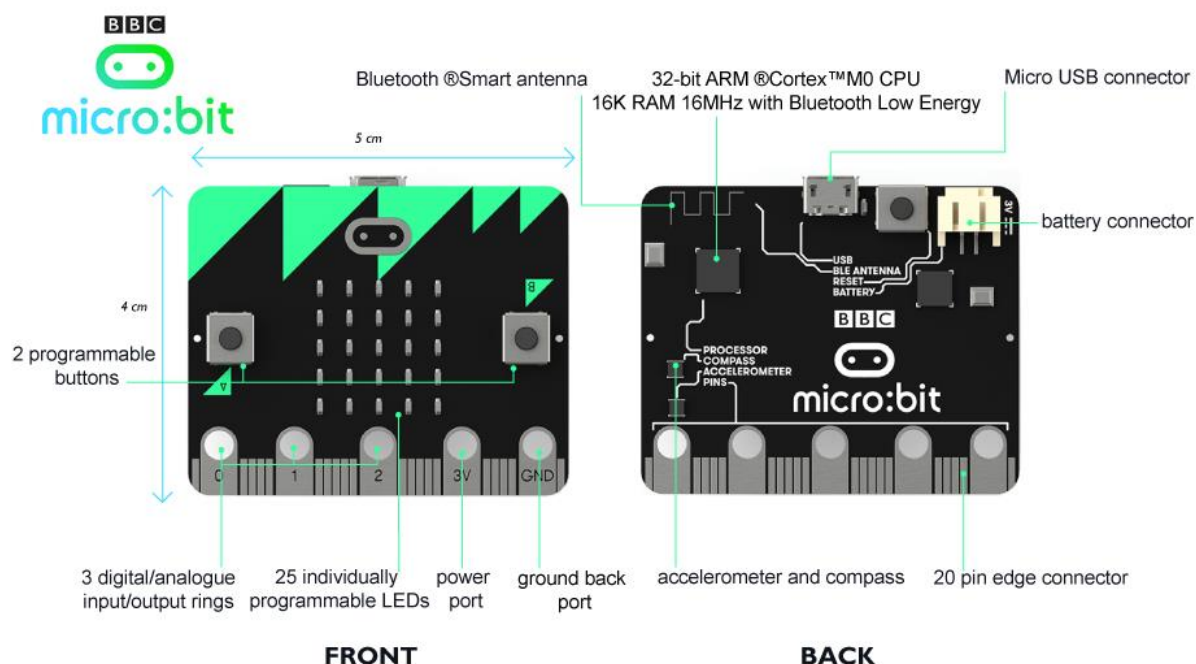


## First steps in using BBC micro:bit for control and physical computing

Adrian Oldknow [adrian@ccite.org](mailto:adrian@ccite.org) 26<sup>th</sup> January 2017

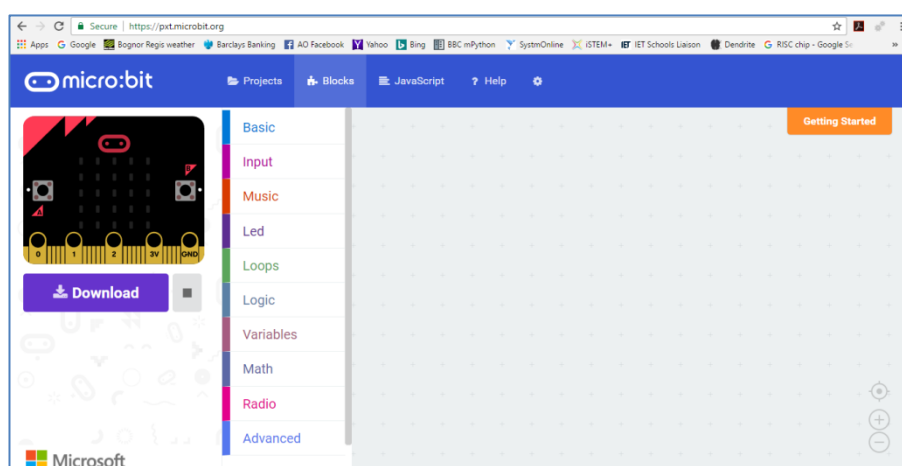
### 0. Writing your first program (please skip this section if you are already happy how to do it!)



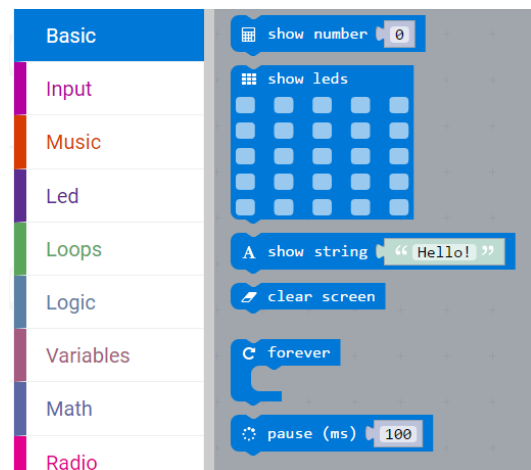
The image above shows the main elements of the BBC micro:bit. One million of these were distributed up to July 2016 to maintained schools in the UK to be given free to all 11-year old students. Since then they have been on general sale for around £15, including a battery case and USB connecting cable. One of the aims of the BBC digital literacy campaign has been to help people understand something now generally called the 'Internet of Things' (IoT for short). Many modern products use the adjective 'smart' to mean that they include a computing device to control it, and probably one which can exchange data with other devices wirelessly. Have a close look at the kinds of things this little device has built in to it. In order to make it work you need to connect a power source such as 2 AAA batteries. What is missing is an On/Off switch! As soon as the micro:bit has power it runs whatever program was loaded into it most recently. When you disconnect the power, the micro:bit continues to store the program – and will run it again as soon as you apply power.

So the first thing we need to find about is the simplest way to get a program to run on it. There are several ways to do this. We will start off with the programming environment developed by Microsoft Research, called the [Programming Experience Toolkit](https://pxt.microbit.org/), or PXT for short: <https://pxt.microbit.org/>.

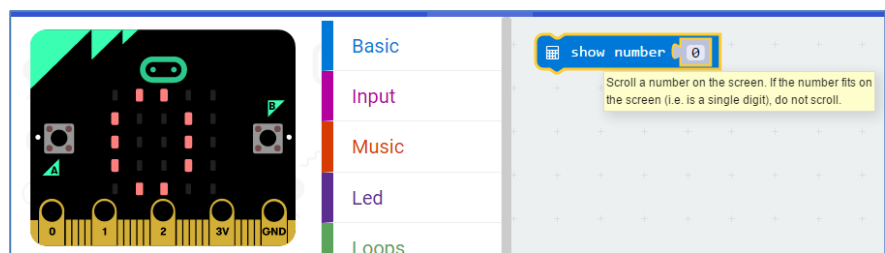
If this is the first time you have used the PXT editor, then you will have a blank program area. Otherwise it will open the last program you built. In which case click on 'Projects' and select 'New Project'. (Note: the PXT site often changes its layout!)



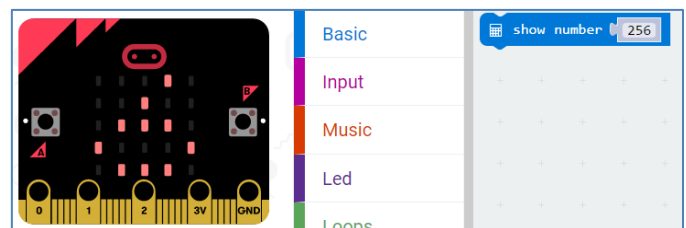
The image of the micro:bit at the top left is a very clever animated device, called an 'emulator', which allows you to test programs without even having your own micro:bit! The coloured words in the next column are links to the various building blocks from which you will build your program. If you click on one of these, such as '**Basic**' you will see a collection of blocks from which you select one and drag it to the programming window on the right. Let's try the '**show number**' block. When you click on it a yellow border shows up, which means you have started to run your simple program. You should see that the emulator shows the number you have chosen to display. There will also be some explanation about this block



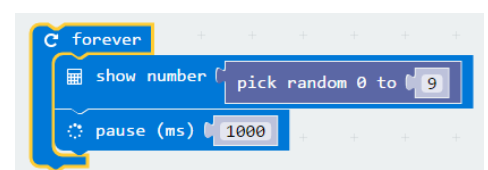
called '**showNumber**'. If you click on this you will open up a new tab with a page of information from the [reference manual](#), including an example. Try deleting the zero and entering your own number into the 'show number' block. I have entered '256', and you will see that the display scrolls to show each digit in turn, and the program ends with the '6' on the display.



See what happens if you enter something which is not a whole number like '3.14' or 'two'. It would be a good idea, now that you have started to build a program, to give it a name like '**First steps**'. Enter the new name in the box next to '**Download**'.



Most programs do something repeatedly, and use what is called a 'loop'. The simplest kind of loop is the '**forever**' loop.



Use the '**Basic**' menu and drag in a '**forever**' block and a '**pause**' block. Place the '**show number**' and '**pause**' blocks within the 'jaws' of the '**forever**' loop. From the '**Math**' menu drag in a '**pick random**' block and place it inside the '**show number**' block. Edit the number and change it from 4 to 9. Edit the number in the '**pause**' block and change it from 100 to 1000. So the new program will continually generate a random whole number (called an 'integer') between 0 and 9, display it and wait 1000 milliseconds (i.e. 1 second) before doing it all again. Check that this is what the emulator does.

Now explore what happens when you click on **'JavaScript'**.

This is exactly the same program but written in a text, rather than a graphical, format. It's not as pretty, but it is much more convenient to use for longer programs, especially if you want to print them out. Click on **'Blocks'** to swap back to the graphical version. Now we are ready

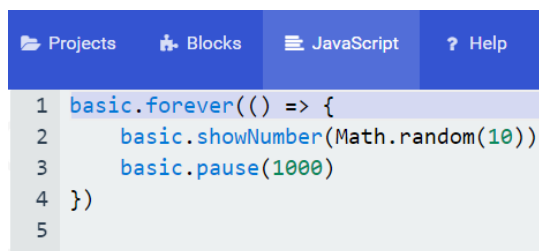
to send our working program to a micro:bit. This has two stages. The first is for the PXT editor to create a version of the program which the micro:bit can understand. This converts all the lines of the JavaScript program using a number code in what is called 'base 16' or 'hex' for short. The resulting hex file is stored on your computer, usually in a folder called something like 'Downloads'. The second stage is to connect a micro:bit to the computer with a USB cable and to transfer the file to it. Once you have got used to this you should find the whole business of writing a program on a computer, and loading it into the micro:bit, pretty painless! Click on **'Download'**. This opens a window at the bottom of the web-browser display which shows you that a file is being saved.

It is called 'microbit-First-steps.hex'. If you right-click on this name you can select the option to 'Show in Folder'. This opens your 'Downloads' folder. You should see your file at the top with the current date and time. My version has a size of 561 Kb. If you now plug your micro:bit into the USB port of your computer, after a few seconds another window will open showing it as an external storage device with a name something like 'MICROBIT (D:)'.

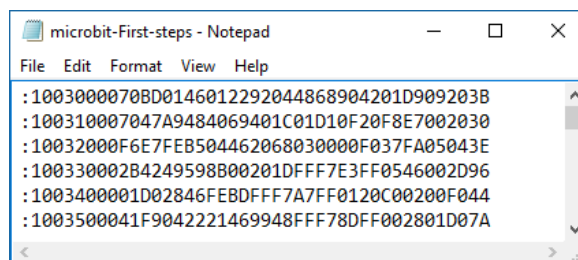
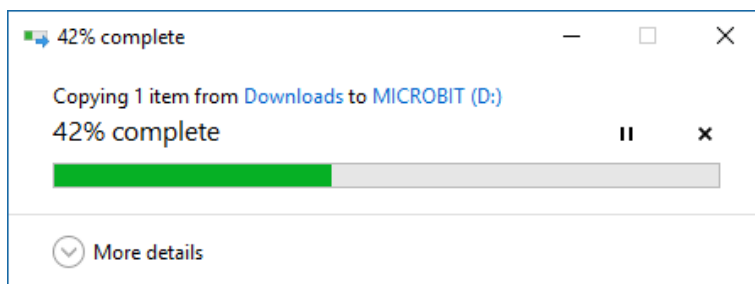
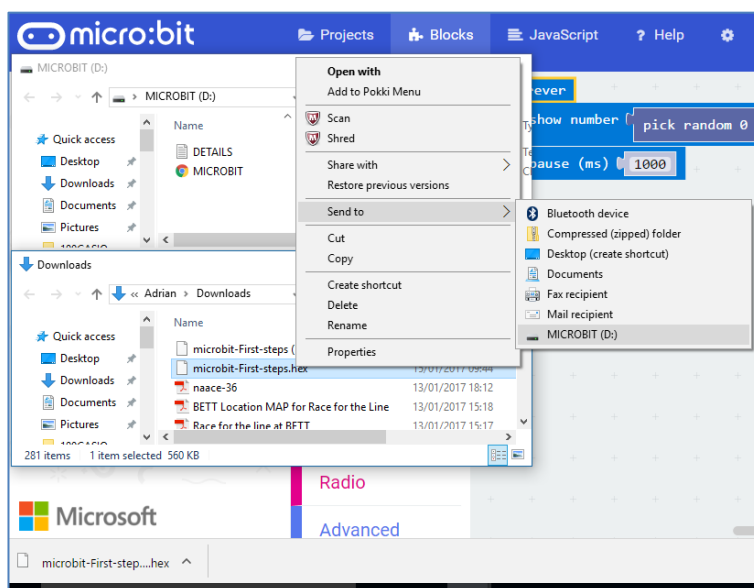
If you right-click on your hex file, you will have the option to 'Send to >' and a list of possible locations. Select 'MICROBIT (D:)' and a little dialogue box opens to show that the file is being transferred. You will also see an LED on the back of the micro:bit flash while this happens. As soon as the transfer is complete your program should start to run. At the moment the micro:bit is being powered through the USB cable from the computer. If you disconnect it, you can then attach a battery pack and the program will happily start to run – at least until you disconnect the power.

If you are curious you can open your hex file with something like 'Notepad' and see that it does indeed just contain a load of numbers using the characters

0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. Once you have created a hex file you can send it to your friends. They can either send it to their own micro:bits, or open it in the PXT editor. Try it now.



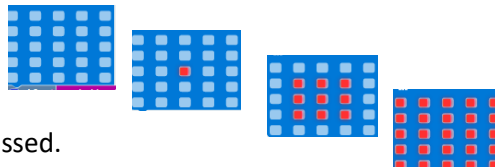
```
1 basic.forever(() => {
2   basic.showNumber(Math.random(10))
3   basic.pause(1000)
4 })
5
```



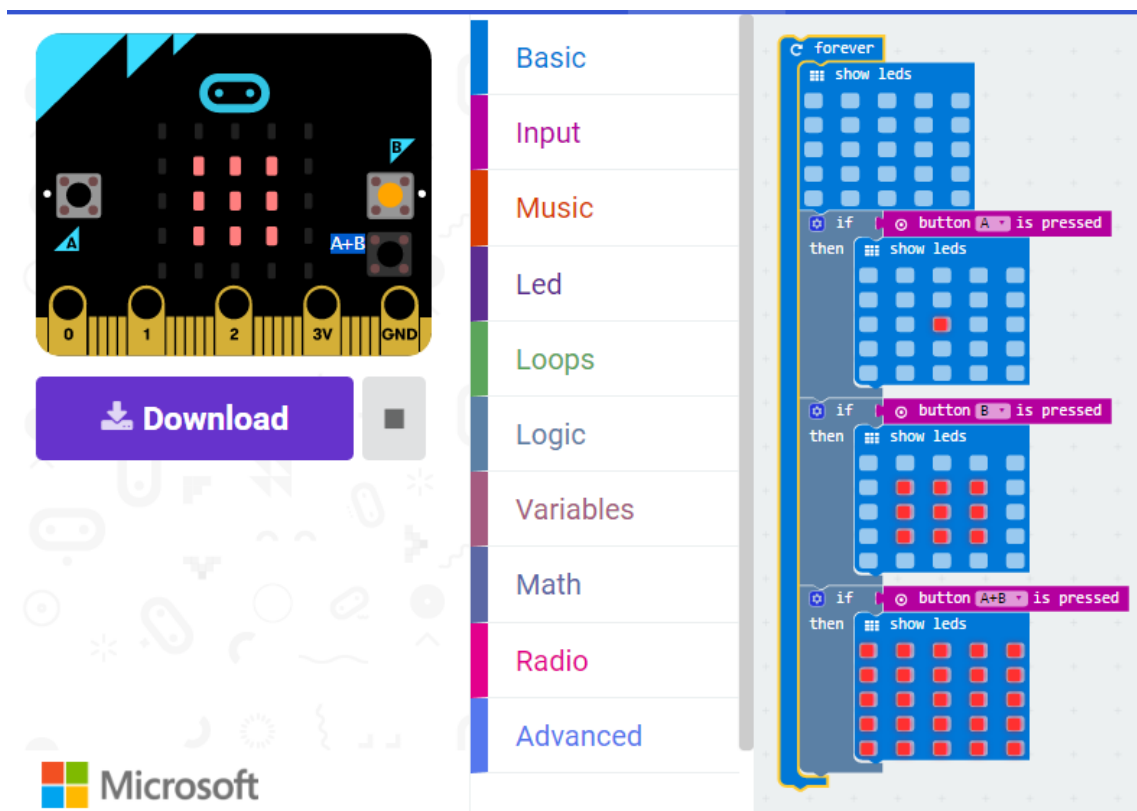
## 1. Making the micro:bit display respond to its buttons and sensors

The BBC micro:bit is quite a different animal from the Raspberry Pi. The RPi has to have input (keyboard and mouse) and output (TV or monitor) devices connected in order to be useable. The micro:bit has its own input (buttons and sensors) and output (the 25 LED display) devices built in. It can also have other ones attached, such as buzzers, speakers and electronic components. In this section you will learn how. We will start with the simple inputs – the A and B buttons. The micro:bit has a very simple way of accepting inputs from its buttons and using them to control the pattern of LEDs illuminated on the display. Start a New program. We will build a simple 4-state device which tells the micro:bit what to display

- (a) when no buttons are pressed:
- (b) when button A is pressed:
- (c) when button B is pressed:
- (d) when both buttons A and B are pressed.

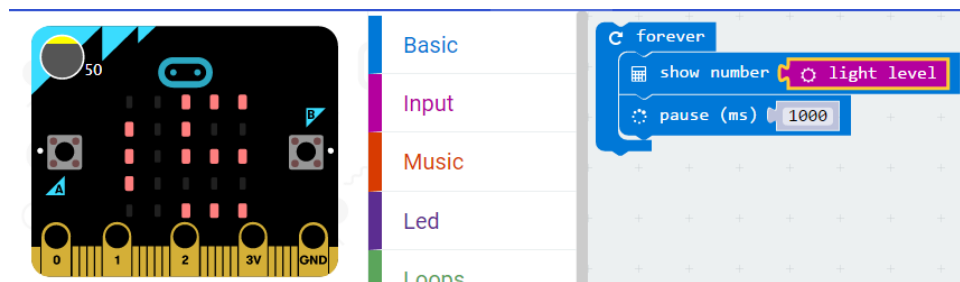


Here is the complete program:



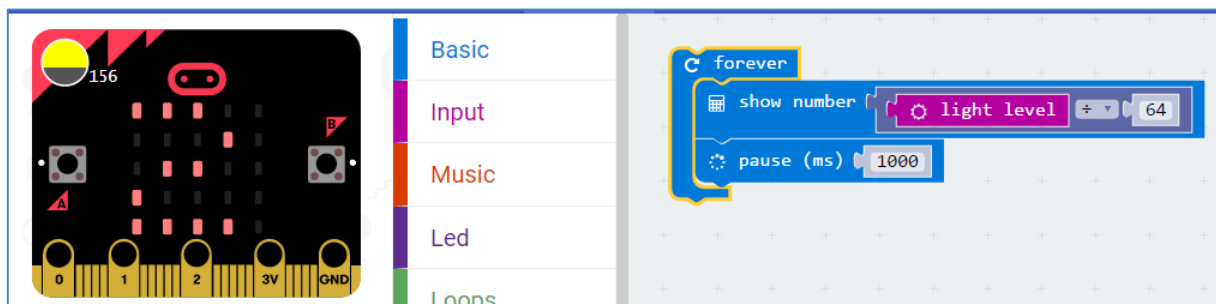
The `show leds` blocks are **Basic** commands. The `if` blocks are **Logic** commands. The `button` blocks are **Input** commands. By default, the led display will be blank unless either or both of the buttons are pressed. See what happens to the emulator when you run the program. Note that it creates an extra A+B button! When you are happy, give your program a name like 'Push buttons'. Download it as the hex file: 'microbit-Push-buttons.hex' and transfer it to your micro:bit. Test that the right things happen when you press any or all of the buttons. Now you have the basic idea we can see how the micro:bit can make decisions about what to display based on its own sensors. The first one we will try is the Light sensor. This isn't shown on the actual micro:bit, but it uses a clever way to detect light intensity falling on the 25 led display. (The technical details are [here](#).) Start a new program and build the following one. The **light level** block is in the

**`Input`** menu. This returns a value between 0 and 255. You can simulate this by using your mouse to pull the yellow `blind` up and down over the simulated sensor at the top left of the emulator. Because the result can have up to three digits, the display will scroll to show you the simulated reading.

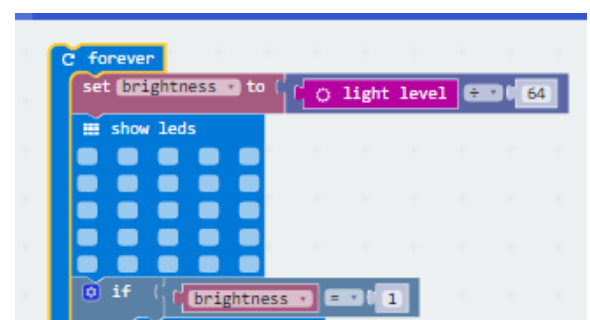
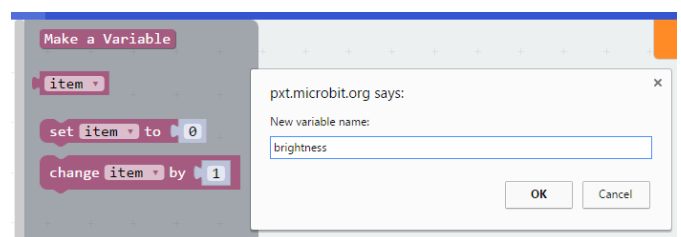


Give your program a title such as `Light level`. Download it as the `microbit-Light-level.hex` file and transfer it to your micro:bit. Try moving it closer and further from a light source, or shine a torch at the display.

We will now use a little trick to make the **`light level`** easier to work with. Find the **`divide`** block in the **`Math`** menu and insert it in the **`show number`** block. Edit the number to the right of the **`÷`** sign to read 64. If you use a calculator to do a divide operation you will usually get a decimal point in the answer. Try dividing 156 by 64 and see what you get. Division on the micro:bit works rather differently. You only get the whole number (aka `integer`) part of the result. Sometimes this operation is called `integer divide`. Test the resulting program with the emulator. Now we see that a light level of 156 produces the single digit `2` on the display. What are the only possible numbers this program can display?



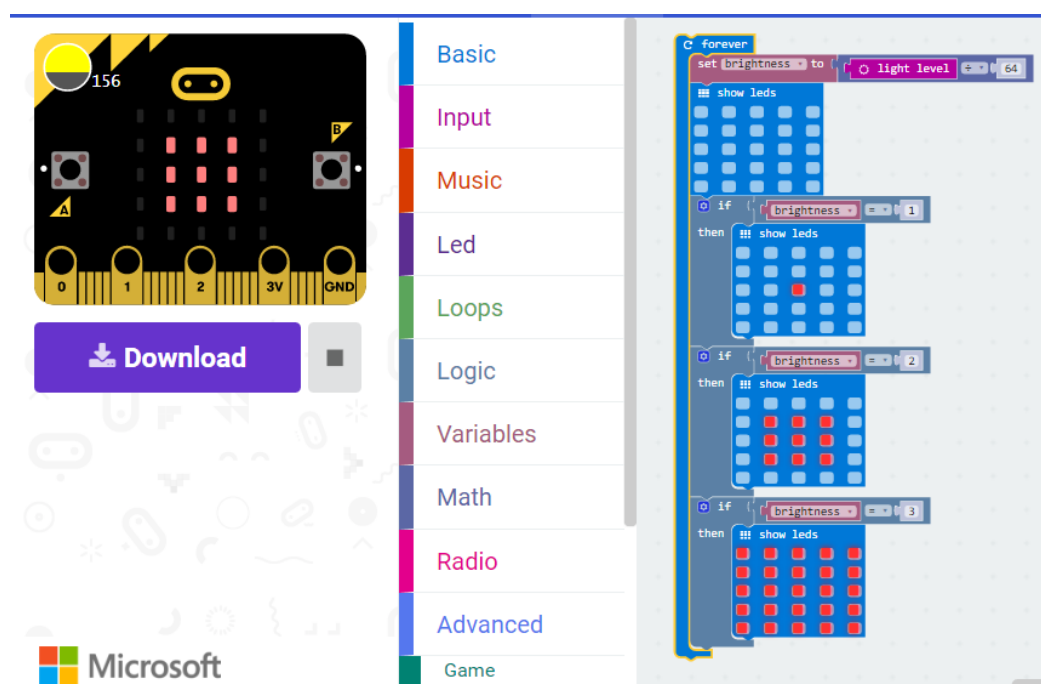
Don't bother saving, downloading and transferring (aka `flashing`) the current program to your micro:bit. We can now merge this idea with our 4-state program, replacing the **`button`** test with a **`light level`** test. We will now introduce the idea of a `variable`. In the **`Variables`** menu there is a block called **`Make a Variable`** which allows you to create a new one e.g. **`brightness`**. Click on **`Projects`** and re-open your `Push buttons` program (or create a new one if you can't find it). Insert a new **`set`** block from the **`Variables`** menu, select **`brightness`** instead of **`item`** as the variable name and insert the **`light level`** and **`divide`** blocks as shown. In each of the **`if`** blocks replace the **`button`** block with an **`equals`** block from the **`Logic`** menu. Insert the variable name **`brightness`** and edit the test value to 1, 2 and 3 in turn.



The completed program is shown here. Check it works using the emulator to simulate changes in light intensity.

**But the logic is all wrong!!!** We need more leds when it gets darker, not lighter!

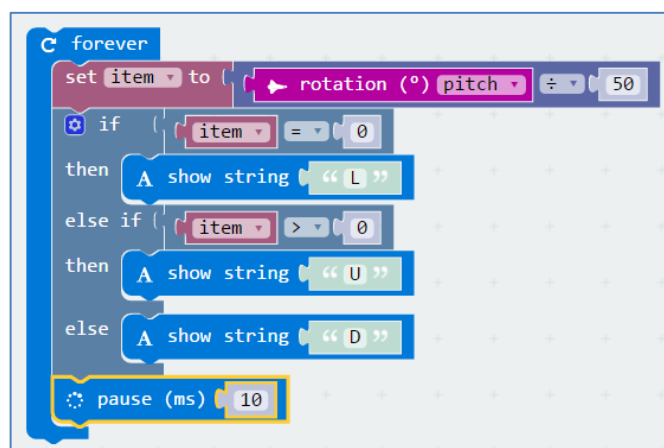
So can you edit the program to fix it? Now you have not only created a program, but also detected and corrected an error.



That process is called 'debugging'. Save your program and transfer it to your micro:bit to check it now works properly. Move the micro:bit so that the light-levels change and check it works OK.

**Congratulations.** You have built your first Internet of Things device, using the light sensor to switch on and off the leds in the display. That's how most current cars have automatic systems to switch their lights on and off as the light level changes. You could attach your micro:bit to the back of your bike's saddle to create a smart rear lamp. I have saved my version of the corrected program on Dropbox with this [link](#). Check you can copy this program to your computer, open it with the PXT editor and transfer it to your micro:bit.

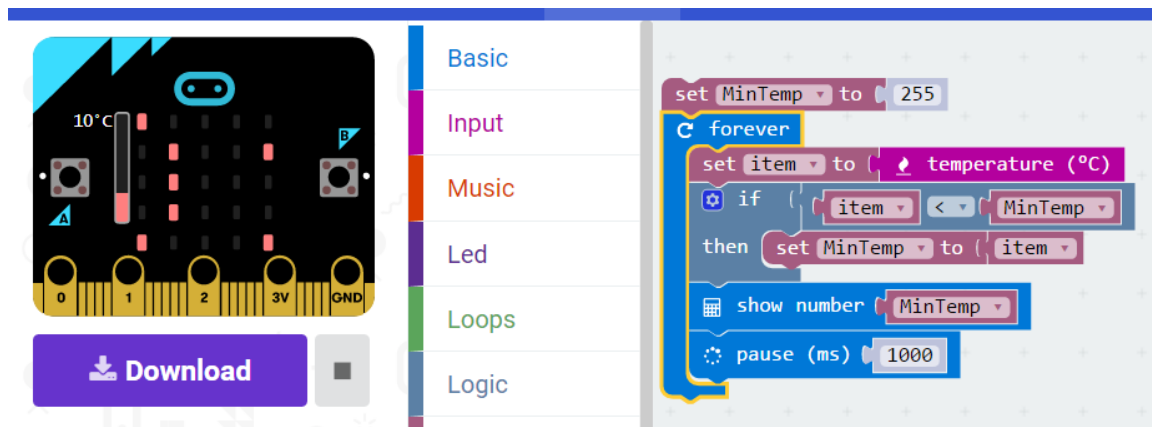
The micro:bit has several sensors other than temperature or light which can be used to detect motion. These include accelerometers and magnetometers which can be used to detect gestures, collisions and direction. The '*rotation*' block from the '*Input More*' menu can measure the '*pitch*' (forward and back) and '*roll*' (side to side) motion of the micro:bit. This program turns the micro:bit into a spirit-level.



So how does the program work? It continuously checks the angle being returned by the '*pitch*' sensor. Dividing this angle by 50 returns a value for the variable '*item*' which you could read using the '*show number*' command from the Basic block. This is a single digit signed number between -4 and +4. We are just going to test whether this is a positive, negative or zero number. If it's zero we display the letter 'L' for Level, if it's negative we display 'D' for Down, and if it's positive we display 'U' for Up. Then we have a slight pause before doing the job again. You can test this with on screen emulator by clicking the mouse somewhere inside the image of the micro:bit at the top left. Check that you can get it to display each one of the 3 letters. So this is another way to use the micro:bit's built-in sensors to control an output. It simulates the way a smart-phone or tablet senses the orientation of the way in which you are holding the device – and



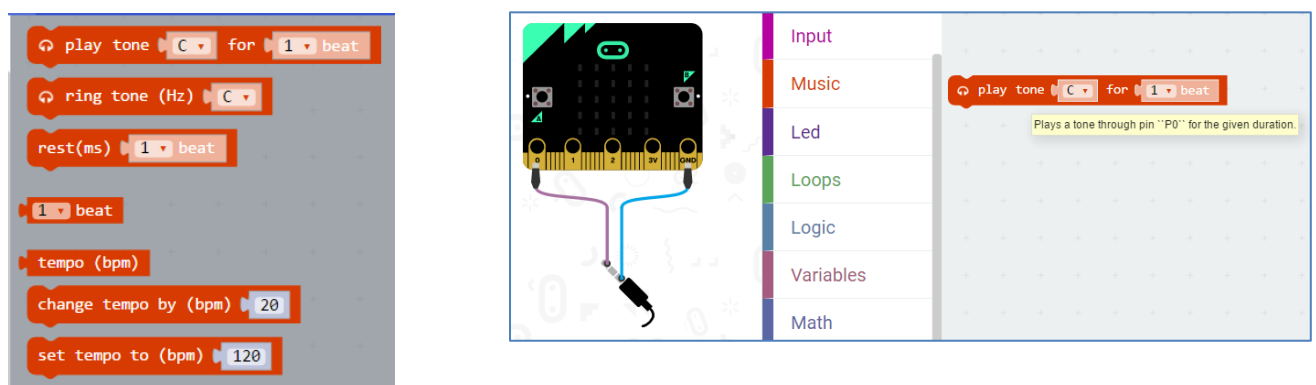
so is able always to display text in the right direction for you to read clearly. The last example in this section turns the micro:bit into another kind of instrument – a thermometer with a difference. We will use the variable **MinTemp** to record the lowest temperature reached while the micro:bit is awake and sending. This starts with a large number stored in it. Every time the temperature sensor records a value lower than the lowest recorded so far, we replace the value stored in the variable **MinTemp** with the current one, saved in **item**. You can test the program with the emulator by sliding the simulated thermometer. Once you have transferred the program to the micro:bit, you can detach it, attach batteries and place the micro:bit in the fridge. When I take mine out after half an hour I find the lowest value was 6°C. So my nice bottle of white wine is probably just a bit too cold!



Can you edit the program to record the maximum temperature reached? I suspect it will not do a micro:bit much good by testing this in a kitchen oven! Can you develop a program which displays the current temperature while also storing the maximum and minimum temperatures reached? Make it display the maximum temperature when the A button is pressed and the minimum when the B button is pressed. Place your micro:bit in a plastic bag and leave it outside for 24 hours to check the max and min temperatures.

## 2. Connecting external devices

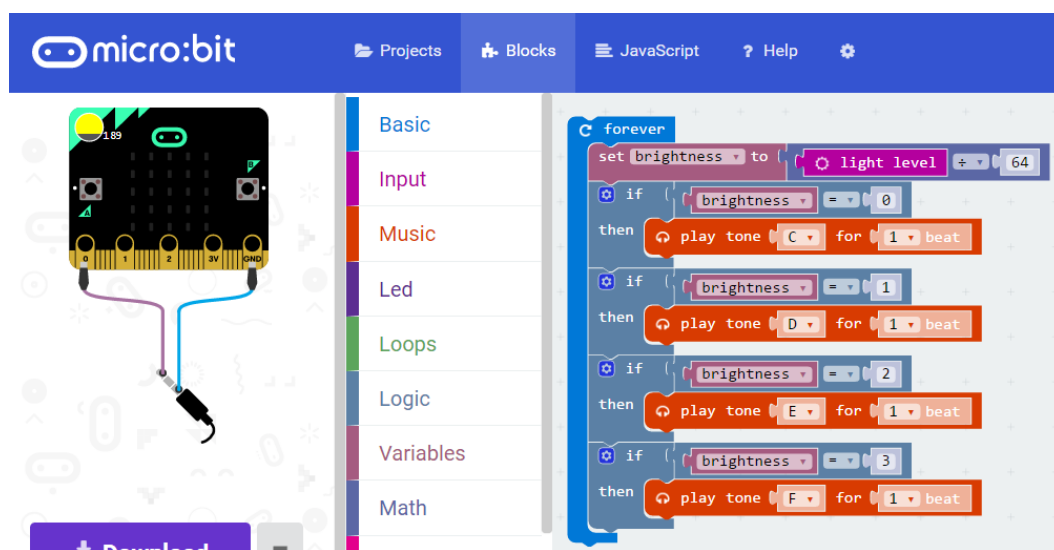
One of the micro:bit menus is called **Music**, but there is no speaker on the micro:bit itself.



When you try to run the simple program to play a single note, the emulator suggests that you need to connect 'pin0' and 'GND' to a headphone or speaker. In the photo below, the bottom left shows a simple black circular buzzer attached to the micro:bit's Pin0 with a green cable and crocodile clips, and to its GND pin with a black cable. You could use the crocodile clips to attach to the separate sections of the jack plug of headphones or speakers.

For £2.25 you can buy a ready-made [jack plug adaptor](#) from 'Handy Little Modules' or for £5 you can buy the Kitronik [M1 power adaptor](#) (bottom right) which has a built in buzzer connected to the right pins.

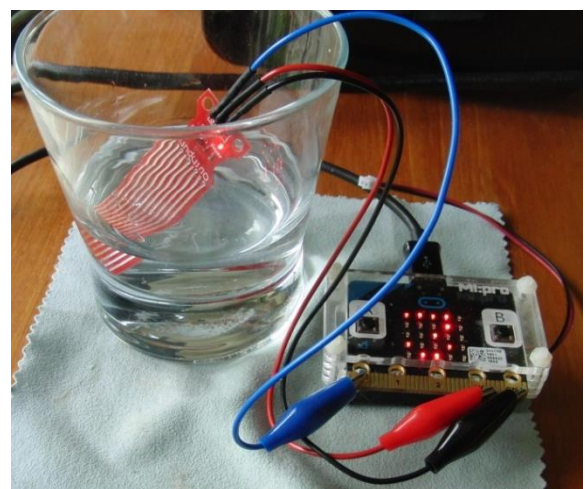
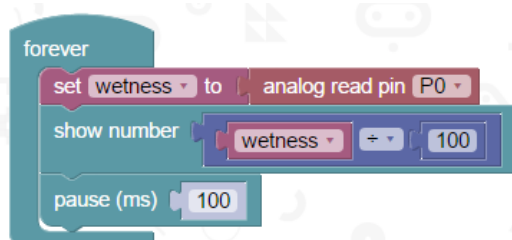
So now you can build a range of alarms to display warning signs on the led array as well, as making noises.



Could you use the temperature sensor to sound an alarm and flash a light if the temperature in the room gets too high, or too cold? This is, of course, the principle used by a thermostat to control the central heating in a home, or the climate control in many modern cars.

As well as external outputs, such as a buzzer, or bright/coloured LEDs, you can also attach external inputs such as sensors. A very nice project to build an automated plant watering system is described [here](#). This uses an external moisture sensor to detect the dampness of the soil in a plant pot. It could be used to sound an alarm so that you are prompted to water the plant. But, better still, the micro:bit could turn on a pump to water the plant automatically. The [water sensor](#) costs £3 and the [pump](#) costs £5. More tutorials and suggestions are on [this site](#).

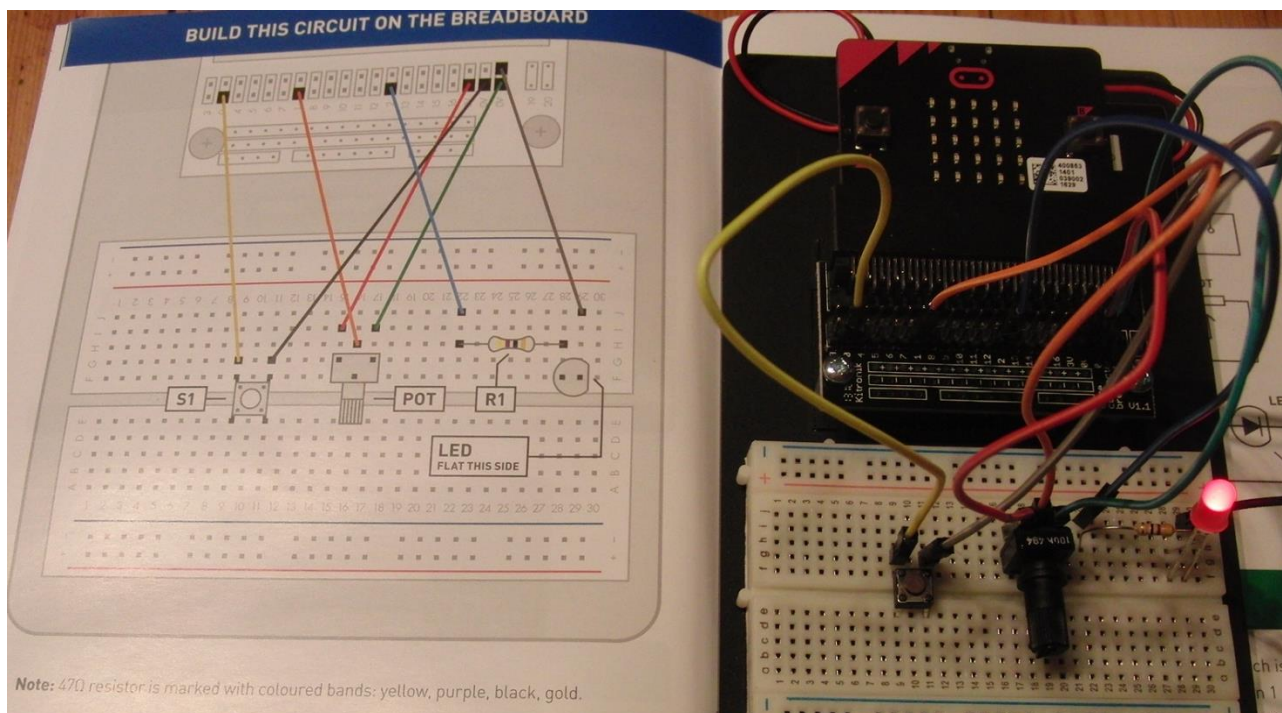
Let's test the water sensor. The red lead is connected to 3V, the black to GND and the blue to pin P0. Here is the simple code to check it works:





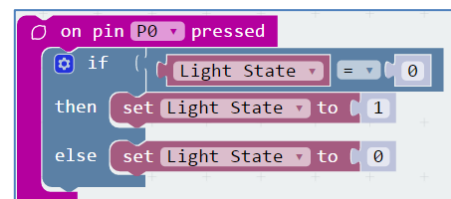
### 3. Designing a working system – smart lighting

In this section, we will simulate working like an electronic systems designer. We will just use the micro:bit as the smart control device. We will design a system in which a dimmable light emitting diode (LED) responds to the amount of light falling on another electronic component called a light dependent resistor (LDR). This simulates the automatic lighting system in a car or house. These components are widely available and quite cheap. A very convenient resource for this kind of activity is ready made kit, such at the Kitronik's ['Inventor's Kit for the BBC micro:bit'](#) costing £25. This consists of an 'edge connector' into which you plug your micro:bit. This is connected to two parallel rows of pins which you use to attach wires. This is stuck on a base board along with an object called a 'bread-board' with many holes in to allow you to place components, like an LED, and connecting wires. The photograph below shows page 24 of the tutorial with the circuit diagram we need to build. On the right is the assembled Inventor's kit with the programmed micro:bit plugged into the edge connector, with the battery boxed tucked away underneath. The coloured leads slip over the I/O pins connected to the micro:bit through the edge-connector and plug into holes in the bread board. We are using three I/O pins, 0, 1 and 2. Also the GND and +3V power pins. The first test components are a push switch, a potentiometer (variable resistor), a red LED and a 470Ω ohm resistor.

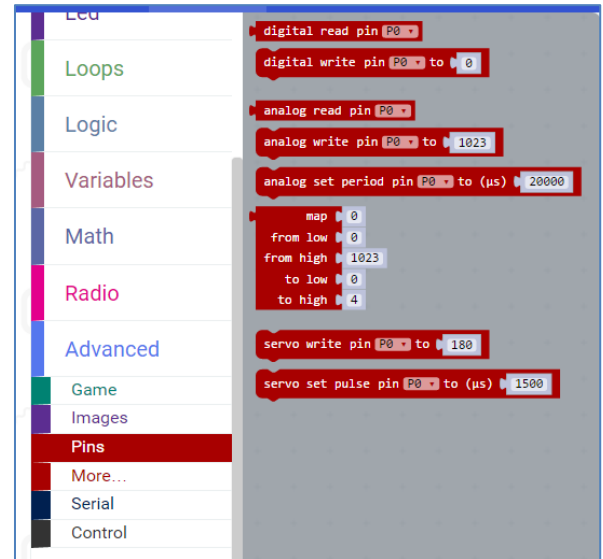
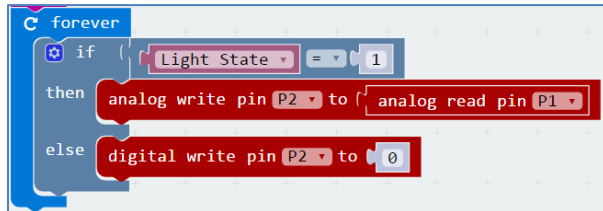


In this first version we will use the potentiometer as a manual dimmer-switch, like we used the A and B buttons in earliest example.

Here is the program written in the current Microsoft PXT editor. I have called it 'Dimmer switch'. We use a variable called '**light state**' to tell whether the LED is switched on (1) or off (0). So the first bit of code just tells the micro:bit to use the push switch attached to pin P0 as a 'flip-flop' to change the state of the LED.

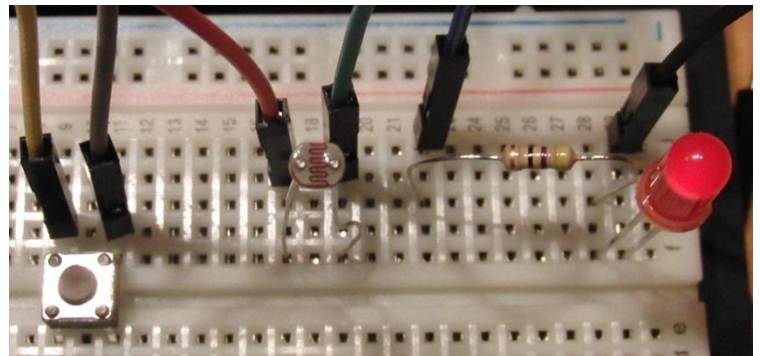


Now is the time to have a look at the **'Advanced'** blocks menu. The one that we need is called **'Pins'**. This allows you to read values from digital (e.g. a switch) or analog (e.g. a potentiometer) inputs and to send out signals to digital and analog devices.

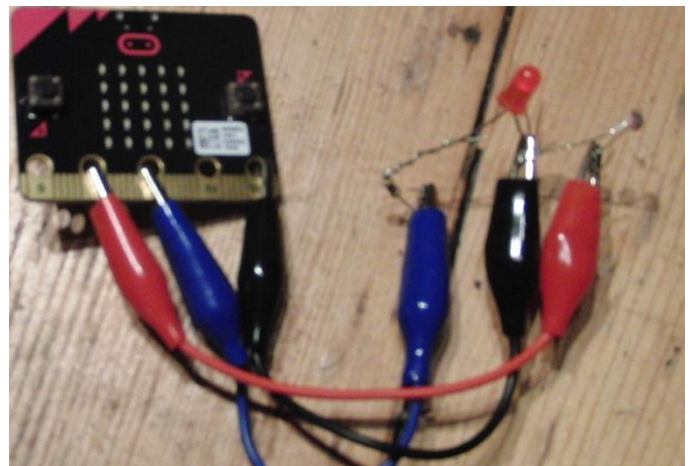


The second block of the program uses the potentiometer attached to the analog pin P1 to control the brightness of the LED connected to the analog pin P2. When you have transferred the program to the micro:bit you can use the push button to switch the LED on and off. You can turn the spindle attached to the potentiometer to control the brightness of the LED.

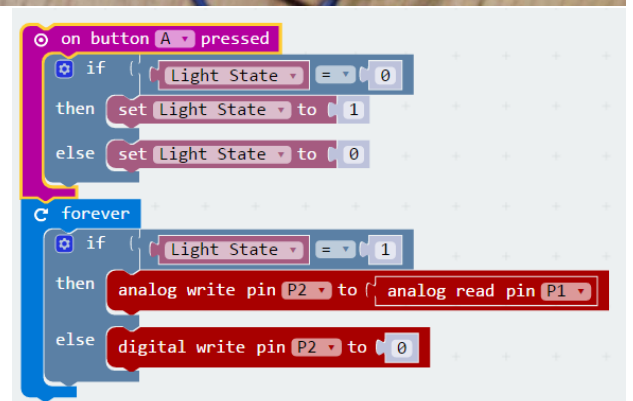
Once your program works OK, you can replace the potentiometer with a light dependent resistor LDR. The program doesn't need any changes. But we no longer need the lead to connect the 3V to the potentiometer. The LDR is just connected to GND and P1.



Now we have developed our working system we can dispense with the bread-board and make a bespoke circuit. I have used 3 crocodile clip leads. The black lead connects GND to one leg of the LDR and to the negative leg of the LED. One leg of the resistor is twisted round the positive leg of the LED. The other leg of the resistor is attached with the blue lead to Pin2. The red lead connects the other leg of the LDR to Pin1. There is no point in adding a push switch to the system as we can use button A, say, instead.



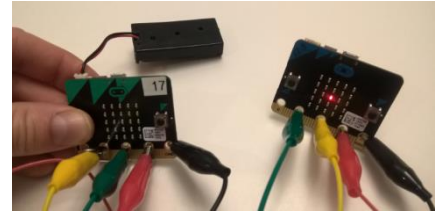
So we just need a very small modification to the program. Of course, if you wanted to market your device you would probably like to put all the components, including the micro:bit and battery, in a nicely designed box for which you could charge a substantial amount!



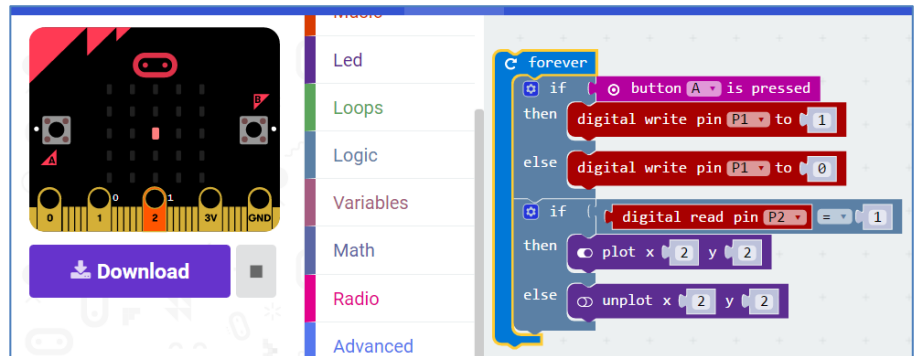
The 10 well described experiments in the Inventor's kit give a pretty good feel for how to design and control your own circuits and devices.

#### 4. Communicating with other devices, including getting micro:bits talking to each other.

There is now an increasing number of other fun projects for use with micro:bits on the Internet. Many of these also have tutorials which you might find helpful. There are lots of ideas on the [BBC site](#), such as connecting [two micro:bits](#) together with crocodile clips. Four crocodile clipped leads are used to connect two micro:bits together.

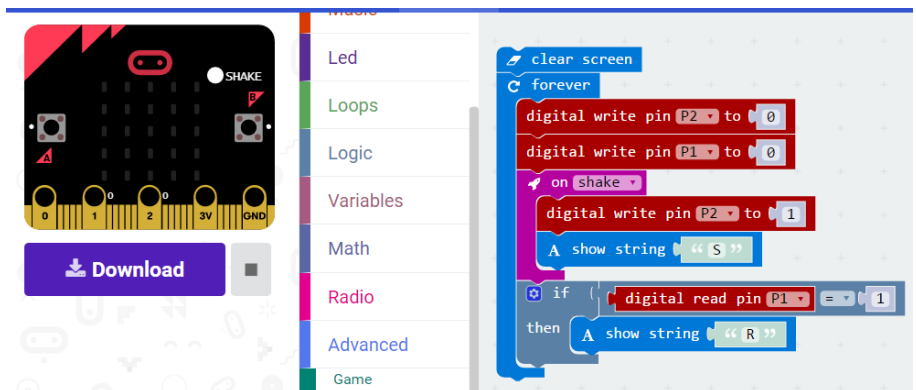


The black and red leads connect the 3V and GND ports together so that one of the units is powered from the other – so we only need one battery pack. The green lead connects P1 on one board to P2 on the other. The yellow lead does the reverse. The same simple



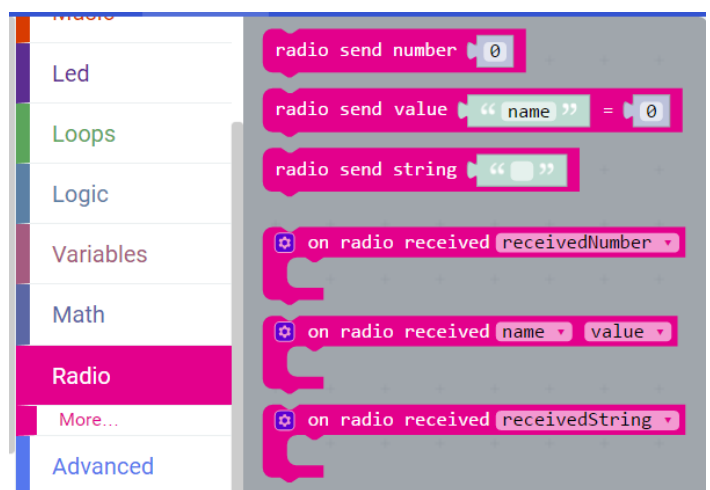
program is used on both units. It just outputs a signal to Pin1 if button A is pressed. If it senses an incoming signal on Pin2 it turns on the central LED – and switches it off if there is no incoming signal. The new commands here are the `'plot'` and `'unplot'` blocks from the `'Led'` menu. These enable you to turn on or off individual 'pixels' on the 5x5 Led display. Could you build a 2-player game around this technique?

Another application would be to send data from a sensor of one micro:bit to the other. Here is a simulation of an intruder alarm. One of the micro:bits simulates the alarm box – with batteries, the other a simulated alarm sensor as might be attached to an entry door. The sensor we are using



for the input is a combination of the accelerometers which detects whether the device has been disturbed by a `'shake'`. Transfer the same program to each micro:bit. If you shake one of the micro:bits it displays the letter "S" to show that it has been shaken, and also that it is sending an alarm message. The other micro:bit displays the letter "R" to show it has received an alarm call. Press the reset buttons on both devices to reset the alarm system. Can you think of other uses for this kind of layout?

Another interesting menu says `'Radio'`. More information about the 2.4GHz radio module is



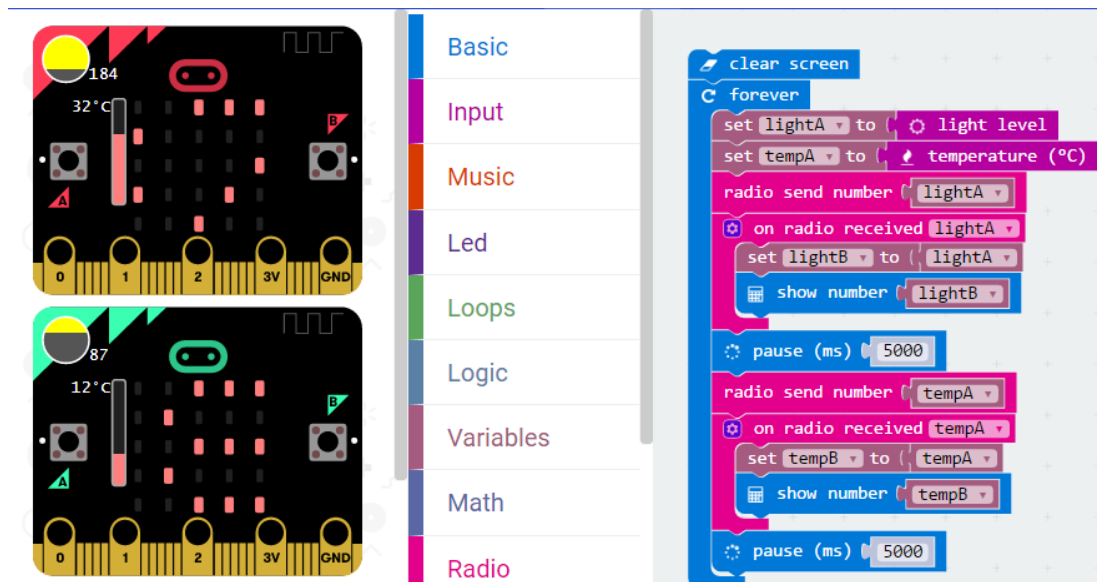
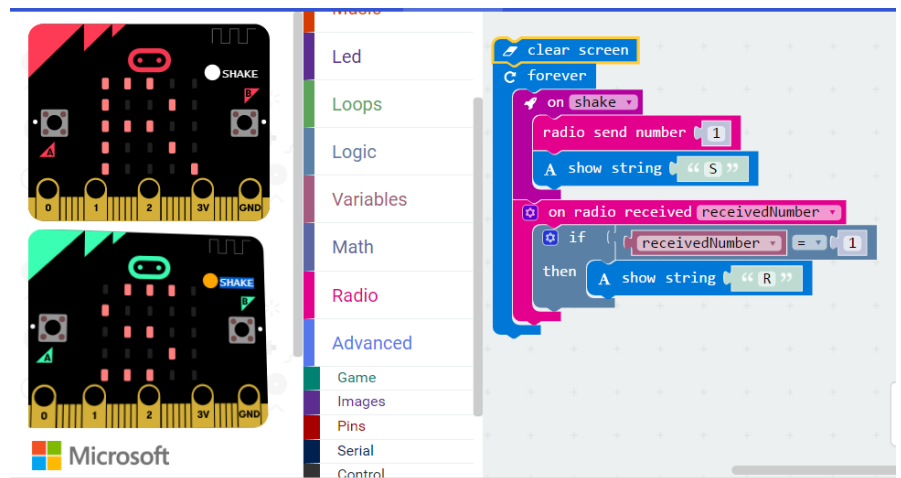
[here](#). This is a simple way to connect micro:bits. Here are some of the blocks which the **'Radio'** menu contains. Use **'More'** to see the others.

It looks as if we could disconnect the green and yellow leads from pins 1 and 2, and send information wirelessly. I have never tried using this menu before!

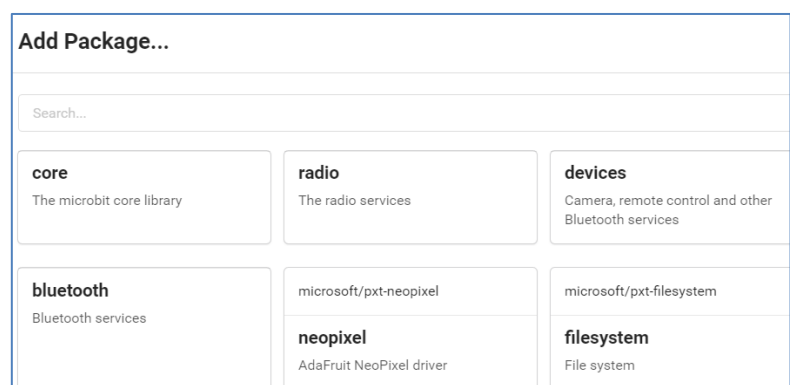
Wonder of wonders – using the **'Radio'** menu opens up a second emulated micro:bit. All we have to do to use **'radio send number'** on one device and **'on radio received'** on the other. We can use the same program on both devices.

Disconnect the red and black leads and connect a battery box to each device. Now you can use either device to check for a shake, and send a message to the other. We can use the technique for telemetry – by reading a value from one of the transmitting micro:bit's sensors and sending it to the receiving micro:bit wirelessly.

Now you could send, say, the temperature read by micro:bit A inside a fridge, and monitor it on micro:bit B outside it. You could also use the same idea to check that the light really does go out when the fridge door is shut. Test how far apart the units can be before the radio connection stops working. On the emulator you can adjust the light and temperature levels on both A and B to check the program.



In the next of these articles we will see how we can also use Bluetooth to send data from a micro:bit to a computer wirelessly. In the blue band to right of **'Help?'** you will find the tool menu.





Select **'Add package'**. You will see there is a **'bluetooth'** option.

There are also other interesting ones, such as **'devices'**. I am going to explore the **'neopixel'** one. There is a range of multi-coloured leds produce by Adafruit. I bought a small Neopixel Stick of 8 leds from [Cool Components](#) for £5.40. If we import the **'neopixel'** package we can look at the kinds of blocks now available.



Further block are available from the **'More'** menu. The vital block is the first one, which imports a library of extra commands for the micro:bit. I couldn't see any easy way to use these blocks. Google found me this [link](#) to an excellent source of micro:bit information.

**MultiWingSpan**

Home Programming Web Design Computer Science Twisting Puzzles Arduino BBC micro:bit

### BBC micro:bit Lighting The Way (PXT)

**Introduction**

'Neopixels' is a term used by the company Adafruit to describe chainable RGB LEDs that come with a constant current driver. You can control a decent number of neopixels using a single GPIO pin and they tend to be wonderfully bright. Since we are running these from battery power, rather than power from the micro:bit, we can afford to run them a little brighter than we might normally do.

There are 12 neopixels on the Bit:Bot, 6 of them on each side. They are helpfully numbered on the PCB from 0 to 11, no excuses for not lighting the correct ones.

The neopixels are connected to pin 13.

**Programming**

In order to use the Neopixel blocks, you need to add the library. You do this by clicking the settings link (cog button at the right of the menu). Choose **import package** and add the Neopixel library.

The following blocks are used to light all of the Neopixels in a single colour.

```
set robotpixels to NeoPixel at pin P13 with 12 leds as RGB (GRB format)
forever
  robotpixels show color red
  robotpixels show
  pause (ms) 1000
  robotpixels show color green
```

**BBC Microbit**

- ★ Programming S
- ★ BBC micro:bit H
- Block Editor - T
- ★ LED Matrix
- ★ Buttons A & B
- ★ Counting
- ★ Menu System
- ★ Touchy Touchy
- ★ Tilting
- ★ Shaking All Ove
- ★ Rock, Paper, Sc
- ★ Two Dice
- ★ Gestures
- ★ The Matrix Revi
- ★ Charlie's Game
- ★ Compass
- ★ Temperature
- Block Editor - C
- ★ Getting Started
- ★ Make Some Noi
- ★ Headphones Ha
- ★ Connecting mic
- ★ More LEDs
- ★ Knock Sensor
- ★ Bi-Colour & Tri
- ★ Using A NOT Ga
- ★ Adding More Bu
- ★ RGB LED
- ★ Rotation Mate

```
NeoPixel at pin P0 with 24 leds as RGB (GRB format)
item range from 0 with 4 leds
item show rainbow from 1 to 360
item show color red
item show bar graph of 0 up to 255
item show
item clear
item shift pixels by 1
item rotate pixels by 1
```

We need to create a variable such as **'array'** to refer to the neopixel strip. We tell it which pin will be used to control it – called DIN for 'digital input'. Also the number of leds it holds. Commands like **'array show color red'** applies to all the leds, but changes only happen when you use an **'array show'** command.

Note how the emulator shows up how to connect the leads. You can produce all sorts of fancy light shows. If you look at the **'More'** menu you will see how to address any individual pixel in the array.

**Basic**

**Input**

**Music**

**Led**

**Loops**

**Logic**

**Variables**

**Math**

**Radio**

**Neopixel**

```
set array to NeoPixel at pin P0 with 8 leds as RGB (GRB format)
forever
  array show color red
  array show
  pause (ms) 1000
  array show color green
  array show
  pause (ms) 1000
  array show color blue
  array show
  pause (ms) 1000
  array clear
  array show
  pause (ms) 1000
```

assign the value of a variable  
Sets this variable to be equal to the

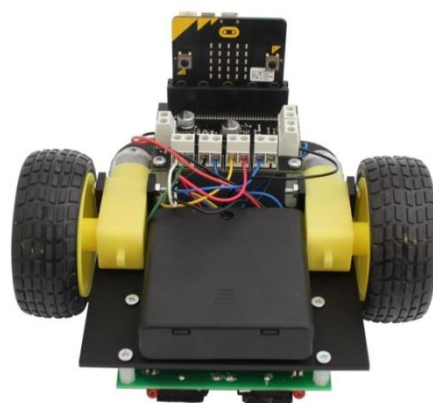
There is a wealth of relatively cheap existing components we can use with in the PXT programming environment. Many more if you can master MicroPython. In the next article we look at some of the options we have for attaching other sensors – and for using Bluetooth for transmitting them.



There are also lots of interesting accessories on the market as well. We have seen a couple of these already.

Kitronik's '[Line-following buggy](#)' at £26

[DIMM](#) and [UFO](#) from Binary Bots (micro:bit included, £40 each)



4Tronik have developed their own bit:bot buggy which includes a strip of 12 RGB leds, as well as motors, light sensors and a buzzer. It costs £30 and there are instructions for it [here](#). For an extra £3 you can add an [ultrasound distance sensor](#).

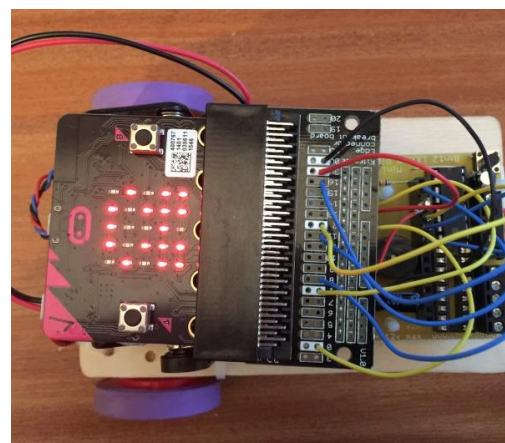
Cheap sensors and accessories are available from '[microbit accessories](#)', such as [head-phone adaptors](#), the natty [plant watering project](#) and the [chick-bot robot](#). Using the head-phone adaptor you can also attach speakers to the micro:bit.

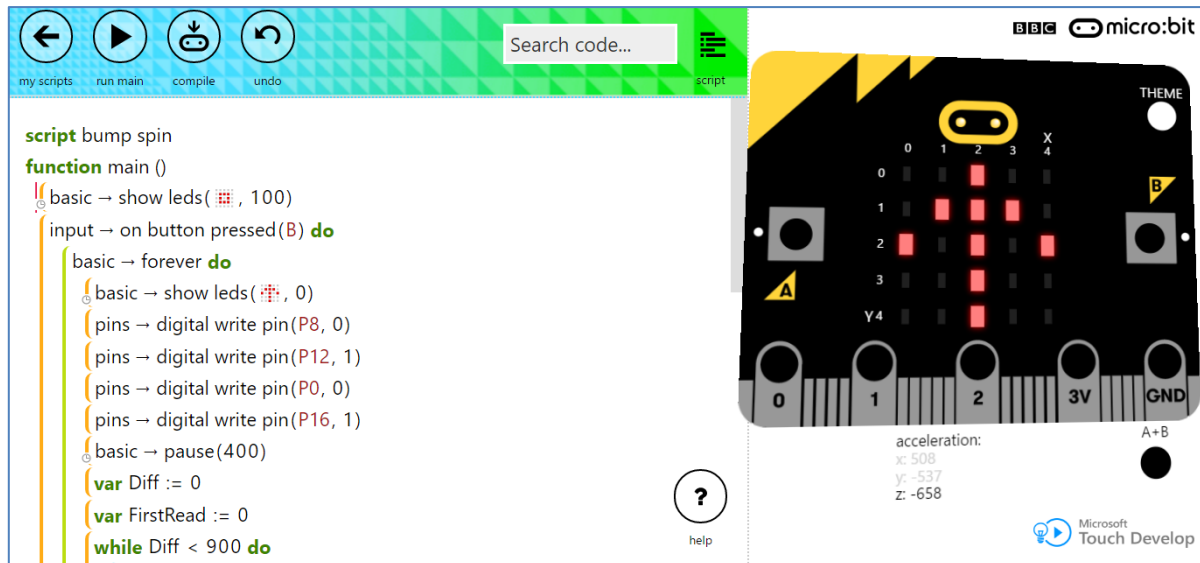
The Kitronik [MI:power board](#) costs £5 and provides a robust casing for the micro:bit as well as compact power from a coin battery.

There are also a large number of sites with excellent help and ideas for projects and techniques – many with helpful instructional videos.

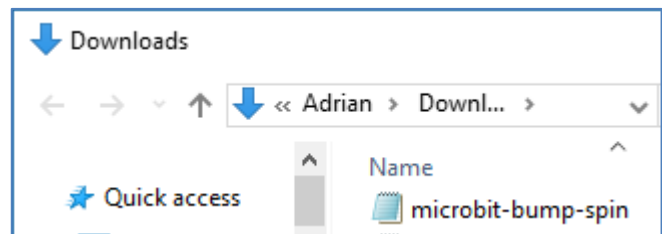
## 5. '[Borrowing code](#)' from others – a bump-and-spin program for a robot car.

You can easily build your own robot car, or buy a ready to build kit like the Kitronik or Chick Bot one shown here. You will need a couple of motors and a cheap motor driver board such as [this](#). A common sort of program to control such a device is to sense when the car bumps into an object, to reverse away from it and to turn in a different direction before driving forward again. We don't need a special sensor to detect the collision. We can use the micro:bit's built in accelerometers instead. Looking at the information on the Kitronik site I found that they have an [example](#) of this kind of program we can learn from. Clicking on that link opens the program in an earlier version of the block editor.

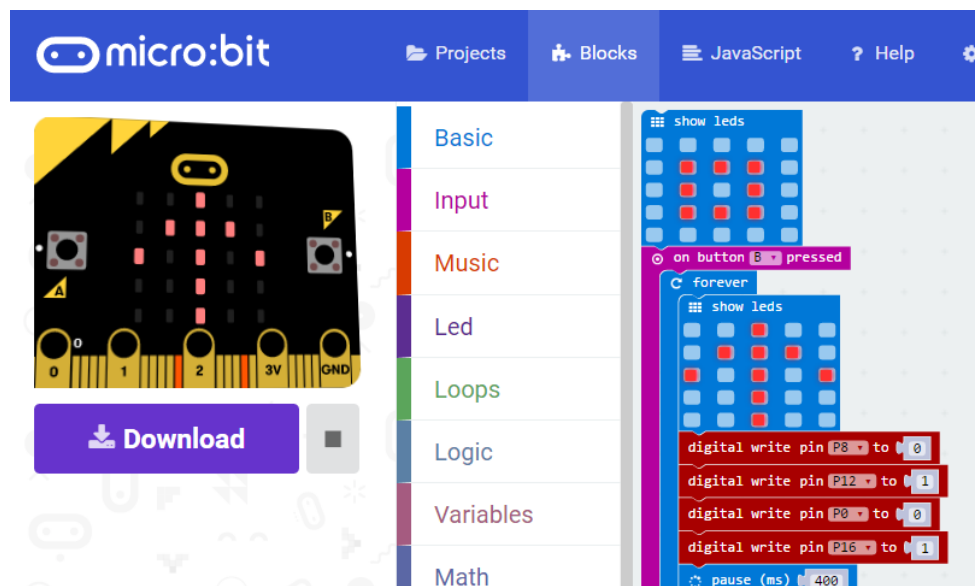




The program is shown in JavaScript. You can use the emulator to simulate changes in acceleration. If you click on **Compile** it will be saved in your 'Downloads' folder as the file 'microbit-bump-spin.hex'. Now open the PXT editor. Click on **Projects** and select **Import from file**. Browse to your Downloads folder and select the appropriate hex file. This will be opened in JavaScript, but you can then click on **Blocks** to read the code.

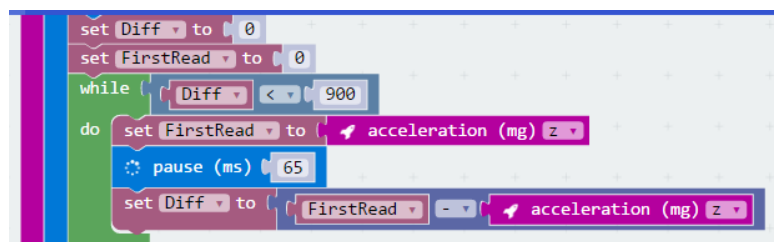


The units of acceleration in 'milli-g', or one thousandth of the acceleration due to gravity. If the micro:bit is mounted vertically its z-acceleration will be measured in the direction of travel. Since force is the product of mass and acceleration, this will indicate the force applied in that direction. Bumping into an object causes a force, which the micro:bit detects and triggers evasive action.

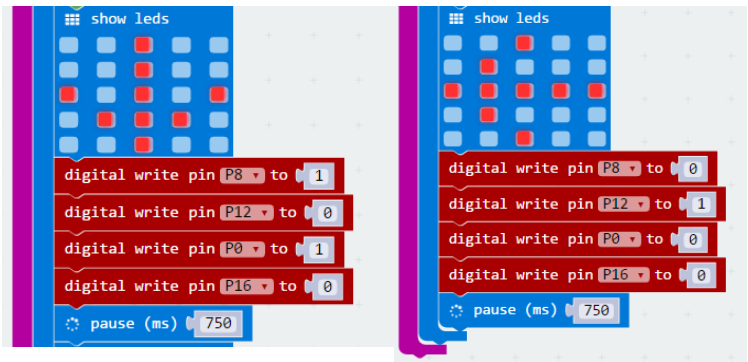


The key technique is in these few lines.

The buggy keeps driving forward until two successive readings from the z-accelerometer differ by more than a certain threshold value.



The rest of the program manages the display of an arrow to show change in direction, and send data to the motors to control their speed and direction.

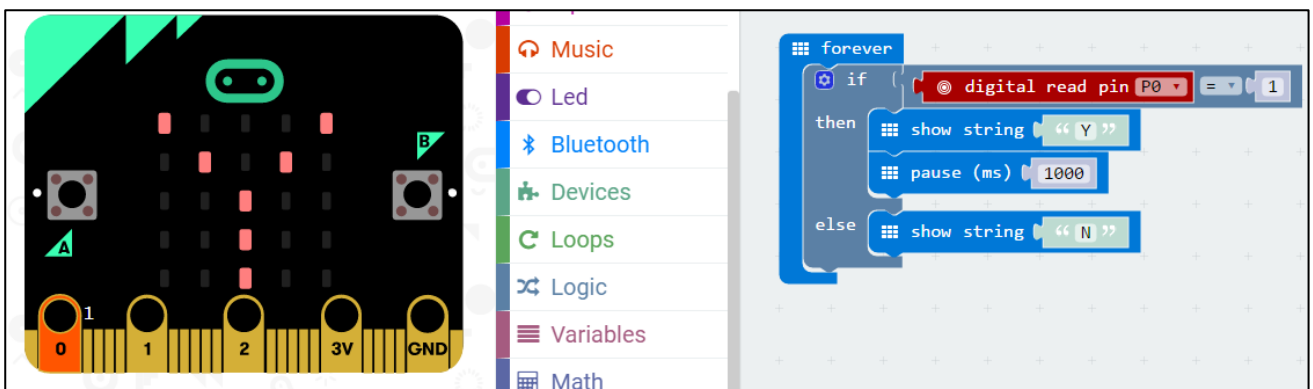
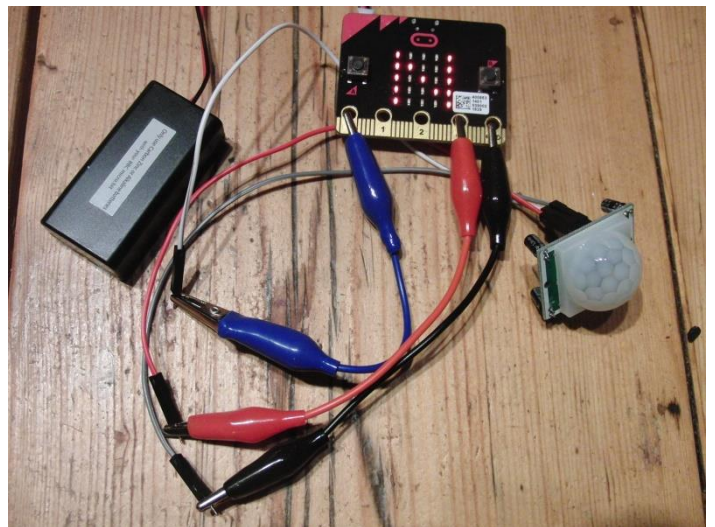
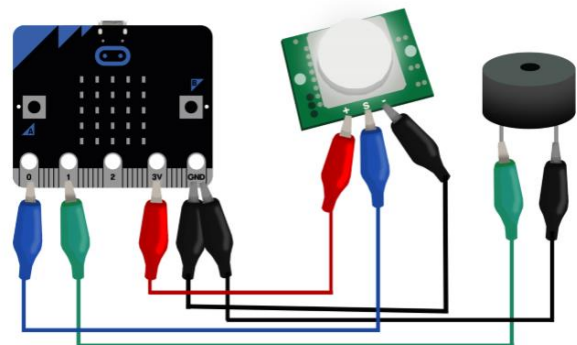


## 6. Detecting objects using infra-red and ultrasound sensors

It is less than a week since I started writing this article. Today I find that the PXT has been given a new look! So I will use this section to show a few of the changes it has just introduced. The first task I want to try is to design an intruder alarm. This is a project I found [on-line](#) at the BBC site.

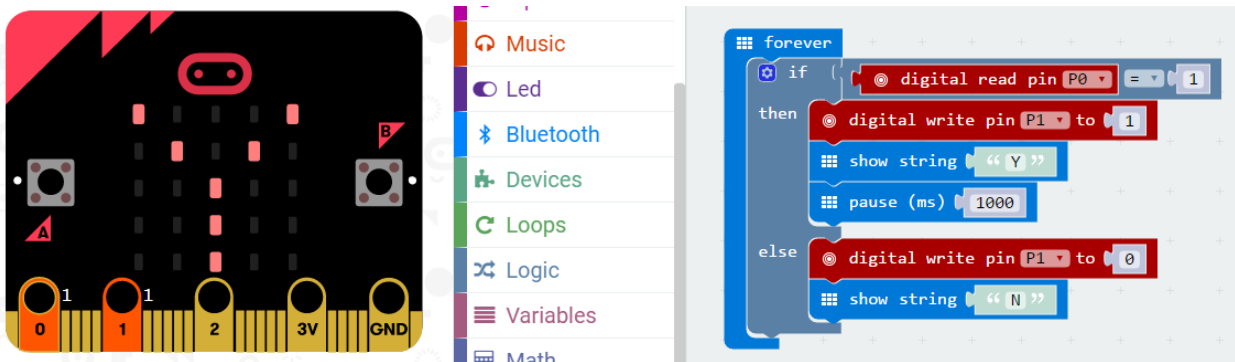
To simplify the project even further I will just display a “Y” or an “N” depending on whether or not the intruder is detected by the PIR (Passive Infra Red) and cut out the buzzer! You can buy a PIR very cheaply. I bought a pair from [here](#) for £5.33 including postage. There are 3 pins on the PIR for connections to GND, power and digital output. It’s a bit fiddly to connect crocodile clips to these pins so I used 3 connectors to bring the pins out so they can be more easily attached to the crocodile clips. The black goes to GND, the red to 3V and the blue to Pin0. The program is very simple. It just checks to see if Pin0 is set to 1 – showing when the PIR detects an object. In the emulator you can click on Pin0 to flip between 0 and 1 as the input.

Step 1: Creating your circuit

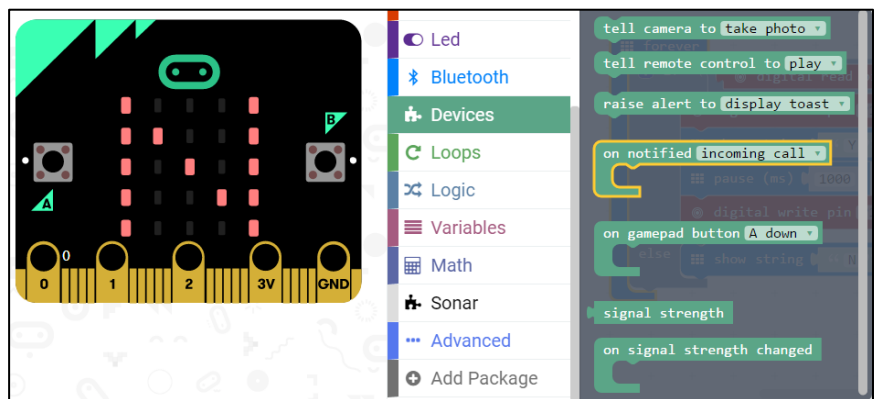




If you connect a buzzer between GND and Pin 1 you just need to add a couple of digital write commands.



The new look web site has integrated **'Add Package'** to the main list of types of block. One of these is called **'Devices'**. I have not tried this, but it looks as if you could use it take a photo of the intruder if you run the program on your Android or Apple mobile device!

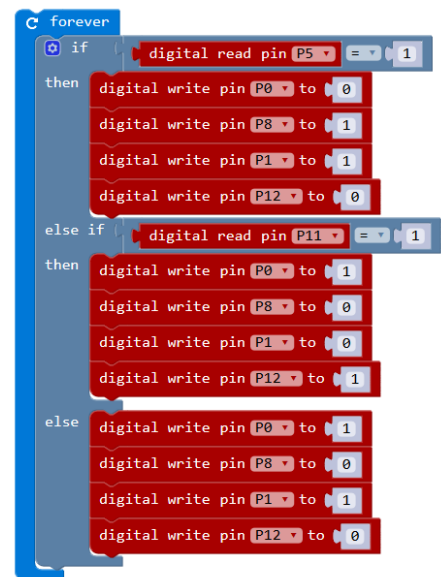


Finally we will take a look at 4Tronik's bit:bot [buggy](#) and explore its ultrasound distance sensor (usually called a motion sensor). The unit is very easy to assemble and has a pair of motors as well as LED emitters and detectors for use as a line-following robot. It also has a buzzer, two rows of 6 neo-pixels each and a pair of analog light sensors. At £30 it packs an impressive array of devices. The 4Tronix blog [here](#) gives building and programming instructions. Under the battery box is an edge connector for the micro:bit which connects its input and output pins to the various components on-board the buggy.



The blog contains programming hints. Here is their suggestion for using the line-following sensors to steer the buggy. The left and right sensor inputs are read on pins P5 and P11. The motor control outputs are on pins P0, P1, P8 and P12. There are also helpful hints on using the neo-pixel arrays, the light sensors and the buzzer.

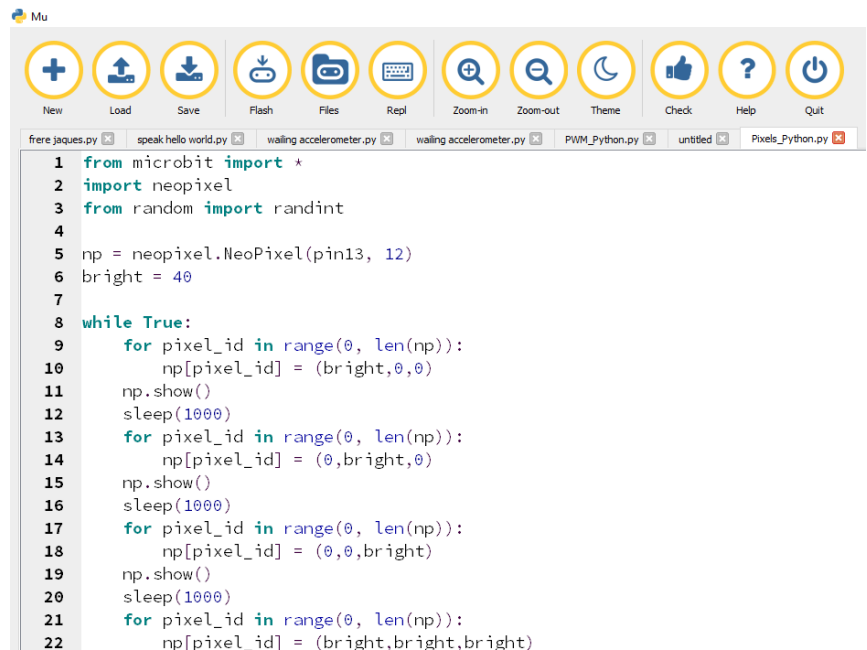
There are very helpful instructions posted on the MultiWingSpan [website](#).



Some of the programming examples are in the version of MicroPython written specifically for the BBC micro:bit. So that might be an easy way to take a look at how the Python programs compare with their PXT blocks and JavaScript equivalents.

Here is a screen shot of the MicroPython version to control the neo-pixels. As with the other editors you can debug the program on screen before you “Flash” the working version to the micro:bit.

One advantage of MicroPython is that there are extensive libraries of tools to make using the micro:bit much easier.

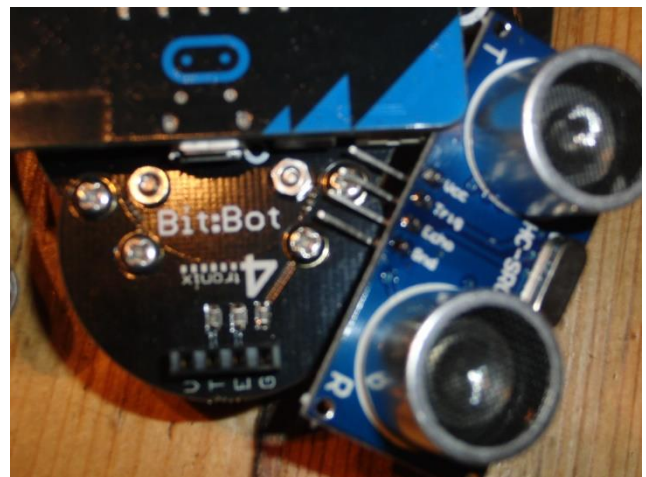


```

1 from microbit import *
2 import neopixel
3 from random import randint
4
5 np = neopixel.NeoPixel(pin13, 12)
6 bright = 40
7
8 while True:
9     for pixel_id in range(0, len(np)):
10         np[pixel_id] = (bright,0,0)
11     np.show()
12     sleep(1000)
13     for pixel_id in range(0, len(np)):
14         np[pixel_id] = (0,bright,0)
15     np.show()
16     sleep(1000)
17     for pixel_id in range(0, len(np)):
18         np[pixel_id] = (0,0,bright)
19     np.show()
20     sleep(1000)
21     for pixel_id in range(0, len(np)):
22         np[pixel_id] = (bright,bright,bright)

```

However I guess it won't be long before most of these get incorporated into the Microsoft PXT editor. We will see one example of this to make it easy to measure distances with the £3 [sensor accessory](#) for the 4Tronix bit:bug. This has four pins which slip into the extension port on the front of the buggy.



They are labelled Vcc (for voltage in), Trig (for the emitter Trigger), Echo (for the capture of the echo) and Gnd (for Ground). The idea is that an ultrasound signal is emitted from one side of the device, which is bounced back from the nearest object in the path, and is received in the other side. The time taken is measured and this is used to calculate the distance from the device to the nearest object. The device is designed to work with a 5V power supply. The 3 AAA battery pack provide 4.5V is fine. In order to use this device we need to install an extension package called **'sonar'** which has been written by a third party to extend the PXT's built-in blocks.



Here is the relevant information from the Micro:bit [website](#):

As you can see, both the 'neopixel' and the 'sonar' packages are examples of user contributed extensions. So let's see how to load in the 'sonar' package and find out what blocks it has.

## Packages

You can publish libraries (also known as packages or extensions) that users can then add to their scripts. These typically provide a driver for a particular hardware device you can connect to a microbit.

- [pxt-max6675](#) – TypeScript
- [pxt-neopixel](#) – TypeScript + ARM Thumb assembly package
- [pxt-sonar](#) – TypeScript
- [pxt-i2c-fram](#) – TypeScript
- [Sample C++ extension](#)
- [Sample TypeScript extension](#)

## Finding packages

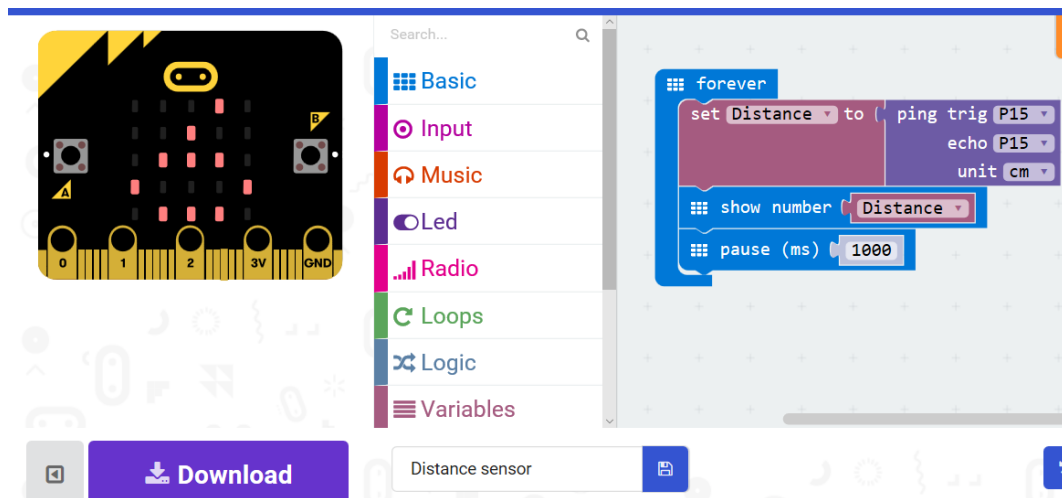
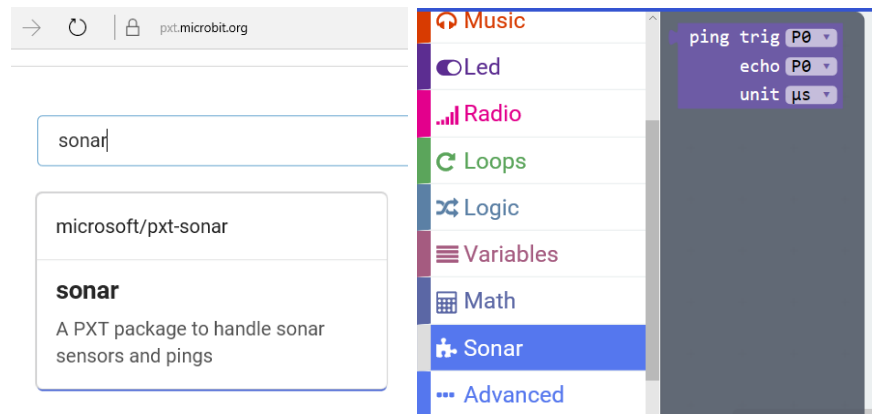
From the editor, the user clicks on **More** then **Add Package** and searches for the package.

To see the list of packages, click on **More** then **Show Files** to see the project file list.

To remove a package, click on the garbage button in the file list next to the package.

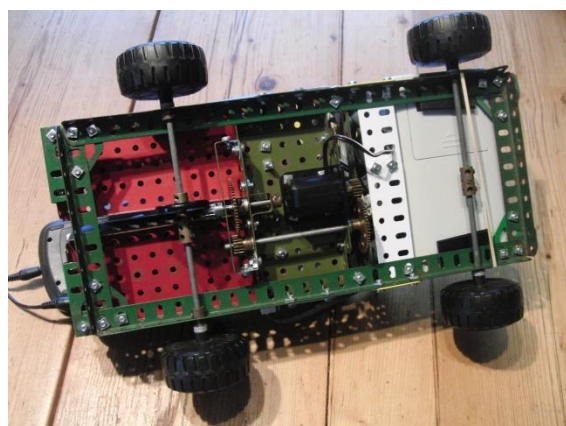
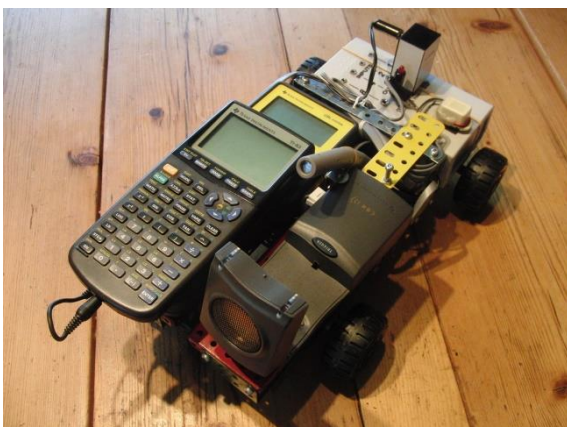


Use 'Add Package' and then search for 'sonar'. Click on the suggested extension and it will appear in the central column of types of block. Click on the 'Sonar' package and the one new block appears. For our sensor both the trigger and the echo use Pin 15, and we want the result as a value in cm. Here is the completed test program.



## 7. Summing it all up: a Mars crawler project

This is a picture of Bridget, the test vehicle build by Airbus Space & Defence in Stevenage to explore the surface of Mars. 20 years ago I built my own simulator using Meccano with an electric motor, a TI-83 graphical calculator as the computer, and various expensive components to sense light, distance and to control the motor.



Now we have the micro:bit and a range of off-the-shelf components to make a much cheaper, simpler and smaller equivalent. I'll use the 4Tronix bit:bot buggy with the ultra-sound range-finder as above. You could do the same, or similar, using other kits and components.

The basic program for a test run will look something like this (expressed in words)

1. While there is not enough light, stay still;
2. When there is enough light, move forward;
3. Until you sense something in your way;
4. Then stop and sound the alarm;
5. Then reverse a small distance.

The bit:bot has two light sensors. Output 0 or 1 to Pin 16 to select which to read, Pin 2 is the analog output. Power to the left motor is set through Pin 0 and direction by Pin 8. Use Pins 1 and 12 for the right one. The neopixels are on Pin 13 and the buzzer on Pin 14. The range-finder uses Pin15. We will develop the program in sections. Here is section 1.

The variable `'light'` holds the current light reading detected by the left and right sensors at the ends of the now-pixel bars. The variable `'minlight'` holds the threshold value we need `'light'` to exceed to declare that the sun is up over Mars. The green `'while'` loop just keeps reading the light sensors until the sun rises. We send 0 to Pin16 and then read the light value from one sensor. We then send 1 to Pin16 and then read the light value from the other. The final value of `'light'` is the average of these readings. To test the program we show the reading on the micro:bit's LED array. When we have enough light we move out of the green loop and display a message while sounding the on-board buzzer at Pin 14 for 1 second. Now we have developed the first of the part of the program we can develop sections 2 and 3.



We will use the motor controls to make both run forward at lower power. We will use the ultra-sound range finder to detect if there's anything in the way. This will use another `'while - do'` loop. It turns out that occasionally the ultra-sound range finder throw up a spuriously low reading! So I have to use a counted loop to add up 5 successive sonar reading and take their average. This seems to do the trick. In order to stop the vehicle we change the power to both motors to zero by analog writes to Pins 0 and 1.

Perhaps you would like to try coding this for yourself before I tell the real truth?

OK, now's the time to come clean. The program we have just been developing should work fine in theory. But in practice I ran into four very different and unexpected problems. The first was my own fault for not having read the instructions about the bit:bot's motor control carefully. I assumed that for each motor you sent a digital output of 0 to go forward and 1 to reverse. Also that you sent a value of between 0 and 1023 to control the speed. But when I switched a motor to reverse and selected a speed of 0 it did not stop, as I would have expected. It roared away at great speed! That's when I followed the link to the

“MultiWingSpan” site and checked out the [motor page](#). So when you are in reverse you have to subtract the speed you want from 1023! So when the motor’s direction is forward (0) then 1023 is its max speed and 0 is rest. When it is in reverse (1) then 0 is its max speed and 1023 is at rest!

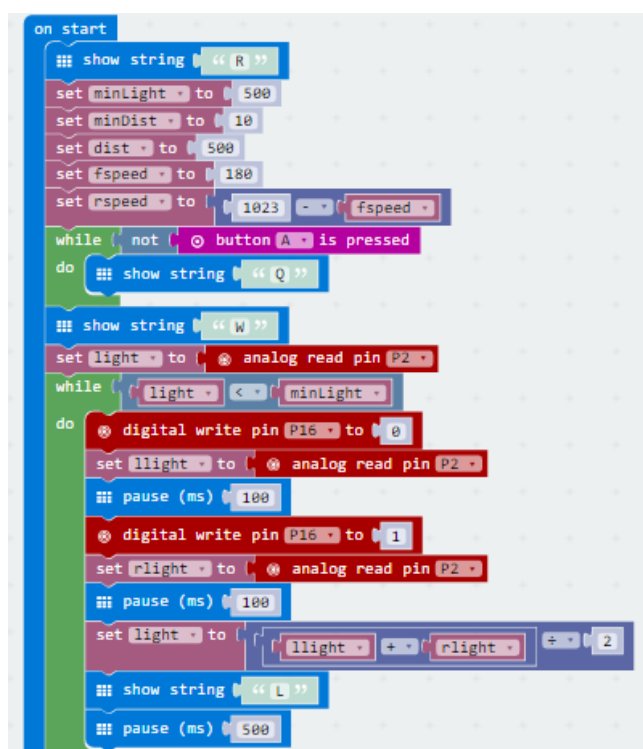
The second problem is a design feature of the bit:bot, which was very frustrating – but turned out to be resolvable. As we have seen the bit:bot is crammed with components which connect to the micro:bit. It turns out, that, because of the limited number of pins available on the micro:bit, two of the components share the same input pins to the micro:bit as the A and B buttons. These are the two light dependent resistors on the underside of the chassis which are used, together with the two LEDs, to sense a black line when being used as a line-following vehicle. The effect is that, when you place the bit:bot down on a surface, the micro:bit thinks you have pressed the A and B buttons down together, and so enters “**Pairing Mode**”! So a program which works when you hold the bit:bot off the ground, never gets a chance to run when placed on the ground. Fortunately an e-mail to the support line at 4Tronix revealed the issue. The solution I came up with is to start running the program by turning on the power switch with the bit:bot off the ground, and to start the program with a *‘while’* loop which waits until button A is pressed. Then when you put the bit:bot on the floor, one of the LDRs will send a signal to the micro:bit on the same pin as button A, and then your program will start to run!

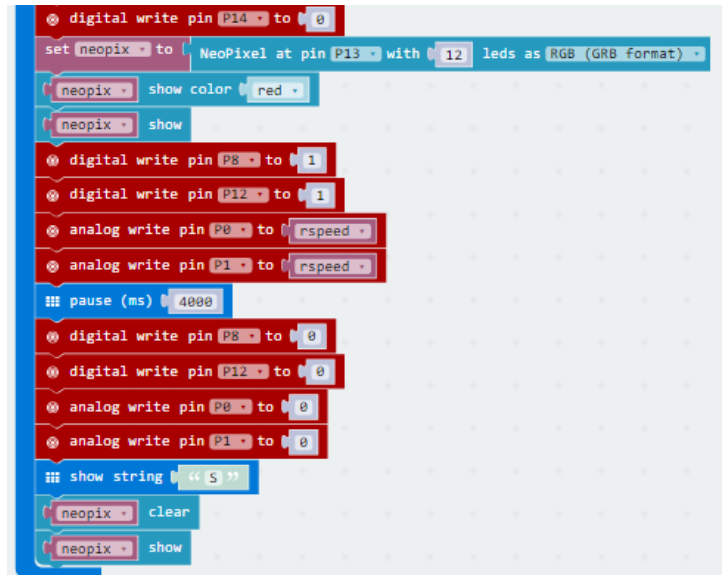
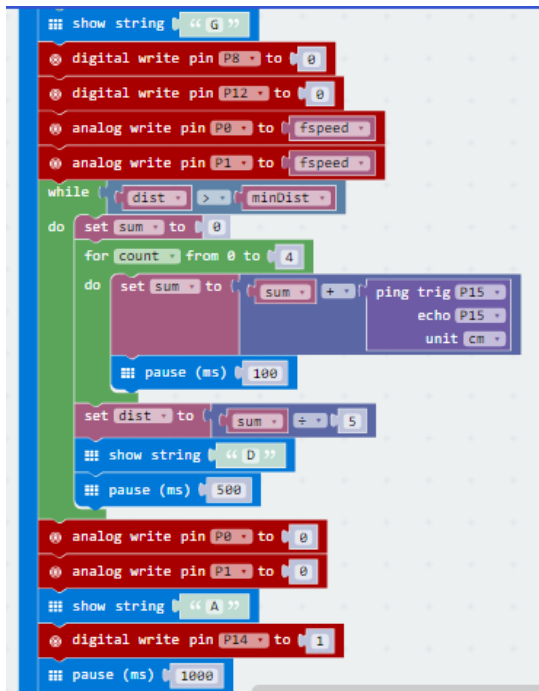
The third issue was a case of a dodgy connection. The ultra-sonic range-finder for the bit:bot is mounted on a little plate which is screwed to the chassis with four small nuts and bolts. Stupidly I dropped one of the nuts on the kitchen floor and have been unable to find it! So I had 3 secure connections and one with just the bolt held in place by gravity. It turns out that these nuts and bolts are also needed to pass the 4 signals for the connections between the sonic sensor and the micro:bit. So, when the bit:bot was crawling forward, the vibrations made the unsecured bolt shake, and hence convey an intermittent signal! So I have now improvised a nut using kitchen foil! (Presumably this was the reason for the intermittent results mentioned earlier for which I had come up with a software “cure” by averaging 5 readings.)

The fourth issue was within the Microsoft PXT editor. As I have learned, this is still under development. I was completely baffled when a program which ran fine on the emulator, got stuck in the middle of a program. That’s when I decided to put a liberal scattering of commands like *‘show string “C”* throughout the program. I found that my program just stuck with a “W” on the display.

At that point I had been using the PXT editor on the micro:bit site <https://pxt.microbit.org>. So I Googled “PXT editor” and found the alternative site: <https://www.pxt.io>. I compiled my program from that site, and was delighted and relieved to find that it worked fine!

So I hope you don’t have too many problems. But the moral is that we all need a bit of help from our friends from time to time! So don’t be afraid to ask for help and advice – you can save a lot of time and frustration as well as making new friends. Here is the complete, debugged, working program.





**Now is the time to tell people about what you have been doing!**

Once you have designed, tested and de-bugged some interesting systems of your own, please write up what you have done to share it with others. Better still, create a YouTube video and upload it together with your notes and hex program files to somewhere you can share it. One possible site is this [wikispace](#). The STEM Learning site has groups for [Student Digital Ambassadors](#) and for the [BBC micro:bit](#). Then send the links to your friends. I have just uploaded a video of the bit:bot Mars Crawler onto my [Dropbox](#) space.

Happy tinkering!