Contents

About this manual

Please note an update version of this manual is under preparation. See www.picaxe.co.uk for latest version

The PICAXE manual is divided into three separate sections:

Section 1 -	Getting Started	(picaxe_manual1.pdf)
Section 2 -	BASIC Commands	(picaxe_manual2.pdf)
Section 3 -	Microcontroller interfacing circuits	(picaxe_manual3.pdf)

This third section provides general microcontroller interfacing circuits, and example programs, for most common input/output transducers used within microcontroller circuits.

For general information on getting started with the PICAXE system please see section 1 of the manual. No prior understanding of microcontrollers is required. A series of tutorials introduces the main features of the system.

For more specific information, syntax and examples of each BASIC Command please see section 2 'BASIC Commands'.

The software used for programming the PICAXE is called the 'Programming Editor'. This software is free to download from www.picaxe.co.uk. This manual was prepared using Version 5.2.0 of the Programming Editor software. Please ensure you are using this version (or later) of the software, as earlier versions may not support all the commands and features described.

The latest version of this document is available on the PICAXE website at **www.picaxe.co.uk**

If you have a question about any command please post a question on the very active support forum at this website.

Microcontroller Interfacing Circuits
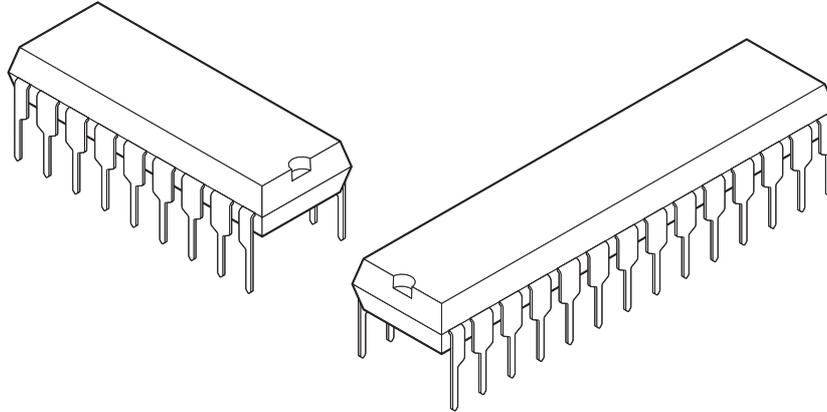
What is a PIC Microcontroller?

A PIC microcontroller is a single integrated circuit small enough to fit in the palm of a hand. 'Traditional' microprocessor circuits contain four or five separate integrated circuits - the microprocessor (CPU) itself, an EPROM program memory chip, some RAM memory and an input/output interface. With PIC microcontrollers all these functions are included within one single package, making them cost effective and easy to use.

PIC microcontrollers can be used as the 'brain' to control a large variety of products. In order to control devices, it is necessary to interface (or 'connect') them to the PIC microcontroller. This section will help to enable those with limited electronics experience to successfully complete these interfacing tasks.

What is a PICAXE microcontroller?

A PICAXE microcontroller is a standard Microchip PICmicro™ microcontroller that has been pre-programmed with the PICAXE bootstrap code. The bootstrap code enables the PICAXE microcontroller to be re-programmed directly via a simple serial connection. This eliminates the need for an (expensive) conventional programmer, making the whole download system a very low-cost simple serial cable!

The pre-programmed bootstrap code also contains common routines (such as how to generate a pause delay or a sound output), so that each download does not have to waste time downloading this commonly required data. This makes the download time much quicker.

As the blank microcontrollers purchased to 'make' PICAXE microcontrollers are purchased in large volumes, it is possible for the manufacturer to program the bootstrap code and still sell the PICAXE microcontroller at prices close to standard catalogue process for single un-programmed PIC microcontrollers. This means the cost of the PICAXE microcontroller to the end user is very economical.

Interfacing to the PICAXE Microcontroller

This section explains how to interface many different input and output devices to the PICAXE microcontroller. Explanations of BASIC commands are provided in the Commands section (available separately). The interfacing circuits can also be used with any other PIC microcontrollers such as the PIC16F84A, although these microcontrollers may require programming in assembler code.

This section is split into four subsections:

- Introduction to 'standard' interfacing circuits

- Output Device Interfacing

- Input Device Interfacing

- Advanced Component Interfacing

Note on the BASIC Code Samples

Simple BASIC code examples are provided within each subsection. The samples are not 'complete' programs but sections of code that can be included within a main program when using that particular component. When using these code samples it must be remembered that:

1. Each pin should be set up as an input or output before using the code (stamp users only).

2. If the hardware pins are changed from those given in the circuit diagrams it will be necessary to modify the pin numbers in the code.

3. Any **'let dirs** =' or **'let pins** =' commands will adjust **all** 8 pins, in the port.

4. Try to keep variables independant of each other. If a sub-procedure uses a variable, do not use the same variable anywhere else in the code. If the same variable must be used again, make sure there is no way it can clash with any other part of the code. This is the most common way of adding 'hard-to-find' bugs into software code.

Note on Component Selection

For convenience and ease of understanding, a single device has been adopted when using standard interfacing components such as transistors and MOSFETS. For instance, the 'standard' transistor selected is the darlington device BCX38C. This does not mean that this device is the only transistor that can be used in all the transistor circuits, as it is not, but it is chosen because it is suitable for the majority of project work applications. All components listed are common devices that can be purchased from most electronics distributors.

Standard Interfacing Circuits

Standard Circuits 1 - The Transistor Interfacing Circuit

Many output devices will require a transistor switching circuit. In most cases a darlington pair formed from two transistors is ideal.
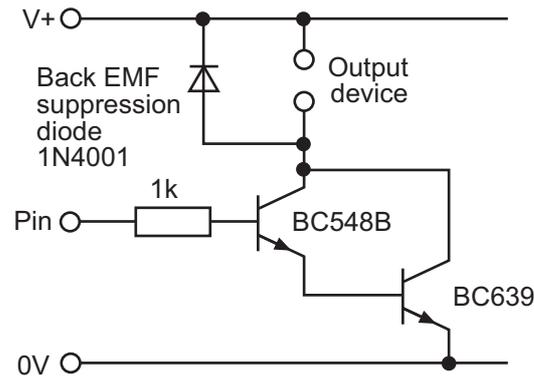
However this circuit requires that two separate transistors are used. It is possible to buy a device that contains the two transistors in a single package. This transistor is called the BCX38C, and can switch currents up to 800mA. This is the transistor used in all the circuits through this book.
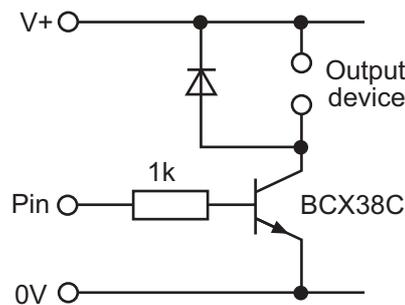
Note that it is usual to connect a back emf suppression diode across the output device. This is essential with devices such as relays, solenoids and motors which create a back emf when power is switched off. The diode type 1N4001 is the device recommended.

Standard Circuits 2 - Using a Darlington Driver IC

If a number of output devices are being controlled it may be necessary to use a number of output transistors. In this case it will often be more convenient to use a ULN2003 Darlington driver IC. This is simply a 16 pin 'chip' that contains 7 darlington transistors similar in value to the BCX38C. The 'chip' also contains internal back emf suppression diodes and so no external 1N4001 diodes are required.
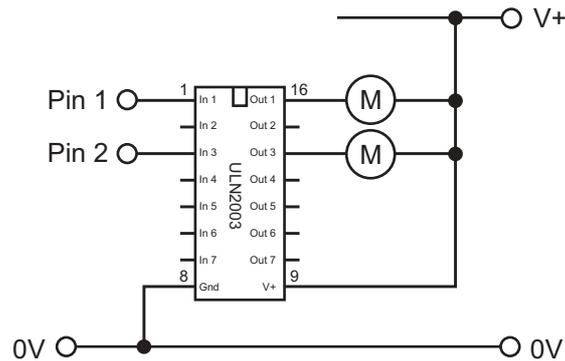
A device called the ULN2803 Darlington Driver IC is also available. This is identical to the ULN2003 except that it is an 18 pin device and contains 8 darlington pairs instead of 7. If it is necessary to pass relatively high currents through a device it can be useful to 'pair up' drivers as shown with this circuit.
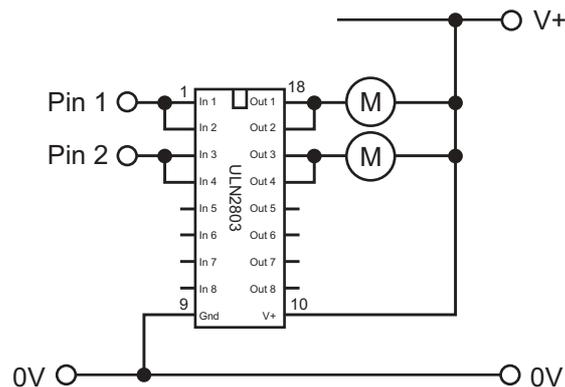
A ULN2803 darlington driver is supplied prefitted to the PICAXE project boards.

Standard Circuits 3 - The Relay Interfacing Circuit

A relay can be used to switch higher power devices such as motors and solenoids. If desired, the relay can be powered by a separate power supply, so, for instance, 12V solenoids can be controlled by the microcontroller. Note the use of a back emf suppression diode across the relay contacts. This is to prevent damage to the transistor when the relay switches off. Diode type 1N4001 is suitable for this diode.
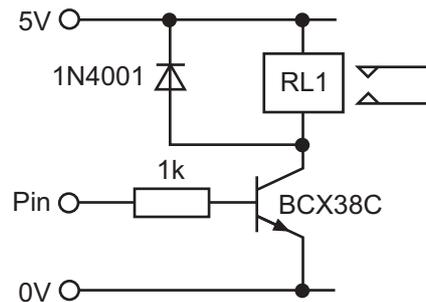
Standard Circuits 4 - The Power MOSFET Interfacing Circuit
Power MOSFETs can be used instead of darlington transistor pairs to switch medium power devices. The standard MOSFET circuit is shown below. The device IRL520 is a suitable logic level MOSFET to use in this circuit.

Note that it is usual to connect a back emf suppression diode across the output device. This is essential with devices such as relays, solenoids and motors which create a back emf when power is switched off. The diode type 1N4001 is the device recommended.

When a PICAXE chip resets the output pin is momentarily not directly driven. Therefore on sensitive circuits it may be necessary to include a 10k pulldown resistor on the MOSFET gate. This holds the gate off until the PICAXE actively drives the output.
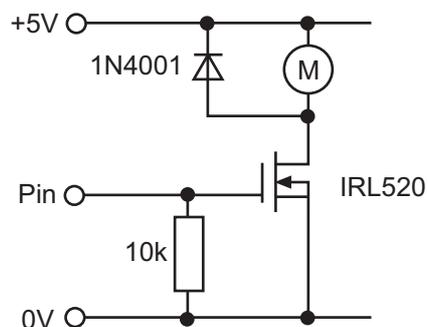
Output Device Interfacing

Output Device 1 - Light Emitting Diode (LED)

The PICAXE microcontroller can sink ("absorb") or source ("give out") a small amount of current, which means that an LED can be connected directly to the output pin. A series resistor (value 330R) is also required to limit the current.
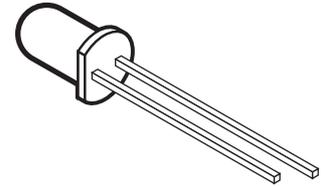
LED connected to Ground Rail.

To switch on LED – `high 1`
To switch off LED – `low  1`
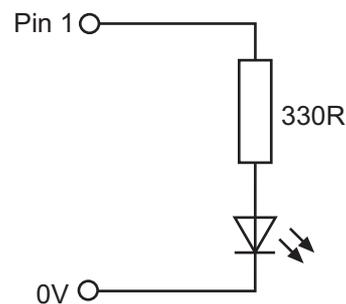
LED connected to Power Rail.

To switch on LED – `low 1`
To switch off LED – `high 1`
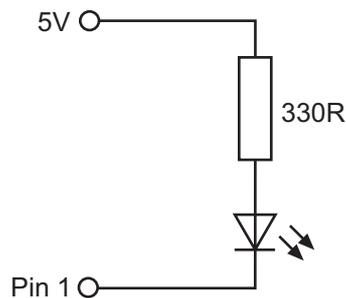
Bi-colour LEDs often contain both green and red LEDs connected in 'inverse parallel'. This means if current flows one way through the device the LED lights green, and if current flows the other way the LED lights red. Therefore by using the sink/source capabilities of the PICAXE microcontroller it is possible to light the LED in both colours.

To switch on LED in red -   high 0
                               low 1

To switch on LED in green - low 0
                               high 1

To switch off LED -   low 0
                       low 1

       or,   high 0
                  high 1
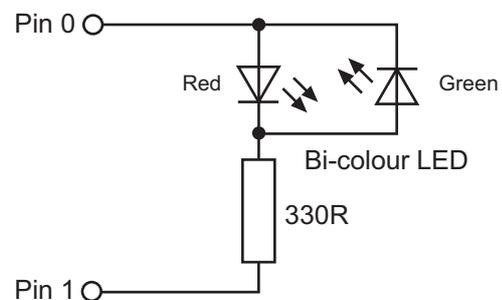
Output Device 2 - Signal Lamp

To interface a signal lamp the standard transistor interfacing circuit is used. Note that if a different power supply is used for the signal lamp, the 0V rails of each power supply must be connected to provide a common reference.

If a battery is used as the power supply, it is worth remembering that LEDs draw much less current than lamps. Therefore, if a simple 'indicator' is required, a LED will be a better solution than a lamp as the batteries will last far longer.

To switch on **Lamp** - `high 1`
To switch off **Lamp** - `low 1`

signal lamp
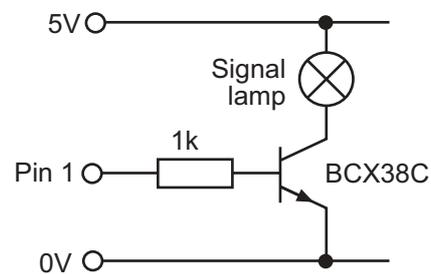
Output Device 3 - Buzzer

To interface a buzzer the standard transistor interfacing circuit is used. Note that if a different power supply is used for the buzzer, the 0V rails of each power supply must be connected to provide a common reference.

If a battery is used as the power supply, it is worth remembering that piezo sounders draw much less current than buzzers. Buzzers also just have one 'tone', whereas a piezo sounder is able to create sounds of many different tones.

To switch on buzzer -   high 1
To switch off buzzer -   low 1
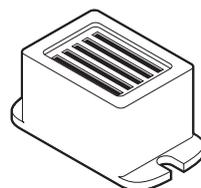
buzzer
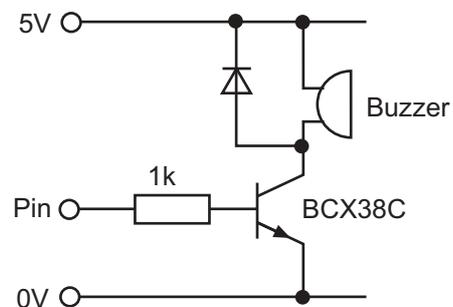
Output Devices 4 - Piezo Sounder & Speaker

A piezo sounder or speaker can be used to produce many different sounds, whereas a buzzer can only produce a single tone. Buzzers produce a noise when power is applied, but a piezo or speaker requires a pulsed signal to generate the noise. Fortunately this is very easy to generate from the microcontroller by using the BASIC 'sound' command.

Pin 1 o———

0V o———

Pin 1 o———
    +
    10uF

    40R

0V o———
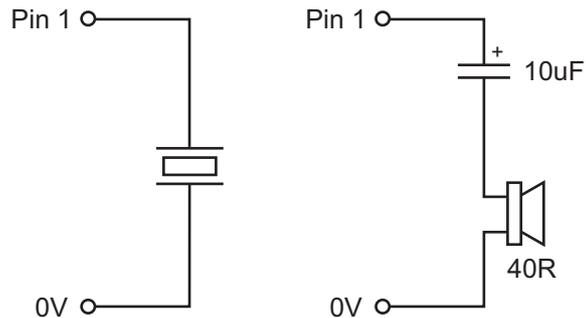
To produce a note of pitch 100, length 50 on pin 1 -

```
sound 1, (100,50)
```

To produce a varying noise using variable b1 -

```
for b1 = 1 to 100
        sound 1, (b1,25)
next b1
```

Output Devices 5 - Solar & DC "Toy" Motors

Many projects require the use of a cheap dc motor to create rotational movement. There are a number of ways motors can be interfaced to the microcontroller.
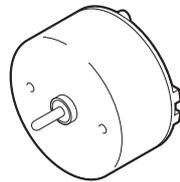
solar motor
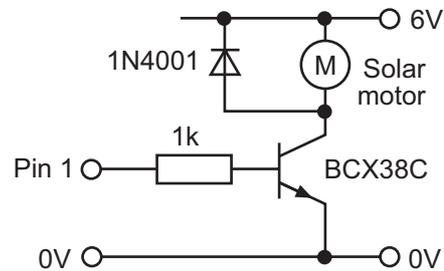
This circuit uses a darlington transistor to switch the motor on and off. This circuit will work with 'solar' motors, but may not function correctly with cheap dc 'toy' motors. This is because this type of motor introduces a lot of electrical 'noise' on to the power rails. This noise can affect the microcontroller, and in some cases can completely stop the control program functioning.

Electrical noise can be reduced by soldering suppression capacitors across the motor contacts, as shown. Use a 220nF polyester (non polarised) capacitor.
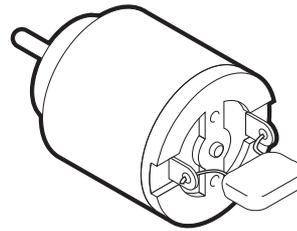
In order to switch medium power motors, a power MOSFET is used instead of a darlington transistor. The MOSFET circuit is shown below. The device IRL520 is a suitable power MOSFET to use in this circuit.
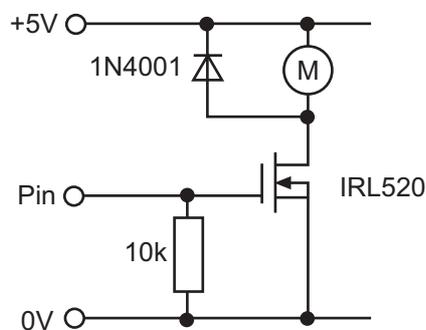
On many occasions it may be necessary to control two motors. A convenient and cheap approach would be to use a motor driver IC such as the L293D. This IC will allow control of two dc motors, using four data lines from the microcontroller. Naturally, if only one motor is to be controlled then only two output lines are used.
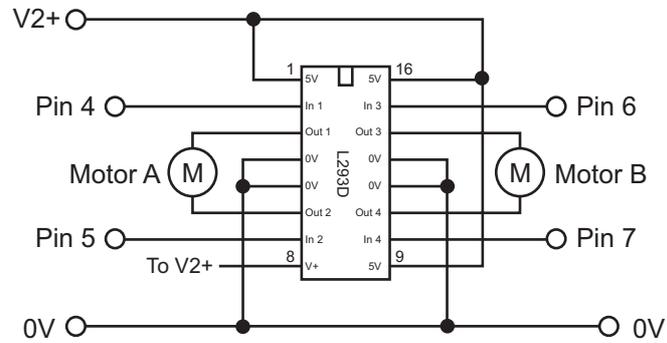
Both inputs low	- motor halt
First output high, second output low	- motor forward
First output low, second output high	- motor reverse
Both inputs high	- motor halt

Changing the states of the input pins has the effect of altering the direction of current flow through the motor, as shown below.
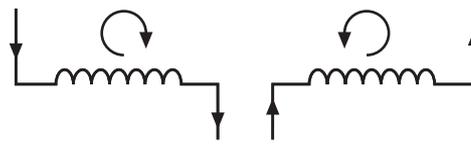
Current flow

Note that the L293D will become warm with continuous use. A heatsink bonded onto the top of the chip will help keep it cool.

One way to prevent electrical noise affecting the microcontroller is to use separate power supplies for the 'control' electronics and the motor. For example, a PP3 battery may be chosen to power the microcontroller and 4xAA cells to power the motors. Naturally it will be necessary to 'link' the two circuits so that the motor can be controlled. A relay is an ideal component to do this.
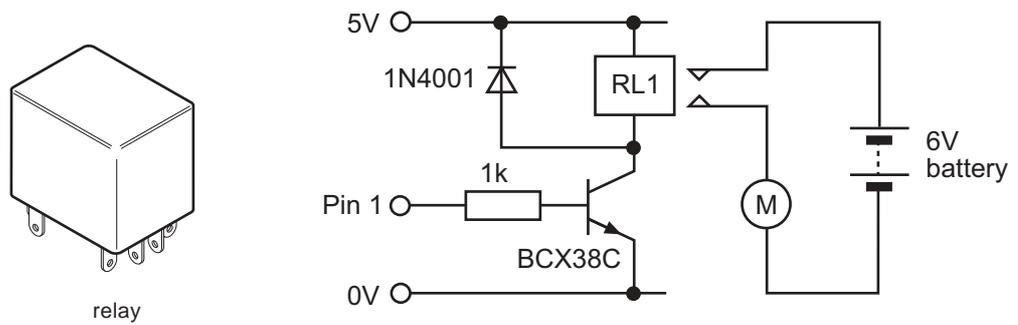
relay

The above circuit will only switch the motor on and off. If the motor is required to run in both directions (forwards and reverse), two relays can be used as shown.

Contacts 2/2 not used
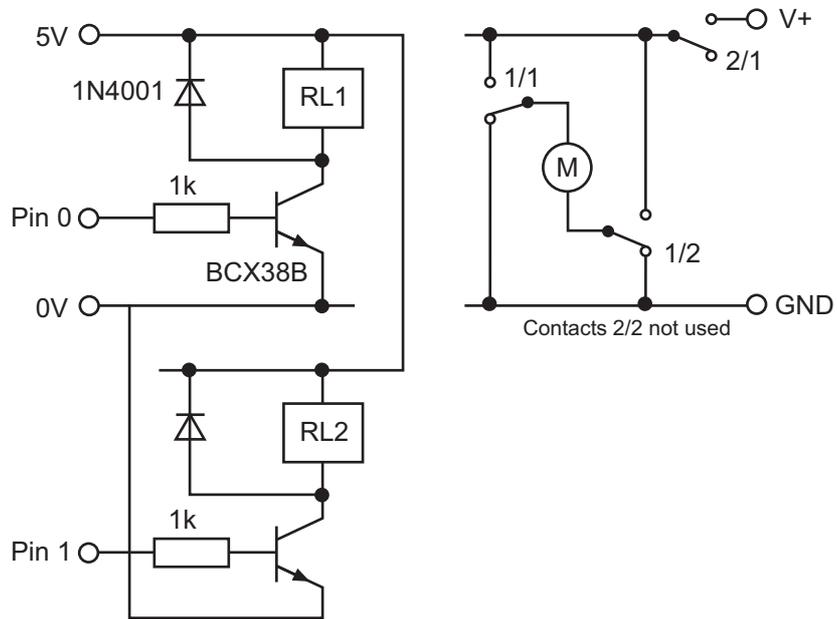
Output Device 6 - Unipolar stepper motor

Stepper motors are very accurate motors that are commonly used in computer disk drives, printers and clocks. Unlike dc motors, which spin round freely when power is applied, stepper motors require that their power supply be continuously pulsed in specific patterns. For each pulse, the stepper motor moves around one 'step', often 7.5 degrees (giving 48 steps in a full revolution).

There are two main types of stepper motors - Unipolar and Bipolar. Unipolar motors usually have four coils which are switched on and off in a particular sequence. Bipolar motors have two coils in which the current flow is reversed in a similar sequence. Use of bipolar motors is covered in the next section.

Each of the four coils in a unipolar stepper motor must be switched on and off in a certain order to make the motor turn. Many microprocessor systems use four output lines to control the stepper motor, each output line controlling the power to one of the coils.
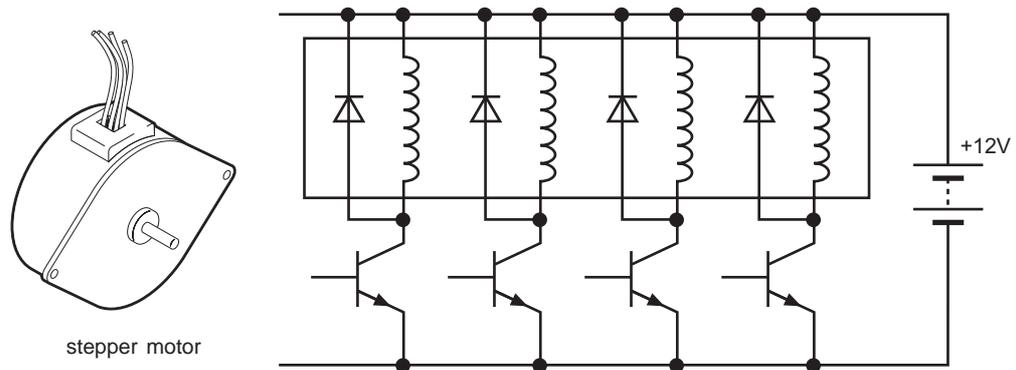
stepper motor

As the stepper motor operates at 12V, the standard transistor circuit is required to switch each coil. As the coils create a back emf when switched off, a suppression diode on each coil is also required. The table below show the four different steps required to make the motor turn.

Step	Coil 1	Coil 2	Coil 3	Coil 4
1	1	0	1	0
2	1	0	0	1
3	0	1	0	1
4	0	1	1	0
1	1	0	1	0

Look carefully at the table, and notice that a pattern is visible. Coil 2 is always the opposite (or logical NOT) of coil 1. The same applies for coils 3 and 4. It is therefore possible to cut down the number of microcontroller pins required to just two by the use of two additional NOT gates.

Fortunately the darlington driver IC ULN2003 can be used to provide both the NOT and darlington driver circuits. It also contains the back emf suppression diodes so no external diodes are required. The complete circuit is shown below.
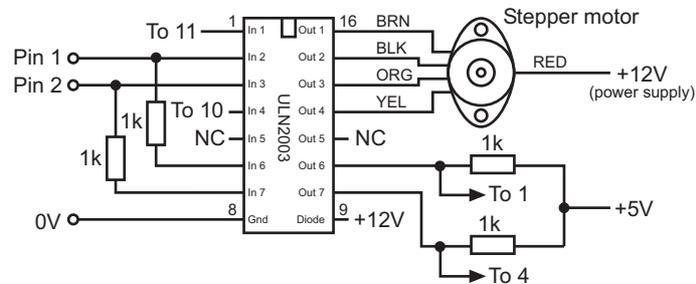
N.B. colours of stepper motor leads may vary

Before programming, there is another pattern to notice in the stepping sequence. Look at this table, which just shows coil 1 and coil 3.

Step	Coil 1	Coil 3	Change
1	1	1	
			coil 3
2	1	0	
			coil 1
3	0	0	
			coil 3
4	0	1	
			coil 1
1	1	1	

Notice the change from step 1 to step 2, just coil 3 changes. Then look at the next change - just coil 1 changes. In fact the two coils take it 'in turns' to change from high to low and back again. This high-low-high changing can be described as 'toggling' state. This makes the programming very simple by using the BASIC toggle command.

```
steps:    toggle 1            ' Toggle pin 1
          pause 200           ' Wait 200 ms
          toggle 2            ' Toggle pin 2
          pause 200           ' Wait 200ms
          goto steps          ' Loop
```

**Note:** If stepper motor 'wobbles', try adjusting wire polarity.

Output Device 7 - Bipolar Stepper motor

Stepper motors are very accurate motors that are commonly used in computer disk drives, printers and clocks. Unlike dc motors, which spin round freely when power is applied, stepper motors require that their power supply be continuously pulsed in specific patterns. For each pulse, the stepper motor moves around one 'step', often 7.5 degrees (giving 48 steps in a full revolution).

There are two main types of stepper motors - Unipolar and Bipolar. Unipolar motors usually have four coils which are switched on and off in a particular sequence. Bipolar motors have two coils in which the current flow is reversed in a similar sequence. Use of unipolar motors is covered in the previous pages.

The bipolar stepper motor has two coils that must be controlled so that the current flows in different directions through the coils in a certain order. The changing magnetic fields that these coils create cause the rotor of the motor to move around in steps.

The circuit that is normally used to control one of the coils is shown below. Notice how there are four 'control' transistors, that are switched on in 'pairs'. Therefore with two coils there are four control transistor pairs (Q1-Q4) which must be switched on and off in a certain sequence.
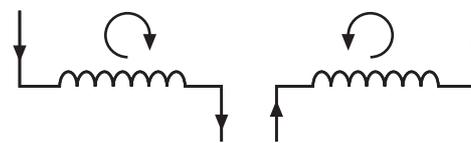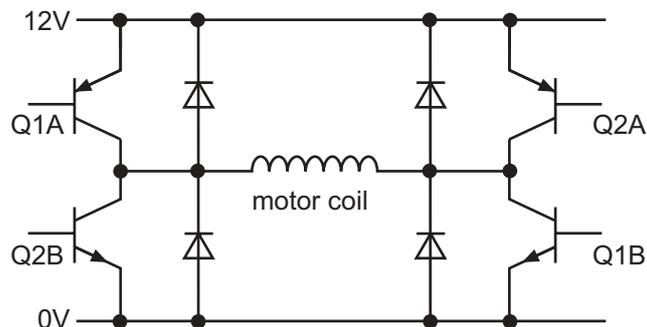
Current flow

Notice that as the coils create a back emf when switched off 8 suppression diodes (4 on each coil) are also required.

The table below show the four different steps required to make the motor turn

Step	Q1	Q2	Q3	Q4
1	1	0	1	0
2	1	0	0	1
3	0	1	0	1
4	0	1	1	0
1	1	0	1	0

Fortunately the motor driver L293D has been specifically designed to provide this transistor switching circuit. The L293D contains all 8 transistors and diodes within one 16 pin package.
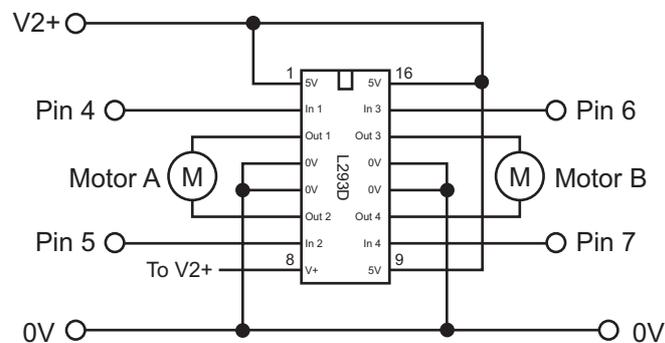
Four pins from the microcontroller are connected to the four transistor 'pairs' via IC pins 2, 7, 10 and 15.

This sample procedure makes the motor spin 100 steps to the left and then 100 steps to the right by using two sub-procedures. lstep causes the motor to move one step to the left, rstep causes the motor to move one step to the right. Variable b1 is used to store the step position and so should not be used elsewhere in the program.

```
main:  for b3 = 0 to 99                      ' start a for...next loop
           gosub lstep                       ' call left step sub-procedure
       next b3                               ' next loop
       for b3 = 0 to 99                      ' start a for...next loop
           gosub rstep                       ' call left step sub-procedure
       next b3                               ' next loop

lstep: let b1 = b1 + 1                       ' add 1 to variable b1
        goto step2                           ' goto the lookup table
rstep: let b1 = b1 - 1                       ' subtract 1 from variable b1
step2: let b1 = b1 & %00000011               ' mask lower two bits of b1
       lookup b1,(%1010,%1001,%0101,%0110),b2 ' lookup code into b2
       let pins = b2                         ' output b2 onto control lines
       return
```

Output Device 8 - Radio Control Servo

Servos are used in most radio controlled cars and planes to control the steering mechanism. They are accurate devices that always rotate the same amount for a given signal, and so are ideal for use in many automated machines.
Servos can be driven directly via the 'servo' command, or via pulsout commands.
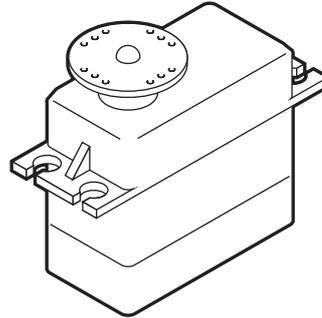
A typical servo has just three connection wires, normally red, black and white (or yellow). The red wire is the 5V supply, the black wire is the 0V supply, and the white (or yellow) wire is for the positioning signal.

The positioning signal is a pulse between 0.75 and 2.25 milliseconds (ms) long, repeated about every 18ms (so there are roughly 50 pulses per second). With a 0.75ms pulse the servo moves to one end of its range, and with a 2.25ms pulse the servo moves to the other. Therefore, with a 1.5ms pulse, the servo will move to the central position. If the pulses are stopped the servo will move freely to any position.

Unfortunately servos require a large current (up to 1A) and also introduce a large amount of noise on to the power rail. Therefore in most cases the servo should be powered from a separate power supply, as shown below. Remember that when using two power supplies the two 0V rails must be joined to provide a common reference point.

```
init:  servo 4,75            ' start servo on 4
main:  servopos 4,75         ' move servo to one end
       pause 2000            ' wait 2 seconds
       servopos 4,150        ' move servo to centre
       pause 2000            ' wait 2 seconds
       servopos 4,225        ' move servo to other end
       pause 2000            ' wait 2 seconds
       goto main             ' loop back to start
```
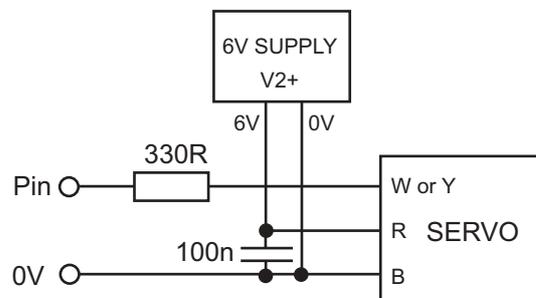
Output Device 9 - Counter module
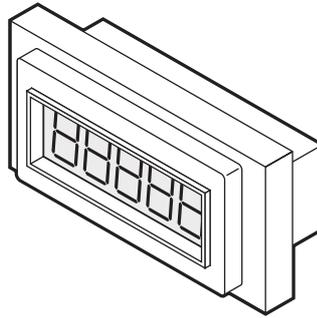
The Counter Module is a numeric LCD display module that can be used to show a 'counter' value. To increment the counter a pulse (between 1 and 1.5V) must be applied to the counter pad 3. As the PICAXE microcontroller operates at 5V a potential divider formed from resistors must be used to reduce the PICAXE microcontroller output signal to 1.5V. As the counter uses it's own, internal, 1.5V battery, the two 0V rails must also be connected.
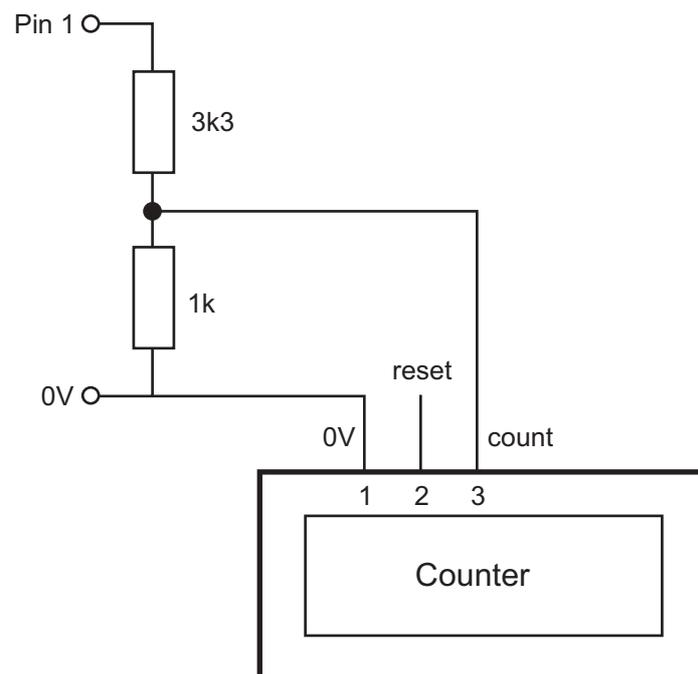
To increment counter: pulsout 1,100

To reset the counter, a second potential divider is added and connected to pin 2.

Output Device 10 - Seven Segment Display

A seven segment display contains seven LED 'bars' that can be lit up in different combinations to show the ten digits 0 to 9. In theory each 'bar' could be connected to one microcontroller output pin, but this would use up 7 of the 8 available pins!
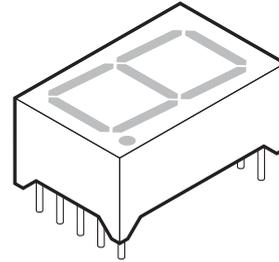
A better solution is to use a dedicated integrated circuit, such as the CMOS 4511B to control the seven segment display. This IC controls the seven segment display according to the binary 'code' on the four data lines. This system uses four pins rather than 7.

IMPORTANT NOTE - Seven segment displays are available in two types, called 'common cathode' and 'common anode'. The following circuits will only work with a '**common cathode**' type display. Use the manufacturer's datasheet to determine the pinout arrangement of the LED bars.
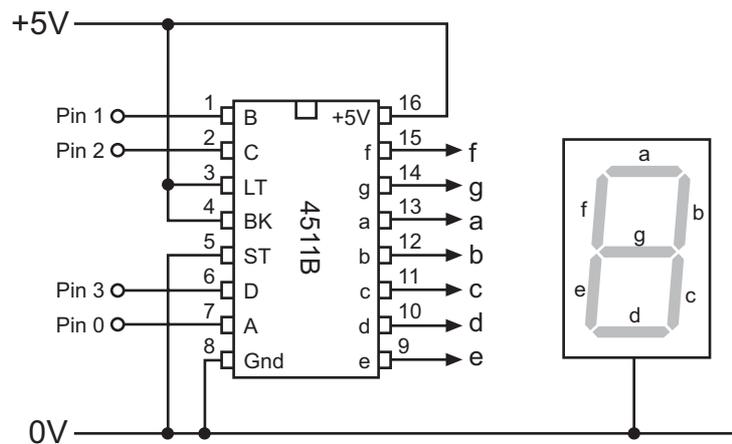
This code example counts through the digits 0 to 9

```
main:  for b1 = 0 to 9      ' Set up a for...next loop using variable b1
          let pins=b1        ' Output b1 onto the four data lines
        pause 1000        ' Pause 1 second
      next b1              ' Next
      goto main           ' Loop back to start
```

Another possible solution is to use the CMOS 4026B to control the seven segment display. This system uses just two pins to control the display. The reset pin is used to reset the display to 0, the clock pin is then used to increment the digit up from 0. This means to display the digit '4' it is necessary to reset and then pulse the clock line 4 times. In reality this means that the display shows the digits 0-1-2-3-4, but, as they are clocked extremely rapidly, the human eye cannot see the changes, and so the number '4' seems to appear immediately!
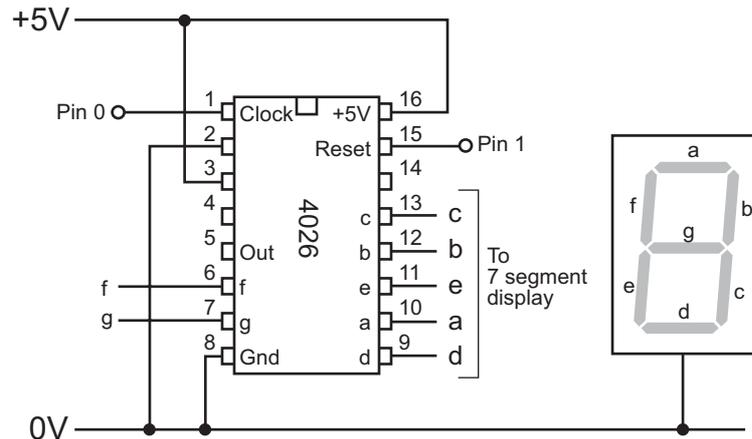
This code example uses sub-procedure 'clock' to display the digit '4', which is stored in the variable b1.

'This is the sub-procedure

```
clock:  pulsout 1,10              ' reset display to 0
        if b1 = 0 then endclk     ' if b1 = 0 then return
        for b3 = 1 to b1          ' start a for...next loop
           pulsout 0,10           ' pulse clock line
        next b3                   ' next loop
endclk: return                    ' return from sub-procedure
```

This is the main code

```
main:   let b1 = 4               ' give variable b1 the value 4
        gosub clock              ' call sub-procedure
        pause 1000               ' wait 1 second
        goto main                ' loop
```

This system can be expanded to two digits by adding a second 4026B IC and a second seven segment display, as shown in the diagram below. No changes to the code are required, just give the variable b1 a value between 0 and 99 and the number will be displayed on the two displays when sub-procedure 'clock' is called.
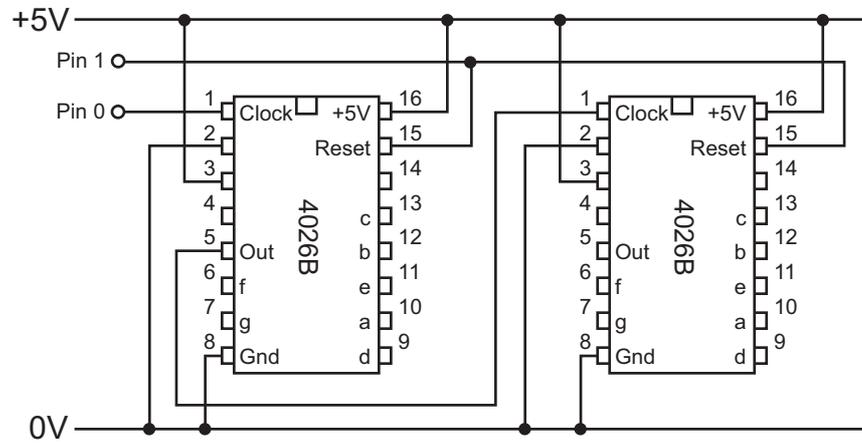
Output Device 11 - Solenoid & Solenoid Valves
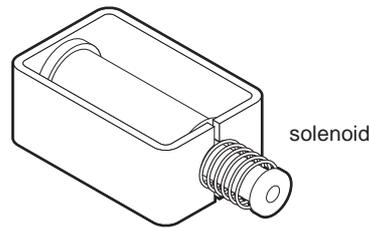
solenoid

A solenoid consists of a steel plunger inside an electric coil which is wrapped around a tube. When the coil is energised a magnetic field is created, and this draws the plunger into the tube. When the coil is de-energised a spring pushes the plunger back out of the tube.

To control a solenoid the standard MOSFET circuit is used.
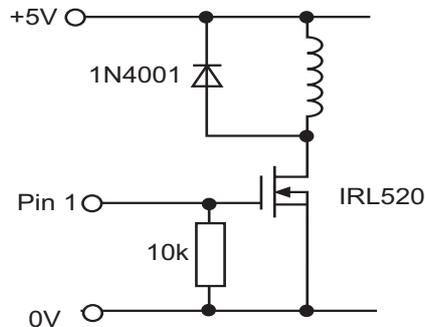
The isonic solenoid valve can be used to control air flow through a pneumatic system. Isonic valves are ideal for battery operated products as operate at a low voltage and draw much less current than traditional solenoid valves. The standard transistor switching circuit can be used to drive the isonic valve.
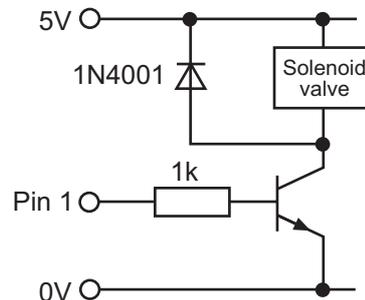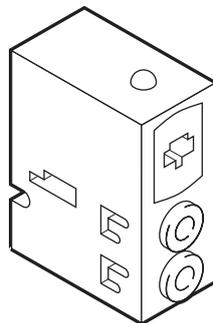
To switch the solenoid on -   high 1
To switch the solenoid off - low 1

Output Device 12 - Smart Wire & Smart Springs
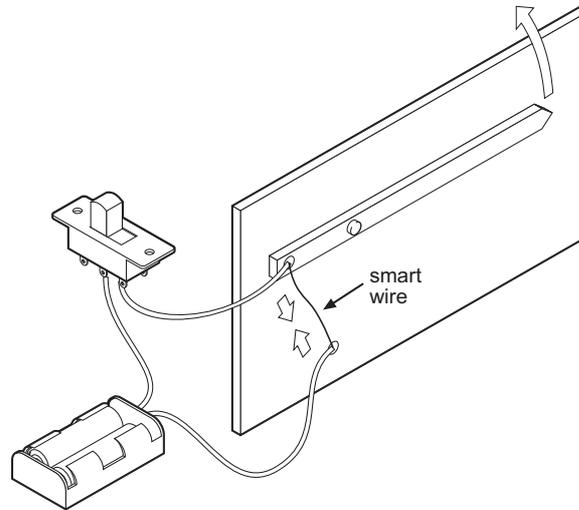
smart
wire

Shape Memory Alloy wire or springs are 'smart' materials that can be used to create mechanical actuation (movement). When an electric current is passed through the wire it heats up and so contracts with a large pulling force. When the current is removed the wire cools and so expands again (a 'traditional' steel spring is sometimes used to pull the smart wire/spring taut as it cools).

Smart wire or springs draw a relatively large current, and so the standard FET interfacing circuit should be used to interface to the microcontroller.
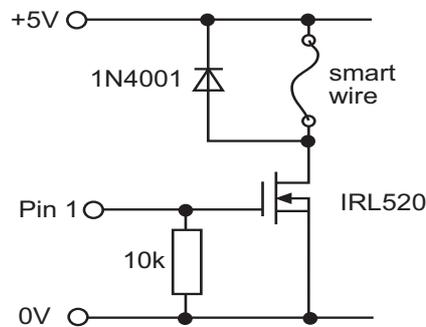
To make the wire / spring contract -   high 1
To allow the wire / spring to expand again -   low 1

Input Device Interfacing

Input Device 1 - Switches

There are a large variety of switches available, but the majority all have two 'contacts' which are either 'open' (off) or 'closed' (on). The two circuits shown below can be used with almost all switches.
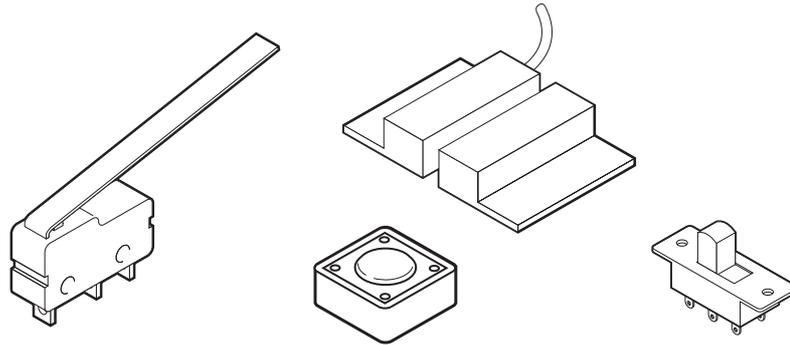
With this circuit the input pin is **low** when the switch is open and **high** when the switch is closed.

Goto 'jump' when switch is open: if pin0 = 0 then jump
Goto 'jump' when switch is closed: if pin0 = 1 then jump
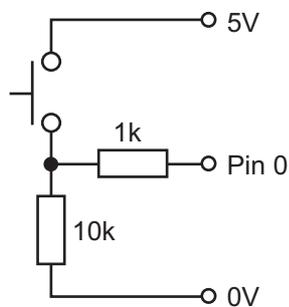
With this circuit the input pin is **high** when the switch is open and **low** when the switch is closed.

Goto 'jump' when switch is open: if pin0 = 1 then jump
Goto 'jump' when switch is closed: if pin0 = 0 then jump
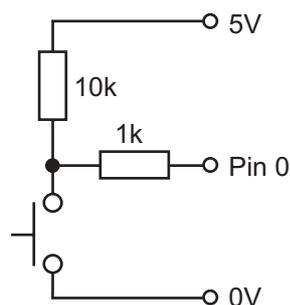
Switch Bounce

All mechanical switches 'bounce' when the switch opens or closes. This means that the switch contacts 'bounce' against each other before settling. As the PICAXE microcontroller operates so quickly it is possible that in some programs the microcontroller may register 2 or 3 of these 'bounces' instead of just registering one 'push'.

The simplest way to debounce a circuit is to simply add a time delay (pause 100) after the if... command. If the section of code after the push is quite long this time delay will occur naturally (as the other code commands are carried out) and so is unnecessary. However if the code does not have a long delay, as in the following example, a pause command can be used instead.
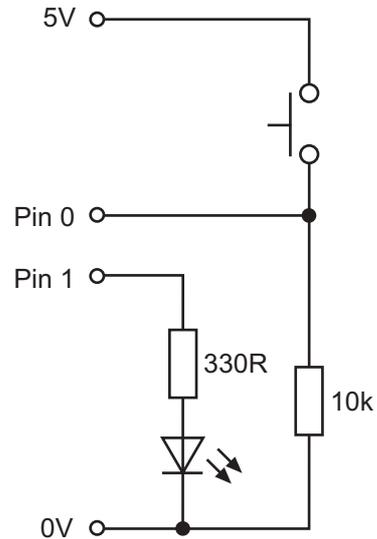
The following two programs show the effect of switch bouncing. The program should light the LED on pin1 when the switch connected to pin0 has been pressed more than 5 times. However, the first listing may not work correctly, because the microcontroller may count 'bounces' rather than actual pushes, and so the LED may light prematurely.

```
init:  let b0 = 0

main:  if pin 1 = 1 then add
       goto main

add:   let b0 = b0 + 1
       if b0 < 5 then main
       high 1
       goto main
```

```
init:  let b0 = 0

main:  if pin 1 = 1 then add
       goto main

add:   pause 100    'short delay
       let b0 = b0 + 1
       if b0 < 5 then main
       high 1
       goto main
```

Input Device 2 - Potentiometer

A potentiometer (or 'variable resistor') has a spindle that can be moved to change the resistance value of the potentiometer. This can be used to measure rotational or linear movement.
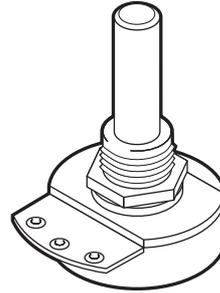
The readADC command is used to measure the value of the resistance by carrying out an Analogue to Digital Conversion. The value of the resistance is given a 'value' between 0 and 255 which is then stored in a variable. After storing the reading in the variable, the if...then command can be used to perform different functions.
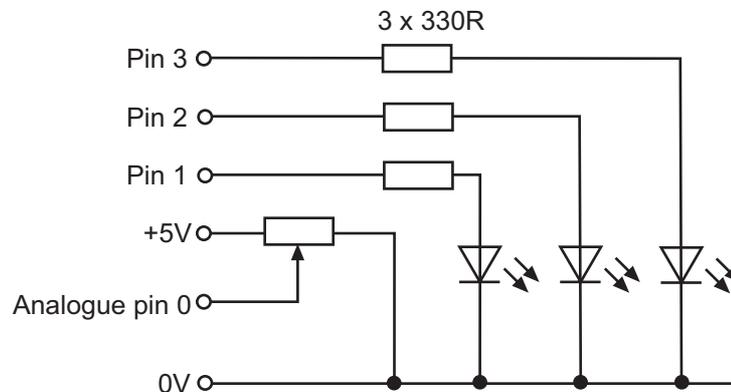
The program below lights three different LEDs (connected to pins 1, 2 and 3), depending on the analogue sensor reading.

```
main:    readadc 0,b1          ' read value on pin0 into variable b1
         if b1<75 then light1   ' if b1 is less than 75 then light 1
         if b1<175 then light2  ' if b1 is less than 175 then light 2
         goto light3            ' if b1 is greater than 175 then light 3

light1: high 1                  ' switch on LED 1
        low 2                   ' switch off LED 2
        low 3                   ' switch off LED 3
        goto main              ' loop

light2: low 1                   ' switch off LED 1
        high 2                  ' switch on LED 2
        low 3                   ' switch off LED 3
        goto main              ' loop

light3: low 1                   ' switch off LED 1
        low 2                   ' switch off LED 2
        high 3                  ' switch on LED 3
        goto main              ' loop
```

Input Device 3 - Light Dependant Resistor (LDR)

A Light Dependant Resistor (LDR) is a resistor that changes in value according to the light falling on it. A commonly used device, the ORP-12, has a high resistance in the dark, and a low resistance in the light. Connecting the LDR to the microcontroller is very straight forward, but some software 'calibrating' is required.
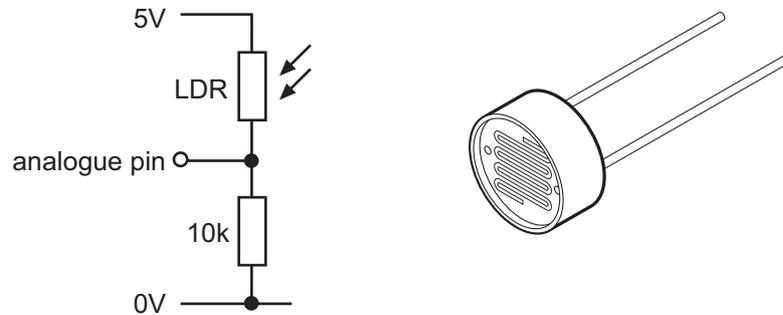
It should be remembered that the LDR response is not linear, and so the readings will not change in exactly the same way as with a potentiometer. In general there is a larger resistance change at brighter light levels. This can be compensated for in the software by using a smaller range at darker light levels. Experiment to find the most appropriate settings for the circuit.

```
main: readadc 0,b1            ' read the value
      if b1<50 then light1    ' range 0-50 = 50
      if b1<100 then light2   ' range 50-100 = 50
      if b1<145 then light3   ' range 100-145 = 45
      if b1<175 then light4   ' range 145-175 = 30
      goto main
```

Input Device 4 - Thermistor

A thermistor is a resistor that changes in value according to it's heat. In actual fact all resistors change in value as they heat up or cool down, but thermistors are manufactured to show a large resistance change. Connecting the thermistor to the microcontroller is very straight forward, but some software 'calibrating' is required.
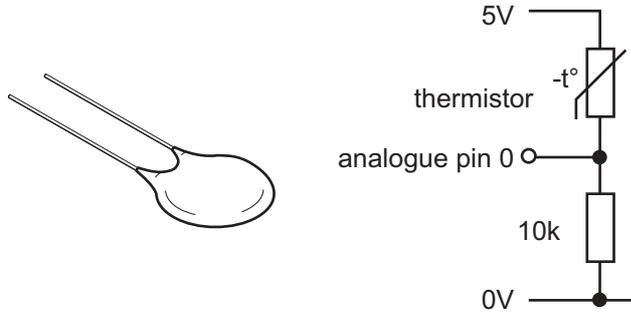
It should be remembered that the thermistor response is not linear, and so the readings will not change in exactly the same way as with a potentiometer. In general there is a larger resistance change at lower temperatures. This can be compensated for in the software by using a smaller range at higher temperatures. Experiment to find the most appropriate settings for the circuit.

```
main:  readadc 0,b1              ' read the value
       if b1<50 then light1      ' range 0-50 = 50
       if b1<100 then light2     ' range 50-100 = 50
       if b1<145 then light3     ' range 100-145 = 45
       if b1<175 then light4     ' range 145-175 = 30
       goto main
```

Advanced Component Interfacing

Advanced Interfacing 1 - LCD Display

A Liquid Crystal Display is an electronic device that can be used to show numbers or text. There are two main types of LCD display, numeric displays (used in watches, calculators etc) and alphanumeric text displays (often used in devices such as photocopiers and mobile telephones).
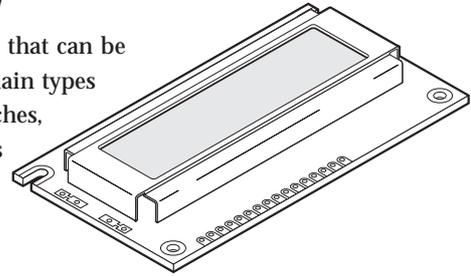
The display is made up of a number of shaped 'crystals'. In numeric displays these crystals are shaped into 'bars', and in alphanumeric displays the crystals are simply arranged into patterns of 'dots'. Each crystal has an individual electrical connection so that each crystal can be controlled independently. When the crystal is 'off' (i.e. when no current is passed through the crystal) the crystal reflect the same amount of light as the background material, and so the crystals cannot be seen. However when the crystal has an electric current passed through it, it changes shape and so absorbs more light. This makes the crystal appear darker to the human eye - and so the shape of the dot or bar can be seen against the background.

It is important to realise the difference between a LCD display and an LED display. An LED display (often used in clock radios) is made up of a number of LEDs which actually give off light (and so can be seen in the dark). An LCD display only reflects light, and so cannot be seen in the dark.

LCD Characters

The table on the next page shows the characters available from a typical LCD display. The character 'code' is obtained by adding the number at the top of the column with the number at the side of the row.

Note that characters 32 to 127 are always the same for all LCDs, but characters 16 to 31 & 128 to 255 can vary with different LCD manufacturers. Therefore some LCDs will display different characters from those shown in the table.

Characters 0 to 15 are described as 'user-defined' characters and so must be defined before use, or they will contain 'randomly shaped' characters. For details on how to use these characters see the LCD manufacturers data sheets.

		Column Value															
		0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
Row Value	0	CG RAM (1)			0	@	P	`	p								
	1	CG RAM (2)	!	1	A	Q	a	q									
	2	CG RAM (3)	"	2	B	R	b	r									
	3	CG RAM (4)	#	3	C	S	c	s									
	4	CG RAM (5)	$	4	D	T	d	t									
	5	CG RAM (6)	%	5	E	U	e	u									
	6	CG RAM (7)	&	6	F	V	f	v									
	7	CG RAM (8)	'	7	G	W	g	w									
	8	CG RAM (1)	(	8	H	X	h	x									
	9	CG RAM (2)	)	9	I	Y	i	y									
	10	CG RAM (3)	*	:	J	Z	j	z									
	11	CG RAM (4)	+	;	K	k											
	12	CG RAM (5)	,	<	L	\	l										
	13	CG RAM (6)	-	=	M	]	m	}									
	14	CG RAM (7)	.	>	N	^	n											
	15	CG RAM (8)	/	?	O	_	o											

The operation of the display is quite complex as the display can actually store more characters than can be displayed at once. A simple model makes this easier to understand. Imagine a piece of paper with a row of letters written across it. If a piece of card is taken, which has a 'window' cut in it, and the card is placed over the paper, only some of the letters will be visible. The other letters are still there, it's just that they cannot be seen. This is how a LCD display works - it stores a lot of characters, but only shows a few, through the 'display window', at once

20 letters stored in display memory

a b c d e f g h i j k l m n o p q r s t

Only 16 letters can be seen at one time

a b c d e f g h i j k l m n o p	

	b c d e f g h i j k l m n o p q	

Start with a piece of paper, on which one letter is written. Place the card over the paper, and the letter will be visible because it shows through the 'display window'. Remove the card, write another letter, replace the card and they will both be visible. In fact all of the first sixteen letters will be visible, but the seventeenth will not, as the 'display window' is only wide enough for 16 letters.

Blank 'paper'

First letter can be seen

a	

Next letter can be seen

a b	

17th letter cannot be seen as it is 'outside' the display window

a b c d e f g h i j k l m n o p	

To be able to see the seventeenth letter it is necessary to move (or 'scroll') the display window one place to the right, but this will also mean that the first letter can no longer be seen. Advantage can be taken of this 'moving' window method to make long messages appear to scroll across the LCD screen. To do this a long message is written into the LCD memory, and then the display window is repeatedly scrolled across the message. This is equivalent to 'pulling' the paper under the window to show the long message. The LCD window does not 'physically' move - so to anyone watching the LCD the letters 'appear' to be moving to the right.

```
a b c d e f g h i j k l m n o p
```

```
  b c d e f g h i j k l m n o p q
```

```
    c d e f g h i j k l m n o p q r
```

```
      d e f g h i j k l m n o p q r s
```

On most LCD displays there is memory for 40 characters on each line. Each space in the RAM memory can be thought of as a 'box' which is ready to hold a single character. Each RAM 'box' has a numbered address to describe it. The first line RAM 'boxes' are at addresses 128 to 191, the second line RAM 'boxes' are from 192 to 255.

16x2 displays have a window that is two lines deep. That means that 16 letters can be seen on each line. If a character is to be printed on the second line, it is necessary to move the cursor to the start of line 2. Moving the cursor is very simple; simply send the RAM address (of the 'box' to be moved) as an instruction. Therefore to move the cursor to the start of the second line, simply send the instruction '192' to the LCD module. To move the cursor to the fifth position on the second line send the instruction '197' (=192+5).

Note about 16x1 displays...
Most 16x1 LCDs are in actual fact 8x2 LCDs, but with the 'second' line positioned directly after the first (instead of underneath it). This makes 16x1 displays confusing to use, as, after 8 characters have been printed, the cursor seems to disappear in the middle of the display! If this type of display is needed, remember that the 'ninth' character is actually the first character of the second line.

There are three main ways of interfacing LCDs to microcontrollers.
1) Serial LCD firmware chip
2) Serial LCD Module with onboard firmware chip
3) Direct Connection

Connecting The LCD using a serial firmware chip(OPTION 1)
The serial LCD firmware is used to allow serial control of an alphanumeric LCD. This allows microcontrollers (and microcontroller based systems such as the PICAXE or Basic Stamp) to visually output user instructions or readings onto a text screen without the need for a host computer. This is especially useful when working, for example, with analogue sensors, as the analogue reading can easily be displayed on the LCD module. All LCD commands are transmitted serially via a single microcontroller pin. A sample instruction, using the **serout** command is as follows:

to print the text 'Hello' the instruction is simply

    serout 7,T2400,("Hello")
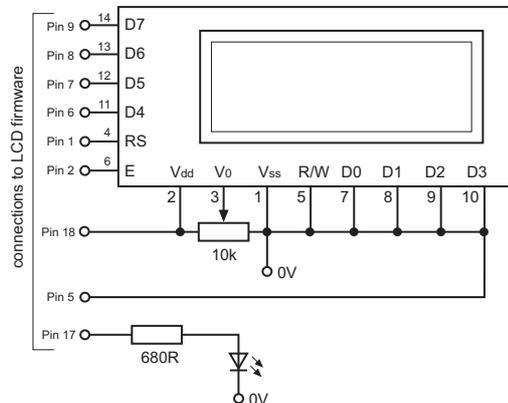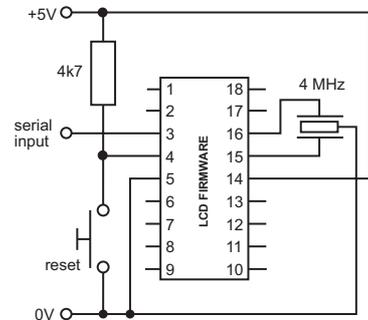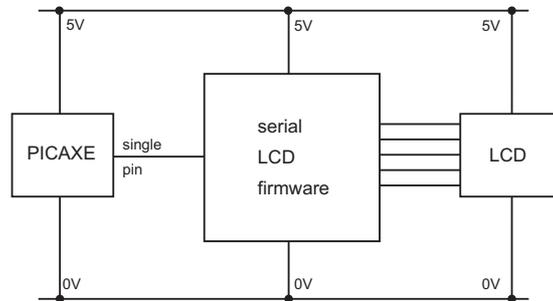
For more information, see the Serial LCD Firmware datasheet:
        www.rev-ed.co.uk/docs/frm010.pdf

Using a Serial LCD Module (OPTION 2)

The serial LCD module, part AXE033, contains a LCD module fitted to a custom printed circuit board fitted with a LCD firmware chip. This enables the user to rapidly build an LCD circuit that uses the single wire connection as with option 1. See the AXE033 Serial LCD/ Clock Module datasheet for more details:

www.rev-ed.co.uk/docs/axe033.pdf

Connnecting The LCD (OPTION 3)
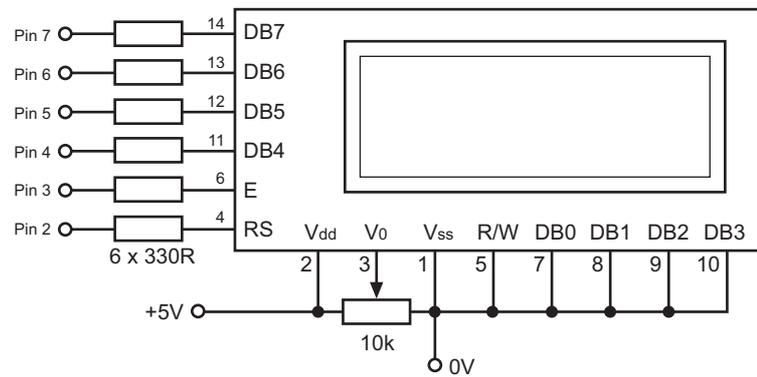
The LCD has 6 lines that can be connected directly to the PICAXE microcontroller pins. However it is a good design practice to add a low value resistor (e.g. 330R) on the lines to protect against static discharges. The 10k potentiometer connected to pin 3 is used to adjust the contrast of the display. All unused lines should be tied to ground as shown.

A Simple LCD Program

The following program will print out the phrase 'Hello there!' on two lines of the LCD display. It uses three sub-procedures called init, wrins and wrchr. These three sub-procedures carry out all the 'difficult' software tasks, and are 'standard' sub-procedures that will not have to be changed. In fact they can be used without understanding how they work, but it is necessary to know what they do:

init   'initialises' the LCD so that it is ready to accept instructions
wrins   sends an instruction stored in variable b1 to the LCD module
wrchr   sends a character stored in variable b1 to be 'printed' on the LCD screen

The three sub-procedures are explained further in the following sections.

```
EEPROM 0,("Hellothere!")     ' store the text in the EEPROM memory


        gosub init           ' initialise LCD

main:   let b1 = 1           ' set b1 to 'clear display' instruction
        gosub wrins          ' send instruction to LCD

        for b3 = 0 to 4      ' setup for...next loop ("Hello" - positions 0 to 4)
           read b3, b1        ' read letter from EEPROM into variable b1
           gosub wrchr        ' send character to LCD
        next b3              ' next loop

        let b1 = 192         ' set b1 to 'start of second line' position
        gosub wrins          ' send instruction to LCD

        for b3 = 5 to 11     ' setup for...next loop ("there!"-positions 5 to 11)
           read b3, b1        ' read letter from EEPROM memory into variable b1
           gosub wrchr        ' send character to LCD
        next b3              ' next loop
```

More Advanced LCD Program

The following program scrolls the message 'Hello there everybody!' across the screen. As the text is longer than 16 letters, the message is first stored in the LCD memory, and then the display window is repeatedly scrolled to show all the message.

```
EEPROM 0,("Hello there everybody!") ' store the text in the EEPROM memory

          gosub init            ' initialise LCD

start:    let b1 = 1            ' set b1 to 'clear display' instruction
          gosub wrins           ' send instruction to LCD

          for b3 = 0 to 22      ' setup a for...next loop
             read b3, b1        ' read letter from EEPROM into variable b1
             gosub wrchr        ' send character to LCD
          next b3               ' next loop

          let b1 = 12           ' set b1 to 'hide cursor' instruction
          gosub wrins           ' send instruction to LCD

main:     let b1 = 24           ' set b1 to 'scroll display left' instruction
          gosub wrins           ' send instruction to LCD
          pause 250             ' pause for 0.25s
          goto main             ' loop
```

Standard LCD Sub-Procedures (Direct Connection)
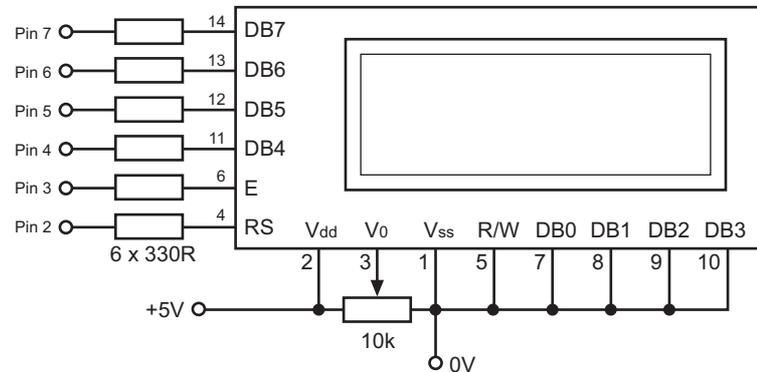
Before the sub-procedures are studied, it is important to understand how the LCD module operates. It has two modes of operation, which are called 'character' mode and 'instruction' mode. The RS pin (pin 2) controls the mode - when high the LCD is in character mode, when low the LCD is in instruction mode.

The character or instruction is sent as a 4 bit binary number down the data lines (pins 7-4). Every time the Enable pin (pin 3) is 'pulsed' the LCD reads the data lines and prints the character (or carries out the instruction) which is given by the number on the data lines.

This is not quite the whole story, as each character or instruction is actually made up of an 8 bit number, which contains a table of all the character and instruction codes. As there are only four data lines, this 8 bit number is split into two 'halves' which are sent one after the other. The two halves are called the 'high nibble' and the 'low nibble'. This means that two nibbles are transmitted down the data lines for each character.

1011		0101	= 10110101
high nibble	+	low nibble	= byte

The three 'standard' sub-procedures described below perform all of the 'complicated' software tasks when using the LCD display. Each sub-procedure is called from the main program to perform a certain task. These tasks are:

init   initialise the display and sets the module to two line operation
wrchr 'prints' one 'character' onto the LCD screen
wrins writes one 'command' to the LCD module.
        (This is actually just the wrchr sub-procedure with the addition of one line that sets the RS line into 'instruction' mode at the start of the sub-procedure).

```
init:   let pins = 0        ' Clear all output lines
        let b4 = 0          ' Reset variable b3
        let dirs = 252      ' Set pins 2-7 as output lines (Stamp only).
        pause 200           ' Wait 200 ms for LCD to reset.
        let pins = 48       ' Set to 8-bit operation.
        pulsout 3,1         ' Send data by pulsing 'enable'
        pause 10            ' Wait 10 ms
        pulsout 3,1         ' Send data again
        pulsout 3,1         ' Send data again
        let pins = 32       ' Set to 4-bit operation.
        pulsout 3,1         ' Send data.
        pulsout 3,1         ' Send data again.
        let pins = 128      ' Set to two line operation
        pulsout 3,1         ' Send data.
        let b1 = 14         ' Screen on, cursor on instruction
        gosub wrins         ' Write instruction to LCD
        return

wrchr:  let pins = b1 & 240 ' Mask the high nibble of b1 into b2.
        high 2              ' Make sure RS is high
        pulsout 3,1         ' Pulse the enable pin to send data.
        let b2 = b1 * 16    ' Put low nibble of b1 into b2.
        let pins = b2 & 240 ' Mask the high nibble of b2
        high 2              ' Make sure RS is high
        pulsout 3,1         ' Pulse enable pin to send data.
        return

wrins:  let pins = b1 & 240 ' Mask the high nibble of b1 into b2.
        pulsout 3,1         ' Pulse the enable pin to send data.
        let b2 = b1 * 16    ' Put low nibble of b1 into b2.
        let pins = b2 & 240 ' Mask the high nibble of b2
        pulsout 3,1         ' Pulse enable pin to send data.
        high 2              ' Back to character mode
        return
```

*Note that init uses a let dirs = commands that will affect all 8 pins, not just the 6 used by the LCD display. The let pins = commands used by wrins/wrchr will **not** alter the state of unused pins 0 and 1. Do not use variable b1 or b2 (or w0 or w1) for any other function within a program.*

NB. The | character is 'SHIFT + \' (next to Z on a UK layout keyboard).

Using the LCD Instruction set

The codes for the LCD instructions are given below. Each code can be sent to the LCD module by using the wrins sub-procedure. These instructions can be used to make the LCD messages more interesting - for instance by flashing the screen or creating 'moving' messages which scroll across the screen.

Code	Instruction
1	Clear display and move to the start of the first line
2	Move the cursor and display 'window' to the start of the first line
4	Set 'right to left printing' mode
5	Set 'scroll printing to the left' mode
6	Set 'left to right printing' mode
7	Set 'scroll printing to the right' mode
10	Turn visual LCD screen off
12	Hide cursor
13	Make cursor flash
14	Turn visual LCD screen (and cursor) on
16	Move cursor left one position
20	Move cursor right one position
24	Scroll display 'window' left one position
28	Scroll display 'window' right one position
128	Move cursor to the start of the first line
192	Move cursor to the start of the second line

Examples:

Clear the display

```
clear:      let b1 = 1             ' Set b1 to clear instruction
            call wrins             ' Send it to LCD
```

Move cursor to the second line

```
clear:      let b1 = 192           ' Set b1 to start of second line
            call wrins             ' Send it to LCD
```

Flash a message 10 times

```
flash:      for b3 = 1 to 10       ' Start a for...next loop using
                                   ' variable b3 Don't use b1!!
               let b1 = 10         ' Set b1 to 'turn visual display
                                   ' off' instruction
               gosub wrins         ' Send instruction to LCD
               pause 200           ' Pause for 0.2 second
               let b1 = 14         ' Set b1 to 'turn visual display
                                   ' back on' instruction
               gosub wrins         ' Send instruction to LCD
               pause 200           ' Pause for 0.2 second
            next b3                ' End of for...next loop
```

Scroll a long message (30 characters long)

```
scroll:     for b3 = 1 to 30       ' Start a for...next loop using
                                   ' variable b3 Don't use b1!!
               let b1 = 28         ' Set b1 to 'scroll display
                                   ' window right' instruction
               gosub wrins         ' Send instruction to LCD
               pause 200           ' Pause for 0.2 second
            next b3                ' End of for...next loop
            let b1 = 1             ' Set b1 to 'move scroll window
                                   ' back to start' instruction
            gosub wrins            ' Send instruction to LCD
            pause 200              ' Pause for 0.2 second
            goto scroll            ' Loop
```

## Advanced Interfacing 2 - Serial Interfacing to a Computer.

Most computers can 'talk' to other devices by serial communication. Serial communication uses a common 'protocol' (or code) where characters are converted into numbers and then transmitted via cables. A computer mouse normally 'communicates' serially with a computer, and computer modems work by turning these numbers into sounds to travel down telephone lines.

As all computers use the same ASCII code for transmitting and receiving characters it is relatively easy to program the PICAXE microcontroller to 'talk' to any type of computer. All that is needed is a suitable cable and some very simple electronic circuits.

Connecting to the Computer

The system we will use requires just three wires between the computer and the microcontroller. The ground wire provides a common reference, the RX wire sends signals from the computer to the PICAXE microcontroller, and the TX wire sends signals from the PICAXE microcontroller to the computer.

The best way to make a serial cable is to buy a serial 'extension' cable and cut it in half. This will give two cables with a suitable connector at each end. The diagrams below show the various wiring connections required.

```
RX = 3              RX = 3              RX = 3
TX = 2              TX = 2              TX = 5
GND = 5             GND = 7             GND = 4

PC/RISC PC        PC (25 way)           Mac
```
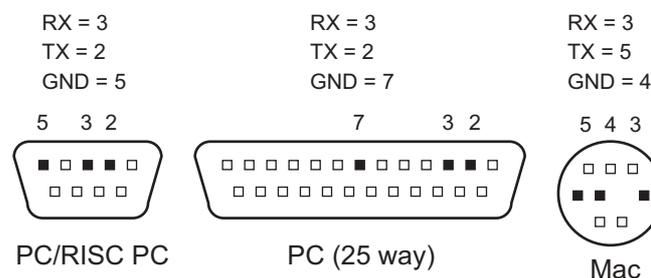
Computer Communication Software

To use this system a communication software package is required for the PC. The examples below use the Terminal option within the Programming Editor software, but any communications package can be used.

There are various different protocols that can be used for serial communication, and it is important that both the computer and the microcontroller use the same setting. The 2400,N,8,1 protocol is used here, which means baud speed 2400, no parity, 8 data bits and one stop bit. This baud speed is quite slow by modern standards, but is quite sufficient for the majority of project work tasks. All 'handshaking' (hardware or software) must also be disabled.

PICAXE Microcontroller Interfacing Circuit

The system described here requires just three wires between the computer and the PICAXE microcontroller. Strictly speaking RS232 serial voltages should be at ±15V, but the standard 5V from the on-board 5V regulator will be used here. This is not the industry standard, but works perfectly OK with the majority of computers. This is the circuit that will be used use for serial communication.

To provide true RS232 voltages another integrated circuit is required. The most common IC used is the MAX232, which has on-board voltage boosters to create the required voltage swing. If this setting is used it is necessary to change the N2400 (negative) in all the serial software commands to T2400 (true positive).
ONLY USE ONE OF THESE TWO CIRCUIT OPTIONS!
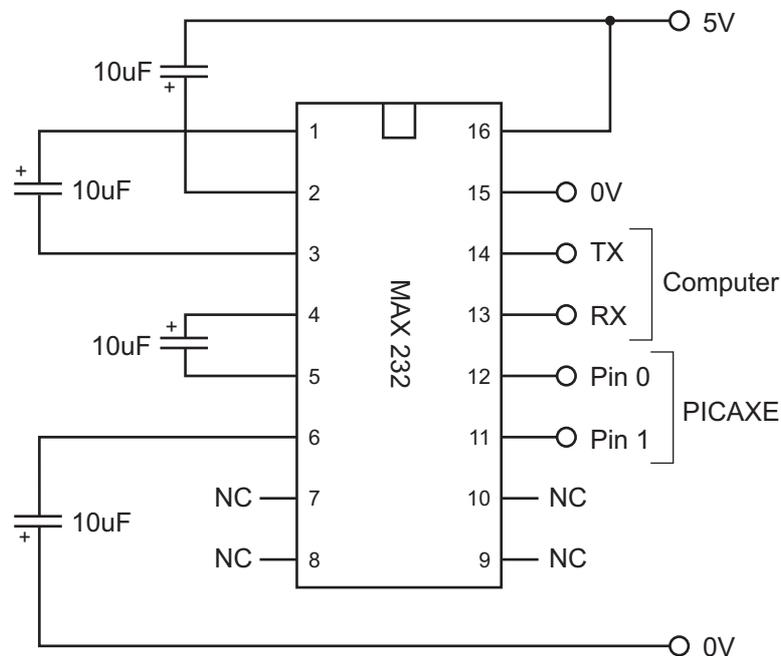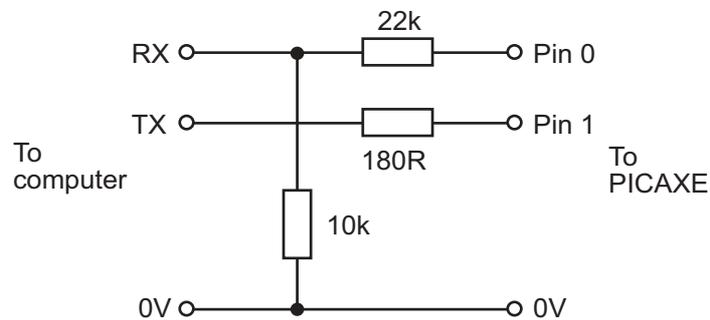
NB. Note polarity - capacitors connected to pins 2 and 6 are connected 'upside down'.

Transmitting Characters to the Computer Screen

The following program will transmit the word 'Hello' to the computer screen over and over again. If the cable is connected and the communication software is operating correctly, the word will appear every second.

```
main:  serout 1,N2400,("Hello")      ' Send the word 'Hello'
       serout 1,N2400,(10,13)        ' Send the 'new line' instructions.
       pause 1000                    ' Wait one second
       goto main                     ' Loop back to the start
```

Notice that "text" must be enclosed within speech marks. This tells the microcontroller to convert the text into a string of ASCII codes. Individual ASCII codes can be transmitted by just giving their numbers. Therefore the two commands below achieve the same task:

```
       serout 1,N2400,("Hello")
       serout 1,N2400,(72,101,108,108,111)
```

Receiving Keyboard Input from the Computer

It can be useful to be able to use a keyboard for people to 'answer' questions. This is achieved by using the serin command as shown below.

```
main:  serout 1,N2400,(10,13)            ' Start a new line
       serout 1,N2400,("Press a key- ")  ' Send a message
       serin 0,N2400,b1                  ' Receive a character into variable b1
       serout 1,N2400,(b1)               ' Transmit character back to the screen
       if b1="a" then hot                ' Is character 'a'? If yes goto hot
       goto main                         ' No, so loop back to start

hot:   serout 1,N2400, (10,13,"A is the Hot Key!")
                                         ' Send message
       goto main                         ' Loop back to start
```

If this program is run and then a key is pressed on the keyboard, the character will appear on the screen. This

is the microcontroller (not the computer) working. The keyboard press has been received from the keyboard and then transmitted back to the screen!

Characters or numbers?

Consider this command: `serout  1,N2400,(65)`
This will send the ASCII character 'A' to the screen.

Now consider this command: `serout  1,N2400,(b1)`
This will send the character stored in variable b1 to the screen, and so if b1=65, the character 'A' will be sent to the screen.

However, variables are often used to store the answers to mathematical sums, and so it may be necessary to send the number '65' to the screen rather than the letter 'A'. To do this, the microcontroller must be told that a number is to be sent rather than a character. This is achieved by adding a hash (#): `serout  1,N2400,(#b1)`

This will send the number '65' (actually the two characters '6' and '5') to the screen rather than the character 'A'.

This is a summary of the serial commands used. Remember that the pin number may have to be changed, and also to the N2400 section to T2400 if the MAX232 interfacing circuit is used.

serout 1,N2400,("Hello")- Sends a message to the screen.
serout 1,N2400,(10) -   Sends a direct ASCII instruction to the screen.
serout 1,N2400,(b1) -   Sends an ASCII character stored in variable to the screen.
serout 1,N2400,(#b1) -   Sends a number stored in a variable to the screen.
serin 0,N2400,b1 -   Receives an ASCII character from a keypress on the
                     keyboard and stores it as the ASCII value in a variable (b1)
serin 0,N2400,#b1 - Receives a real number from the number keys on the
                     keyboard and stores it in a variable (b1)