PSP3
RD4	PSP4
RD5	PSP5
RD6	PSP6
RD7	PSP7

Table 5: Port D Alternate Functions

Port D uses TRISD which need setting to control the port. On power up or reset, the port configures as an input. When running as a parallel slave port, TRISE bits 4-7 are used to set up this mode.

14.6. Port Specific Information- Port E

Port E has alternate functions as the read, write and chip select control signals for the parallel slave port, and three further analogue inputs. These are shown in table 6.

Port Pin	Alternate 1	Alternate 2
RE0	\(\overline{RD} \)	AN5
RE1	\(\overline{WR} \)	AN6
RE2	\(\overline{CS} \)	AN7

Table 6: Port E Alternate Functions

Port E uses ADCON in the same way as port A to define the analogue inputs. In PSP mode, bits 4-7 of TRISE act as flags and mode bits as listed in table 7 below. On power up or reset, the port configures as an input.

TRISEbit.	Name	Function
7 IBF	Input Buffer Full	Data has been sent from the external source which has not yet been read by the PIC
6 OBF	Output Buffer Full	Data has been written to the buffer which has not yet been read
5 IBOV	PSP input buffer overflow	The input buffer has been written to for a second time before being read and cleared. If this error occurs it needs manually clearing in software.
4 PSPMODE	PSP Mode	1- PSP active 0- Normal port digital/analogue mode

Table 7: TRISE Parallel Slave Port Control

15. PIC Interrupts

Microcontrollers need to respond to many events which happen periodically, or even spasmodically. Microcontrollers run fast enough that even an event which occurs several times every second could be classed as occasional. Other events are much more random and rarer in their occurrence, for example a power failure alarm sensor signal which may occur once a year or even less.

There are two ways that the CPU can detect that an event like this has occurred, known as *polling* and *interrupt driven*.

Suppose for example, you need to get up at five o'clock in the morning. You have two options:

1. You keep looking at the alarm clock every 5 minutes all night until it is five o'clock and then you get up.
2. You go to sleep and when the alarm goes off you wake up respond by getting up.

The first is an example of polling and the second interrupt driven. The first will work but in the morning, you would be very tired having had a disturbed night's sleep.

The same principles apply to the microcontroller, it can poll an input source, checking every so often to see if the input has changed and needs responding to, or it can wait until the input sends a signal which flags to the microcontroller that something has happened to which it needs to respond. Both of these would work but polling will use up processor time having to keep doing the checking, particularly if the response needs to be quick and so the checks need to be made frequently. An interrupt driven response means that the processor can get on with doing other tasks but as soon as the interrupt is generated it will shelve those tasks and attend to the input which has generated the input.

In this session we will look at how interrupts work in hardware on the PIC and how we can handle them in software in C.

We have already seen that settings on the PIC are controlled by writing values to memories known as *special function registers* or SFRs, and it won't surprise you to learn that interrupts are handled in just the same way. Interrupts are turned on and off by writing 1s or 0s to specific bits in SFRs and an interrupt is triggered when a 1 is written to the appropriate bit in an SFR. It will also not surprise you that these SFR bits are given names in the header file to make them easier to identify and access in software.

15.1. Levels of Importance

Depending on the specific device you are using, some PICs operate a multi-level interrupt structure. Basic PICs just have a single level of interrupts (all interrupts are equally important). The PIC we are using has two levels of interrupt, high and low. Some others may have more.

When an interrupt occurs, the program will jump from what it is doing to handle the interrupt but no other interrupts can be processed until the first one is dealt with. In a mutli-level interrupt structure 'more important' interrupts can interrupt a lower level interrupt but a lower level one cannot interrupt a more important one.

You could think in terms of a more senior ranking officer taking priority.

This can be useful in larger systems with lots of interrupt sources. If many sources flag interrupts at the same time it could still take a few moments for the last one to be responded to. If it is *vital* that a particular interrupt is handled *now* you can make that the high priority interrupt and then all others will have to wait until that one is handled.

15.2. Interrupt Process

Just because a module in the system (or an external input) can generate an interrupt does not mean that is necessarily will. Before any source can generate an interrupt, you must:

1. Turn on the interrupt system. Setting a bit in one of the interrupt control SFRs, the *global interrupt enable,* turns on the interrupt system. No interrupts are active unless this is set.
2. If using a *peripheral interrupt,* of which more shortly, you also need to set a second bit which is the global enable for this set of interrupts.
3. Finally, you need to turn on the relevant specific interrupt sources by setting the appropriate bit for each interrupt being used. Normally, you would only activate the interrupt sources you definitely need to use to prevent unnecessary interrupts being triggered.

When the interrupt situation occurs, the interrupt will be generated- technically, this is known as an interrupt request. This occurs by the source setting the appropriate *interrupt flag* in one of the interrupt registers. Internally, within the PIC, the setting of any of these bits will be detected and the interrupt responded to.

The most common way for the interrupt response to occur is in the form of an *interrupt vector*. This is a specific address in program memory to which the microcontroller will jump when an interrupt is detected. There are a number of slightly different ways that this can be implemented.

1. In old Z80-based systems, the interrupt vector was stored in the interrupt generating device and could be any value. When the interrupt was detected, the Z80 would ask for the vector to be placed on the data bus and would then jump to that location.
2. 8051-based microcontrollers have fixed interrupt vectors, with a different vector for each of the standard sources. In that way each source can have its own unique program, known as the *interrupt service routine,* to respond to that request.
3. The PIC is slightly different, it has either one or two interrupt vectors. One for low priority and (if implemented) a second for high priority interrupts. It is necessary for the program to determine which source has generated the interrupt and respond accordingly. Fortunately, as you will see, the version of the C compiler we use makes this quite a simple task. Some of the more complex devices implement full interrupt vectoring, but not those we will be using.

It is obviously important to ensure that no other program code is stored at these addresses otherwise something very strange could occur. It is also important to ‘archive’ all data relating to the current task before running the interrupt service routine, otherwise you could not successfully return afterwards. Fortunately, the C compiler interrupt handling will handle a lot of this for you (in assembly language you would need to remember to do this yourself).

One thing that must be remembered is that the interrupt flags do not automatically reset themselves so part of the interrupt service routine must to be reset the flag, otherwise the interrupt will continually recur.

15.3. Interrupt Registers

On the PIC18F452 there are numerous interrupt sources. As the facilities available on the PICs have increased, so the number and variety of interrupt sources have increased. Over time, interrupts have been divided into two groups: *interrupts* and *peripheral interrupts*. However, the division between the two is somewhat arbitrary because some timers, for example, appear under the interrupt area and some under the peripheral interrupt area. There are three general interrupt control SFRs, *INTCON*, *INTCON2* and *INTCON3*. There are used to control the interrupts- and also to provide overall interrupt control.

INTCON

INTCON is the main interrupt control SFR and contains the global and peripheral enable bits in addition to some specific enables and flags.

It can be addressed as a byte by using the INTCON name, or one bit at a time by using the '*INTCONbits*' structure followed by a full stop and the name of the bit to be accessed. Table 1 shows the bit names for the bits in the INTCON SFR. You will notice that some bits have alternative names declared. Any of these can be used to access the relevant bit.

To set the global interrupt enable you could use any of the following:

1. INTCONbits.GIE_GIEH=1;
2. INTCONbits.GIE=1;
3. INTCONbits.GIEH=1;
4. INTCON=0b10000000;

Note that option 4 writes to INTCON as a byte and not only sets the global interrupt enable, but also resets the peripheral interrupt enable and the receive buffer, external interrupt 0 and timer 0. If you were wishing to set a number of these at the same time, then writing as a byte might be more efficient, otherwise, writing to an individual bit is normally easier. Obviously, you could also express option 4 in decimal or hex too.

Bit	Function	Name	Alt Name 1	Alt Name 2
0	Port B Change Flag	RBIF		
1	External Interrupt Zero Flag	INT0IF	INT0F	
2	Timer Zero Interrupt Flag	TMR0IF	TMR0F	
3	Port B Change Enable A change on any of the port B pins RB7, RB6, RB5 or RB4 will cause an interrupt.	RBIE		
4	External Interrupt Zero Enable	INT0IE	INT0E	
5	Timer Zero Interrupt Enable	TMR0IE	TMR0E	
6	Peripheral Interrupt Enable (Also known as Global Interrupt Enable Low)	PEIE_GIEL	PEIE	GIEL
7	Global Interrupt Enable (Also known as Global Interrupt Enable High)	GIE_GIEH	GIE	GIEH

Table 1: INTCON Bit Names and Functions

INTCON2

Only six of the bits in INTCON2 are used on the PIC18F452. Note that the remaining bits may be used by other PIC devices so care should be taken not to write to these bits by accident in case you change target at a later date.

INTCON2 contains the interrupt priority bits for port B and timer 0, and selection bits to allow the external interrupts to be triggered by either a rising (leading) edge (0 to 1 transition on the pin) when the bit is a 1 or a falling (trailing) edge (1 to 0 transition) when the bit is a 0.

Table 2 shows the bit names and functions for the INTCON2 SFR. You will notice that bit 7 is slightly anomalous in that it doesn't seem to have anything to do with the interrupt system. It contains an active low (0 turns on, 1 turns off) global control for the port B pull-ups. When this bit is zero, the internal pull-ups on port B are activated.

Note that, unlike many SFRs, the power up value for these bits is one except the two unused bits. This means that high priority is selected by default, rising edge triggering on the external interrupts and port B pull-ups are off.

Bit	Function	Name	Alt Name 1
0	Port B interrupt Priority 0 – low, 1 – high	RBIP	
1	*Unused*		
2	Timer 0 interrupt priority 0 – low, 1 – high	TMR0IP	T0IP
3	*Unused*		
4	External interrupt 2 edge selector 0 = falling 1 = rising	INTEDG2	
5	External interrupt 1 edge selector 0 = falling 1 = rising	INTEDG1	
6	External interrupt 0 edge selector 0 = falling 1 = rising	INTEDG0	
7	Port B weak pull-up global activation bit (Active low- a zero turns this function on)	NOT_RBPU	RBPU

Table 2: INTCON2 Bit Names and Functions

INTCON3

In some ways, the structure of INTCON3 is easier to understand than INTCON2. It consists of the enable, flag and priority bits for the two other external interrupt pins, *INT1* and *INT2*. Again, 6 of the 8 bits in this register are used.

Table 3 shows the names and alternate names for the bits in INTCON3, which are hopefully fairly self-explanatory.

Bit	0	1	2	3	4	5	6	7
Name	INT1IF	INT2IF		INT1IE	INT2IE		INT1IP	INT2IP
Alt	INT1F	INT2F		INT1E	INT2E		INT1P	INT2P

Table 3: INTCON3 Bit Names

Peripheral Interrupts

On the PIC18F452, there are then a total of 12 interrupt sources which come under the peripheral interrupt grouping. As with most of the standard interrupts, for each of these peripheral interrupts there are three control bits:

1. Interrupt Enable. This is the bit which is used to turn the interrupt on or off. For consistency, all the interrupt enable names include 'IE' in them.
2. Interrupt Flag. This is the bit which is set by the source when interrupt occurs. These all contain 'IF' in their names.
3. Interrupt Priority. The value on this bit determines whether the interrupt is high priority (1) or low priority (0) which is the default. These all contain 'IP' in their names.

The interrupt bits are arrange across a set of 6 SFRs, with the enable, flag and priority bits for any given source being in the corresponding bits of the enable, flag or priority registers. A total of 6 SFRs are required because with 12 possible sources, it takes one and a half bytes of register space to store each. It would be possible to arrange the bits so that one SFR was half enable bits and half flags but that would over-complicate matters, so there are two interrupt enable registers: *PIE1* and *PIE2*; two interrupt flag registers: *PIR1*and *PIR2*; and two priority registers: *IPR1* and *IPR2*. PIE stands for **P**eripheral **I**nterrupt **E**nable, PIR for **P**eripheral **I**nterrupt **R**equest and IPR for **I**nterrupt **Pr**iority.

Each of these registers can be written to as a byte, by writing a value to that name, or by assigning a variable to the register. However, since each bit in the registers refers to a different interrupt source, there is little benefit in addressing these registers as a whole byte since it could accidentally change the status of a different interrupt source unintentionally.

Each bit is available individually using the standard bit-naming convention:

[bytename]bits.[bitname]

So to enable the interrupt for timer 1- which is stored in the PIE1 register write:

PIE1bits.TMR1IE=1;

Where TMR1 is the shorthand for **T**i**m**e**r 1** and IE is **i**nterrupt **e**nable.

Table 4 shows which bits in each of the enable, request and priority SFRs PIE1, PIR1 and IPR1 are associated with which interrupt source. It also shows what I have called the *base mnemonic*. This is the standard start to the source name. To access a particular bit, type the base mnemonic followed by IE for the enable bit, IF for the flag (request) bit and IP for the priority bit.

Table 5 shows the same information for the second set of SFRs, PIE2, PIR2 and IPR2. Some of the sources shown in this table may need further explanation, because they are not quite so obvious.

PIE1, PIR1, IPR1 bit	Source	Base Mnemonic	Resetting Interrupt Flag
0	Timer 1	TMR1	Must be cleared by user in their code
1	Timer 2	TMR2	Must be cleared by user in their code
2	Capture/Compare/PWM module 1	CCP1	Must be cleared by user in their code
3	Master Synchronous Serial Port	SSP	Must be cleared by user in their code
4	Serial UART (Universal Asynchronous Receiver Transmitter) Transmitter	TX	Cleared when TXREG is written
5	Serial UART Receiver	RC	Cleared when RCREG is read
6	Analogue to Digital Convertor	AD	Must be cleared by user in their code
7	*This bit is not implemented on the PIC18F252 but is used for the Parallel Slave Port Interrupt on other devices including the PIC18F452.*	PSP	Must be cleared by user in their code

Table 4: PIE1, PIR1 and IPR1 bit sources

PIE2, PIR2, IPR2 bit	Source	Base Mnemonic
0	Capture/Compare/PWM module 2	CCP2
1	Timer 3	TMR3
2	Low Voltage Detect Interrupt	LVD
3	Bus Collision Detect Interrupt	BCL
4	Data EEPROM/Flash write operation interrupt	EE
5-7	*Unused*	

Table 5: PIE2, PIR2 and IPR2 bit sources (All flags need clearing in software)

Low Voltage Detect

Low voltage detect is an interrupt which is generated if the supply voltage to the PIC drops below a certain voltage. This may be useful in battery powered devices to indicate that the battery is running low. The PIC data sheet suggests activating this occasionally to test and then deactivating to save power since batteries normally run out quite slowly.

Bus Collision Detect

In many situations more than one output may be connected to a common data line. Generally speaking, in introductory digital systems courses it is taught that you cannot connect digital outputs directly together. However, provided the right precautions are taken it is possible in some situations. In the days before CAT5 etc cabling was used for network connections, computers were connected to the Internet using Tbase10 coaxial cabling. This was a common line connecting all computers together. If your computer wanted to send data to the Internet, it would first check that the line was clear and would then start transmitting. However, because these cables were very long, it was possible that another computer could also have started transmitting at the same time, thinking the line was clear. This would be detected if the value your computer placed on the line was not the same as that detected. This was known as a collision, and would trigger the need to resend the message. This interrupt will trigger in that sort of situation on the I^2C bus which also a multipoint common line system.

EEPROM/Flash memory write

The PIC18 series contains an amount of data EEPROM or Flash memory which can retain the data written to it for many years even which the power is removed. Writing to this type of memory takes longer than to a normal register, so this interrupt can be used to confirm the completion of a memory write cycle.

Although they are included for completeness, on this course you will probably not need to use the sources in this second set of peripheral interrupt registers.

15.4. Interrupt Handling in C

The first thing to say is that interrupts aren't handled as standard in C. This is because when writing C for a PC, or Mac or similar, any code written by the programmer sits above the operating system and it will be this which will be responsible at a much lower level for handling interrupts. As such, C was not written with interrupt-handling built in.

This is not a major problem because the compiler writers have developed 'add-ons' to allow programmers writing for microcontrollers and similar devices at this lower level to handle interrupts. The only difficulty with this approach is that because it is not part of standard C, each compiler writer has developed their own solution to the problem, so whereas you could transfer a segment of standard C code from one compiler to another and it would (most of the time) still compile, for the interrupt handling each uses different structures and different syntax. The interrupt handling is actually implemented via compiler directives. These are commands which tell the compiler that the code is an interrupt service routine (and which interrupt it is in response to), and when the code is converted to the machine code to actually be downloaded to the microcontroller, the code is placed at the correct memory location (the interrupt vector).

The XC8 compiler that we are using allows one low priority and one high priority interrupt function to be defined for the PIC18F452.

Specifying an Interrupt Function

To specify the high priority interrupt, use the function:

```
Void __interrupt() HighInt(void)
```

Or…

```
Void __interrupt(high_priority) HighInt(void)
```

Looking in detail at this line:

Void is the standard function specifier at the start of the line and is used at the end of the line to indicate that no values are transferred to the function. This is always the case for an interrupt service routine.

(Underscore underscore) Interrupt() is the interrupt service routine directive. This tells the compiler that this function is an interrupt service routine. This is the high priority interrupt directive where two priorities are used. The PIC is set up such that high priority is the default (that is it uses the same interrupt vector as is used on single priority PICs). Low priority is then a *reduced priority* interrupt. Note that you can explicitly state 'high_priority' but it is not needed.

HighInt is the name of the function. This could be anything you choose but make it something meaningful. *You should note that although I have called it HighInt here, this does not have any effect on the level of the interrupt priority.*

To specify the low priority function use:

```
Void __interrupt(low_priority) LowInt(void)
```

You may wish to change the function name too whihc again does not set the interrupt priority.

Structure of an Interrupt Function

Interrupt functions have a fairly standard structure to them:

1. Test for the interrupt. This is needed because all interrupt sources will either generate the high or low interrupt so you need to check which source it was in order to respond accordingly.
2. Respond to the interrupt. This will include resetting the interrupt flag
3. Return. At the end of the interrupt function you need to have a return keyword

The resulting function will look something like this:

```
Void __interrupt() HighInt(void)

{

    If(PIE1bits.TMR1IE && PIR1bits.TMR1IF){
```

```
        PIR1bits.TMR1IF=0;
            ⋮
    }
    If…
    Return;
}
```

If(PIE1bits.TMR1IE && PIR1bits.TMR1IF)

This 'if' statement is the *test for interrupt* in step 1 above. It tests that timer 1's interrupt is enabled and that its flag is set. It is necessary to test the peripheral interrupts in this way because some will still assert the flag even if interrupts are not enabled. This statement would be repeated several times for the different interrupt sources in use.

{

The standard function start, remember to include the corresponding '}' to end each function

PIR1bits.TMR1IF=0;

This resets the flag. You need to do this manually.

⋮

The vertical dots are just to indicate that this is where the rest of your code goes. Try to keep this to a minimum so you don't end up with interrupts coming in on top of each other.

If…

This is the start of the test for the next interrupt. Obviously, this can be left out if only one interrupt is used, or should be fully completed if needed.

Return;

This should be the last line of the function. It is the *return from interrupt* command.

In some cases, for example some timer modes, if you wish to restart the timer this needs to be done manually, and should be included as part of the interrupt service routine too.

16. PIC Timers

Timers are central to many operations of microcontroller systems. If you need to wait, or determine a certain period of time you have two basic options:

1. Use a 'For Loop' which does nothing. This will *waste* the required amount of time but it does this by fully occupying the CPU of the microcontroller during this time. The CPU cannot do anything else and as a result it is quite an inefficient way to proceed.
2. Use a timer, which will *flag* (by means of an interrupt) when the allotted time has elapsed. This has a number of advantages.
    a. It is a separate module within the PIC so it runs outside the CPU, freeing the CPU up to do other things.
    b. It will be more accurate than the CPU count because it won't get interrupted but other tasks.
    c. It is more flexible in its way of operation, and has a clearly defined way of calculating the relationship between the clock frequency of the PIC and the timing rate.

The PIC18F452 has four timers, *timer 0, timer 1, timer 2* and *timer 3*. Each has slightly different facilities, and so can be used for slightly different jobs. The timers can also be used alongside the CCP modules to provide a range of timing and counting functions:

1. Timing: The timer will give a signal after a fixed interval from when it was triggered.
2. Interval Measurement: The timer will detect an input (normally on an external pin) and will start counting. It will then flag when the next occurrence of that signal occurs. You end up with a time value in 'PIC timer units'. Depending on how often and/or how regularly these triggers occur, there are options to slow the timer down and/or flag completion after a number of occurrences have occurred- this has the effect of averaging or smoothing out slight variations in timing of occurrences.
3. Event Counting: The timer 'clock' signal is redirected to what is normally an external source. Rather than measuring the time elapsed between occurrences of an event on the pin, it will count the number of occurrences.
4. Pulse Width Modulation (PWM): In this mode, the timer will count all the way from zero to its maximum, and will then start again at zero. The time it takes to do this will set the *frame rate* of the pulse width modulator. A second data value is also loaded in to the module. The PWM output will be set (1) when counting starts, and will reset when the timer count passes the data value loaded. Changing the data value, allows the programmer to adjust the ratio of high to low time on the output pin, normally called the *mark space ratio*.

16.1. Timer Module General Principles

Focussing on the timer modules in timing mode, the clock signal which drives the timer is normally derived from the system clock. Because this clock is quite fast, it is fed through what is known as a *prescaler* before arriving at the heart of the timer. This prescaler slows the clock rate down, and because it is a digital system the prescaler operates using a set of binary values. The values available are different for the different timers.

This slowed-down clock is then used to drive the heart of the timer, which is actually a counter. If you count through a set number at a given rate, then this task will take a certain, fixed, time to complete. This is the principle on which the timer works. This is the same principle which is used in the children's game, *hide and seek*. The seeker counts up to 100 (hopefully at a set speed) and this gives the hiders a certain amount of time in which to hide. In order to be able to vary the length of time taken, it is necessary to be able to load values in to the counter either to set the starting value or the ending value.

There are different ways in which timers can be constructed:

1. The starting value is loaded in to the counter. The counter will then count up from this value until it *overflows*- that is it reaches to 1 more than its maximum.
2. The ending value is loaded into a register. The counter starts at zero and counts up until its value equals that stored. At which point a flag is set. This type of operation is common with *free running* timers, these are ones which keep running once they flag the time. This is because the reference value is retained here, in option 1 the reference value is changed during the counting process.

Because this time might still be too short, some timers then have a further slowing down mechanism, known as a *post scaler*. It is somewhat unfortunate that both prescaler and post scaler abbreviate to ‘ps’ so it can be confusing working out which you are referring to.

As with all modules in the PIC, each timer has a control byte to set it up, and a data byte (or bytes) to load the start/stop value. It also accesses the relevant bits in the interrupt registers to flag when timing or counting is complete.

The larger the count, the longer the time taken, so it is common for the heart of the timer to be a 16-bit counter. This requires two bytes to operate, and the data ‘byte’ will actually be a two-byte pair (high and low)

Because each of the timers is slightly different it is necessary to look at each in turn to understand how they work.

16.2. Timer 0

Timer 0 can be an 8-bit or 16-bit timer, and can be clocked either from the system clock or an external source. It will always start counting from the value loaded in to the data register pair TMR0L:TMR0H.

A block diagram of timer 0 is shown in figure 1.

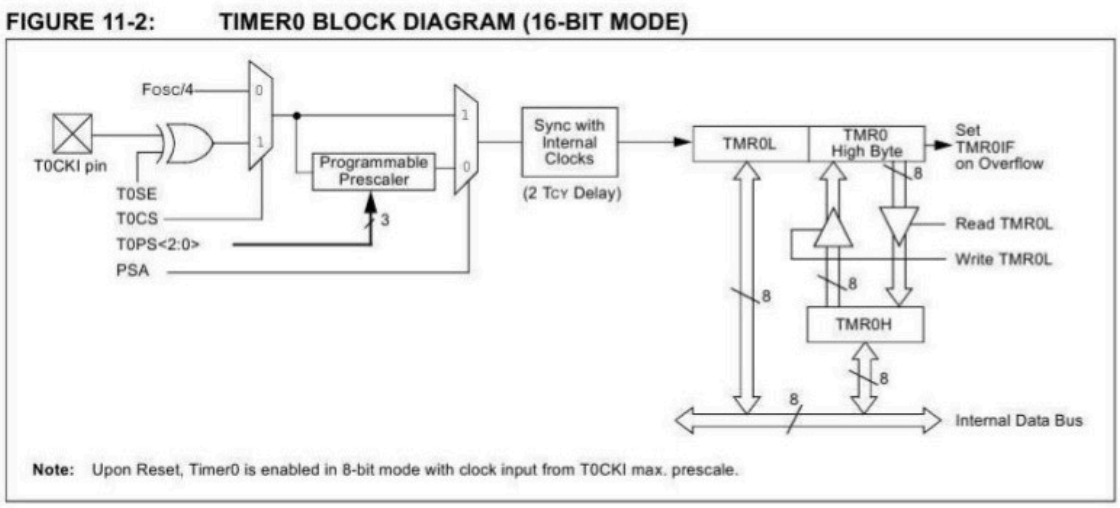

Figure 1: Timer 0 Block Diagram

Data Register Operation

Before looking in detail at the operation of the timer, we will quickly look at the operation of the data register pair.

These are two 8-bit registers which form the heart of the counter part of the timer. Because the PIC is an 8-bit device, it normally requires two transfers to load a 16-bit value, and timer 0 is particular about the order in which the bytes are written to and read from these registers.

When writing, the high byte *must* be written first.

When reading, the low byte *must* be read first.

The reason for this is that the high byte is actually temporarily held in a second 8-bit register between the data bus and the actual TMR0H register. Writing to TMR0H actually writes to this buffer register. TMR0L is directly accessed from the data bus and writing to this register actually writes to the whole 16-bit timer 0 data register, so TMR0L is written from the data bus, and at the same time TMR0H is transferred from the buffer register into the 8 more significant bits. If writing was to occur in the other order, the high byte would not be available for transfer.

Similarly, when reading, reading TMR0L reads the whole 16-bits of the TMR0 data register, placing bits 0:7 on to the data bus, and bits 8:15 into the internal buffer. Reading TM0H, then actually reads the internal buffer. This can be seen in the block diagram in figure 1, to the right hand side of the diagram.

Having said all this, the XC8 compiler actually makes life simple for us by providing a 16-bit TMR0 variable. If you write a 16-bit value to this variable, the compiler will then take care of loading (or reading) TMR0L and TMR0H in the right order.

Timer 0 Control

The control byte for timer 0 is shown in table 1 below.

MSB							LSB
TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0

Table 1: T0CON Special Function Register

TMR0ON: Note that this is TMR zero ON, there is a zero followed by a capital O in the middle of the name here. Writing a 1 to this bit, will activate the timer, a 0 will deactivate it.

T0CONbits.TMR0ON=1;//Timer 0 on

This bit defaults to a zero.

T08BIT: This bit selects 8-bit or 16-bit mode for timer 0. Note that this bit defaults to a 1, so timer 0 defaults to 8-bit mode. In 8-bit mode TMR0L is used as the data byte.

T0CS: Timer 0 Clock Source. The default zero in this bit means that the clock for the timer will be derived from the internal system clock. Setting a 1 here will allow the external pin T0CKI to be the clock source.

T0SE: Timer 0 Source Edge. This is effectively a programmable inverter, as can be seen in figure 1 because it is fed through an XOR gate with the external source. If this bit is set, clocking will occur on a high-to-low transition of the external clock pin (falling edge triggered). A zero in this bit will cause the timer to be rising or leading edge triggered. By default this bit is set, so falling edge triggering is selected.

PSA: Pre-Scaler Active. This determines whether the clock source is fed through a pre-scaler before being fed to the counter. Note that this bit is *active low* so a zero needs to be written to this bit to activate the pre-scaler.

T0PS: Timer 0 Pre-Scale. There a three bits which select the pre-scale value. These can be written to individually, or as a single 3-bit value. The pre-scaling is a binary weighting based on the pre-scale values, ranging from 000= ÷2, to 111=÷256.

Technically, the equation to determine the pre-scale value is:

$$PS=2^{(T0PS+1) }$$

More usefully, to determine T0PS:

$$T0PS=\frac{log(PS)}{log(2)} -1$$

Remember that PS must be a whole power of 2.

If you wish to set the pre-scaler to divide by 32 you would write:

T0CONbits.T0PS=100; //Set the pre-scale value

T0CONbits.PSA=0; //Pre-scale active

Calculating TMR0 and T0PS

The speed of counting of the timer is obviously governed by the frequency of the incoming clock signal.

If we are using the clock derived from the system clock, then the timer clock frequency is:

$$f_{tclk}=\frac{f_{osc}}{4}$$

The actual counting frequency, is then scaled down by the pre-scaler (if used) so:

$$f_{counting}=\frac{f_{tclk}}{PS}=\frac{f_{tclk}}{2^{(T0PS+1) }}$$

If the pre-scaler isn't used then $f_{counting}=f_{tclk}$.

On each counting pulse, the timer will count up by one, and will count from the value loaded in to the TMR0 register pair to the rollover value.

A little care needs to be taken here because timer 0 can operate in both 8-bit and 16-bit modes.

8-bit mode

TMR0 is a value between 0 and 255

Rollover value is 256

16-bit mode

TMR0 is a value between 0 and 65,535

Rollover value is 65,536

The number of counts completed before the timer times out is:

$$counts=Rollover-TMR0$$

So the frequency of timer 0 is:

$$f_{T0}=\frac{f_{counting}}{counts}=\frac{f_{osc}}{4\times2^{(T0PS+1)}\times(Rollover-TMR0)}$$

$$t_{T0}=\frac{1}{f_{T0}}=\frac{4\times2^{(T0PS+1)}\times(Rollover-TMR0)}{f_{osc}}$$

It is also useful to express this in terms of the value loaded into TMR0 to get a particular frequency at the output of the timer:

$$TMR0=Rollover-\frac{f_{osc}}{\left(4\times2^{\left(T0PS+1\right)}\times f_{T0}\right)}$$

An Example

We wish timer 0 to run at 440Hz, what values should be loaded into T0PS and TMR0 to achieve this. We will assume we are working in 16-bit mode and that the system clock is 12MHz.

Step 1:

We will first see if this can be achieved without the use of a pre-scaler:

$$TMR0=Rollover-\left(\frac{f_{osc}}{4\times f_{T0}}\right)$$
$$=65,536-\frac{12M}{4\times440}$$

$$=58,717.82\approx58,718$$

This will work, so we don't need the pre-scaler, and need to load 58,718 into TMR0. Fortunately, because the XC8 compiler can handle this directly, we don't need to calculate the 8-bit values for TMR0L and TMR0H. We will, however need to set up T0CON to 16-bit mode. This can be achieved by:

```
T0CONbits.T08BIT=0;
```

16.3. Timer 1

Timer one is a 16-bit timer which, unlike timer 0 cannot be operated other than in 16-bit mode.

The control register for timer 1 is very similar to that for timer 0, but with 1 replacing 0 as appropriate **BUT** the various bits are, for some reason, arranged in the opposite order, for example, the timer 1 on bit is bit 0, but the timer 0 on bit is bit 7. Fortunately, this does not matter to a great extent because we refer to these bits by name, and let the IDE sort out the exact bit allocations.

MSB							LSB
RD16	-	T1CKPS1	T1CKPS0	T1OSCEN		TMR1CS	TMR1ON

Table 2: T1CON Special Function Register

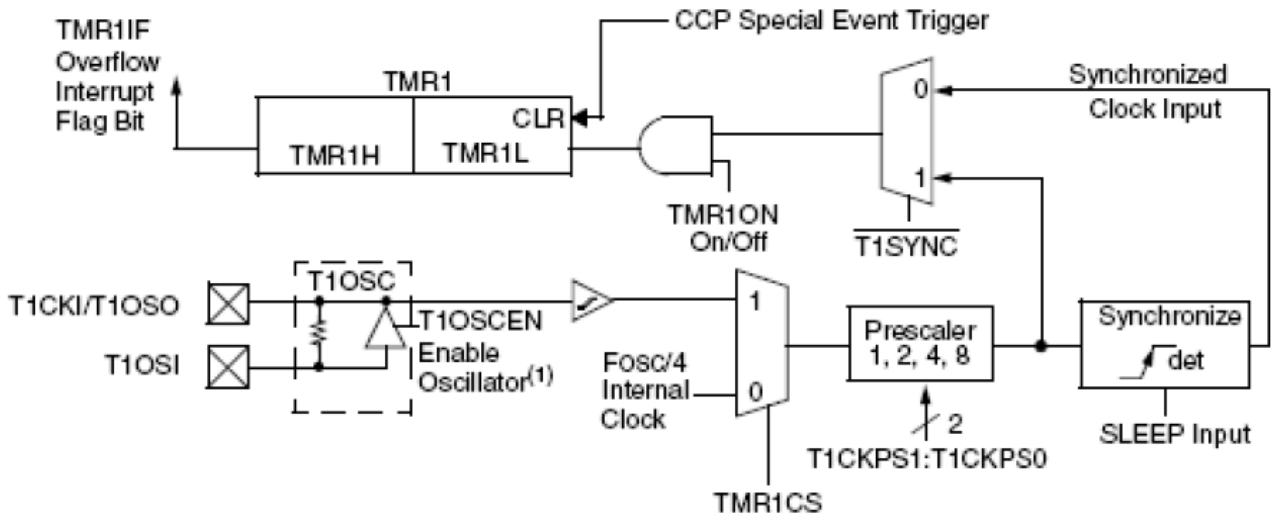

Figure 2: Timer 1 Block Diagram

From figure 2 you can also see that the block diagram is similar but not the same. In this section we will focus on the differences between timer 0 and timer 1.

The first difference is in the external clock input section. For timer 1, this is just a single pin into which is fed the clocking pulse. For timer 1, two external pins are available and it is possible to use these to build a simple RC oscillator to act as the clock source. Whether this oscillator is used is determined by the T1OSCEN (Timer 1 oscillator enable) bit of the T1CON SFR. A buffer and resistor are provided internally, the designer must provide a suitable capacitor externally to control the frequency of oscillation. Turning this off if not used saves power because it stops current flowing through the internal resistor.

The system/external clock select is the same as timer 0, but the pre-scaler, is both simpler and always connected. Only four options are available, as listed in table 3.

Pre-scale	**T1CKPS**
÷1	00
÷2	01
÷4	10
÷8	11

Table 3: Timer 1 Pre-scale Values

Even when using an external clock source, it may be desirable for the operation of the timer to be synchronised with the system clock. \(\overline{T1SYNC} \) is an active low bit, which allows this to occur. When this bit is reset, the external clock input will only be allowed through to clock the timer on the next system clock pulse.

Timer 1 contains a 16-bit data register, which can either be written to in two 8-bit data write cycles, or in one 16-bit write cycle depending on the value of RD16. Note that this is a slightly misleading description (although it is how the data sheet describes it). When RD16 is set, the read and write of timer 1's data registers is exactly the same as described above for timer 0. When in 8-bit mode, TMR1H and TMR1L can be written to independently.

The final difference to note is the CCP Special Event Trigger input. This allows the CCP module to control and/or interact with Timer 1. This allows timing measurements of captured inputs to be made for example.

Timer 1 can be operated on one of three modes:

Mode	TMR1SC	T1SYNC	Description
1	0	N/A	Normal Timer Mode, clocked from the system clock
2	1	0	Synchronous Counter Mode. External clock but synchronised to the system clock
3	1	1	Asynchronous Mode. External clock independent of the system clock.

16.4. Timer 2

Timer 2 is quite different in its structure from timers 0 and 1. It is an 8-bit timer, but with both a pre-scaler and a post-scaler. It also features an 8-bit comparator, so flags an output when the timer data register value equals the value stored in a second register known as *PR2*. A block diagram of timer 2 is shown in figure 3.

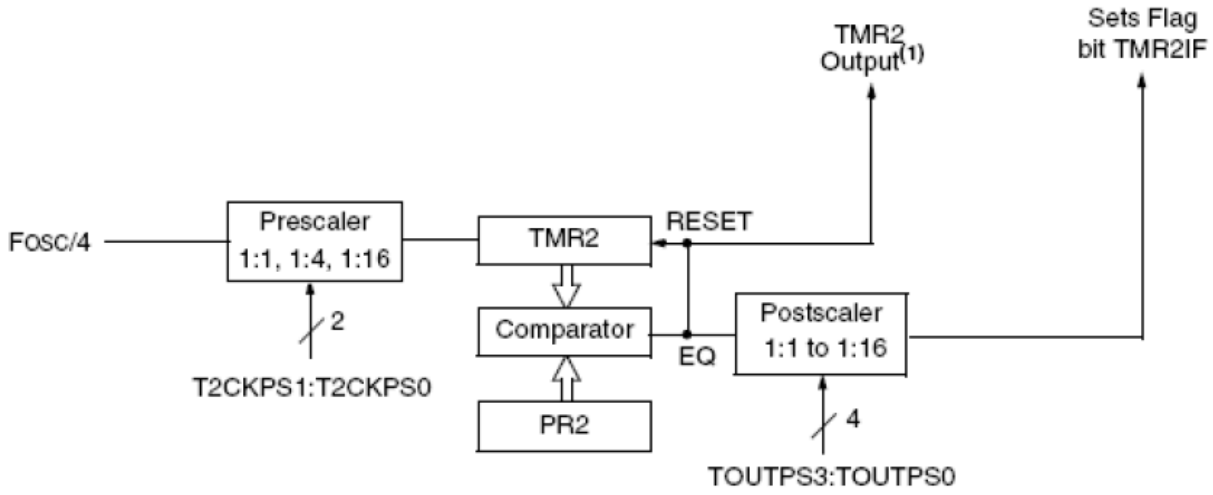

Figure 3: Timer 2 Block Diagram.

From the block diagram it is clear that the structure of the timer is simpler, with no access to external clock sources. Unlike timers 0 and 1 which can only set an interrupt flag when timing is complete, the basic timer signal is available as an external output. It should be noted, as can be seen in the block diagram, that this is before the post-scaler which determines the interrupt flag. Timer 2 is designed to be free-running, that is, when it times out, the timer automatically resets and starts over.

There are three settings which are contained in the T2CON register and which need to be set in order to make the timer function correctly: TMR2ON, T2OUTPS and T2CKPS. You will immediately notice that two of these have 'PS' as part of the name but both stand for different things, and this is pre/post issue highlighted earlier.

TMR2CKPS is the timer 2 (clock) pre-scaler. This is a 2-bit binary value in bits 0 and 1 of the T2CON SFR.

T2OUTPS is the timer 2 (output) post-scaler. This is a 4-bit binary value in bits 3 to 6 of the T2CON SFR.

TMR2ON is bit 2 of the T2CON SFR and bit 7 is unused.

Calculating the time of Timer 2

Timer 2 starts counting at the value loaded into the TMR2 timer data register and times out when the timer data register value equals the PR2 register value. This occurs when (PR2-TMR2)+1 counts have occurred. A count will occur on every timer clock signal, but this is after the pre-scaler, so the pre-scaler value will affect the timing. Finally, the post-scaler means that there is also a divider on the output of the timer.

Timer 2 interrupt rate $= prescale \times postscale \times ((PR2-TMR2)+1) \times clock$

It is helpful to put some more precise values on this and because there are a lot of ps's and pr's around, we will define:

$prescale=A$

$postscale=B$

And if we are using the system clock

$clock=\frac{4}{f_{osc}}$

So

$$T2_{int}=\frac{4\times A\times B\times((PR2-TMR2)+1)}{f_{osc}} \quad \text{Equation 1}$$

Tables 4 and 5 give values for the multipliers A and B, based on the values loaded in to the pre-scale and post-scale modules.

Pre-scale

T2CKPS	T2CKPS1	T2CKPS0	Multiplier (A)
00	0	0	1
01	0	1	4
10	1	0	16
11	1	1	16

Table 4: Timer 2 Pre-Scaler Values

Post-scale

TOUTPS	Multiplier (B)	TOUTPS	Multiplier (B)
0000	1	1000	9
0001	2	1001	10
0010	3	1010	11
0011	4	1011	12
0100	5	1100	13
0101	6	1101	14
0110	7	1110	15
0111	8	1111	16

Table 5: Timer 2 Post-Scale Values

It is common to need to calculate the value we need to load in to PR2 in order to achieve a certain frequency output (or a certain time period at $T2_{int}$). Remember $T=\frac{1}{f}$.

If we rearrange equation 1 in terms of PR2 and TMR2 we get:

$$(PR2-TMR2)=\left(\frac{T_{int} f_{osc}}{4AB}\right)-1$$

An Example

Suppose we wish to generate a frequency of 440Hz. The oscillator frequency is 12MHz. Determine the values of A, B, PR2 and TMR2.

Remember: A can be 1, 4 or 16, B must be a whole number between 1 and 16, and PR2 and TMR2 must be a whole number between 0 and 255.

Step 1:

We need to work out what pre-scale and post-scale values are needed.

$$T_{int}=\frac{1}{f}=\frac{1}{440}=2.27ms$$

Step 2:

Assume A and B are 1, and see if PR2/TMR2 fits

$$PR2-TMR2=\left(\frac{\left(\frac{12M}{440}\right)}{4}\right)-1=6,817.18$$

6,817 is much greater than 255 so pre-scaling and possibly post-scaling will be required.

Step 3:

Determine values for A and B:

Start by dividing the current PR2-TMR2 value by 255. This tells you how many times too big it is:

$$\frac{6817.18}{255}=26.73$$

This gives the lowest value multiplier we need ($A \times B > 26.73$) and because this value is greater than 16, it tells us we need pre-scaling and post-scaling.

We now need to look for a combination of which gets us close to 26.73. Then we adjust the value of PR2-TMR2 to get as close as possible.

If A is 16:

16 would appear not to work because, if A is 16 and B is 1, then we are a long way short but if B is 2 then we overshoot by quite a way.

If A is 4:

When A is 4, setting B to 6 would give 24, or B to 7 would give 28, so this is what we would need.

Step 4:

Repeat the initial calculation to see what the new value of PR2-TMR2 is:

$$PR2\text{-}TMR2=\left(\frac{\left(\frac{12M}{440}\right)}{4 \times 7 \times 4 }\right)-1=242.5$$

This isn't a whole number so we'd either need to round down or round up (242 or 243).

Step 5:

Check the error;

If we now feed the values of 242 and 243 into equation 1, we can find out how far out we are from our target of 440Hz.

$$f=\frac{1}{\left(\frac{4AB(PR2+1)}{f_{osc}} \right)}$$

$$=\frac{1}{\left(\frac{4\times28\times243}{12M} \right)}$$

$$=440.9Hz$$

Repeating for PR2-TMR2=243:

$$f=\frac{1}{\left(\frac{4AB(PR2+1)}{f_{osc}} \right)}$$

$$=\frac{1}{\left(\frac{4\times28\times244}{12M} \right)}$$

$$=439.1Hz$$

So in both cases we would be 0.9Hz out.

We could also look at increasing B to 8 and repeat the calculation of PR2-TMR2 to see if that gets us closer.

If we recalculate using B=8:

$$PR2\text{-}TMR2=\left(\frac{\left(\frac{12M}{440}\right)}{4 \times 8 \times 4 }\right)-1=212.07$$

This gives a value for PR2-TMR2 which is much closer to a whole number, so should give a smaller error, but again it is useful to check:

$$f=\frac{1}{\left(\frac{4AB(PR2+1)}{f_{osc}} \right)}$$

$$=\frac{1}{\left(\frac{4\times23\times213}{12M} \right)}$$

$$=440.1Hz$$

If you are being very keen, you can repeat these calculations over a range of different value for A and B to find the combination which give the lowest error. This is tedious to do by hand, but it is a simple task using a spreadsheet such as Excel. Table 6 below shows a whole series of alternatives, calculating the value for PR2-TMR2, and the error.

Looking at the errors shown, it is noticeable that towards the bottom of the table the errors start getting much larger. As the pre and post scale values increase, the time taken for an individual count to occur increases. This can be seen because the numbers for PR2-TMR2 towards the bottom of the table are that much smaller. Whilst any of these combinations *would* work, it is better to select a value nearer the top of the table. Actually, although the attempt at A=4 and B=8 gives a good accuracy, the best option would be:

A	4
B	11
PR2-TMR2	154

At this point we have two options. We could start with TMR2=0 and count up to PR2=154, or we could leave PR2 at its default value of 255, and set TMR2 accordingly:

$$PR2-TMR2=154$$

$$\therefore TMR2=PR2-154$$

$$=255-154=101$$

A	B	PR2-TMR2	INT PR2-TMR2	F (Hz)	Error (Hz)
4	7	242.5065	243	439.1101	0.88993
4	8	212.0682	212	440.1408	-0.14085
4	9	188.3939	188	440.9171	-0.91711
4	10	169.4545	169	441.1765	-1.17647
4	**11**	**153.9587**	**154**	**439.8827**	**0.117302**
4	12	141.0455	141	440.1408	-0.14085
4	13	130.1189	130	440.3993	-0.3993
4	14	120.7532	121	439.1101	0.88993
4	15	112.6364	113	438.5965	1.403509
4	16	105.5341	106	438.0841	1.915888
16	1	425.1364	425	440.1408	-0.14085
16	2	212.0682	212	440.1408	-0.14085
16	3	141.0455	141	440.1408	-0.14085
16	4	105.5341	106	438.0841	1.915888
16	5	84.22727	84	441.1765	-1.17647
16	6	70.02273	70	440.1408	-0.14085
16	7	59.87662	60	439.1101	0.88993
16	8	52.26705	52	442.217	-2.21698

16	9	46.34848	46	443.2624	-3.26241
16	10	41.61364	42	436.0465	3.953488
16	11	37.73967	38	437.0629	2.937063
16	12	34.51136	35	434.0278	5.972222
16	13	31.77972	32	437.0629	2.937063
16	14	29.43831	29	446.4286	-6.42857
16	15	27.40909	27	446.4286	-6.42857
16	16	25.63352	26	434.0278	5.972222

Table 6 Possible values for PR2-TMR2, and the error.

16.5. Timer 3

Operationally, timer three is the same as timer one. Again, it is a sixteen-bit timer, with the same requirements in terms of reading and writing data to the timer registers in the correct order.

Timer three can be used, alongside timer one, as a timing source for the capture, compare, pulse width modulation modules with options for timer three being the only source, timers one and three being the timing source for different CCP modules, or timer 1 being the only source.

16.6. Using the Timers

In order to use the timers successfully a number of things need to be done:

1. Select 8-bit or 16-bit mode (timer 0)

2. Determine the value to be loaded in to the timer data register.

3. Determine the value of any pre-scaler or post-scaler.

4. Load required values in to the timer control register. Remember for timer 0, internal/external clock needs selecting as does the resolution and whether the pre-scaler is used.

5. Activate the interrupts.

6. Write the interrupt handling routine for the timer- this should include resetting the interrupt flag. Timer 2 can run continuously, timers 0 and 1 need the data value reloading, and restarting as well.

17. PIC ADC

The PIC is a resolutely digital system, designed to handle data presented in the form of 1s and 0s, but much of the outside world in which the PIC is required to operate is analogue, with sensors and other data sources presenting information which is continuous both in time and voltage.

To make the task of inputting this data into the PIC easier, it contains an on-board analogue to digital convertor, or ADC, module.

The PIC18F452 works off a 5V supply and, by default, the analogue to digital converter module is designed to work of that voltage too. This means that applying a voltage of 0V to the analogue input will give a 0 out and 5V will give the maximum out. In the case of the PIC18F452, 10-bits are used for the conversion, so the maximum output is 1023.

It is important that the analogue input range does not exceed this voltage range in either direction. This is important to remember because many analogue signals are symmetrical about 0V- that is, they go equally positive and negative. Before feeding such a signal in to the ADC on the PIC it would need to be *level shifted* so that the signal is symmetrical about 2.5V.

If the analogue voltage range is greater than 5V peak-to-peak, then signal conditioning would need to be applied to the input to reduce it to 5V.

In some cases it may be that the input voltage range is less than 5V. There are two options in this case:

1. Input signal conditioning could be applied to boost the voltage range to 5V.
2. The PIC can be set up to use an input voltage range other than 5V. Using the Port Configuration settings, it is possible to use some of the analogue inputs as minimum and maximum reference voltages (V_{ref-} and V_{ref+} respectively)

It is important to try to use as much of the input voltage range as possible (without exceeding it) in order to get the best quality conversion. If the input signal only actually varies by 2.5V rather than 5V then only 9 of the 10 available bits would be used in the conversion.

There are three main stages to using the ADCs on the PIC:

1. Set up the ports
2. Set up the ADC module.
3. Process the data that the ADC produces

17.1. Setting up the Port for ADC Operation

As stated previously, the analogue input is the alternate function for Port A and Port E.

To configure the port pins as analogue, they must be set as inputs using the TRISA(TRISE) SFR, and they must then be configured for analogue using the PCFG bits in the ADCON1 SFR.

TRISA=0bxxx1x111;

ADCON1bits.PCFG=**pppp**;

In the listing above, the non-analogue bits are shown as 'x' because they could still be set as digital outputs if necessary.

The port configuration bits should be set according to table 1.

pppp	AN7/ RE2	AN6/ RE1	AN5/ RE0	AN4/ RA5	AN3/ RA3	AN2/ RA2	AN1/ RA1	AN0/ RA0
0000	A	A	A	A	A	A	A	A
0001	A	A	A	A	V_{ref+}	A	A	A
0010	D	D	D	A	A	A	A	A
0011	D	D	D	D	V_{ref+}	A	A	A
0100	D	D	D	D	A	D	A	A
0101	D	D	D	D	V_{ref+}	D	A	A
011x	D	D	D	D	D	D	D	D
1000	A	A	A	A	V_{ref+}	V_{ref-}	A	A
1001	D	D	A	A	A	A	A	A
1010	D	D	A	A	V_{ref+}	A	A	A
1011	D	D	A	A	V_{ref+}	V_{ref-}	A	A
1100	D	D	D	A	V_{ref+}	V_{ref-}	A	A
1101	D	D	D	D	V_{ref+}	V_{ref-}	A	A
1110	D	D	D	D	D	D	D	A
1111	D	D	D	D	V_{ref+}	V_{ref-}	D	A

Table 1: Port Configuration settings (Also in the Ports Section).

You will notice that the various different configuration options allow for anything from zero to five analogue inputs. You will also see options with either or both of V_{ref+} and V_{ref-} available externally. It should be noted that whether the reference voltages are internal or external, all analogue inputs are required to use the same references.

17.2. Configuring the ADC

It is now necessary to configure the ADC itself.

Across the two ADC control registers *ADCON0* and *ADCON1*, there are a number of aspects of the performance and operation of the ADC which can be configured. This is in addition to the port configuration we have already looked at.

17.3. ADC Clock Speed

It is possible to vary the speed at which the ADC operates with respect to the system clock. This may seem like an unnecessary complication- surely faster is better? However, if the input is a slowly changing analogue signal, running the converter very fast would be unnecessary and if the ADC is set to restart as soon as the previous conversion is finished, then an unnecessary amount of data will be produced placing increased processing load on the rest of the system.

There are three bits to select the ADC clock speed, two of these, *ADCS1* and *ADCS0*, are located in bits 6 and 7 of ADCON0, and the third, *ADCS2*, is located in bit 6 of ADCON1.

There is a link between the bits which are set and the ADC clock speed, but it's not immediately obvious. Table 2 below gives the bit values and the clock speeds.

ADCON1 ADCS2	ADCON0 ADCS1 ADCS0	Clock Speed
0	0 0	$F_{osc}/2$
0	0 1	$F_{osc}/8$
0	1 0	$F_{osc}/32$
0	1 1	Clock derived from ADC's own RC oscillator
1	0 0	$F_{osc}/4$
1	0 1	$F_{osc}/16$
1	1 0	$F_{osc}/64$
1	1 1	Clock derived from ADC's own RC oscillator

Table 2: ADC oscillator selection.

The basic relationship is that the base is half the oscillator frequency.

- Selecting bit 0 causes a further division by 4
- Selecting bit 1 causes a further division by 16
- Selecting bit 2 causes a further division by 2.

If more than one bit is selected, all the factors are multiplied together (along with the base factor if 2) to give the overall division.

However, if *both* bits 0 and 1 are selected, the clock source changes completely.

Because these bits are spread cross two SFRs, it is not possible to set them as a single 3-bit value. As they are likely to be set on initialisation and then left, however, this should not be too much of an issue.

17.4. ADC Channel Selection

Unlike the clock selection bits, this second set of control bits might change more at run time. If more than one ADC input is being used, it is necessary to specify which of the ADC inputs should be used for the current conversion. This would probably need to be reset after each conversion, because if three ADC inputs were used, it would probably be required that each is read at the same rate, so channel 1 would be read, then channel 2, then channel 3 and back to channel 1.

There are three bits used for this purpose *CHS2, CHS1* and *CHS0*. They are located in bits 5, 4 and 3 of ADCON0. The binary value placed on these bits selects the current input. Note that not all combinations are valid on the PIC18F252, and others may not be valid if only some inputs are configured as analogue. Care should be taken not to select an invalid number here.

You can access these bits individually as:

ADCON0bits.CHS0

ADCON0bits.CHS1

ADCON0bits.CHS2

They can also be accessed as a 3-bit value:

ADCON0bits.CHS

17.5. Activating the ADC

Before the ADC can do any converting, it is necessary to switch it on. By default, this module is switched off at power-up to save power. You would only activate it if the ADC is required.

The on bit is located at bit 0 of ADCON0 and is called *ADON*. Activating the ADC is achieved with the command.

```
ADCON0bits.ADON=1;
```

17.6. Starting and Completing a Conversion

Having set the rest of the settings for the ADC, a conversion can now be started. This is achieved be setting the GO/DONE bit (bit 2 of ADCON0).

The program writes a 1 to this bit to start conversion, and the ADC resets this bit to zero when conversion is complete. This bit has several alternate names listed in the header file. Any of these will work in accessing this bit:

ADCON0bits.GO_NOT_DONE

ADCON0bits.GO_nDONE

ADCON0bits.GO

ADCON0bits.GO_DONE

ADCON0bits.DONE

ADCON0bits.nDONE

Generally speaking, selecting one and sticking with it is the best policy.

Running and detecting the completion of a conversion can be achieved using the following code snippet:

```
ADCON0bits.GO_DONE=1;

While(ADCON0bits.GO_DONE){

}
```

Basically, this code sets the ADC running, then the while loop waits for the go/done bit to reset before continuing. After the while loop, code would be required to handle the output of the ADC.

Alternatively, and possibly a more elegant solution from the coding perspective, is to handle the conversion finishing via the interrupt system. To do this, you would need to set up the interrupt system, as we talked about in the section on interrupts, then in the interrupt service routine, detect that the ADC interrupt is flagged, you would then need to download the ADC data value and restart the ADC if required.

17.7. Output Formatting

As stated previously, the output of the ADC is 10-bit resolution, which requires two bytes of data to store. These are two SFRs, *ADRESH* and *ADRESL*. These stand for **A**-**D** **RES**ult **H**igh and **A**-**D** **RES**ult **L**ow. Because the PIC is an 8-bit system, a 10-bit result will not full occupy both of these registers, so 6-bits of 'padding' are added. This padding is in the form of zeros added to the unused bits.

It is possible to select which end the result registers are filled from using the *ADFM* bit, which standard for **A**-**D** **F**or**M**atting.

ADFM=1

The result is filled from the least significant end.

Bits 7 to 0 occupy bits 7 to 0 of ADRESL, bits 8 and 9 occupy bits 0 and 1 of ADRESH and bits 7-3 of ADRESH are filled with zeros.

ADFM=0

The result is filled from the most significant end.

Bits 9 to 2 occupy bits 7 to 0 of ADRESH and bits 1 and 0 occupy bits 7 and 6 of ADESL. Bits 5-0 of ADRESL are filled with zeros.

Why?

The PIC18F452 is an 8-bit system and in a lot of situations an 8-bit resolution form the ADC may be perfectly acceptable. It will mean that the number can be processed in one machine cycle making processing faster.

The 8 most significant bits would want to be used for this, but using ADFM=1 formatting (Right Justified) would require some maths to marry up the two bits in ADRESH with the 6 in ADRESL and this would all take time.

Using ADFM=0 (Left Justified) allows the 8-bit version of the result to be directly read from ADRESH and ADRESL is ignored.

18. MPLAB X

Most computer (and microcontroller) programs are written using a program known as an *Integrated Development Environment* or IDE. Such environments contain a number of different but connected facilities to enable them to handle the whole program-writing process. These facilities include:

- Code and project management: This allows code files to be added to projects, and for code to be written. It also includes file management facilities to ensure files are included in a common folder for example.
- Code-writing window: This is normally an enhanced text editor. In theory you could use an editor such as *notepad* to write the code files, but the editor included within the IDE will normally include some form of error or syntax checking, for example, it will highlight keywords in a different colour, and will check that there is an end of function curly bracket to match each start of function bracket.
- Simulator: The simulator will 'run' the program before it is loaded on to the microcontroller and provide information about the amount of memory used, or the status of SFRs at any given point in the program.
- Debug: Alongside the simulator, debugging facilities are provided to assist in solving problems with the code.
- Compiler: This will do a full syntax check on the code, and convert it to something which can be run on the target device.
- Linker: The linker will organise all the files for a particular project and ensure they are ready for downloading
- Downloader: This will manage the process of downloading the program and any other information such as microcontroller settings on the target.

The IDE we use is *MPLAB X* which is the latest version of the MPLAB IDE written by microchip, the makers of the PIC microcontrollers and, as such, is closely aligned with the PIC chips.

18.1. Starting MPLAB X

These notes are based on MPLAB X Version 6.20 which is the current version as at February 2025. It is noted that in April 2025, version 6.25 has been released. I haven't upgraded to this version because the PICKit 3 which I use to download the programs onto the PIC itself to demo them doesn't work on this later version. There may be slight variations if using a different version.

To launch MPLAB, either double click on the MPLAB X icon on the desktop, or go to the *Start* menu, find *Microchip* and double-click on *MPLAB X IDE*.

You will see the splash-screen appear and then, once the program has launched you will get an empty project space as shown in figure 2.

Figure 1: The MPLAB X Icon[i]

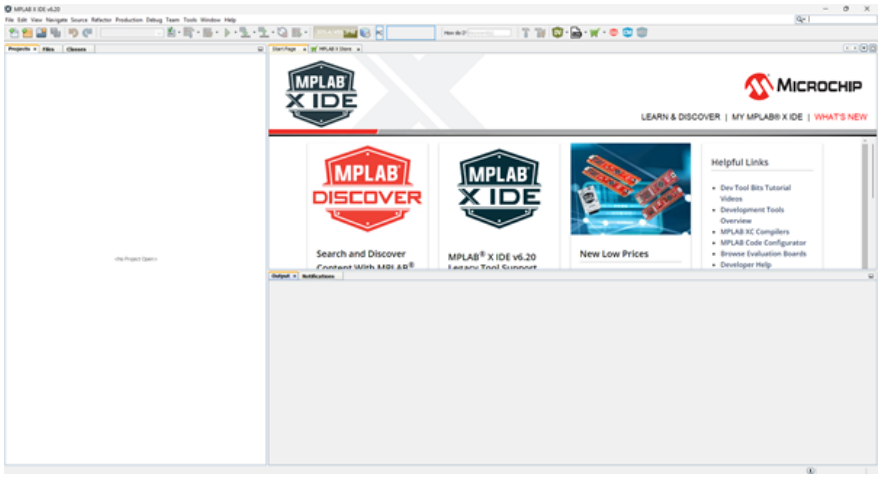

Figure 2: MPLAB launched.

Much of the layout will look familiar to users of any modern *Windows* program, with file, edit, view, window and help menus. The reference to 'modern' programs is made specifically because MPLAB X has adopted current trend for having a number of different areas or panes containing different information. Note that MPLAB will remember how it was previously set up, not how your project was set up, so different windows may be visible or invisible, and they may be in different positions. If a window you need is not visible go to the *View* menu and then select the window you wish to hide or make visible.

One word of caution as we start however. MPLAB organises 'programs' into *Projects*. Before you can do anything, you need either to open a current project, or to create a new project.

18.2. Starting a New Project

Either click on or go to:

File/New Project

Finally, you can click CNTL+SHIFT+N

The *New Project Wizard,* as shown in figure 3, will open to guide you through the setting up of the new project. It should be noted that a wide variety of different, and possibly quite complex, projects can be written using MPLAB X.

PICs are Microchip products, and in this course we will build a set of 'Application Projects'. If you have used an older version of MPLAB X, these used to be called 'Standalone Projects'. This is normally the default, so click 'Next'...

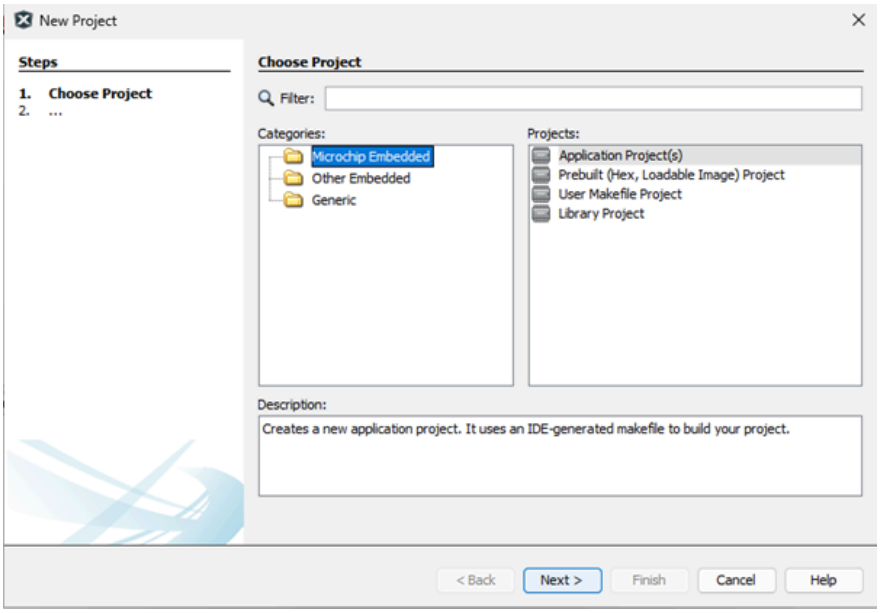

Figure 3: Project Wizard Dialogue Page 1

This takes you to page 2 of the dialogue, as shown in figure 4.

Again, you have two options here:

1) Directly select the device you require

2) Select the 'Family' first, and this will help limit the number of options in the device list.

The dropdown list of 'Family' options is shown in figure 5.

In this course we will always use the same microcontroller which is a member of the PIC18 family, so select 'Advanced 8-bit MCUs (PIC18)'.

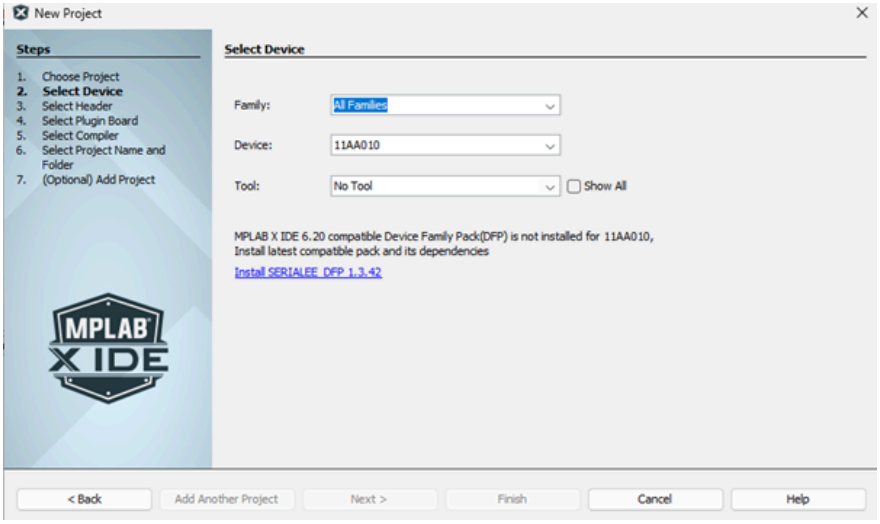

Figure 4: Project Wizard Page 2

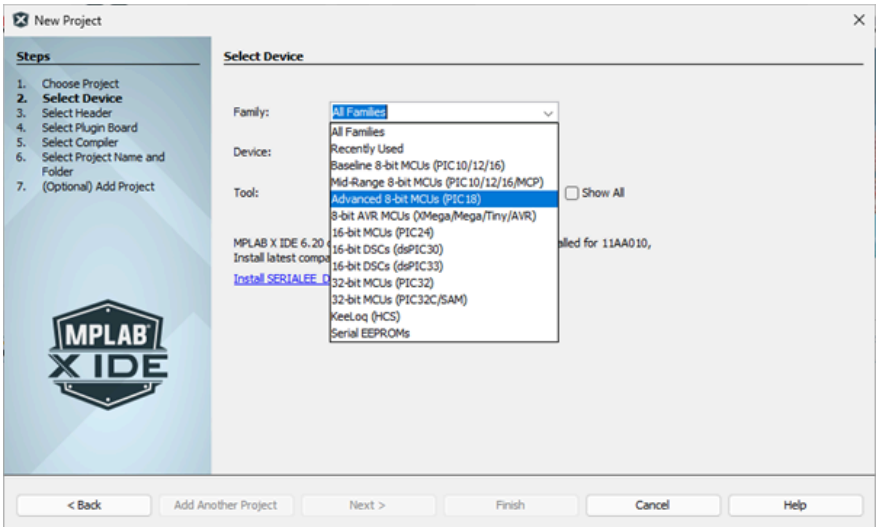

Figure 5: Selecting the Required Family

Even selecting just the 'PIC18s' gives a very long list of alternative devices, as you can see if you now scroll down the 'Devices' list, as shown in figure 6.

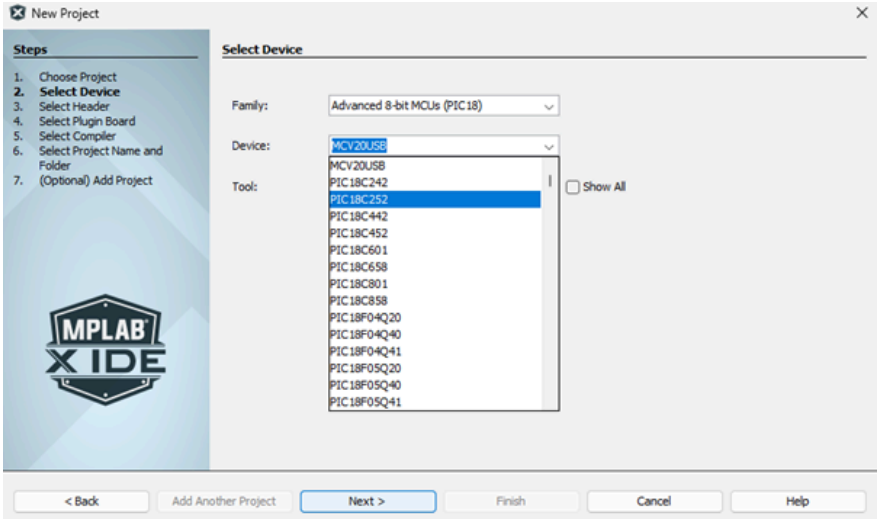

Figure 6: Selecting the PIC18 Device.

We want the **PIC18F452**. Note that there is also a PIC18C452. Make sure you select the right one, otherwise you will get errors later on.

1) You can scroll down the list until you find this option.

2) You can start typing and the list will shrink based on what you type.

Note that if you select PIC18C452 by mistake, it seems to confuse MPLAB X and, although you can change the device later on, it seems to cause the system not to recognise commands and things correctly. If you get a lot of errors, even on basic commands and Special Function Register names, check that you haven't got this setting wrong. If you have, the best solution is to close the project down, and open a new project with the correct device.

Having selected the microcontroller, you now need to select the tool. For the moment, we will select 'Simulator' which is the only option available as shown in figure 7.

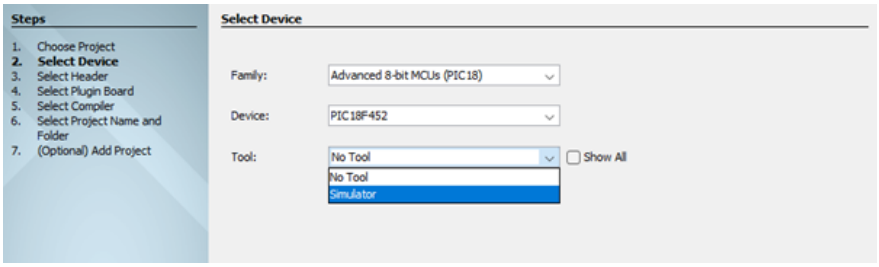

Figure 7: Selecting the Tool.

We can now click 'Next'.

On page 3 of the dialogue box, you will be asked to specify the language and tool suite you wish to use. MPLAB X can program PICs in C and Assembly Language, and has more than one C compiler available. Figure 8 shows page three of the dialogue box with the *Select Compiler* window.

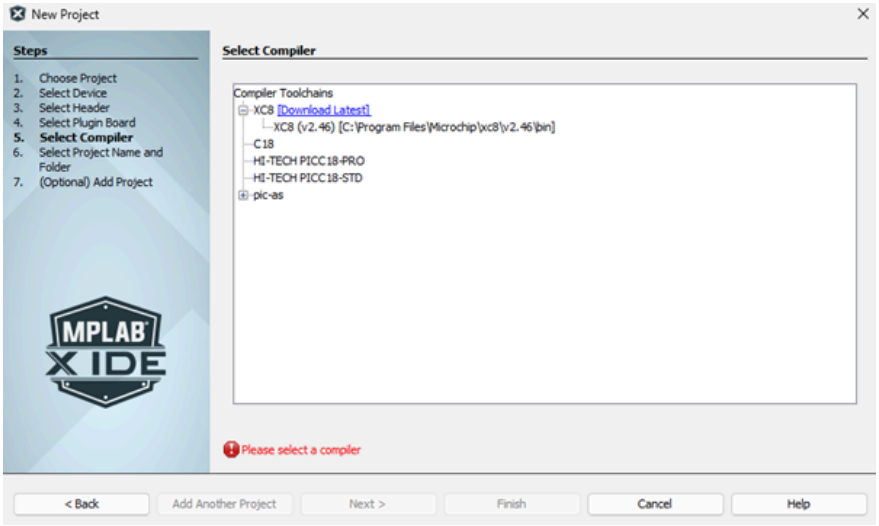

Figure 8: Selecting the Correct Tool Suite

Select *XC8* and click on next. You will notice that 'Next' is greyed out until a compiler is selected.

Page 4 of the dialogue box allows you to give your project a name and specify the folder in which it is saved as shown in figure 9. I would recommend that you set up a folder within which to store all your projects. You might want to call your project something like 'Exercise1'. This will then create a folder for that project. You will notice that MPLAB X will sometimes complain about 'non-standard' characters such as spaces in file or path names. Leave the 'Encoding' set to 'ISO-8859-1'. You should now have finished setting up. If everything is correct click 'Finish', otherwise click 'Back' and correct what is wrong.

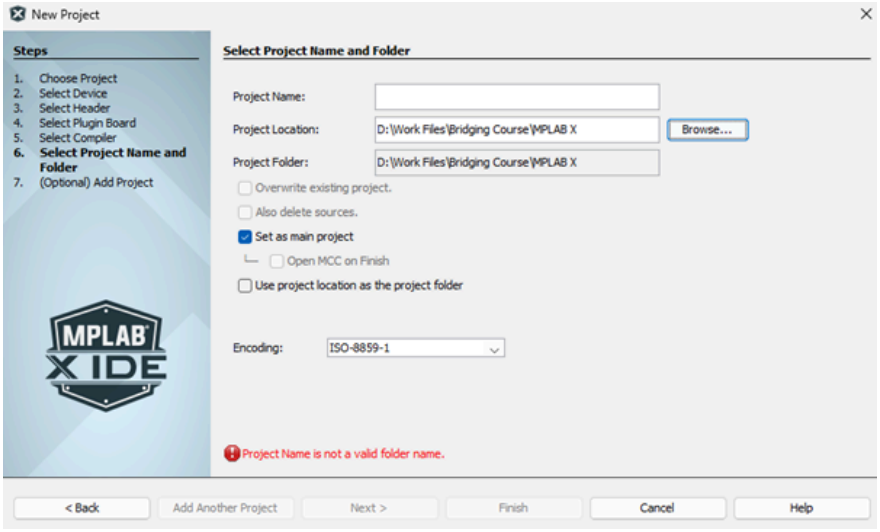

Figure 9: The Completed Project Setup

When you click 'Finish' the project will open and you see the main screen as shown in figure 10.

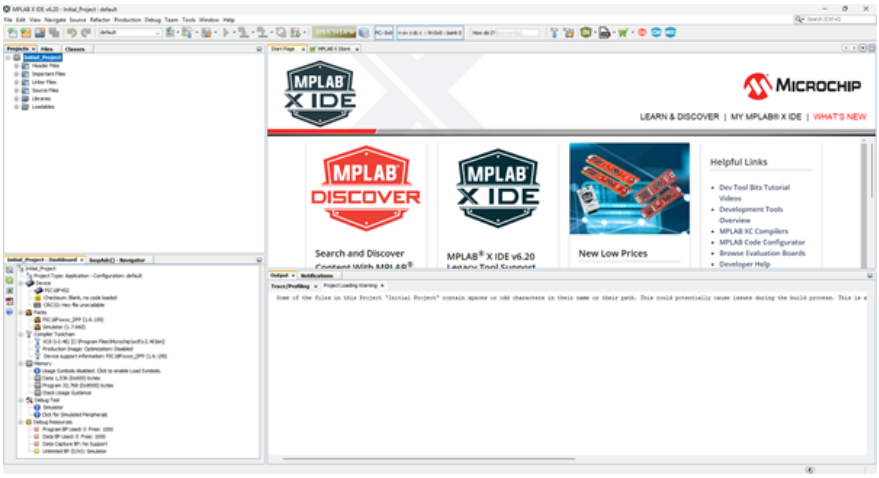

Figure 10: The New Project Set-up Complete.

In figure 10, you can clearly see the various different areas containing different information about the project. We can briefly look round these before continuing.

Project Navigator

The top-left window is the project navigator, which contains a list of all the files included in the project. Many of these will be added automatically. In a moment, we will need to add a *source file* which is where our code will be written.

Project Dashboard

The lower-left window is the project dashboard. This contains a large amount of information relating to the project, including details of the simulator and debugger selected, and the compiler. To the left of this area are a set of five icons. The top one of these is 'Project Properties' and allows you to alter the information specified in the set-up just now. The fourth option is the device datasheet, and the final option a link to the compiler help.

Status Window

Along the bottom of the screen is a status window. Exactly what is displayed here will change as programming and simulation proceed, but can include information about the success of the building of a project and simulation information.

There is a lot more which these windows can tell you, but I don't want to overload you with information at this stage. Microchip provide extensive help for MPLAB X and a full user manual for the XC8 compiler, which I would recommend reading.

When you have successfully managed to open a project, click on the button to start writing the code.

Starting MPLAB X Video

18.3. Adding a Source File

Before we can add code to a project, we need to add a new source file to our project. To do this Right Click on 'Source Files' and a dropdown menu will appear. Select 'New' and 'main.c' as shown in figure 11.

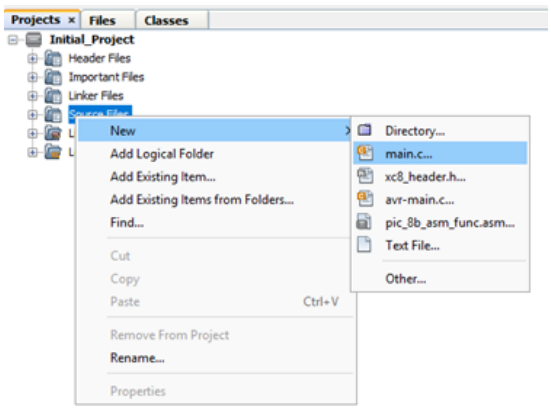

Figure 11: Adding a New Source File.

A dialogue box will open asking for the filename, choose a name to suit.

When you click 'finish' a source code window will open up in the top right area of the screen as shown in figure 12.

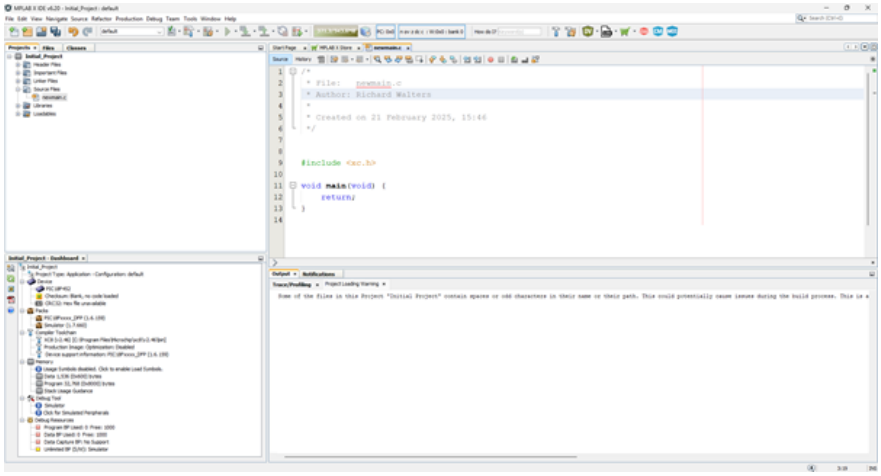

Figure 12: The Editor window and the New Source Code

You will notice that some bits have been included automatically for you. The Author Name, is taken from the username set up in Windows, so might not be your name on a shared system.

The '#include' line adds a *header file* which includes all the compiler information for the XC8 compiler we are using. If needed, other standard C header files can be added too.

The only other bit included is an 'empty' main function. We will actually overwrite this.

18.4. Building a Project

Before a project can be simulated or downloaded to the PIC, it is necessary to compile the project (or in MPLAB 'build' the project).

- Click on *Production* and then *Build Main Project* as shown in figure 13 below, or.

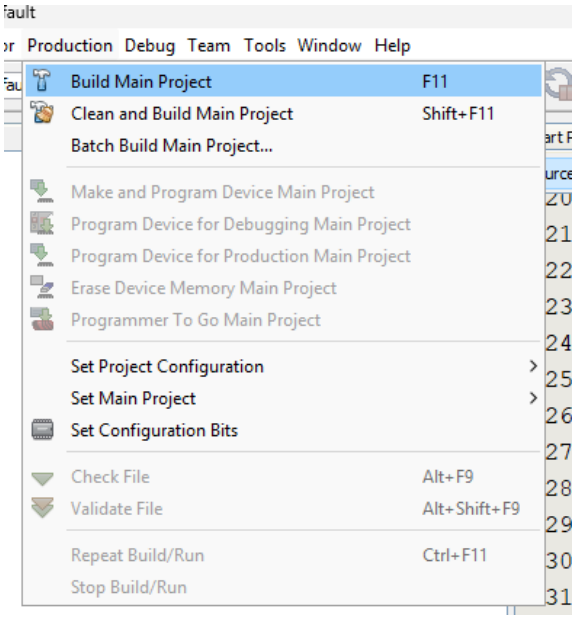

Figure 13: Selecting *Build Main Project*.

- Click on the 'hammer' icon,  , or.
- Press F11

Hopefully, if you have copied the file correctly, the build should be successful and you will see a message in the bottom-right window as shown in figure 14.

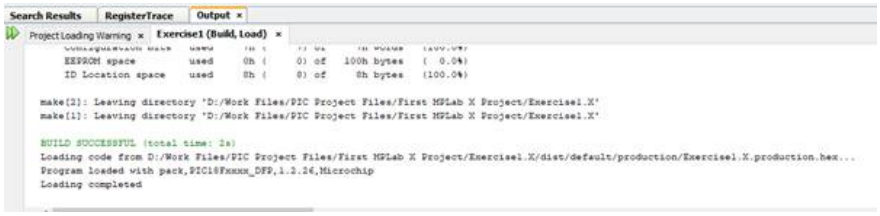

Figure 14: Project Successfully Built.

You should remember that this only means that the compiler did not spot any syntax or other errors in the file, not that the program necessarily does what you want.

If there are errors in your file, the output window may look something like figure 15.

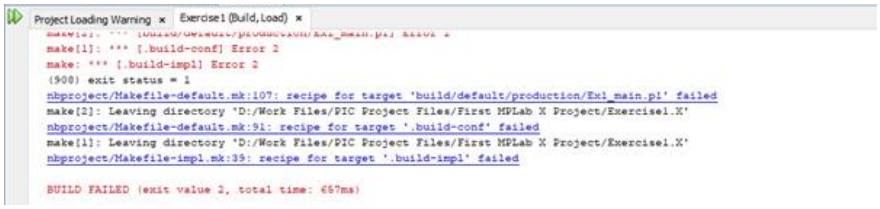

Figure 15: Failed Build.

If you scroll up this file, you will find some lines of blue text which tell you what error has been found. If you double-click on the blue text line informing you about the error then it will take you to the line in the code which generated the error, as shown in figure 16. You will see query marks to the left from the 'void delay (void){' line.

Figure 16: Error in File.

An important note:

An error may be detected in a certain line shown but this does not necessarily mean that the error actually is in this line.

Here, and I know because I did so deliberately, the '}' in the 'j' for loop is missing. The compiler has therefore associated what should be the end of function '}' with the for loop and so is complaining that the end of function '}' is missing.

18.5. Starting the Simulator

When it comes to simulating and testing your code there are a number of things you need to consider.

Much of the operation of the code is in the form of port and other SFR (Special Function Register) values changing as the code runs. This is all hidden inside the PIC itself and so can be difficult to see. To verify that the software is doing what is expected of it, it is advisable always to simulate it before downloading to the PIC.

Just downloading code to a microcontroller and measuring port voltages will not tell you if the code is working as expected- and if you see nothing on the output, you won't know what is not working.

To that end, you need to consider the following:

1. Break the code down into small testable chunks and simulate each of these individually.
2. Simulation is different from 'running the program in the IDE'. You need to adapt your way of thinking about how the program will run in order to get the most out of the simulator.
3. All SFRs and other internal registers can be 'watched' to see how they change.
4. Functions can be run, jumped into or jumped over to ease the monitoring of the running of the program.

In order for the simulator to be able to determine time delays etc correctly, the simulator needs to know the clock speed the PIC is running at. Click on the *project properties icon*, the spanner to the left of the dashboard,  . This will open the project properties window. The *Oscillator Frequency* is 24 MHz and the PIC takes four clock cycles, so the instruction frequency should be set to 6MHz, as shown in figure 17.

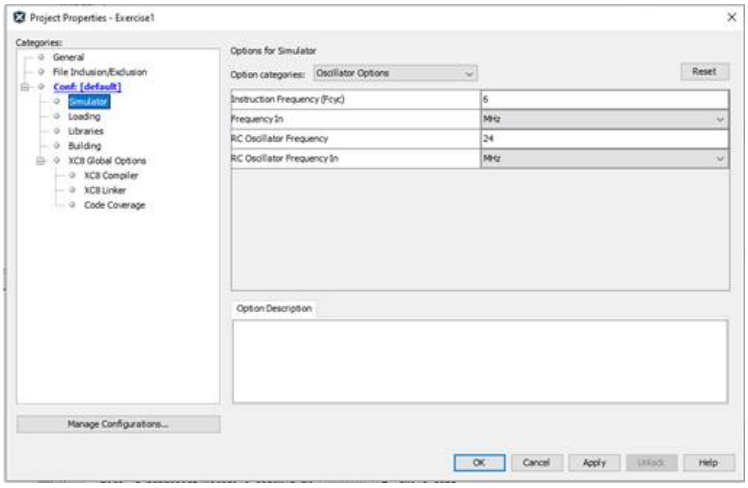

Figure 17: Setting the Clock Speed

To start a simulation click on the 'simulate' button, .

18.6. Simulation Controls

When the simulation is running, it is helpful to be able to control the operation of the code to allow you to follow what happens at those points in the code which are of particular interest.

Table 23: Simulator Controls

Icon Description

End the simulation- this allows you to return to editing the code
SHIFT+F5

Pause- this will halt the simulation, but not close the simulator
CNTL+ALT+8

Restart the simulator- this allows you to run the code again from the start

Run- this will start the code running again after a pause at a breakpoint for example. When this is clicked the code will run continually until the next break point is arrived at
F5

Step Over. Clicking on this would cause the simulator to ignore the next command. So, for example, if the line calls a delay function, stepping over will ignore the delay and go straight to the next line.
F8

Step Into. This will run just the next line of code and then pause again.
F7

Step Out. This will exit the current function and go to the next line in the calling function. If it is not possible to resolve this location at the op-code (execution) level, an error may result.
CNTL+F7

Run to Cursor. If you place the cursor at the line of interest. The code will run up until the line at which the cursor is placed and then pause. This can be helpful if you want to check a particular part of the code.

Set PC (the Program Counter) to the cursor. The program counter is the internal log stored inside the microcontroller of where in the code, the program has got to. Clicking on this icon will force the internal program counter to be placed at the value which corresponds to the current instruction at which the cursor is placed.

Focus the cursor at the program counter. This is the opposite of the previous option. The cursor will be moved to show which instruction is currently being executed.

18.7. Adding Breakpoints

Because values are only updated when the simulator pauses, breakpoints are central to the operation of the simulator. These are points in the execution of the program at which it will automatically pause and the status of the system can be checked at that point.

To insert a breakpoint, you can click on the 'debug' menu and then 'New breakpoint' or you can right-click on the line in the source code and select 'Toggle line breakpoint'. Finally, and most simply, click on the line number to the left of the line you want the breakpoint on and a breakpoint will be inserted on that line. The line will turn red and a red square will in place of the number. In figure 18, you can see that I have added two breakpoints, one either side of the call to the delay function. If we start the program running again, execution will stop before line 93, and then again before line 95- and repeatedly so each time these lines are reached. When the breakpoint is reached and the program stops, the current line will be highlighted in green as shown in figure 19. You should note that the program stops *before* the line is run, so any changes which occur in that line will not have happened.

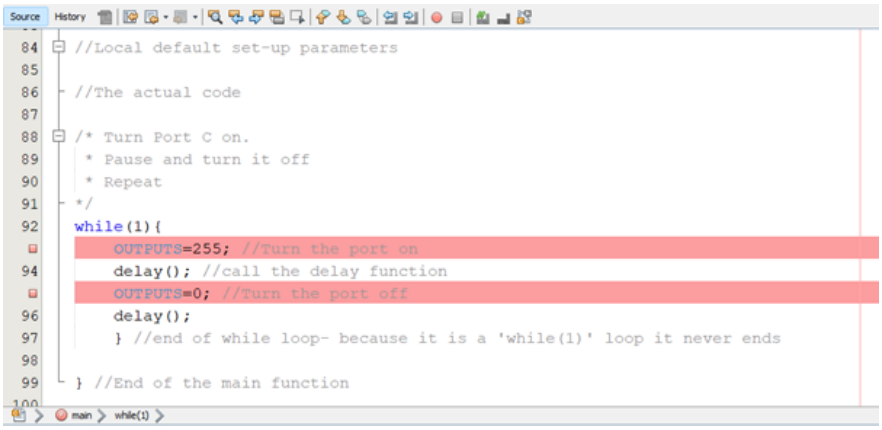

Figure 18: Breakpoints Added to the Code Window

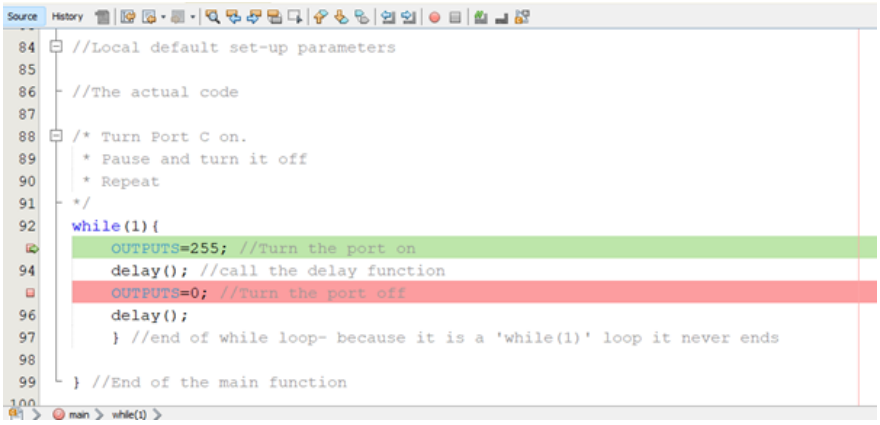

Figure 19: Execution Paused at Breakpoint.

If you need to check how one particular part of the code is operating, you could place a breakpoint, at the start of that section, run the code up to the breakpoint, and then 'single step' or step into each line for the bit you are interested in. This will allow you to check that the code is working as expected. What you need to remember here is that the simulator will tell you what is happening in each step, but you will need to determine whether what is happening is actually what you need to happen.

18.8. Watching an SFR or Variable

To watch either a variable within the program, or a special function register such as a port, use the watch window, as shown in figure 20.

Figure 20: Opening the Watch Window

The blank watch window will open as shown in figure 21.

Figure 21: The Blank Watch Window

To add a watch, double-click on the page icon, , but if you miss slightly, it will open a text entry option where it says <enter new watch>.

Alternatively, right-click on the line to open the add watch window, as shown in figure 22.

Figure 22: New Watch Window

You have two options in this window, either to add a global symbol- such as a global variable, or an SFR. It defaults to global symbols, check the SFR's option to add an SFR such as the Ports, as shown in figure 23.

Figure 23: A Watch Added

Having added an item to the watch list, we can update the columns shown by right-clicking on the title bar. The options list will appear, as shown in figure 24.

Figure 24: The Watch Window Column List. Ticked Items are currently shown.

Changes on the items listed are only updated when the simulator pauses. Any changes since the last update are shown in red, as shown in figure 25.

Figure 25: The Watch Window Updating

Although it is available as a separate option from the debugging menu, watching a variable takes you to the same window. Figure 26 shows this window with two variables added, i, and j. These are the two delay variables in the initial program. In figure 26, the code is paused in the delay function.

Figure 26: Variables in the Watch Window

The variables, i, and j are only available within the delay function. When the code pauses outside of this function, as shown in figure 27, the variables which don't exist at that point are not shown.

Figure 27: The Watch Window Outside the Delay Function

18.9. The Input/Output Viewer

A second way of looking at the operation of various parts of the system is by use of the I/O Viewer, which can be selected from the debug menu, as shown in figure 28.

Figure 28: Selecting the I/O Viewer

The I/O Viewer is divided into two areas. The upper part of the window shows the various parts of the system, which can be selected, and the lower part shows all SFRs associated with that part of the system. In figure 29, below, the window, and particularly the upper part has been expanded so that all of the options in the upper part are visible.

Figure 29: The I/O Viewer with all system resources shown.

Selecting an item, such as Timer 1, in figure 29, displays all the SFRs associated with that item in the lower area, as shown in figure 30.

Figure 30: SFRs Associated with Timer 1.

As you single step through the code, changes since the last pause are shown in red, as shown in figure 31.

Figure 31: Updating the I/O Viewer

In figure 31, bits 7 and 6 of ADCON0 updated in the last line executed. These changed to ones, which is shown because the 'bit blobs' are filled in. A zero is indicated by the blob being empty. Note, too, that there is a sign to the left of each SFR name. If you click on this, it will expand the SFR to show all the bits contained, as shown in figure 32.

Figure 32: Expanded ADCON0 SFR

You can see that the two bits set are the ADCS (the A to D converter clock select bits). The names of the other bits or groups of bits are also shown.

18.10. I/O Pin

An alternative to using the Watch window or the I/O Viewer window is to use the IO Pin window. Selecting this is shown in figure 44. Note that this is selected through the 'Simulator' option rather than the 'debugging' option.

Figure 44: Selecting the IO Pin

The main benefit of the IO Pin window is that it is a way that you can feed data into the simulator is a port is acting as an input for example. Again, this is only possible when the simulator is paused.

The IO Pin window is shown in figure 45.

Figure 45: The IO Pin Window

You can only add pins to the list when the simulator is stopped. To do this, click on the window, on the line under 'Pin' and a text box and dropdown arrow will appear as shown in figure 46.

Figure 46: Adding a Pin

Where pins have alternate functions, and have different names in these situations, for example, Port A bit zero is listed as RA0, the port name, and AN0, when used as an analogue input, both of these alternatives are given in the list.

Once the pins are added, the simulation can be started. As ever, remember that the outputs only update, and inputs can only be input when the simulator is paused.

Figure 36, below, shows the IO Pin window when the simulator is paused.

Figure 36: The IO Pin Window when the Simulator is Paused

Here, you can see the current value on these pins, but also whether they are analogue or digital in the case of port A, and whether they are inputs or outputs. In the case of an input, you can click on the current value, and a drop down box appears to allow you to change that input value, as shown in figure 37.

Figure 37: Changing an Input Value

18.11. The Stimulus Window

A second method of triggering inputs to the system is by using the stimulus window. Again, this is selected from within the simulator option on the Window menu, as shown in figure 38.

Figure 38: Selecting the Stimulus Window

This, in some ways is more flexible than the IO Pin window because it allows a wider range of inputs to be sent into the PIC. This system is quite comprehensive and so we won't cover everything it can do here, but will go through some of the basics. the empty window is shown in figure 39.

Figure 39: The Stimulus Window

As with the IO Pin window, you add items by clicking under 'pin', however, in the stimulus window, you can add the same pin multiple times, but with different actions applied.

Figure 40: Options for a Digital Input

Figure 40 shows the options for a digital input. You will notice that these include setting the port pin high and setting it low, toggling the pin, and generating a high and a low pulse.

In the input example program, where RA0 follows RB0, we could use two lines, to set RB0 high and low, or we could use one line to toggle it.

To add another item, click the icon to the left of the window, and then add the new item.

Figure 41 shows all the options above on separate lines.

Figure 41: The Populated Stimulus Window

To trigger one of these stimuli, click the fire button on that row. The action performed is listed at the bottom of the window, as shown in figure 42- note that I have shrunk the window to save space.

Figure 42: The Toggle Stimulus Being Fired

The stimulus window can also be used to inject analogue inputs into the ADC block on the PIC. The stimulus window with the analogue input AN0 set up to inject voltages from 0V to 5V in 1V steps is shown in figure 43. By default, when you select 'Set Voltage', 0V is listed. Click on the zero to change the voltage.

Figure 43: Injecting Analogue Voltages

Notice, here, that AN0 is the label of the input, not a reference to the analogue voltage being zero volts. Clicking the appropriate fire arrow will inject the voltage listed into the input.

18.12. Logic Analyser Video

A second method of triggering inputs to the system is by using the stimulus window. Again, this is selected from within the simulator option on the Window menu, as shown in figure 38.

Figure 38: Selecting the Stimulus Window

This, in some ways is more flexible than the IO Pin window because it allows a wider range of inputs to be sent into the PIC. This system is quite comprehensive and so we won't cover everything it can do here, but will go through some of the basics. the empty window is shown in figure 39.

Figure 39: The Stimulus Window

As with the IO Pin window, you add items by clicking under 'pin', however, in the stimulus window, you can add the same pin multiple times, but with different actions applied.

Figure 40: Options for a Digital Input

Figure 40 shows the options for a digital input. You will notice that these include setting the port pin high and setting it low, toggling the pin, and generating a high and a low pulse.

In the input example program, where RA0 follows RB0, we could use two lines, to set RB0 high and low, or we could use one line to toggle it.

To add another item, click the icon to the left of the window, and then add the new item.

Figure 41 shows all the options above on separate lines.

Figure 41: The Populated Stimulus Window

To trigger one of these stimuli, click the fire button on that row. The action performed is listed at the bottom of the window, as shown in figure 42- note that I have shrunk the window to save space.

Figure 42: The Toggle Stimulus Being Fired

The stimulus window can also be used to inject analogue inputs into the ADC block on the PIC. The stimulus window with the analogue input AN0 set up to inject voltages from 0V to 5V in 1V steps is shown in figure 43. By default, when you select 'Set Voltage', 0V is listed. Click on the zero to change the voltage.

Figure 43: Injecting Analogue Voltages

Notice, here, that AN0 is the label of the input, not a reference to the analogue voltage being zero volts. Clicking the appropriate fire arrow will inject the voltage listed into the input.

18.13. The Stopwatch

The stopwatch can help to determine the time elapsed between break points. Again, it is found from within the debugging option in the window menu, as shown in figure 48.

Figure 47: Selecting the Stopwatch

The empty stopwatch window is shown in figure 48, below. Note that if you open the stopwatch after you have run the simulator, it may not be empty when you open it.

Figure 48: The Stopwatch Window.

Every time the simulator pauses, the elapsed time, both in seconds and instruction cycles is returned, as shown in figure 49.

Figure 49: Stopwatch Return on Pause.

Note that the number of instruction cycles will always be constant because it depends on the number of instructions executed. The time elapsed, which is given in brackets, depends on the clock frequency set in the simulator, so you need to ensure that you have set this correctly in the simulator properties window as given in figure 17 in section 18.5.

Table 24: Stopwatch Controls

Icon Function

This allows you to adjust settings on the stopwatch, but is not available for the PIC18F452 so gives a window with all options greyed out

Reset Stopwatch on Run. This option will toggle between giving a total value, or a value since the last stop. In figure 61, you can see that this option is on for the first few runs, but then switched off after the stopwatch is cleared, and so the values accumulate.

Figure 50: Reset on Run On and Off

Clears the stopwatch. This will return the stopwatch window to the state shown in figure 48

Clear Stopwatch. This will reset the stopwatch to zero, as shown in figure 50 above.

18.14. Organising Multiple Windows

f you want to run input a stimulus and watch the output, you might want both visible on the screen. You can do this by dragging the title of the window and dropping it where you want the new window to be displayed. Figure 51 shows the IO Pin Window having been undocked.

Figure 51: Separating the Simulator Windows

18.15. Setting the Configuration Bits

There are a number of settings on the PIC which need to be made before the code is downloaded. These are configuration settings. Depending on the particular PIC you are using, the number of these will vary. Fortunately, to help you with this, MPLAB X contains a system for setting the configuration bits. Go to 'Production' and the select 'Set Configuration Bits' as shown in figure 52.

Figure 52: Selecting Configuration Bits

This opens the window shown in figure 53.

Figure 53: The Configuration Bit Window

Many of the bits can stay at their default value but you want to make sure that the oscillator is set to high speed, and that the watchdog timer is off- if you leave this on, then the PIC will reset periodically because, unless you service the timer, it will think that the code has stalled. We don't want to worry about that here, so we'll just turn it off.

As settings are changed, they highlight in blue, as in figure 53.

When done, click and a text window opens as shown in figure 54.

Figure 54: The configuration bits ready for copying into the code. Use CNTL+A and then CNTL+C to copy the whole listing.

You can then paste this code into the code listing in the configuration bits area.

Note that #include <xc.h> is the last line. Ideally this wants to go in the included files area, and only include this once. Having the line repeated causes problems when building.